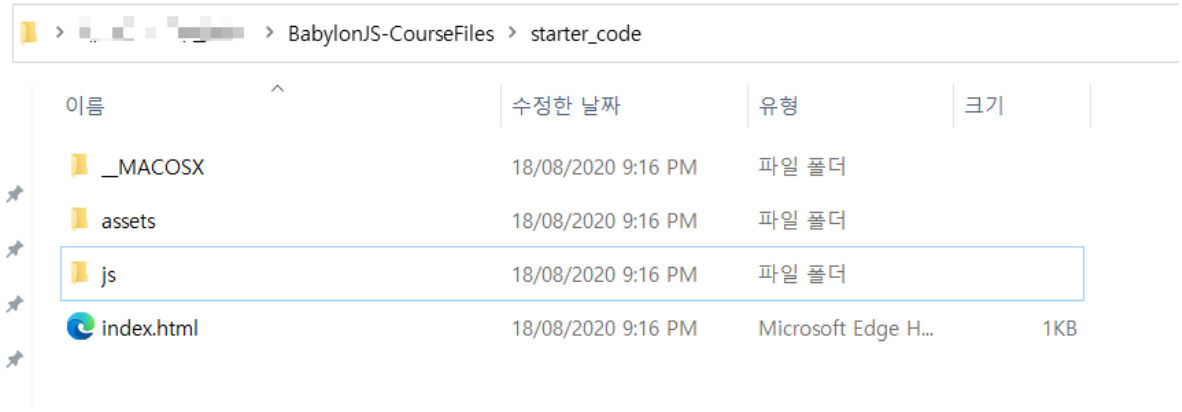
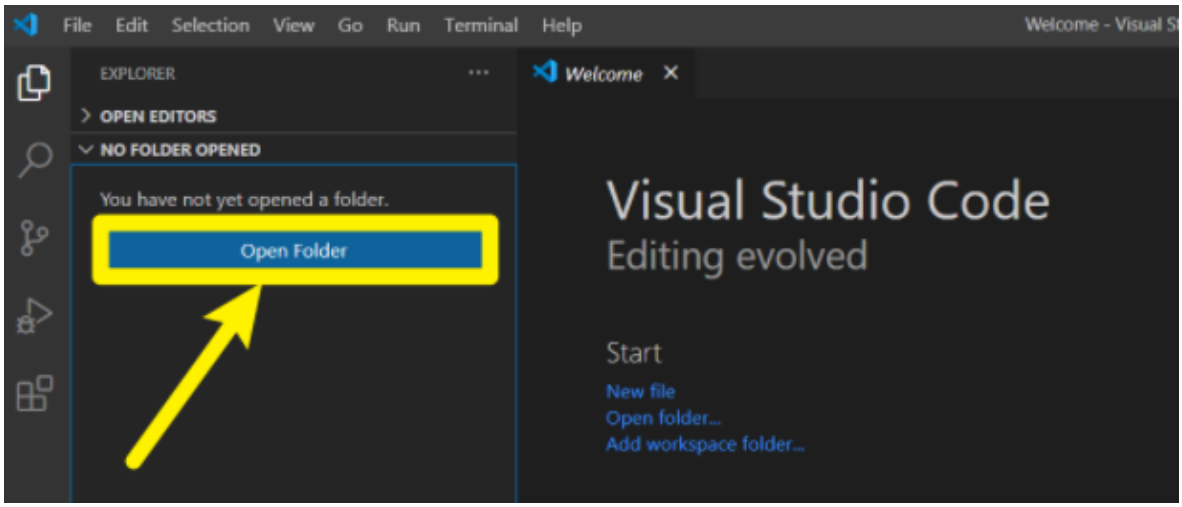
To get started, we're going to set up our development environment.

Download the course files, and extract the contents of **starter_code.zip**.



Open up **Visual Studio Code**, click on 'Open Folder'.



Choose the folder that we extracted earlier, i.e. **starter_code**.

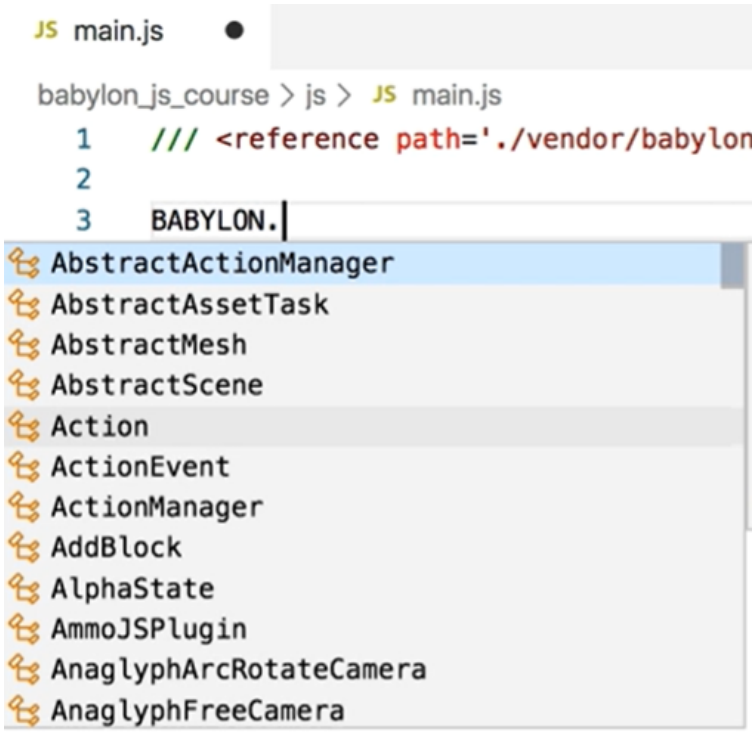
Course Files Overview

The **index.html** have these two scripts included: **babylon.js** and **main.js**.

The first one is for the babylon.js library. There is also a **type definition file** named **babylon.d.ts**, which is loaded by this reference code in main.js.

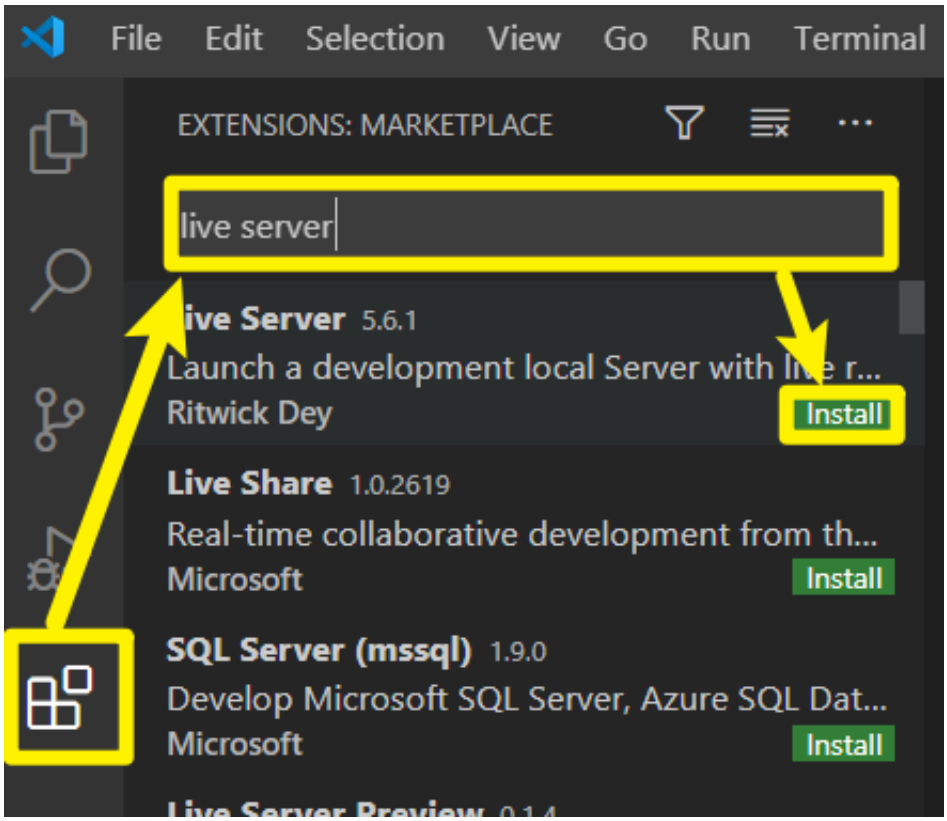
```
<reference path='./vendor/babylon.d.ts' />
```

It allows us to access to all of the properties and the methods that are available on this global object called **BABYLON**.

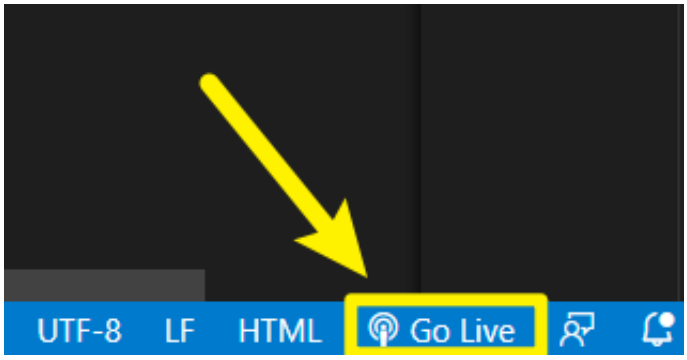


Installing Live Server Extension

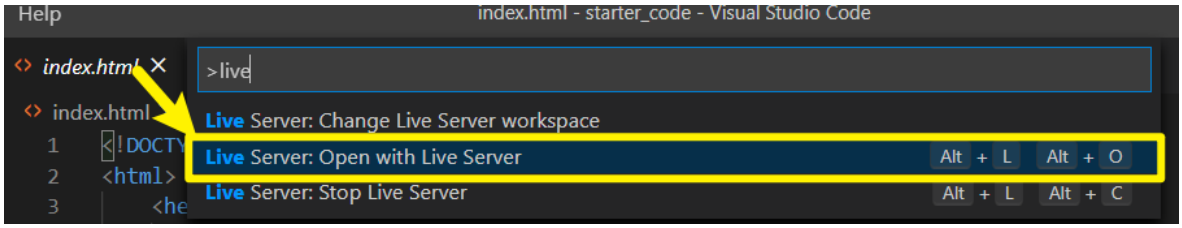
Open up the **Extensions** tab, and search for “Live Server”.



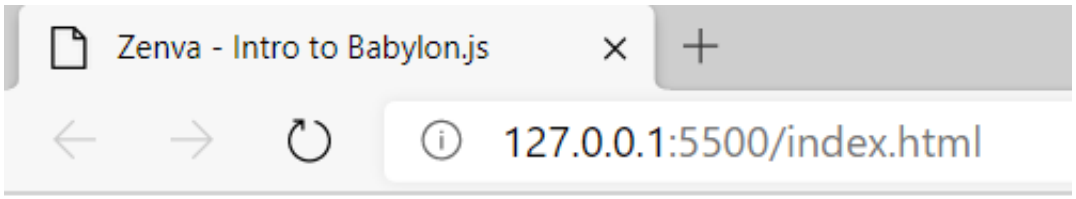
Once it's installed, and click on the **'Go Live'** button on the bottom left corner of the screen.



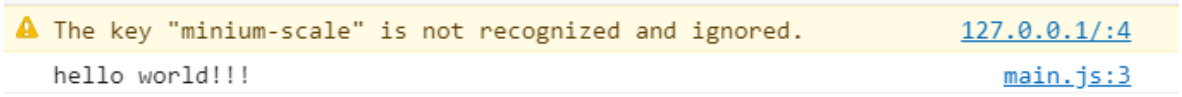
Alternatively, you can go to **View > Command Palette**, and search for "**Live Server: Open with Live Server**".



A new tab should open in your default browser at **port 5500**.



If you open up the **console (F12)** it should say “hello world!!!”.



Note:

- Browser security will prevent you from loading files from the local file system: file://
- By using a local web server, you can serve up files and access them over http.

With the development environment set up, we can start creating our first **3D scene**.

Learning Objective

- Understand the coordinate system in Babylon.js
- Create a scene in Babylon.js
- Render a basic scene with a light and a camera

Creating A Canvas Element

In our **index.html** file, we're going to create a canvas element in **<body>**:

```
<canvas id="renderCanvas"></canvas>
```

Next, we're going to style our page in **<head>:**

```
<style>
    canvas {
        width: 100%;
        height: 100%;
    }
    html, body {
        margin: 0;
    }
\</style>
```

And we're going to switch to **main.js** and render our canvas.

```
// Get our canvas
const canvas = document.getElementById('renderCanvas');
```

Creating A BabylonJS Engine

After setting up the canvas, we're going to create a new engine.

BabylonJS engine is a **WebGL-based** engine, and it is required to load up everything we need in our 3D scene, including objects, cameras, lights, etc.

```
// Create a BabylonJS engine
const engine = new BABYLON.Engine(canvas, true);
```

Note that we are passing the canvas that we just created in the parameter.

Creating A Scene

To create a scene, we're going to create a new function called **createScene( )**.

Inside the function, we're going to create a new **Babylon Scene**, a **FreeCamera**, and a **HemisphericLight**.

```
function createScene(){

// Create a 3D Scene
const scene = new BABYLON.Scene(engine);

// Create a Camera
const camera = new BABYLON.FreeCamera('camera', new BABYLON.Vector3(0, 0, 0), scene)

// Create a Light
const light = new BABYLON.HemisphericLight('light', new BABYLON.Vector3(0, 1, 0));

return scene;
}
```

For more information, please visit the documentations:

- Scene – https://doc.babylonjs.com/api/classes/babylon.scene
- FreeCamera – https://doc.babylonjs.com/api/classes/babylon.freecamera
- HemisphericLight – https://doc.babylonjs.com/api/classes/babylon.hemisphericlight

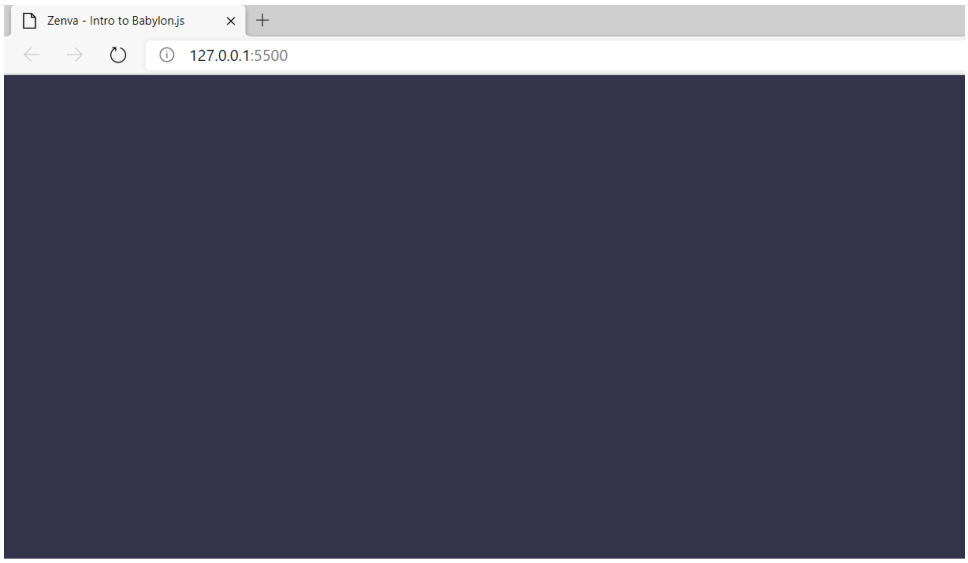
Then, we need to call the function **createScene** to actually create the scene.

```
// Create our scene
const scene = createScene();
```

Lastly, we're going to tell the engine to **render** the scene to our canvas element.

```
engine.runRenderLoop(() => {
    scene.render();
})
```

Once that's done, you should be able to see that the background colour is now added to our canvas.



