

# **Introduction à la 3D en JavaScript : three.js**

# Table des matières

<b>I. Contexte</b>	<b>3</b>
<b>II. Le WebGL et three.js</b>	<b>3</b>
<b>III. Exercice : Appliquez la notion</b>	<b>7</b>
<b>IV. Scène, caméra et renderer</b>	<b>8</b>
<b>V. Exercice : Appliquez la notion</b>	<b>12</b>
<b>VI. Ajouter des éléments</b>	<b>12</b>
<b>VII. Exercice : Appliquez la notion</b>	<b>18</b>
<b>VIII. Animer des objets</b>	<b>19</b>
<b>IX. Exercice : Appliquez la notion</b>	<b>23</b>
<b>X. Essentiel</b>	<b>23</b>
<b>XI. Auto-évaluation</b>	<b>24</b>
A. Exercice final .....	24
B. Exercice : Défi .....	25
<b>Solutions des exercices</b>	<b>28</b>

## I. Contexte

**Durée** : 1 h

**Environnement de travail** : Local

**Pré-requis** : Connaître les bases du JavaScript, savoir gérer ses dépendances avec npm

### Contexte

Au fil des années, JavaScript est devenu un langage incontournable du Web. D'abord utilisé pour animer des pages web, il est aujourd'hui la pierre angulaire qui permet à la fois de gérer la partie frontale d'un site Internet et son serveur HTTP.

Mais avec l'apparition du HTML5, le JavaScript a étendu une nouvelle fois son domaine d'utilisation : il n'est plus limité aux sites, mais permet également de réaliser des graphismes en 3 dimensions, directement dans le navigateur. Interfaces futuristes, animations, jeux vidéo ou environnements de réalité virtuelle, les possibilités de cette technologie sont nombreuses !

Dans ce cours, nous allons voir comment tout ça est possible, puis nous apprendrons à utiliser la librairie three.js, qui permet de manipuler des objets 3D très simplement.

## II. Le WebGL et three.js

### Objectifs

- Découvrir le WebGL
- Comprendre l'utilité de three.js

### Mise en situation

Le HTML5 a été une révolution qui a changé de manière flagrante la manière de créer des sites Internet. Nouveaux blocs, possibilité d'ajouter simplement des sons ou des vidéos, nouveaux éléments de formulaire... Ses apports sont nombreux et leurs bienfaits sont indéniables.

Mais parmi tous ces ajouts, un en particulier permet d'étendre les capacités du navigateur pour ne plus se limiter à l'affichage de sites web : le canevas.

### Définition

#### Canvas et WebGL

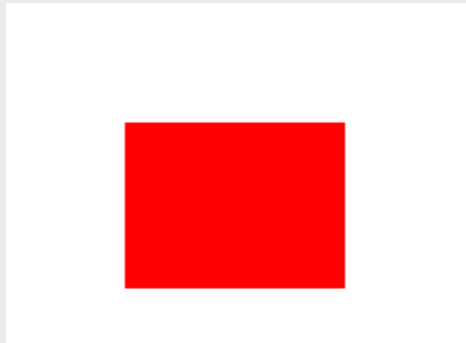
L'élément `<canvas>` est une balise apparue en HTML5 permettant de réaliser des dessins en JavaScript. Le canevas peut utiliser plusieurs technologies pour créer et afficher le dessin, appelées contextes de rendu (ou *rendering contexts*). On peut voir un contexte de rendu comme une boîte à outils contenant tout ce dont JavaScript a besoin pour interagir avec le canevas.

Il existe deux contextes principaux qui sont implémentés nativement dans les navigateurs récents : **Canvas2D**, qui permet de réaliser des dessins en 2D, et **WebGL**, qui permet de faire de la 3D.