Summary

- Scenes get rendered in the render loop
- A camera gives you “eyes” into this 3D world
- Hemispheric light represents ambient light

In this lesson, we're going to add **3D objects** to our scene.

Learning Objective

- Create Basic Shapes and Parametric Shapes

Types of BabylonJS Elements

In BabylonJS, we have two different types of elements that we can create.

(1) We have **Basic elements**, which are basic shapes such as a box, a sphere, or a plane.

(2) We have **Parametric shapes**, which can be formulated by custom mathematical data. For example, we can create multiple lines by providing coordinates and the lines will be drawn between these points.

Creating Basic Shapes – (1) Box

The **Mesh Builder** function allows us to call the method of the shape that we want to create.

For example, we can create a box shape by calling **MeshBuilder.CreateBox()**.

```
// Create a Box
const box = BABYLON.MeshBuilder.CreateBox('box', {}. scene);
```

The first parameter **' '** takes in a **name** of the shape, we're simply calling it 'box'.

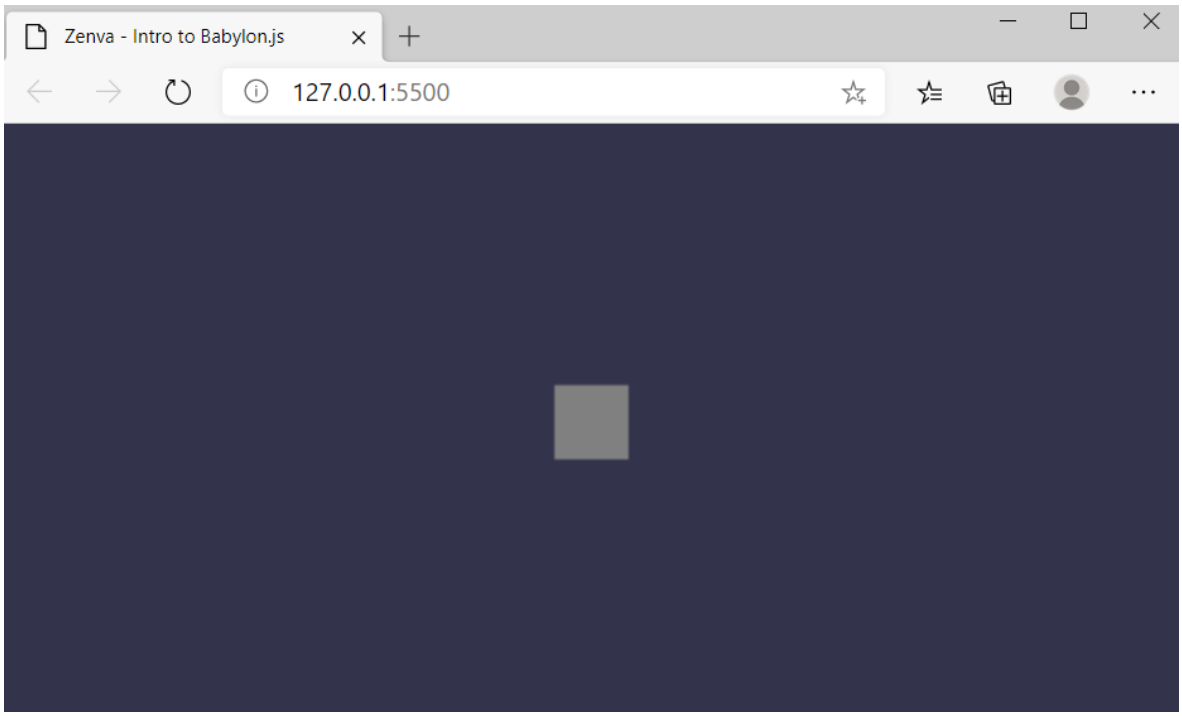
The second parameter **{ }** allows us to set the **options** of the shape, such as width, height, etc.

The third parameter is to define the hosting **scene** that will create the shape.

Note that our box is currently being placed at the exact same position as our camera.

```
// Create a Camera
const camera = new BABYLON.FreeCamera('camera', new BABYLON.Vector3(0, 0, -5), scene)
```

After we updated the position of our camera, we can see that a box is created in our scene.



To change the size of the box, go ahead and simply add a **size:** property inside the curly brackets of the second parameter.

```
// Create a Box
const box = BABYLON.MeshBuilder.CreateBox('box', {
    size: 1
}, scene);
```

Creating Basic Shapes – (2) Sphere

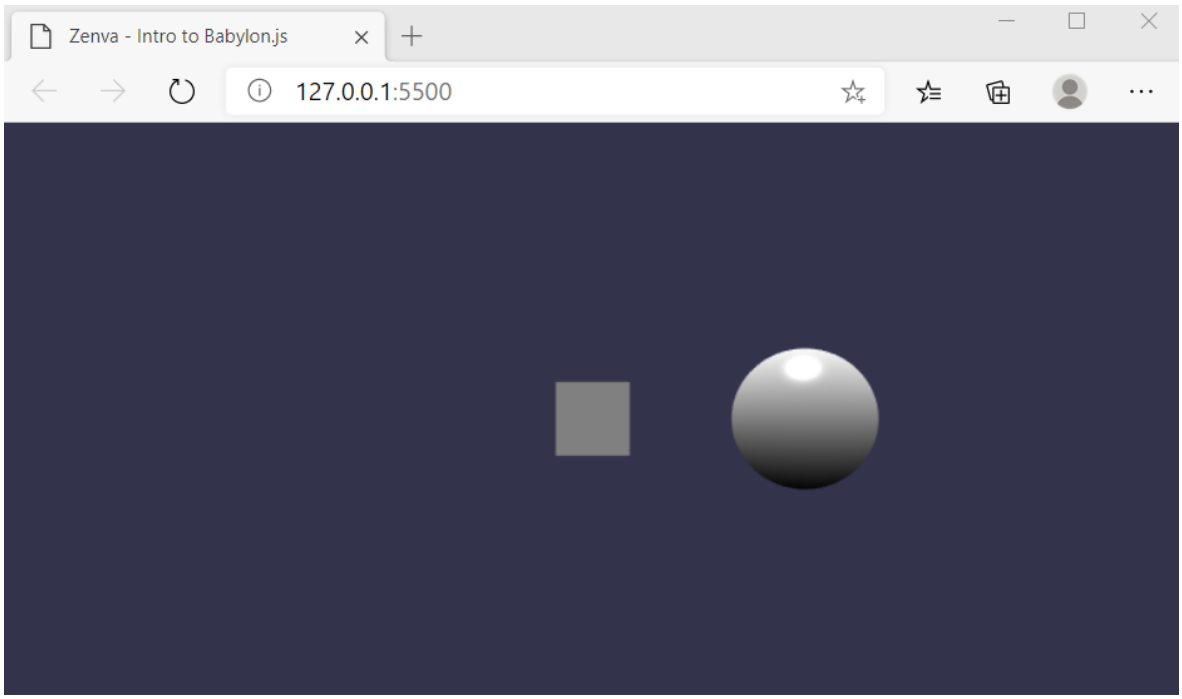
Similar to creating a box, we can call **MeshBuilder.CreateSphere()** to create a sphere.

```
// Create a Sphere
const sphere = BABYLON.MeshBuilder.CreateSphere('sphere', {}, scene);
sphere.position = new BABYLON.Vector3(3, 0, 0);
```

This time, we've manually set the sphere's position to be at (3, 0, 0).

We can also set the **diameter** and the number of **segments** of our sphere. (By default, the segments property is set to 32 and the diameter is 1.)

```
// Create a Sphere
const sphere = BABYLON.MeshBuilder.CreateSphere('sphere', {
    segments: 32,
    diameter: 2,
}, scene);
```

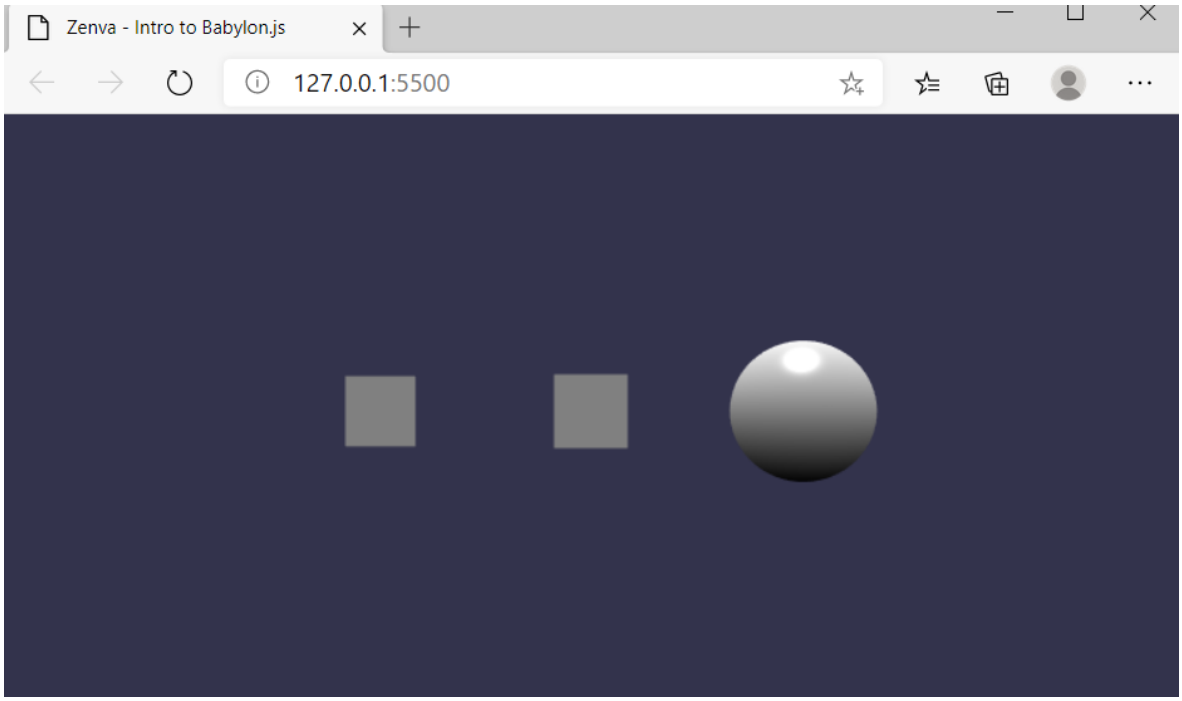



Creating Basic Shapes – (3) Plane

Similar to creating a box/sphere, we can create a plane by calling **MeshBuilder.CreatePlane()**.

```
// Create a Plane
const plane = BABYLON.MeshBuilder.CreatePlane('plane', {}. scene);
plane.position = new BABYLON.Vector3(-3, 0, 0);
```

However, it is quite difficult to distinguish the plane from the box.



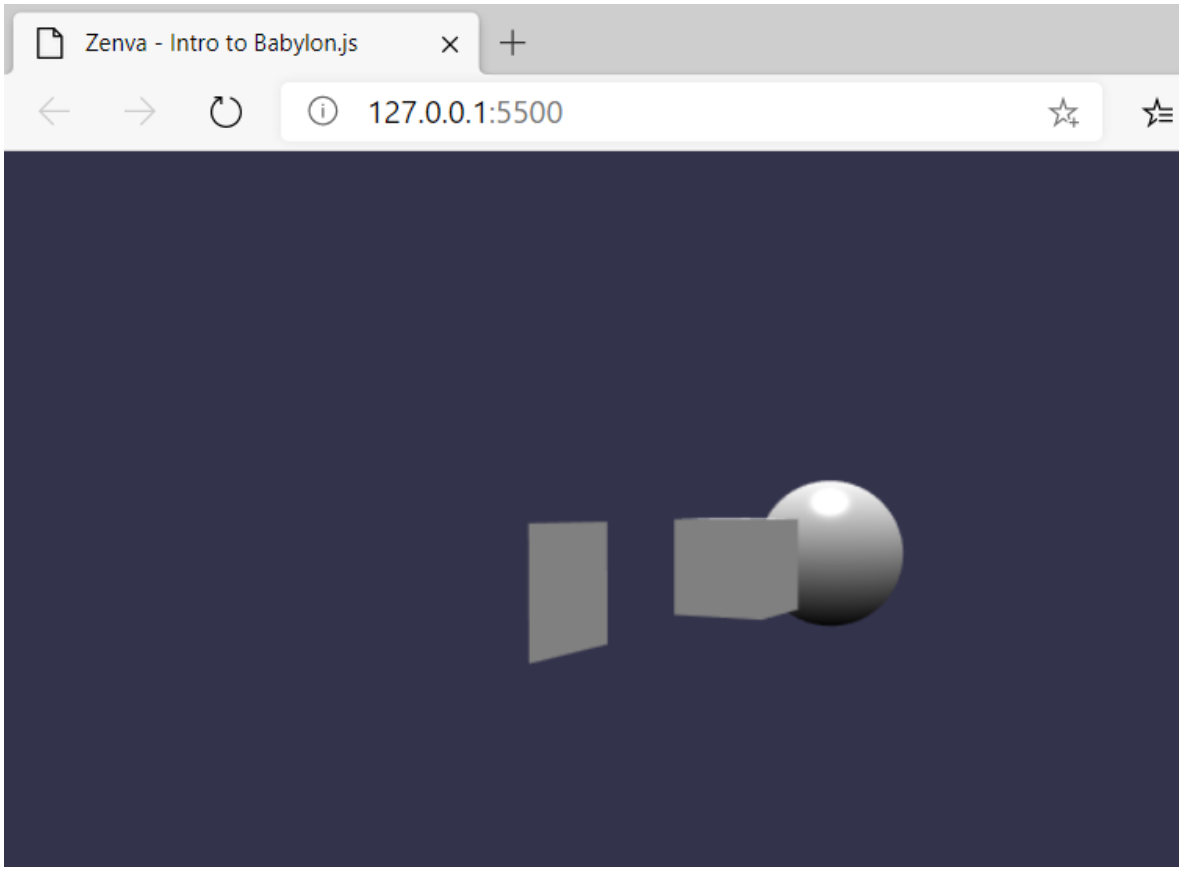
To fix this, we're going to add a controller to our camera.

Adding A Controller To Camera

We can add a controller to our camera by simply calling the **attachControl()** function.

```
camera.attachControl(canvas, true);
```

Now we can click on the canvas to **rotate** the view, and use the arrow keys to **move** our camera.



Creating A Parametric Shape

To create a **parametric shape**, we need to provide some type of mathematical data. In this example, we're going to draw a line consisting of multiple points.

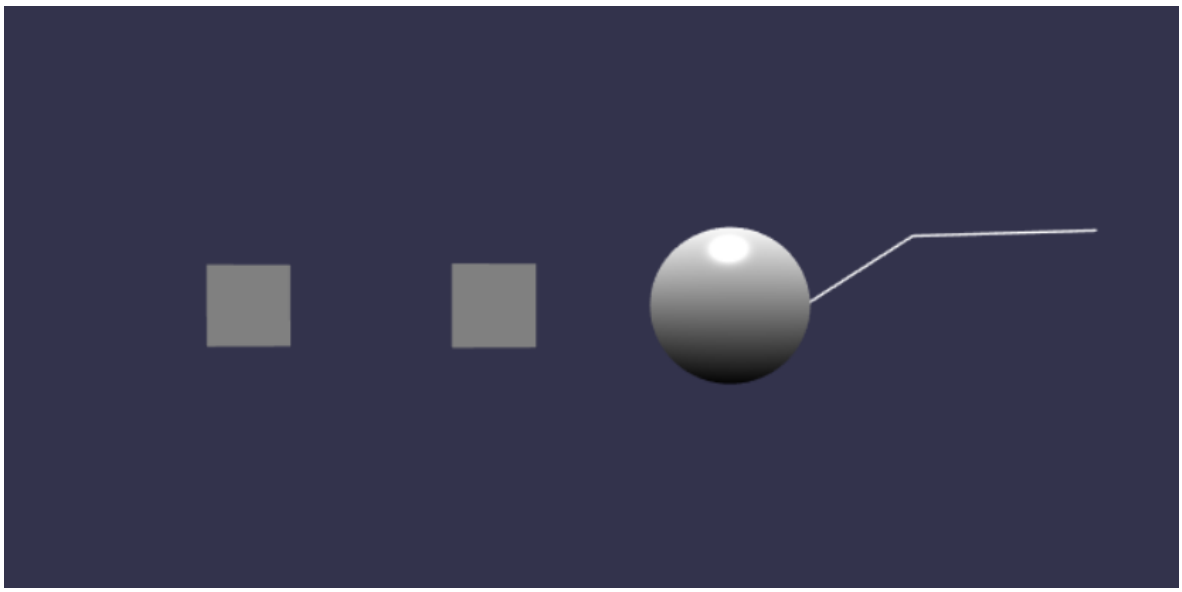
First of all, we're going to create three points at different locations.

```
// Create a Line
const points = [
    new BABYLON.Vector3(4, 0, 0),
    new BABYLON.Vector3(6, 1, 1),
    new BABYLON.Vector3(8, 1, 0),
];
```

Then, we're going to call the **MeshBuilder.CreateLines()** function and pass in our points data.

```
const lines = BABYLON.MeshBuilder.CreateLines('lines', { points }, scene);
```

Now there's a line created in our scene.

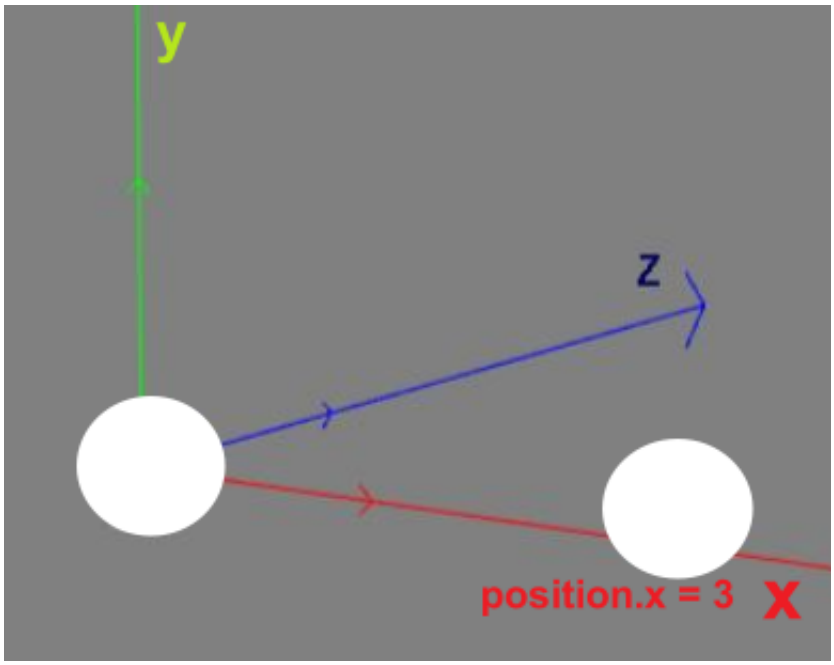


In this lesson, we're going to learn how to update the **position**, the **rotation**, and the **scale** of shapes.

Moving Objects

To update the position of an object, we have to provide a **Vector3** containing three coordinate values **(x,y,z)**. These values are in relation to the **origin** of the world axes (0, 0, 0).

```
sphere.position = new BABYLON.Vector3(3, 0, 0);
```



We can also update the position on an individual axis.

```
sphere.position.x = 3;
```

For more information, visit the documentation: https://doc.babylonjs.com/babylon101/position

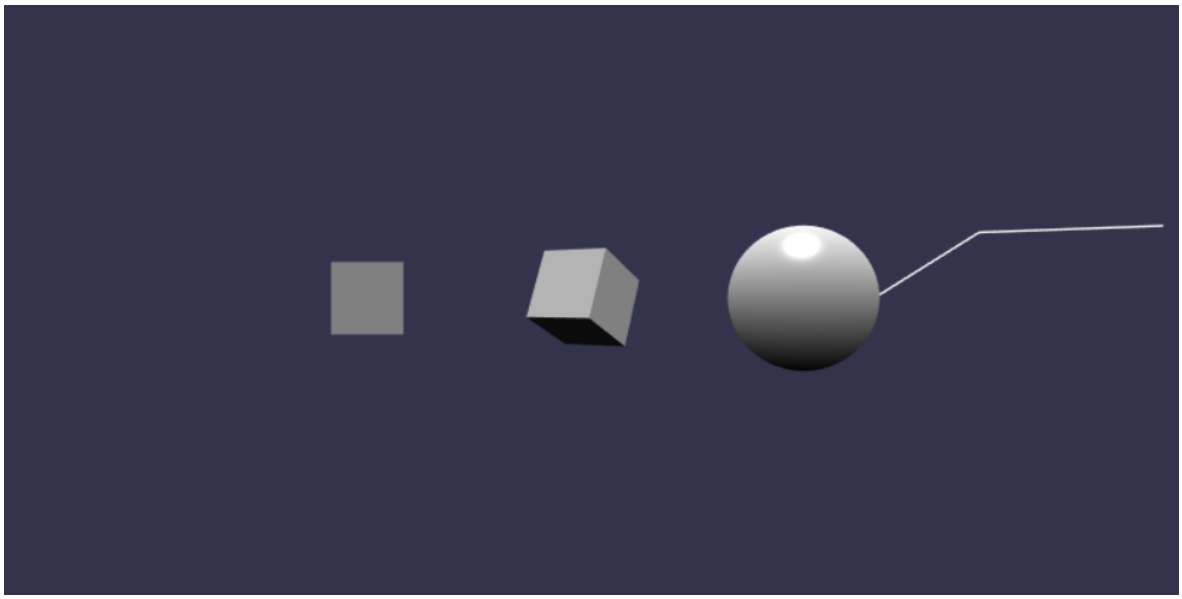
Rotating Objects

Similar to updating the position, we can provide a **Vector3** to rotate an object.

```
box.rotation = new BABYLON.Vector3(20, 10, 0);
```

or set the degree of rotation for each axis.

```
box.rotation.x = 20;
box.rotation.y = 10;
```



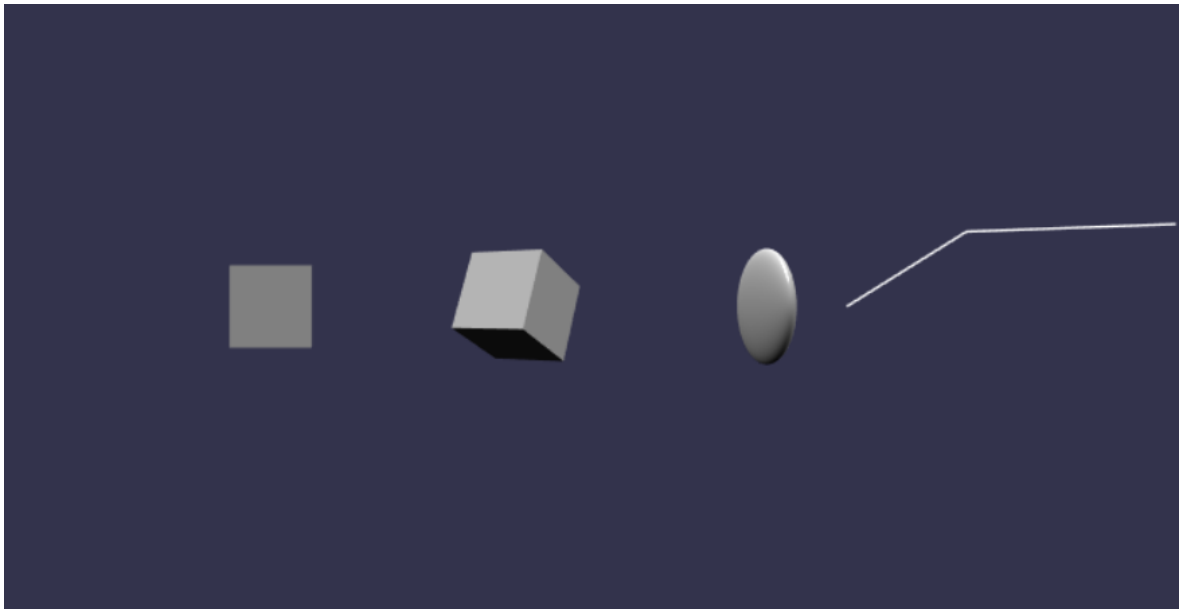
Scaling Objects

By default, the scale of an object is always set to (1, 1, 1). You can provide a new **Vector3** to scale along the x, y and z axes,

```
sphere.scaling = new BABYLON.Vector3(0.2, 0.7, 1);
```

or you can scale on individual axes by:

```
sphere.scaling.x = 0.2;
```



```
//This is the default size of the sphere.
sphere.scaling = new BABYLON.Vector3(1, 1, 1);
```

```
//This code below will double the size of the sphere.
sphere.scaling = new BABYLON.Vector3(2, 2, 2);

//This code below will halve the size of the sphere.
sphere.scaling = new BABYLON.Vector3(0.5, 0.5, 0.5);
```

In this lesson, we're going to modify **materials** of objects in BabylonJS.

Learning Objectives

- Understand the concept of materials
- Assign a diffuse color or texture to materials

Creating Standard Materials

Materials allow us to cover our meshes in color and textures. To create a material, simply call **BABYLON.StandardMaterial**.

```
const material = new BABYLON.StandardMaterial('material', scene);
```

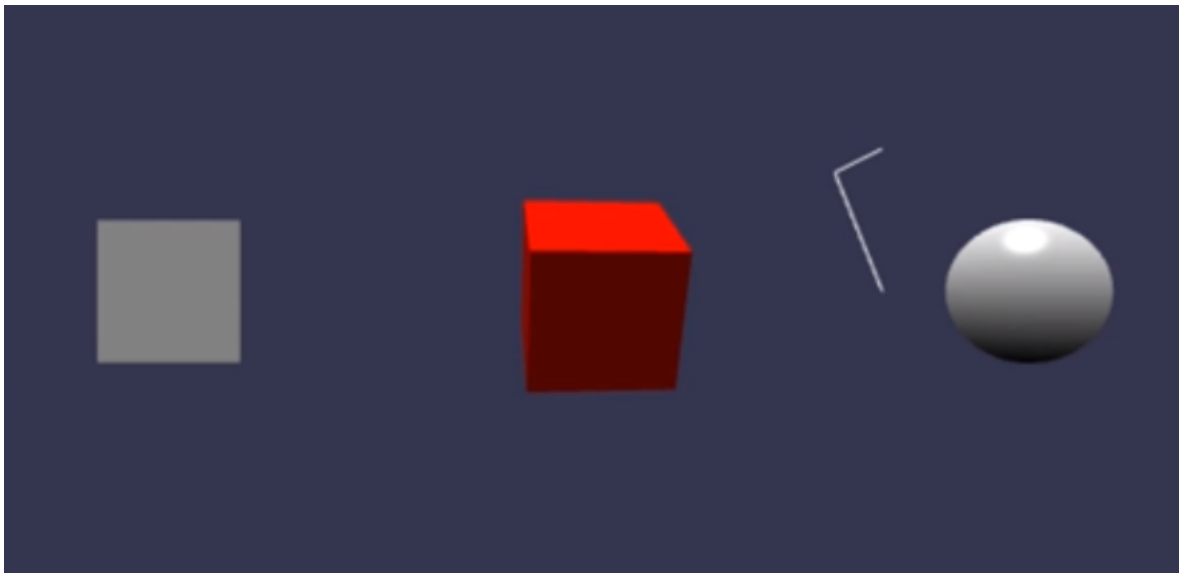
Applying Colors To Materials

We can assign a new diffuse color to the material by calling **BABYLON.Color3** and passing RGB values (between 0 and 1).

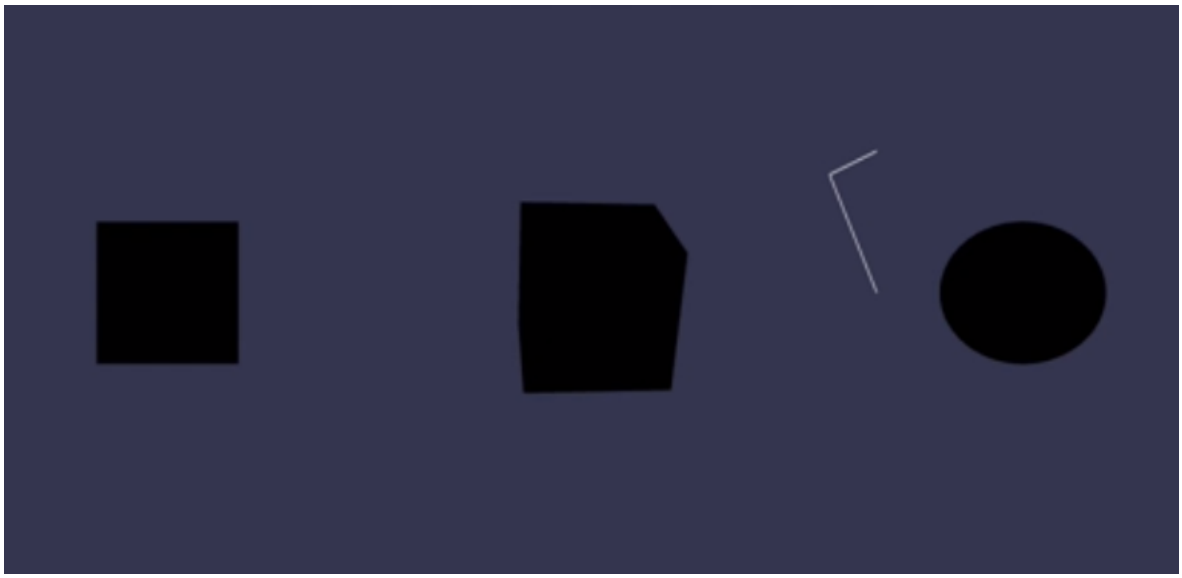
```
//White Color
material.diffuseColor = new BABYLON.Color3(1, 1, 1);

//Green Color
material.diffuseColor = new BABYLON.Color3(0, 1, 0);

//Black Color
material.diffuseColor = new BABYLON.Color3(0, 0, 0);

//Red Color
material.diffuseColor = new BABYLON.Color3(1, 0, 0);
```

We can then **apply** the material to our objects.

```
box.material = material;
```

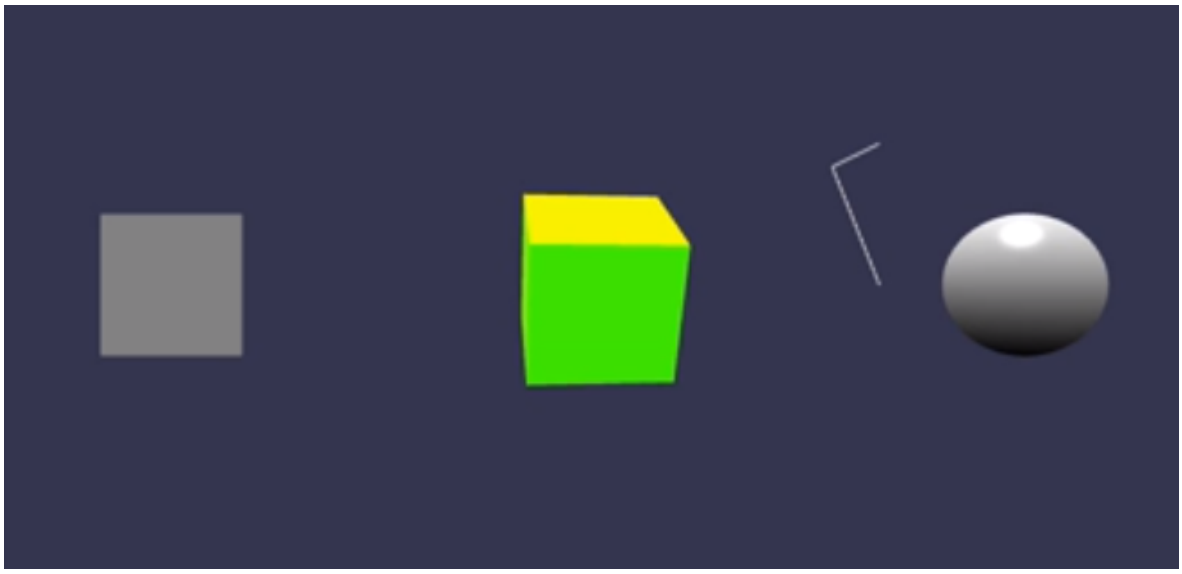


Note that we need to place a light source in our scene in order to make diffuse colors visible. If there is no light in the scene, the materials will just appear black.