### Exemple

La forme la plus primitive qu'il est possible de faire avec Canvas2D est un rectangle. Sans rentrer dans les détails, voici le code nécessaire à la création d'un rectangle rouge :

```
1 <canvas id="canvas" width="300" height="300"></canvas>
2 <script>
3   let canvas = document.getElementById('canvas');
4   let ctx = canvas.getContext('2d'); // Récupération du contexte Canvas2D
5   ctx.fillStyle = "#FF0000"; // Couleur de remplissage (rouge)
6   ctx.fillRect(50, 50, 100, 75); // Dessin d'un rectangle de 100x75 aux coordonnées (50,50)
7 </script>
```



En trois dimensions, la forme de prédilection est un triangle. Voici le code nécessaire à la création d'un triangle rouge :

```
1 <canvas id="canvas" width="300" height="300"></canvas>
2 <script>
3   let canvas = document.getElementById('canvas');
4   gl = canvas.getContext('webgl'); // Récupération du contexte Canvas2D
5
6   let vertices = [
7     -0.5, 0.5, 0.0,
8     -0.5, -0.5, 0.0,
9     0.5, -0.5, 0.0
10  ];
11  let indices = [0, 1, 2];
12
13  // Création de deux buffers de données : un pour les sommets et un pour les indices
14  let vertexBuffer = gl.createBuffer();
15  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
16  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
17  gl.bindBuffer(gl.ARRAY_BUFFER, null);
18  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
19
20  let indexBuffer = gl.createBuffer();
21  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
22  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);
23  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
24  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
25
26  // Création du shader des sommets et des fragments, puis du shader général
27  let vertCode =
28    'attribute vec3 coordinates;' +
29    'void main(void) {' +
30    '  gl_Position = vec4(coordinates, 1.0);' +
31    '}'
```

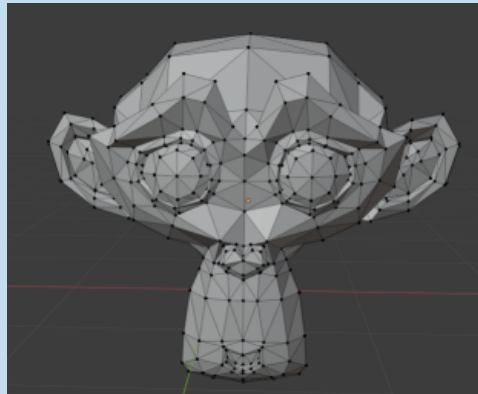
```
32
33 let vertShader = gl.createShader(gl.VERTEX_SHADER);
34 gl.shaderSource(vertShader, vertCode);
35 gl.compileShader(vertShader);
36
37 let fragCode =
38     'void main(void) {' +
39     ' gl_FragColor = vec4(1, 0.0, 0.0, 1);' +
40     ' }';
41 let fragShader = gl.createShader(gl.FRAGMENT_SHADER);
42 gl.shaderSource(fragShader, fragCode);
43 gl.compileShader(fragShader);
44
45 let shaderProgram = gl.createProgram();
46 gl.attachShader(shaderProgram, vertShader);
47 gl.attachShader(shaderProgram, fragShader);
48 gl.linkProgram(shaderProgram);
49 gl.useProgram(shaderProgram);
50
51 let coord = gl.getAttribLocation(shaderProgram, "coordinates");
52 gl.vertexAttribPointer(coord, 3, gl.FLOAT, false, 0, 0);
53 gl.enableVertexAttribArray(coord);
54
55 // Dessin sur le canvas
56 gl.clearColor(1, 1, 1, 1);
57 gl.enable(gl.DEPTH_TEST);
58 gl.clear(gl.COLOR_BUFFER_BIT);
59 gl.viewport(0, 0, canvas.width, canvas.height);
60 gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT, 0);
61 </script>
```



Les exemples ci-dessus permettent de mettre en évidence une différence flagrante entre les deux contextes : WebGL est bien plus complexe à manipuler que Canvas2D. L'ajout d'une troisième dimension nécessite de gérer des éléments qui ne sont pas forcément présents en 2D, comme les *shaders*, les *buffers*, les sommets...

### Remarque

Cela peut sembler étrange que la forme de base d'un objet 3D soit un triangle, mais, en réalité, un modèle 3D est souvent composé d'un assemblage de triangles (un rectangle peut être composé de deux triangles).