We can also make the material to emit light by setting its **emissive** color values.

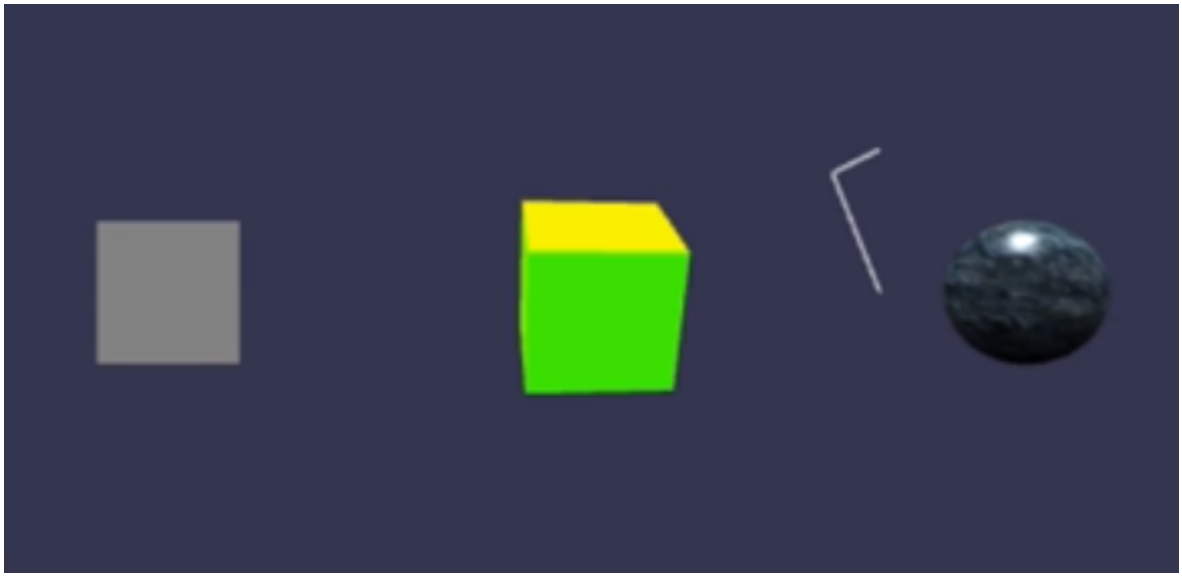
```
//Emit Green Light
material.emissiveColor = new BABYLON.Color3(0, 1, 0)
```



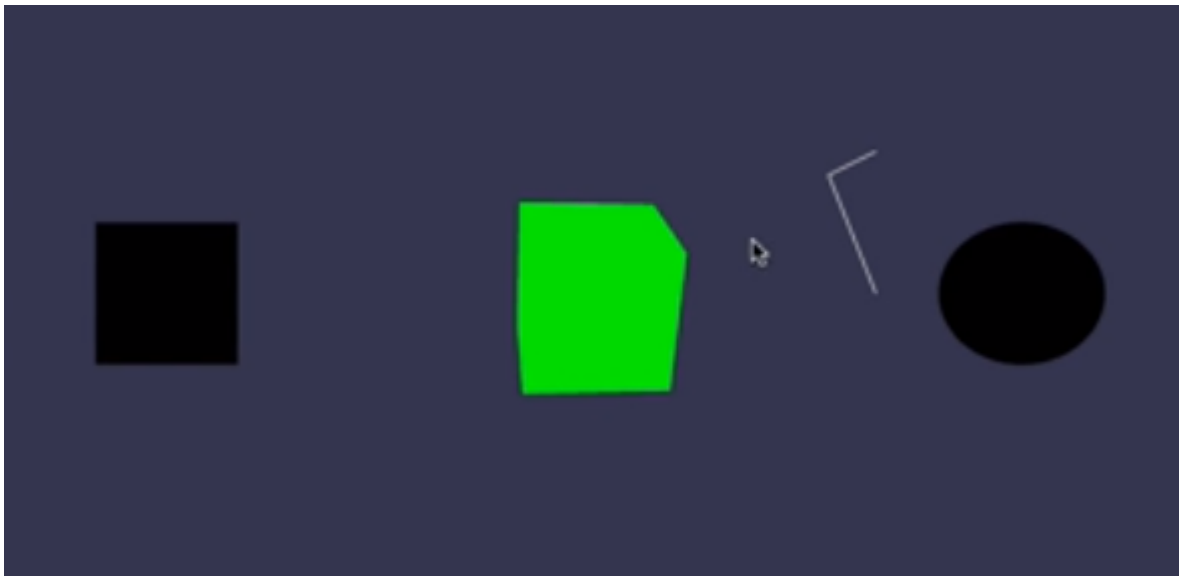
Applying Textures To Materials

Textures allow us to use external images to modify the look of our objects. Let's create another material and apply a texture to it.

```
//Creating a material
const material2 = new BABYLON.StandardMaterial('material', scene);

//Applying a diffuse texture
material2.diffuseTexture = new BABYLON.Texture('assets/images/dark_rock.png', scene);

//Applying the texture to our sphere
sphere.material = material2;
```



Similar to colors, there are **diffuse** textures and **emissive** textures. When we remove the light source from our scene, the diffuse texture is no longer visible.



Summary

- A material allows you to give shapes color, texture and light properties
- The diffuse color is the color the shape has when exposed to light
- Instead of color, you can define a diffuse texture using an image file

In this lesson, we're going to explore different types of lights and cameras.

Learning Objectives

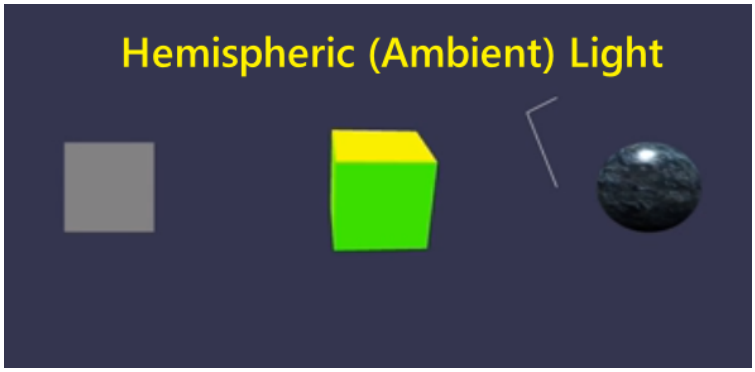
- Create point and directional lights
- Create universal and follow cameras

Different Types of Lights

There are four types of lights that can be used with a range of lighting properties.

Currently, we're using a **hemispheric** light (= ambient environment light) to light up our scene:

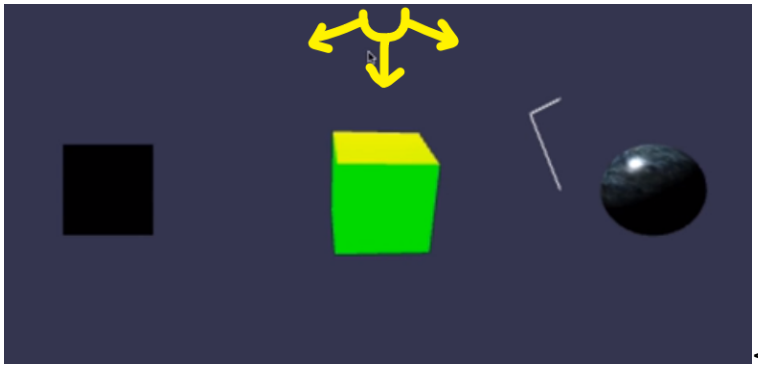
```
//Creating a Hemispheric Light

const light = new BABYLON.HemisphericLight('light', new BABYLON.Vector3(0, 1, 0), sce
ne);
```



Different Types of Lights – Point Light

```
//Creating A Point Light

const light = new BABYLON.PointLight('light', new BABYLON.Vector3(0, 5, 0), scene);
```

A point light can be defined by a unique point in world space. The light is similar to a light bulb, which means it can be emitted in every direction from this point.

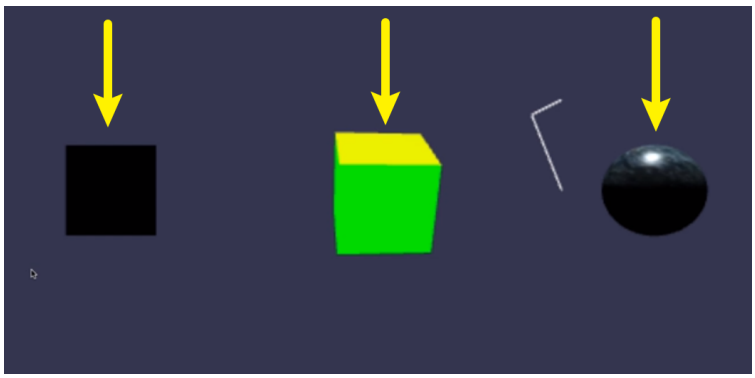

<Point Light

Different Types of Lights – Directional Light

A directional light is a light that can be applied in one direction, meaning it will be emitted from everywhere in the specified direction and has an infinite range. It is often used to simulate sunlight.

```
//Creating A Directoinal Light (Vector3 represents the direction of the light, not th
e position)

const light = new BABYLON.DirectionalLight('light', new BABYLON.Vector3(0, -1, 0), sc
ene);
```

Since we're pointing in the negative y-direction, the light will come from the top of our scene to the bottom.



Free Camera

The camera we're currently using in our scene is the **Free Camera**, which can detect inputs from a keyboard or a mouse.

```
//Create a Free Camera

const camera = new BABYLON.FreeCamera('camera', new BABYLON.Vector3(0, 0, -5), scene)
;
```

If you're planning to make an app for multiple platforms, it is recommended to use the **Universal Camera**.

Universal Camera

This extends from the Free Camera– it can be controlled by the **keyboard, mouse, touch,** or **gamepad** depending on the input device used.

```
//Create a Universal Camera

const camera = new BABYLON.FreeCamera('camera', new BABYLON.Vector3(0, 0, -5), scene)
;
```

The Universal Camera is now the default camera used by Babylon.js if nothing is specified, and it's your best choice if you'd like to have an **FPS-like control** in your scene.

The default actions are:

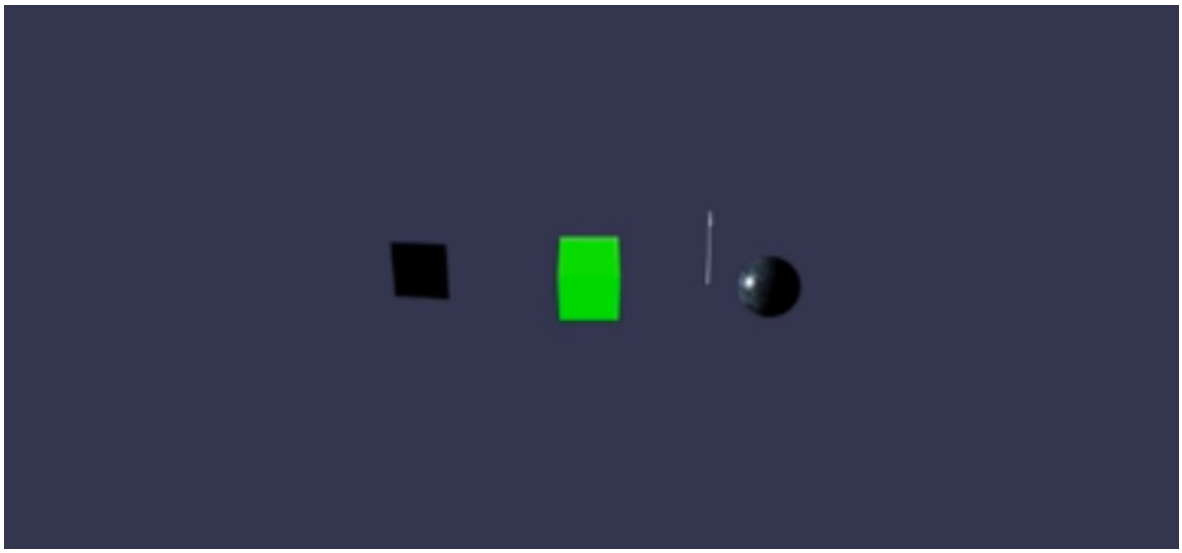
- Keyboard – The left and right arrows move the camera left and right, and up and down arrows move it forwards and backward;
- Mouse – Rotates the camera about the axes with the camera as origin;
- Touch – Swipe left and right to move the camera left and right, and swipe up and down to move it forward and backward;
- Gamepad – corresponds to the device.

Follow Camera

A **Follow camera** allows us to provide a target that we want the camera to follow.

```
//Create a Follow Camera

const camera = new BABYLON.FollowCamera('camera', new BABYLON.Vector3(0, 25, -25), sc
ene);

//Provide the target
camera.lockedTarget = box;
```

If we hit save and refresh the page, we can see that the camera starts zooming in towards the target.



We can also set additional options, such as the distance between the target and the camera.

```
// The goal distance of camera from target
camera.radius = 5;

// The goal height of camera above local origin (centre) of target
camera.heightOffset = 10;

// Acceleration of camera in moving from current to goal position
```

```
camera.cameraAcceleration = 0.005

// The speed at which acceleration is halted
camera.maxCameraSpeed = 10
```


In this lesson, we're going to start creating a **Solar System Project** and explore how we can add more constraints on our camera using **ArcRotateCamera**.

Fixing A Camera On An Object

First, we're going to create a new camera of type **ArcRotateCamera**.

```
function createCamera(scene){
    //Create a new camera that rotates around Vector3.Zero
    const camera = new BABYLON.ArcRotateCamera('camera', 0, 0, 15, BABYLON.Vector3.Ze
ro(), scene);
}
```

Then we're going to allow the user to move our camera by attaching control.

```
function createCamera(scene){
    //Create a new camera
    const camera = new BABYLON.ArcRotateCamera('camera', 0, 0, 15, BABYLON.Vector3.Ze
ro(), scene);

    //Let user move our camera
    camera.attachControl(camera);
}
```

The **ArcRotateCamera** always faces towards a given target position and rotate around it, just like a satellite orbiting the earth. In our case, the target is at the origin of the world space, as we put down Vector3.Zero for the target position.

Testing The Camera

Now that we have our camera updated, we're going to add a sun to our scene.