Heureusement, il existe des bibliothèques JavaScript permettant de simplifier l'utilisation de WebGL. Nous allons voir la plus connue et la plus utilisée : **three.js**.

### Définition Three.js

Three.js est une bibliothèque JavaScript qui permet de manipuler simplement des objets en 3D. Grâce à elle, il est possible de dessiner des formes primitives tridimensionnelles ou d'importer des modèles 3D en quelques lignes de code, et de les afficher en utilisant de nombreuses technologies de rendu, dont WebGL.

Elle dispose également d'outils permettant d'animer ces objets ou de gérer les collisions entre eux. Il est même possible de créer des environnements accessibles depuis un casque de réalité virtuelle.

### Méthode

Pour utiliser la bibliothèque three.js, il va d'abord falloir l'importer dans le projet. Pour cela, il existe deux solutions : l'utilisation d'un CDN et le téléchargement via npm.

- Un **CDN** (pour *Content Delivery Network*) est un hébergeur de fichiers dont le but est de mettre du contenu à disposition de la manière la plus performante possible. Au moment de l'importation des outils de three.js, il faudra ainsi indiquer un lien vers le fichier JavaScript distant. Le CDN recommandé par three.js est unpkg.com et le lien est [https://unpkg.com/three@\[METTRE LA VERSION ICI\]/build/three.module.js](https://unpkg.com/three@[METTRE LA VERSION ICI]/build/three.module.js). Par exemple, pour récupérer la version 0.122.0, il faut utiliser le code suivant :

```
1 <script type="module">
2   import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
3   // Reste du code
4 </script>
```

- Il est également possible de télécharger three.js grâce à **npm**, avec la ligne de commande `npm install --save three`. Il suffira alors d'importer la bibliothèque en utilisant le code suivant :

```
1 <script type="module">
2   import * as THREE from 'three';
3   // Reste du code
4 </script>
```

**Attention**

Bien que three.js ne soit composé que de JavaScript, il importe ses propres modules. Il faut donc toujours utiliser `type="module"` dans la balise de script, ce qui provoque des appels HTTP qui peuvent échouer si l'application n'utilise pas de serveur web. Pour utiliser cette librairie, il y a donc deux options :

- Mettre tout le code JavaScript sur l'index du site, afin d'éviter les appels HTTP
- Utiliser un serveur web, tel qu'Apache (via une solution comme XAMPP, WAMP...) ou un *bundler*, comme Webpack. Node est déconseillé par three.js.

Pour des raisons d'accessibilité, tous les exemples montrés dans ce cours sont prévus pour fonctionner en local, sans serveur web. Tout le code JavaScript est donc situé entre des balises `<script>`. Bien entendu, ce n'est pas une bonne pratique : il est recommandé d'exporter son code dans plusieurs fichiers JavaScript. N'hésitez donc pas à mettre en place votre propre serveur web.

**Syntaxe**   **À retenir**

- La balise HTML5 `<canvas>` permet de faire du dessin en JavaScript. Il existe plusieurs technologies, appelées « contextes », permettant d'utiliser un canevas : Canvas2D, plus simple, mais limité au dessin en deux dimensions ; et WebGL, beaucoup plus complexe, mais permettant la 3D.
- Pour simplifier le dessin en 3D, il est possible d'utiliser la librairie three.js.

**Complément**

Le site officiel de three.js<sup>1</sup>

### III. Exercice : Appliquez la notion

**Question 1**

[solution n°1 p.29]

Le code ci-dessous permet d'afficher un objet 3D en utilisant la version 0.122.0 de la librairie three.js, mais l'import de la librairie a été oublié. En vous servant de vos nouvelles connaissances, importez three.js en haut du code JavaScript et déterminez la forme de l'objet 3D affiché.