First, we're going to create a new Sphere:

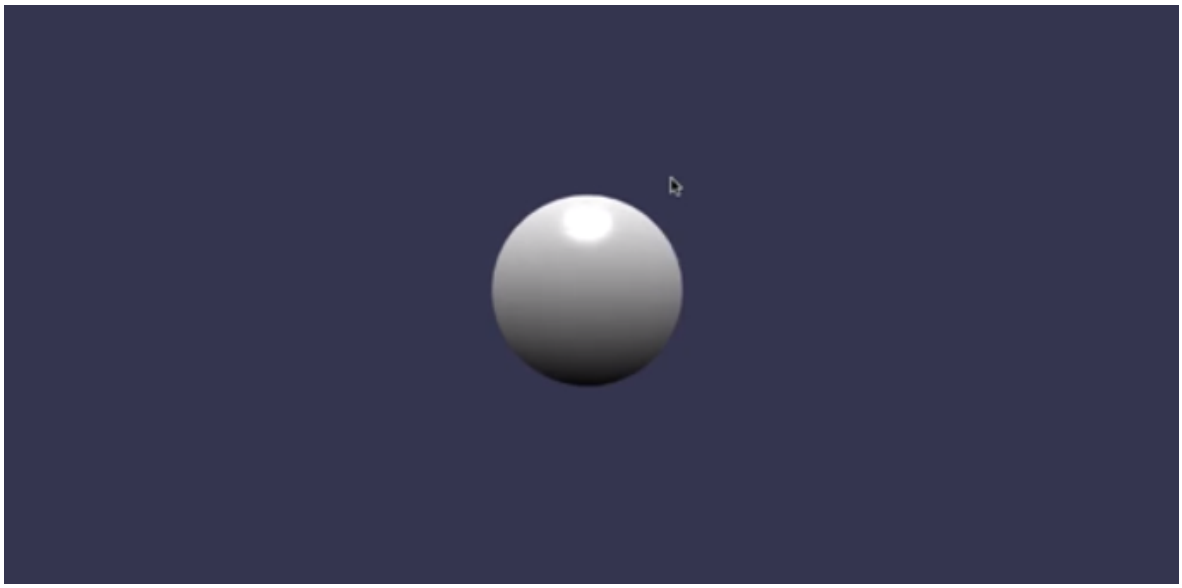
```
function createSun(scene) {
    const sun = BABYLON.MeshBuilder.CreateSphere('sun', { segments: 16, diameter: 4 }
, scene);
}
```

And then we will call the function in our createScene().

```
function createScene() {
    //Create a scene
    const scene = new BABYLON.Scene(engine);

    //Create a camera
    createCamera();

    //Create a Light
```

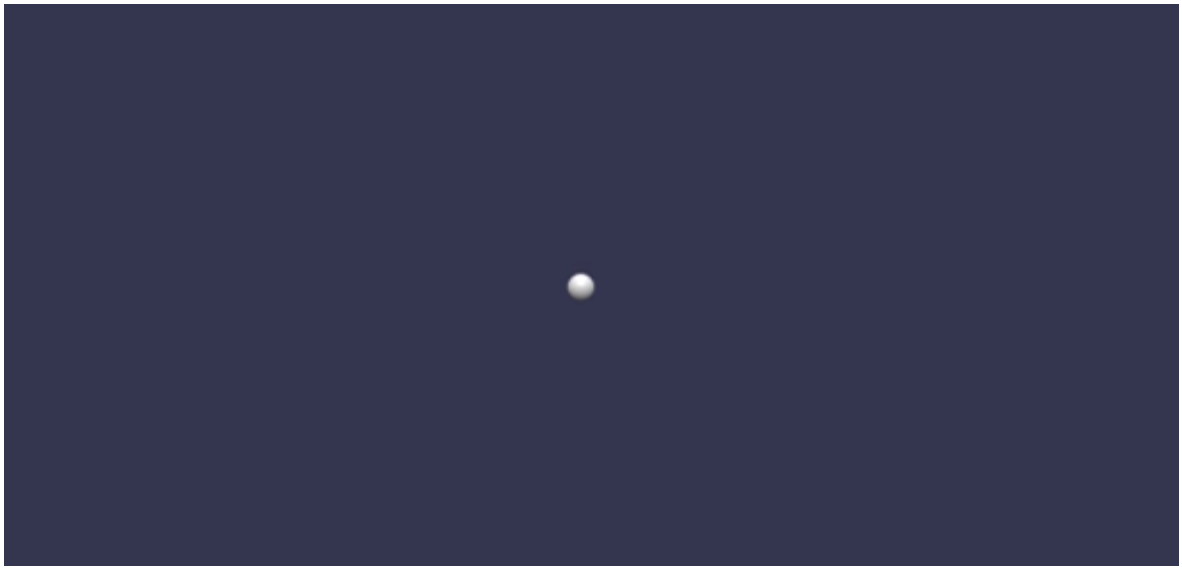
```
    createLight(scene);

    //Create the sun
    createSun(scene);

    return scene;
}
```

Now you'll see that we have our sphere, and our camera only orbits around the sphere.



Adding Constraints to Camera

Right now, we can go ahead and zoom in/out to the point we can't see the object.



To fix this issue, we're going to set a **min distance** (how close we can get to our target) and a **max distance** (how far we can get from our target).

In the ‘createCamera’ function, we’re going to call two functions that set radius limits to our camera: **lowerRadiusLimit** and **upperRadiusLimit**.

```
function createCamera(scene){
    //Create a new camera
    const camera = new BABYLON.ArcRotateCamera('camera', 0, 0, 15, BABYLON.Vector3.Ze
ro(), scene);

    //Let user move our camera
    camera.attachControl(camera);

    //Limit camera movement
    camera.lowerRadiusLimit = 6;
    camera.upperRadiusLimit = 20;
}
```

Summary

- An **ArcRotate** camera allows you to rotate around a point
- You can specify the starting point in terms of **Alpha**, **Beta** and **Radius**.

In this lesson, we're going to **add a planet** to the solar system and set the **scene light properties**.

Creating A Planet

First, we'll create a function that adds a new sphere to the scene.

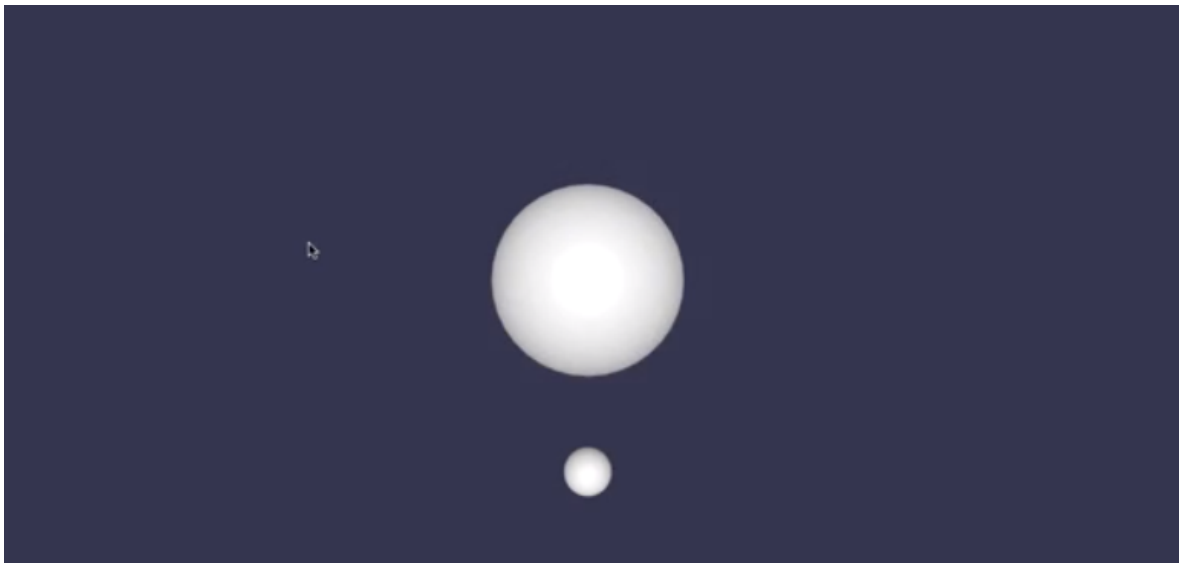
```
function createPlanet(scene) {
    const planet = BABYLON.MeshBuilder.CreateSphere('planet', { segments: 16, diameter: 1 }, scene);
}
```

Then we're going to call the function in **createScene()**.

```
function createScene() {
    //Create a scene
    const scene = new BABYLON.Scene(engine);

    //Create a camera
    createCamera();

    //Create a Light
    createLight(scene);

    //Create the sun
    createSun(scene);

    //Create first planet
    createPlanet(scene);

    return scene;
}
```

Right now, the planet won't be visible as it will be positioned inside the sun. Let's update the position of the planet:

```
function createPlanet(scene) {
    const planet = BABYLON.MeshBuilder.CreateSphere('planet', { segments: 16, diameter: 1 }, scene);
    planet.position.x = 4;
}
```



Applying Texture

Now, we're going to create a new standard material inside the **createPlanet()** function:

```
function createPlanet(scene) {
    //Create a new standard material
    const planetMaterial = new BABYLON.StandardMaterial('planetMaterial', scene);

    //Create a new sphere for the planet
    const planet = BABYLON.MeshBuilder.CreateSphere('planet', { segments: 16, diamete
r: 1 }, scene);

    //Update the position of the planet
    planet.position.x = 4;
}
```

And then we're going to **apply a sand texture** to the material.

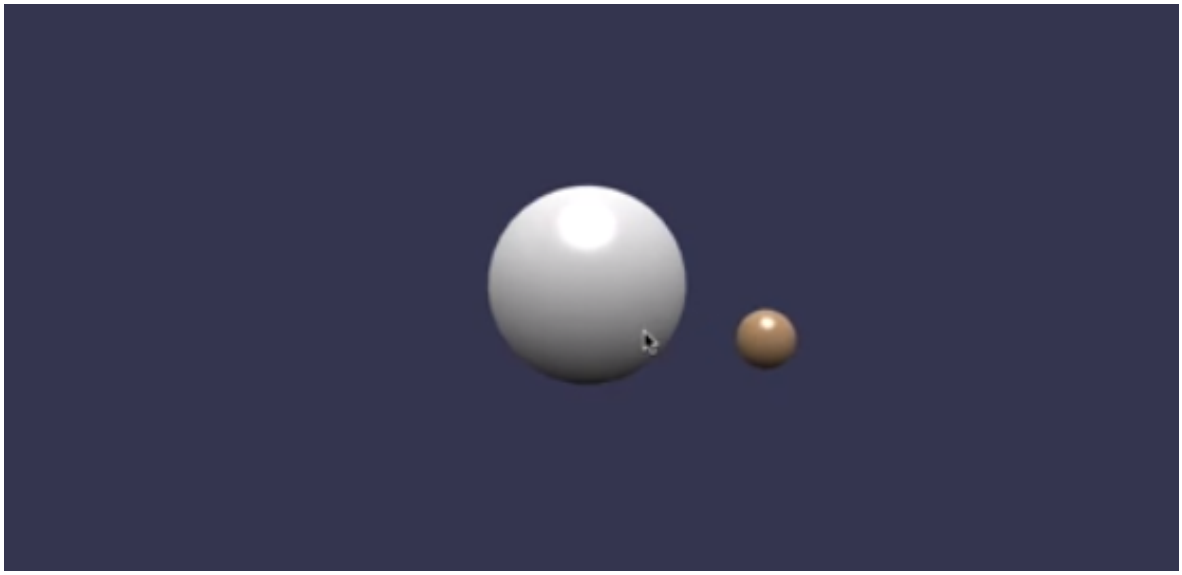
```
function createPlanet(scene) {
    //Create a new standard material
    const planetMaterial = new BABYLON.StandardMaterial('planetMaterial', scene);

    //Apply a sand texture to the material
    planetMaterial.diffuseTexture = new BABYLON.Texture('assets/images/sand.png', sce
ne);

    //Create a new sphere for the planet
    const planet = BABYLON.MeshBuilder.CreateSphere('planet', { segments: 16, diamete
r: 1 }, scene);

    //Update the position of the planet
    planet.position.x = 4;
}
```

Lastly, we're going to **apply the material** to our planet.

```
function createPlanet(scene) {
    //Create a new standard material
    const planetMaterial = new BABYLON.StandardMaterial('planetMaterial', scene);

    //Apply a sand texture to the material
    planetMaterial.diffuseTexture = new BABYLON.Texture('assets/images/sand.png', sce
ne);

    //Create a new sphere for the planet
    const planet = BABYLON.MeshBuilder.CreateSphere('planet', { segments: 16, diamete
r: 1 }, scene);

    //Update the position of the planet
    planet.position.x = 4;

    //Apply the material to the planet
    planet.material = planetMaterial;
}
```

The current material appears to be too shiny, so we're going to update the **specular** color on our material.

(**Specular**: the reflection/highlight given to the material by light.)

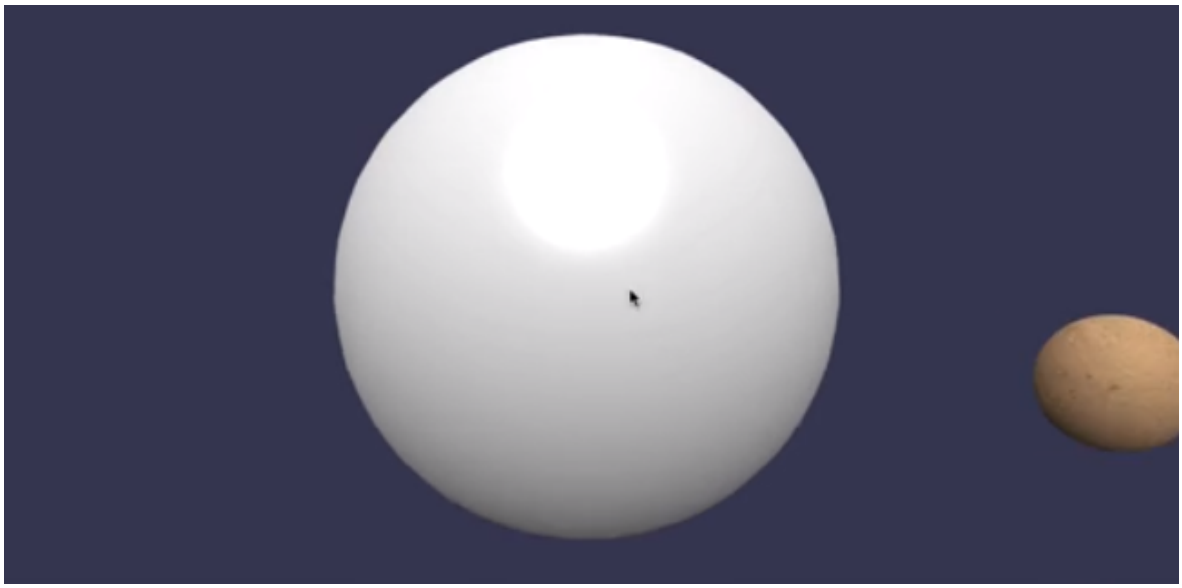


We're going to set the specular color to black as it will remove the highlight.

```
function createPlanet(scene) {
    //Create a new standard material
    const planetMaterial = new BABYLON.StandardMaterial('planetMaterial', scene);
```



```
    //Apply a sand texture to the material
    planetMaterial.diffuseTexture = new BABYLON.Texture('assets/images/sand.png', sce
ne);

    //Remove highlight from the material
    planetMaterial.specularColor = BABYLON.Color3.Black();

    //Create a new sphere for the planet
    const planet = BABYLON.MeshBuilder.CreateSphere('planet', { segments: 16, diamete
r: 1 }, scene);

    //Update the position of the planet
    planet.position.x = 4;

    //Apply the material to the planet
    planet.material = planetMaterial;
}
```



Setting Background Color

To create a more space-like environment, we're going to change the **background** color of the scene to black.