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Three.js</title>
5   <style>
6     body {
7       margin: 0;
8     }
9
10    canvas {
11      display: block;
12    }
13  </style>
14 </head>
15 <body>
16 <script type="module">
17   // Import de three.js
18
19   // Affichage de la forme
20   let color = 0xFF0000;
```

1 <https://threejs.org/>

```

21   let xRotation = 0.005;
22   let yRotation = 0.005;
23   let cameraZPosition = 3;
24
25   const scene = new THREE.Scene();
26   const camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight,
0.1, 1000);
27   camera.position.z = cameraZPosition;
28
29   const renderer = new THREE.WebGLRenderer();
30   renderer.setSize(window.innerWidth, window.innerHeight);
31   document.body.appendChild(renderer.domElement);
32
33   const geometry = new THREE.BoxGeometry();
34   const material = new THREE.MeshBasicMaterial({color: color});
35   const forme = new THREE.Mesh(geometry, material);
36   scene.add(forme);
37
38   const animate = function () {
39       requestAnimationFrame(animate);
40       forme.rotation.x += xRotation;
41       forme.rotation.y += yRotation;
42       renderer.render(scene, camera);
43   };
44
45   animate();
46 </script>
47 </body>
48 </html>

```

## Question 2

[solution n°2 p.30]

Le script contient 4 variables, déclarées au début de l'exécution du code :

- `color`, la couleur de la forme au format `0xRRVVBB`, où `RR` est la quantité de rouge, `VV` la quantité de vert et `BB` la quantité de bleu, au format hexadécimal (donc de 00 à FF)
- `xRotation`, la vitesse de rotation horizontale
- `yRotation`, la vitesse de rotation verticale
- `cameraZPosition`, la distance de la caméra par rapport au cube

Modifiez ces valeurs pour avoir un cube vert foncé, tournant deux fois plus vite horizontalement que verticalement et avec une caméra plus éloignée.

## IV. Scène, caméra et renderer

### Objectifs

- Comprendre les notions de scène, de caméra et de *renderer*
- Implémenter ces concepts en `three.js`

### Mise en situation

Organiser son espace 3D avec `three.js` peut s'apparenter à la réalisation d'un film. Les objets 3D vont être nos acteurs, mais ils ne se suffisent pas à eux-mêmes. Pour fonctionner, ils vont avoir besoin de 3 choses : une **scène** sur laquelle jouer, une **caméra** pour les filmer, puis, une fois le film tourné, il faudra le mettre sur un **support** pour être visionné (DVD, Blu-ray, VHS, format mp4, format AVI...).



On retrouve ces trois concepts dans de nombreux logiciels de conception 3D, et three.js ne fait pas exception.

### Définition La scène

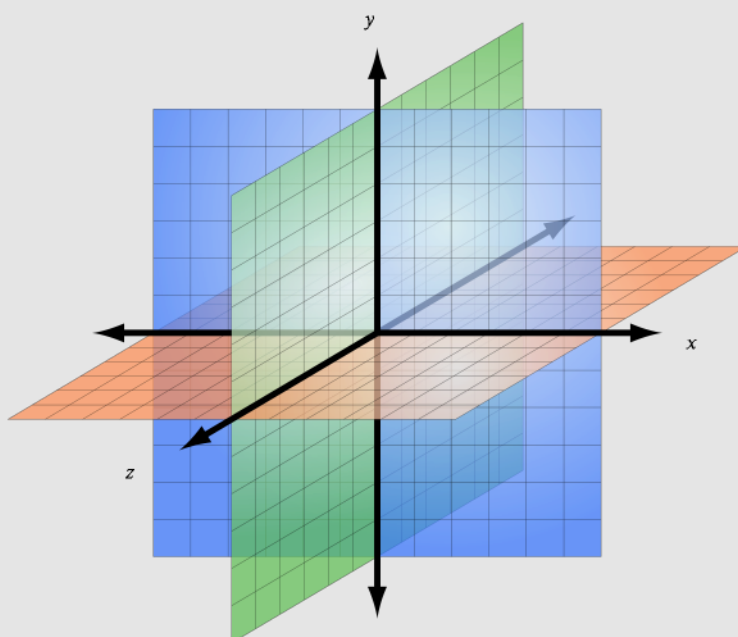
Une **scène** est un espace 3D qui va accueillir les différents éléments à afficher. Tous les objets 3D que three.js va manipuler doivent être ajoutés à une scène pour être visibles. Il est possible de créer plusieurs scènes, mais une seule sera affichée à la fois.

### Méthode

Pour créer une scène en three.js, il suffit d'instancier un objet de type `Scene`. Le constructeur de cette classe ne possède pas de paramètres.