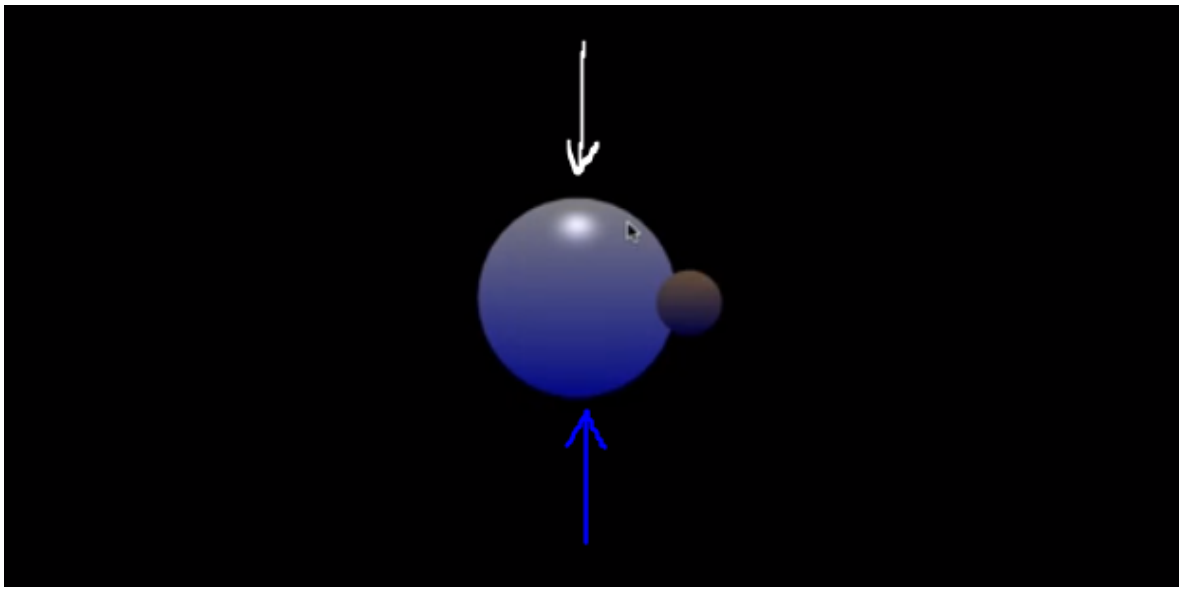
```
function createScene() {
    //Create a scene
    const scene = new BABYLON.Scene(engine);

    //Set the background color to black
    scene.clearColor = BABYLON.Color3.Black();
...
```

We will also halve the **intensity** of our light to make the scene darker.

```
function createLight(scene){
    const light = new BABYLON.HemisphericLight('light', new BABYLON.Vector33(0, 1, 0)
, scene);

    //Halve the light intensity
    light.intensity = 0.5;
}
```

And lastly, we're going to give the light a **ground color** so that there is some blue color hitting from the opposite direction.

```
function createLight(scene){
    const light = new BABYLON.HemisphericLight('light', new BABYLON.Vector33(0, 1, 0)
, scene);
    light.intensity = 0.5;

    //Set the ground color to blue
    light.groundColor = new BABYLON.Color3(0, 0, 1);
}
```



In this lesson, we're going to create a **point light** to simulate start shine and set up its light properties.

Creating A Shining Star

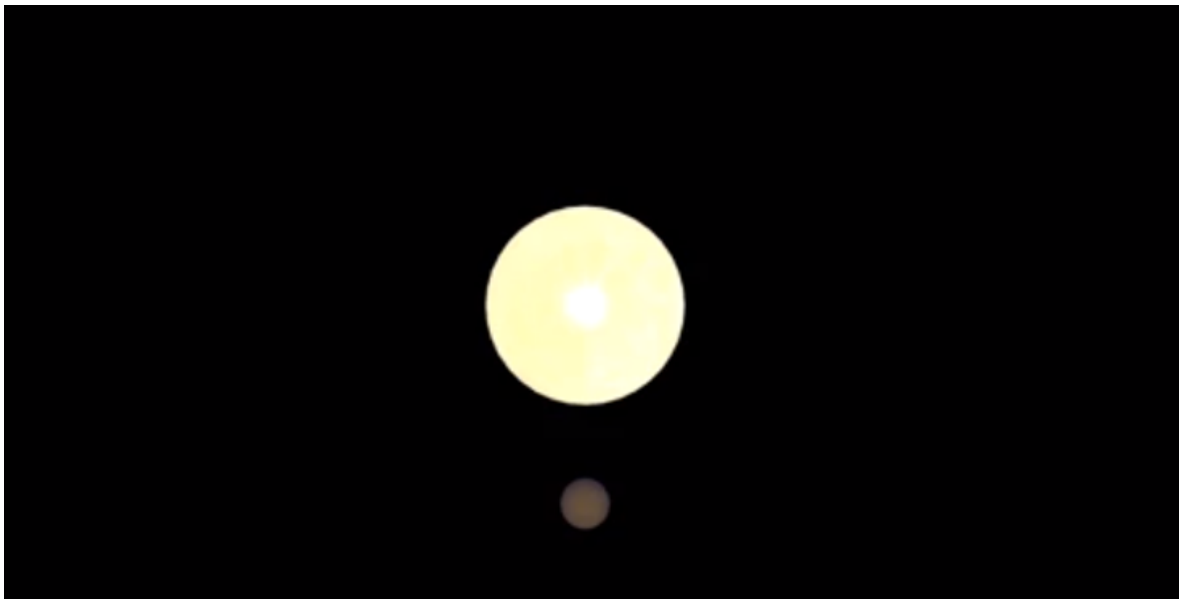
First of all, we're going to create a **material** for our sun inside the **createSun()** function.

```
function createSun(scene){
    //Create a material
    const sunMaterial = new BABYLON.StandardMaterial('sunMaterial', scene);

    //Create a Sphere
    const sun = BABYLON.MeshBuilder.CreateSphere('sun', { segments: 16, diameter: 4 }
);
}
```

Then we're going to give our material an **emissive texture** so it emits light, and then apply the material to our sun.

```
function createSun(scene){
    //Create a material
    const sunMaterial = new BABYLON.StandardMaterial('sunMaterial', scene);

    //Assign an emissive texture to the material
    sunMaterial.emmisiveTexture = new BABYLON.Texture('assets/images/sun.jpg', scene)
;

    //Create a Sphere
    const sun = BABYLON.MeshBuilder.CreateSphere('sun', { segments: 16, diameter: 4 }
);

    //Assign the material
    sun.material = sunMaterial;
}
```

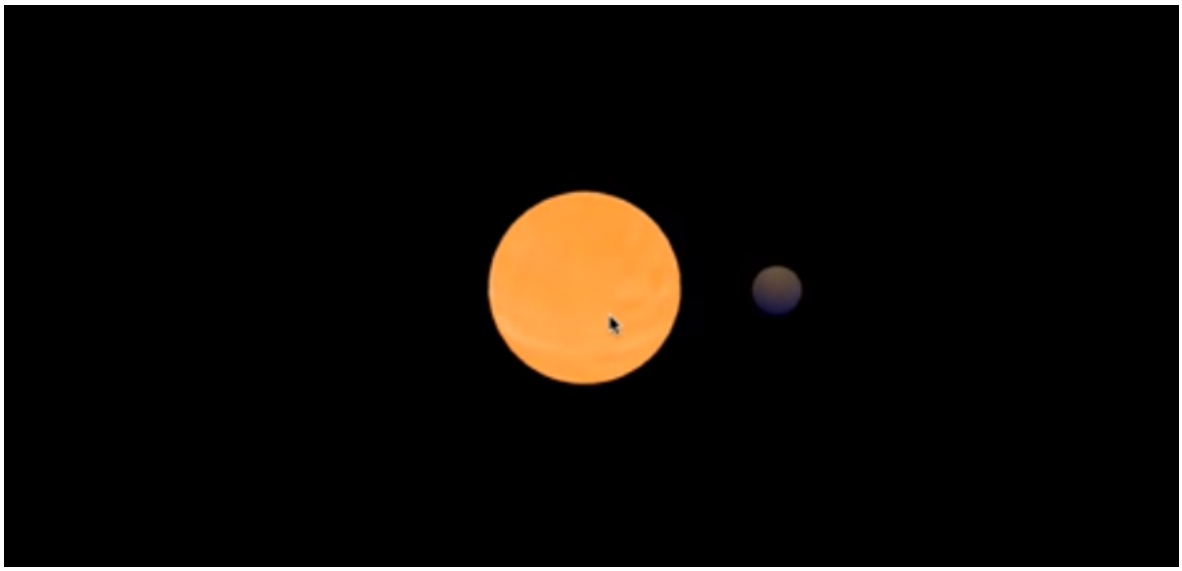
Right now our sun is glowing, but the light is not hitting on another planet.



To create a more realistic light effect, we're going to **reduce the glare effect** on our sun, and then add a new light object.

Reducing the glare effect is as simple as assigning **black** color to the **diffuse** color and **specular** color.

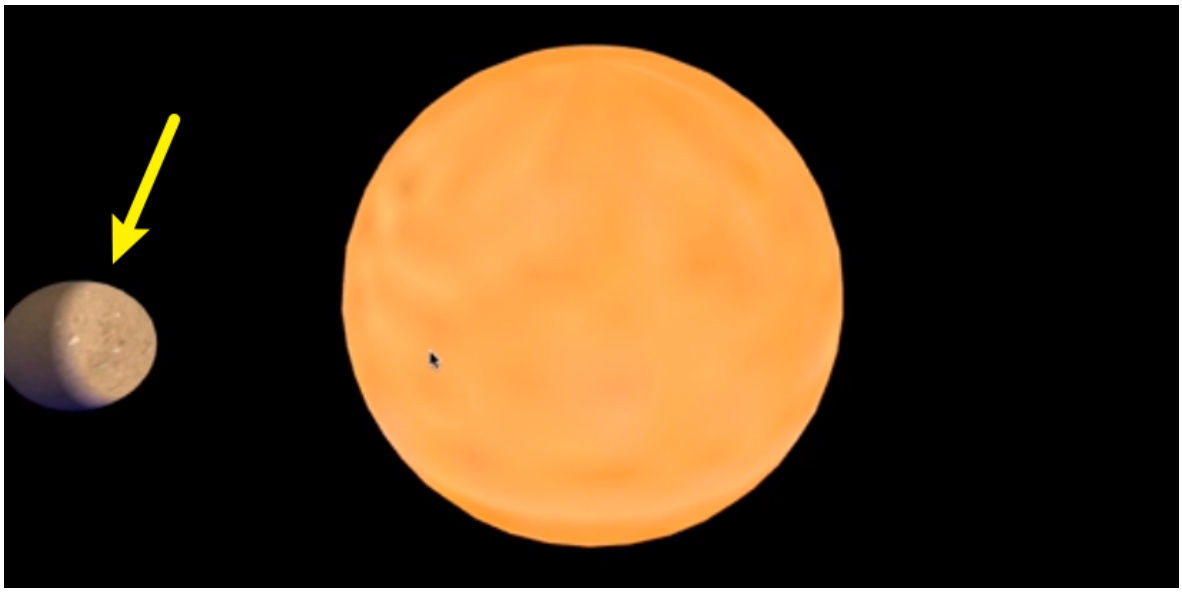
```
function createSun(scene){
    //Create a material
    const sunMaterial = new BABYLON.StandardMaterial('sunMaterial', scene);

    //Assign an emissive texture to the material
    sunMaterial.emmisiveTexture = new BABYLON.Texture('assets/images/sun.png', scene)
;

    //Reducing light glare
    sunMaterial.diffuseColor = BABYLON.Color3.Black();
    sunMaterial.specularColor = BABYLON.Color3.Black();

    //Create a Sphere
    const sun = BABYLON.MeshBuilder.CreateSphere('sun', { segments: 16, diameter: 4 }
);

    //Assign the material
    sun.material = sunMaterial;
}
```



Now we're going to create a new light source for our sun, by adding a **point light**.

```
function createSun(scene){
    //Create a material
    const sunMaterial = new BABYLON.StandardMaterial('sunMaterial', scene);

    //Assign an emissive texture to the material
    sunMaterial.emmisiveTexture = new BABYLON.Texture('assets/images/sun.png', scene)
;

    //Removing Glare from the sun
    sunMaterial.diffuseColor = BABYLON.Color3.Black();
    sunMaterial.specularColor = BABYLON.Color3.Black();

    //Create a Sphere
    const sun = BABYLON.MeshBuilder.CreateSphere('sun', { segments: 16, diameter: 4 }
);

    //Assign the material
    sun.material = sunMaterial;

    //Creating a point light
    const sunLight = new BABYLON.PointLight('sunLight', BABYLON.Vector3.Zero(), scene
);
    sunLight.intensity = 2;
```

Our sun is now emitting light that is hitting another planet.



Summary

- Use an emissive texture on the sun
- Create a point light in the position of the start to simulate star shine
- Regulate the light intensity with the intensity property

In this lesson, we're going to create a **skybox** to simulate star background.

Creating A Skybox

A **Skybox** is a large standard box that we can use to simulate the sky. It has special reflective textures and a group of six images, one for each face of the cube.

We're going to start off by creating a box inside a new function called **createSkybox**:

```
function createSkybox(scene) {

    //Creating a skybox
    const skybox = BABYLON.MeshBuilder.CreateBox('skybox', {size: 1000}, scene);

}
```

Then, we're going to create a new standard material and assign it to the box.

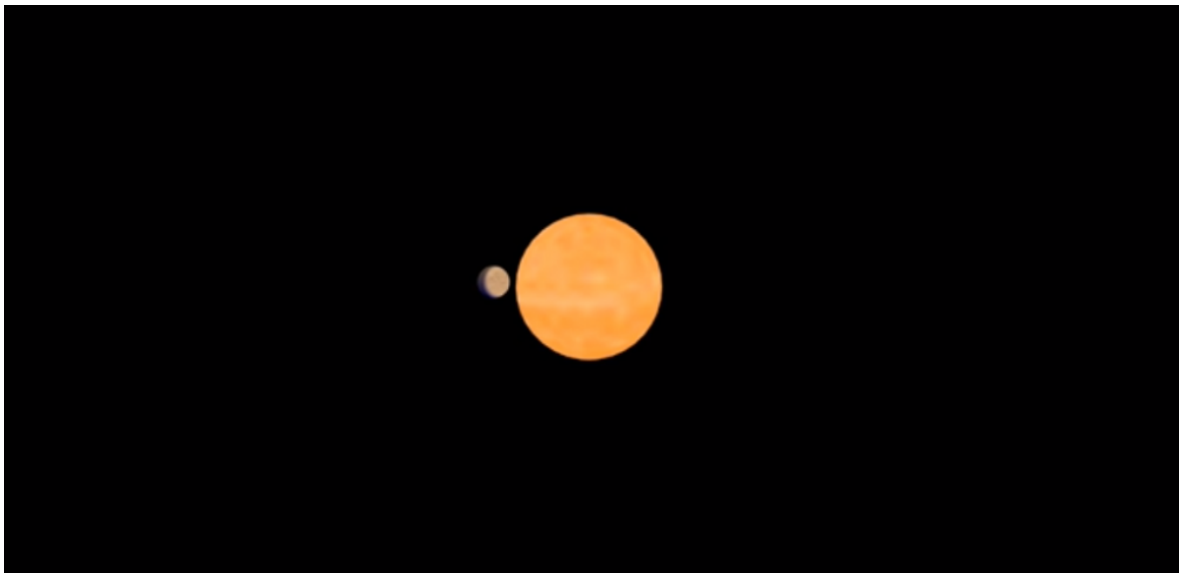
```
function createSkybox(scene) {

    //Creating a skybox
    const skybox = BABYLON.MeshBuilder.CreateBox('skybox', {size: 1000}, scene);

    //Creating a material
    const skyboxMaterial = new BABYLON.StandardMaterial('skyboxMaterial', scene);

    //Applying the material
    skybox.material = skyboxMaterial;

}
```

Lastly, we're going to call the createSky function inside the **createScene()**.

```
function createScene() {

...
//Create A Skybox
createSkybox(scene);
}
```

We can't see the box right now, because the **backface culling*** property of our box is set to true.

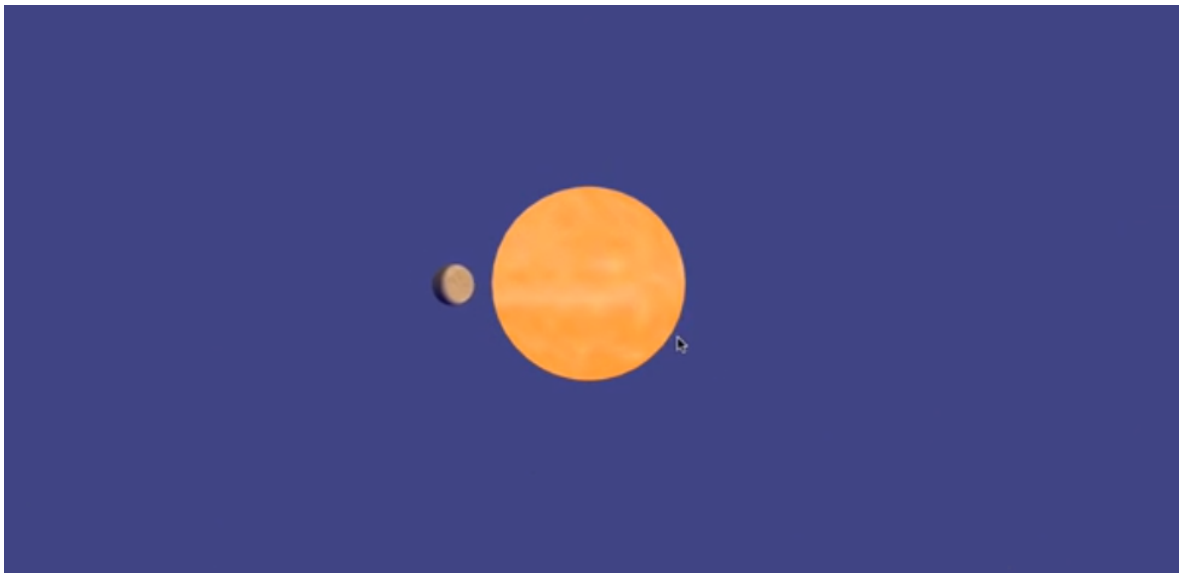
***Backface culling**: The back face of an object is usually hidden by default in order to minimize unnecessary rendering.



We need to manually set this property to '**false**'.

```
function createSkybox(scene) {

    //Creating a skybox
    const skybox = BABYLON.MeshBuilder.CreateBox('skybox', {size: 1000}, scene);

    //Creating a material
    const skyboxMaterial = new BABYLON.StandardMaterial('skyboxMaterial', scene);

    //Disabling Backface Culling
    skyboxMaterial.backFaceCulling = false;

    //Applying the material
    skybox.material = skyboxMaterial;

}
```

Now we can actually see the skybox being rendered to our scene.



Since we don't want our camera to be able to move out of the skybox, we're going to set its **infiniteDistance** property to true. This makes the skybox follow our camera's position.

```
//Move the skybox with camera
skybox.infiniteDistance = true;
```

We're also going to update our **specular** and **diffuse** colors to be black.

```
//Remove reflection in skybox
skyboxMaterial.specularColor = BABYLON.Color3.Black();
skyboxMaterial.diffuseColor = BABYLON.Color3.Black();
```

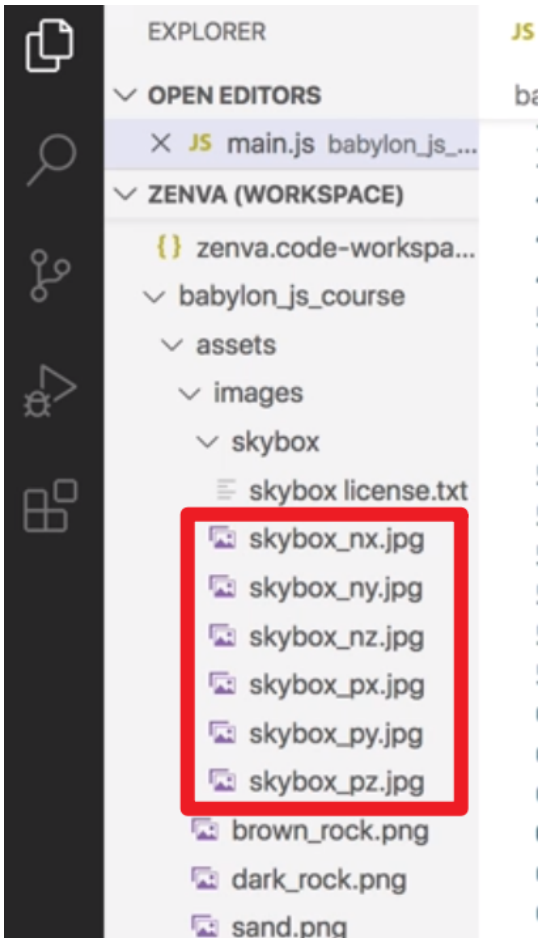
Applying Texture To Skybox

Now we're going to load our sky image and texture it to the different sides of our box. One way of applying different textures to each side of a cube is **CubeTexture**.

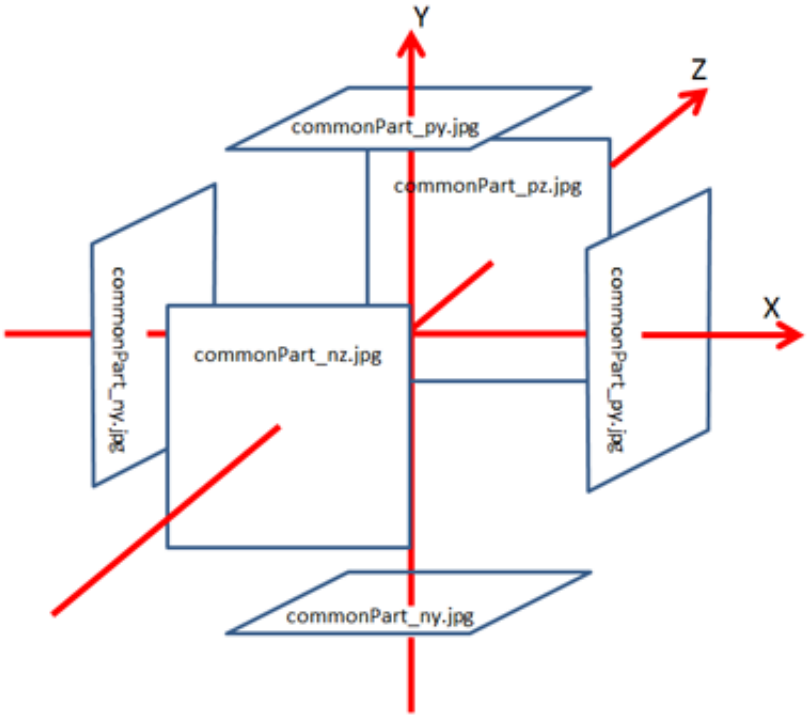
```
//Create a reflection texture
skyboxMaterial.reflectionTexture = new BABYLON.CubeTexture('assets/images/skybox/skyb
ox', scene);
skyboxMaterial.reflectionTexture.coordinatesMode = BABYLON.Texture.SKYBOX_MODE;
```

The **CubeTexture** allows us to specify different materials for each of the sides and apply them correctly.

Note that instead of providing one particular asset, we provided a path to a **set** of assets (/assets/images/skybox/).



Inside the skybox folder, we have six different images with different endings (**_nx, _ny, _nz** ...), referencing which side of the cube each image should be applied to.



Refer to the documentation for more information: https://doc.babylonjs.com/how_to/reflect

Summary

- A skybox is a giant cube that has an image for each inner face
- They are used to simulate 3D backgrounds
- Create a box and give it a reflection texture with coordinates mode skybox

In this lesson, we're going to add more planets to our scene and make them orbit around the sun.

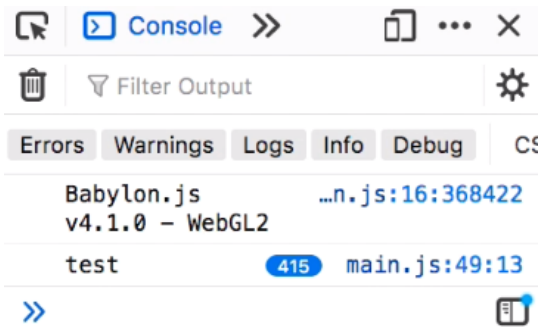
Adding Animations

First, we're going to add animation to our existing planet by updating its position **before** our scene finishes rendering. To do that, we're going to use:

```
scene.registerBeforeRender(() => {});
```

This allows us to invoke our function before render, that is every time our canvas gets updated. If we put in **console.log** here:

```
scene.registerBeforeRender(() => {
  console.log('test');
});
```



We can see that the 'test' is being called multiple times per second (once per frame).

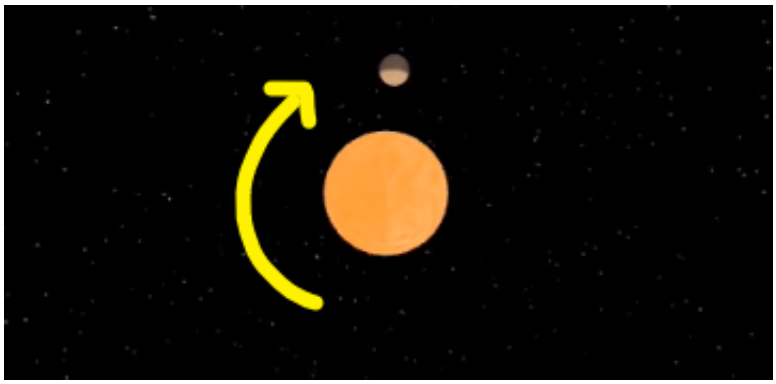
We're going to create a new custom property called **planet.orbit** so we can reference the values such as **radius**, **speed**, **angle** of the planet.

```
planet.orbit = {
  radius: planet.position.x,
  speed: 0.01,
  angle: 0,
};
```

Now we're going to delete the console.log and **update the position** of our planet inside the registerBeforeRender() function:

```
scene.registerBeforeRender(() => {
  planet.position.x = planet.orbit.radius * Math.sin(planet.orbit.angle);
  planet.position.z = planet.orbit.radius * Math.cos(planet.orbit.angle);
  planet.orbit.angle += planet.orbit.speed;
});
```


If we **save** and **refresh**, we should see that our planet starts orbiting around the sun.



Adding More Planets

Now we're going to add a few more planets to our scene by calling certain parts of the **createPlanet** function multiple times in a **for loop**:

```
for (let i=0; i < 3; i++) {

}
```

However, make sure that everything related to the material is placed outside the **for** loop, as we're going to apply the same material for all the planets.

```
function createPlanet(scene) {

    //Create a new s<code>andard material
    const planetMaterial = new BABYLN.StandardMaterial('planetMaterial', scene);

    //Apply a sand texture to the material
    planetMaterial.diffuseTexture = new BABYLON.Texture('assets/images/sand.png', sce
ne);

    //Remove highlight from the material
    planetMaterial.specularColor = BABYLON.Color3.Black();


  for (let i=0; i < 3; i++) {

    //Create a new sphere for the planet
    const planet = BABYLON.MeshBuilder.CreateSphere(`planet${i}`, { segments: 16, dia
meter: 1 }, scene);

    //Update the position of the planet
    planet.position.x = (2 * i) + 4;

    //Apply the material to the planet
    planet.material = planetMaterial;

    //Create a custom property for reference
    planet.orbit = {
```

```
      radius: planet.position.x,
      speed: 0.01,
      angle: 0,
    };

    //Update the position every frame
    scene.registerBeforeRender(() => {
      planet.position.x = planet.orbit.radius * Math.sin(planet.orbit.angle);
      planet.position.z = planet.orbit.radius * Math.cos(planet.orbit.angle);
      planet.orbit.angle += planet.orbit.speed;
    });

  }

}
```

Note that we're using an **{i}** variable inside the for loop to give each planet a unique name and position.

```
//Create a new sphere for the planet
const planet = BABYLON.MeshBuilder.CreateSphere(`planet${i}`, { segments: 16, diamete
r: 1 }, scene);

//Update the position of the planet
planet.position.x = (2 * i) + 4;
```

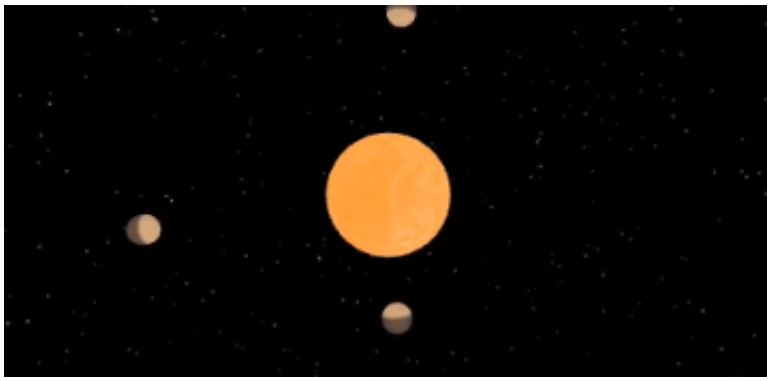
Lastly, we're going to create an **array** in order to make them rotate at different speeds:

```
const speeds = [0.01, -0.01, 0.02];
```

And change the speed property to **speeds[i]** within the orbit definition so it can select the appropriate speed value:

```
planet.orbit = {
  radius: planet.position.x,
  speed: speeds[i],
  angle: 0,
};
```

Now we can see that our planets are rotating at different speeds.



Summary

- Create animations by editing the shapes properties in beforeUpdate
- Use basic physics to move elements in a particular way

In addition to using Babylon MeshBuilder to create 3D objects, we can also use plugins that load in external 3D object files into our scene.

These plugins allow us to load any 3D file types, including .obj, STL, glTF, and more.

Loading A Library File

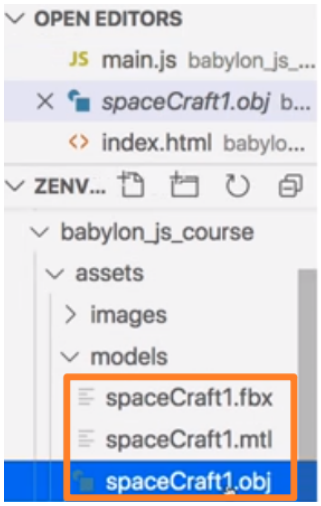
By default, the Babylon Main Library file doesn't include loader plugins to keep the file size down.

Let's open up our **index.html** file, and add a new script tag below our Babylon import:

```
<script src="js/vendor/babylonjs.loaders.min.js"
```

This plugin allows us to **import a file** into a currently active scene.

Now we can import sample 3D object files from **assets/models/** into our scene.



We're going to create a new function that uses the **ImportMesh** feature of **SceneLoader**:

```
function createShip(scene) {
  BABYLON.SceneLoader.ImportMesh()
}
```

It takes certain arguments in the following order:

- **Mesh name**(s). We're going to provide an empty string (' ') for now.
- **Root URL**, where the file is contained at. ('/assets/models/')
- **Scene File Name**, the name of the file that we want to load ('spaceCraft1.obj')
- **Scene reference** ('scene')
- **Success Callback**, the function that will be invoked once the asset is loaded (meshes)

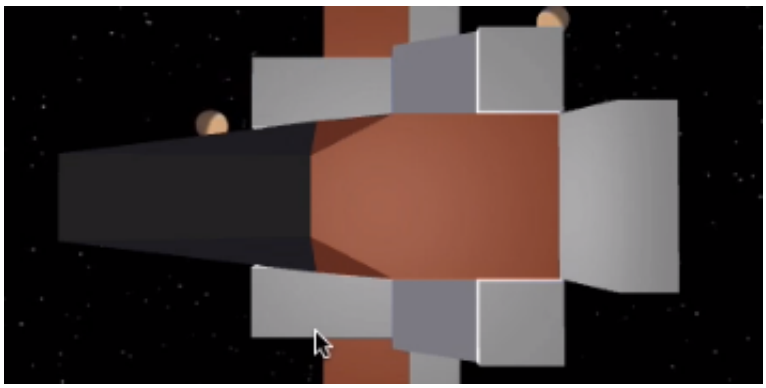
```
function createShip(scene) {
  BABYLON.SceneLoader.ImportMesh('', '/assets/models/', 'spaceCraft1.obj', scene, (me
shes) => {
    console.log(meshes);
```

```
  })
}
```

Then inside the **createScene()** function, we're going to call the **createShip()**.

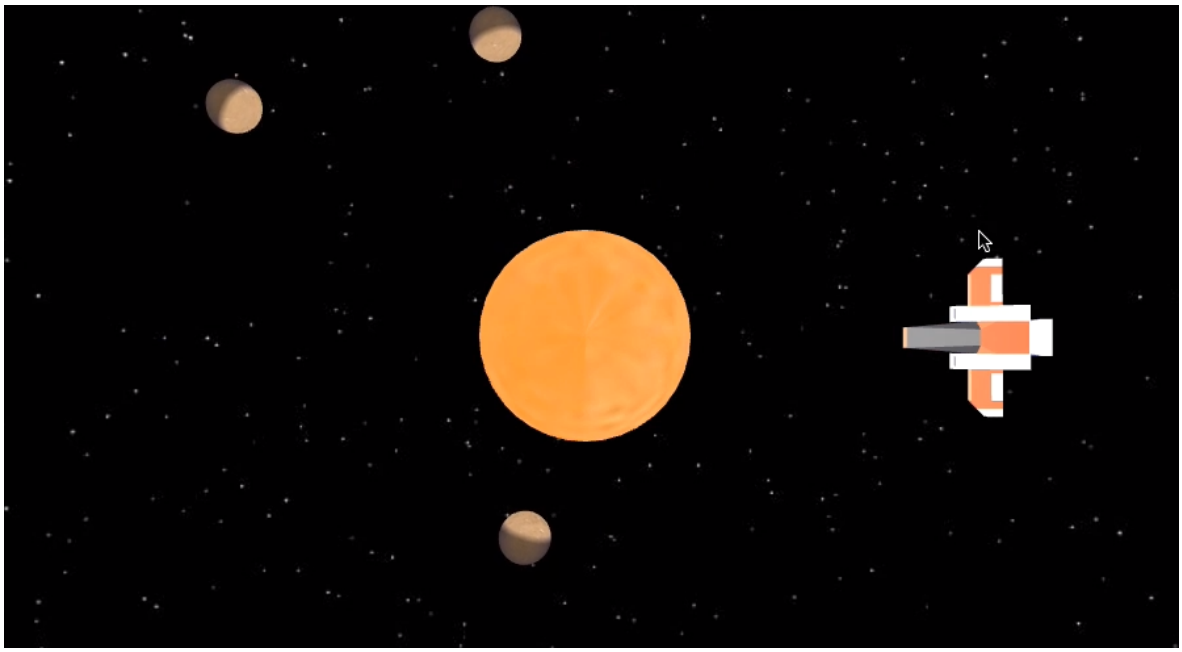
```
function createScene() {

...
createShip(scene);
}
```

We'll see that our spaceship loads into our scene.



Lastly, we're going to set the **scaling** and the **position** of the mesh:

```
function createShip(scene) {
  BABYLON.SceneLoader.ImportMesh('', '/assets/models/', 'spaceCraft1.obj', scene, (me
shes) => {
    console.log(meshes);
    meshes.forEach((mesh) => {
      mesh.position = new BABYLON.Vector3(0, -5, 10);
      mesh.scaling = new BABYLON.Vector3(0.2, 0.2, 0.2);
    });
  })
}
```



Summary

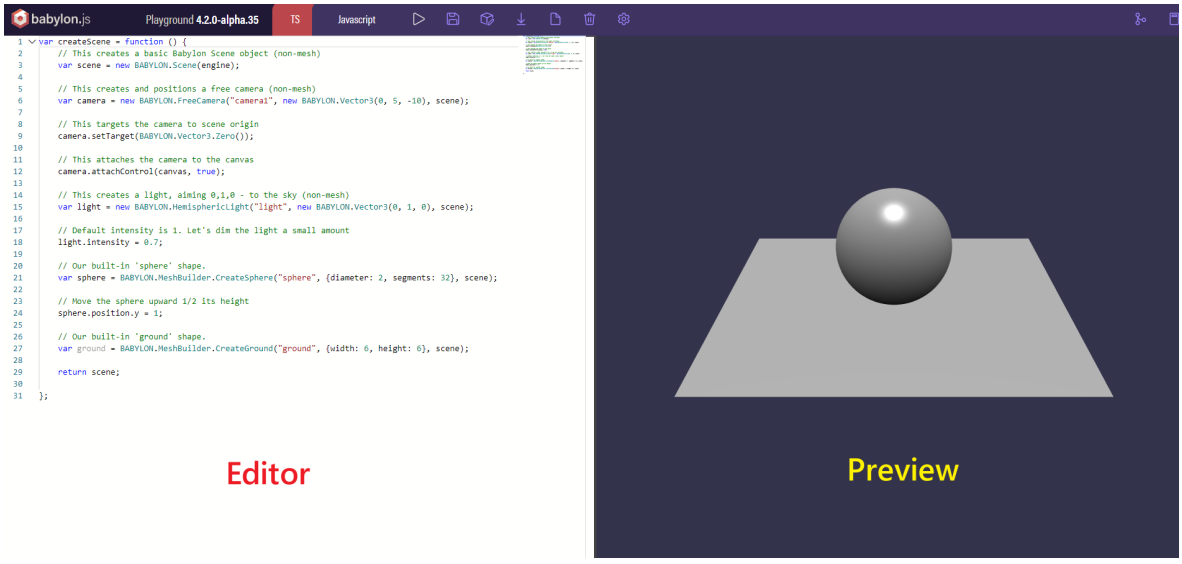
- Using **BABYLON.SceneLoader.ImportMesh** to load external files.
- After the file is loaded, you can modify the properties on the new objects as normal
- For more information, refer to Documentation: https://doc.babylonjs.com/api/classes/babylon.sceneloader#importmesh

In this lesson, we're going to learn about **Babylon.js Playground**, by viewing, modifying, and sharing examples in the Playground.

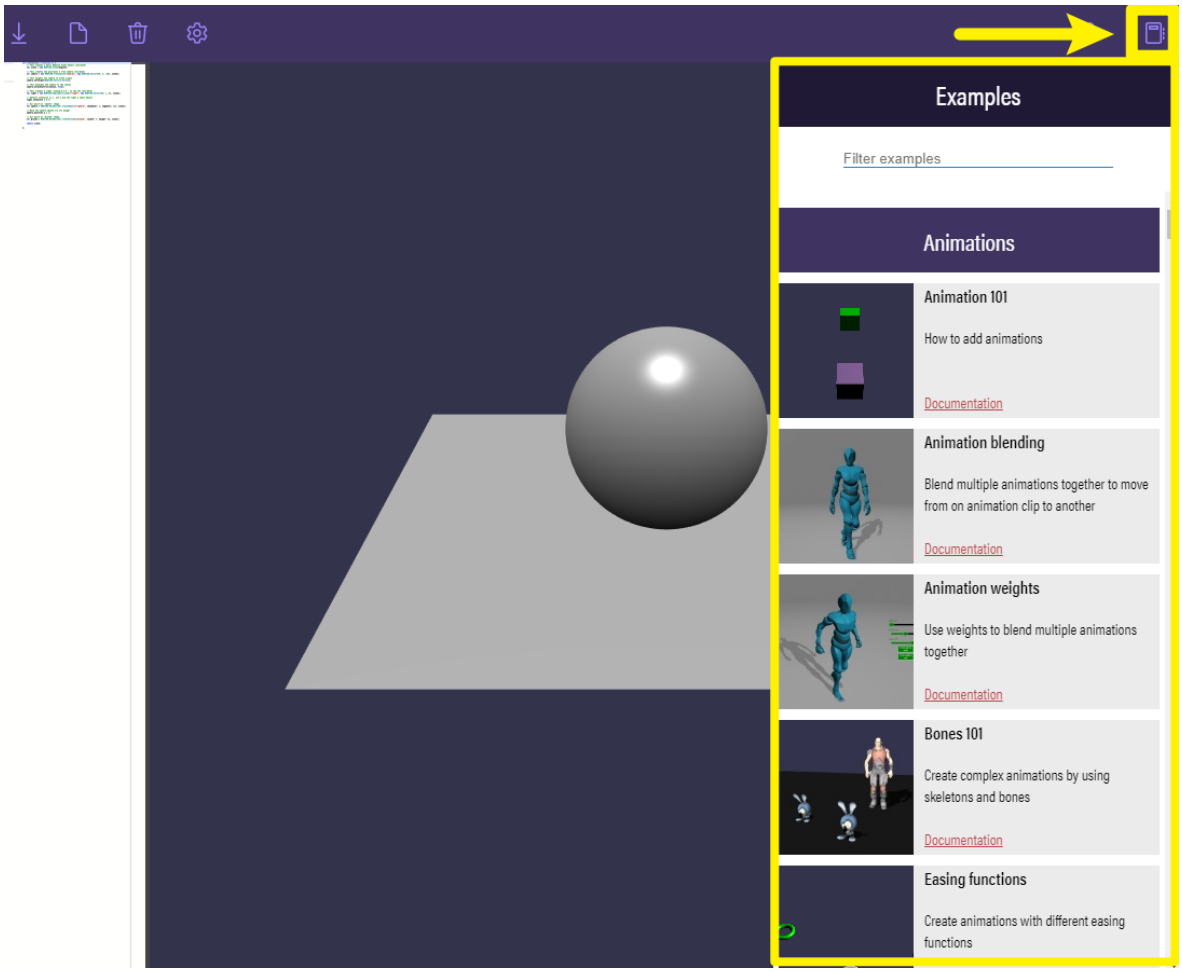
To access the playground, go to the official documentation site:

or click on the URL: https://playground.babylonjs.com/

You can access the **editor** on the left, and all of your updates will be applied to the **preview** on the right.

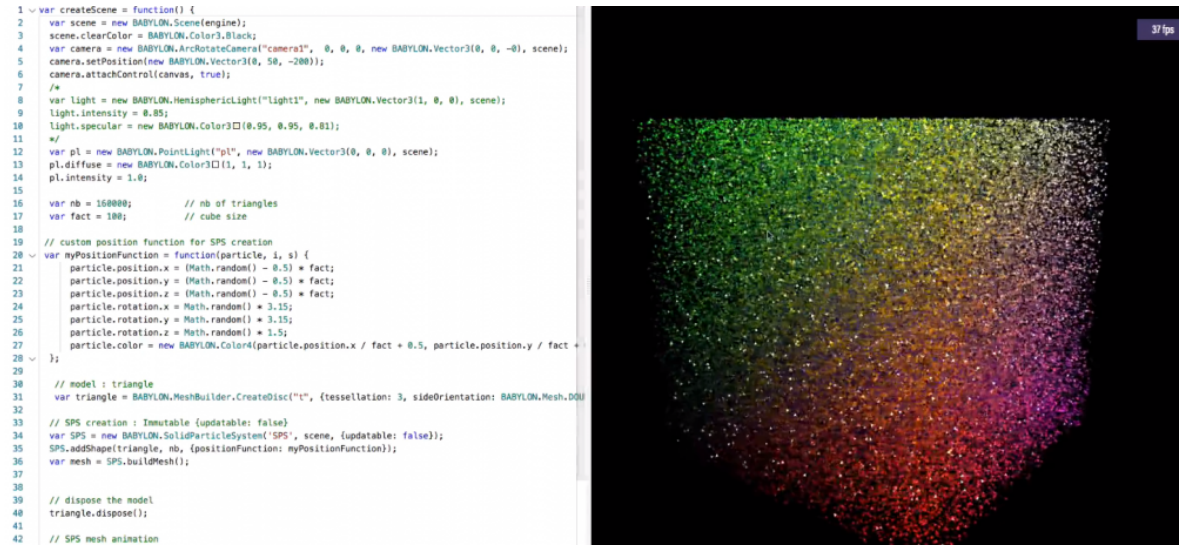


You can open up an **Example** panel by clicking on the **note** icon.

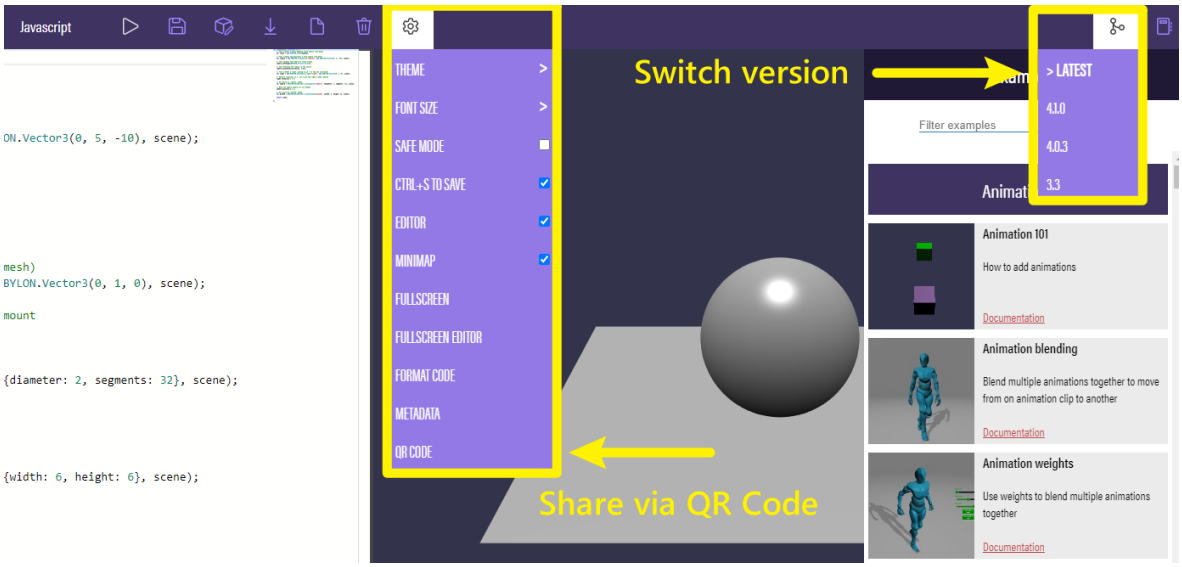


The examples are grouped by various tags including: **Animations, Skeletons, Audio, Cameras, GUI,** etc.

If you go ahead and click on any example, it will be loaded in the Playground and we can start playing around the code.



You can also share the playground project with other people by going to **Settings > QR Code**.



Summary

- The Babylon.js Playground is a live editor for Babylon.js WebGL 3D scenes.
- You can view, edit, and share examples in the Playground.