```
1 import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';  
2  
3 // Déclaration de la scène  
4 let scene = new THREE.Scene();
```

Une scène étant un espace tridimensionnel, la position de chaque élément qui va l'occuper sera définie par ses coordonnées sur les axes  $x$ ,  $y$  et  $z$ . Par défaut, les objets auront pour position  $(0, 0, 0)$ .



### Définition La caméra

La scène seule ne sert pas si personne n'est là pour la voir. La **caméra** permet ainsi de définir le point de vue qui sera affiché dans le canevas. Si on prend, par exemple, une scène contenant une maison, c'est la caméra qui va décider si le point de vue est à l'intérieur ou à l'extérieur de la maison. Il existe plusieurs types de caméra, mais la plus utilisée en 3D est la `PerspectiveCamera`, qui simule la vision de l'œil humain.

### Méthode

Pour créer une caméra, il faut instancier un objet de type `PerspectiveCamera`. Le constructeur de cette classe possède 4 paramètres :

- `FieldOfView`, le champ de vision de la caméra, en degrés.
- `AspectRatio`, le ratio largeur/hauteur de l'image finale. Ce ratio est directement lié à la taille du *container* : en mettre un différent provoquerait des effets d'étirement de l'image.
- `NearClippingPlane`, la distance d'affichage minimale. Tous les objets placés en dessous de cette distance de la caméra ne seront pas affichés.
- `FarClippingPlane`, la distance d'affichage maximale. Tous les objets placés au-dessus de cette distance de la caméra ne seront pas affichés.

Un objet `camera` possède une propriété `position` contenant ses coordonnées dans l'espace. Une `position` possède ainsi des composantes `x`, `y` et `z`.

### Exemple

Le code suivant permet de créer une caméra avec un champ de vision de 75 degrés, dans un *container* prenant toute la fenêtre (dont la largeur et la hauteur sont accessibles respectivement via `window.innerWidth` et `window.innerHeight`) et dont la distance d'affichage est entre 0.1 et 1000 :

```
1 import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
2
3 let camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1,
  1000);
```

Il est ensuite possible de modifier sa position. Par exemple, pour la décaler sur la droite, il faut jouer sur l'axe des `x` :

```
1 camera.position.x = 3;
```

### Remarque

Décaler la caméra sur la droite aura pour effet de décaler tous les objets du champ de vision sur la gauche de l'écran.

### Définition Le renderer

Une fois la scène créée et la caméra placée, il va falloir capturer l'image : c'est le rôle du **renderer**. C'est lui qui va déterminer comment l'image finale sera générée. Le *renderer* principal est `WebGLRenderer`, qui permet de dessiner dans un `<canvas>` en utilisant le WebGL.

### Méthode

Pour créer un *renderer*, il suffit de créer une instance de la classe `WebGLRenderer`, dont le constructeur possède un paramètre facultatif sous forme d'objet permettant de configurer le *renderer*.

Par défaut, ce *renderer* crée son propre `<canvas>`, dont il faut définir la taille grâce à la méthode `setSize`, qui prend en paramètres une longueur et une hauteur. Ce `canvas` est stocké dans la propriété `domElement` et doit être ajouté à la page pour être visible. Il est également possible d'utiliser un `canvas` existant en renseignant la propriété `canvas` dans l'objet de configuration du constructeur.

Pour dessiner l'image finale, il faut utiliser la méthode `render`, qui prend en paramètres la scène à dessiner et la caméra à utiliser.

**Remarque**

Dans le cas d'un rendu dans un canevas, l'« aspect ratio » de la caméra doit être calculé à partir de la largeur et de la hauteur de ce dernier.

**Exemple**

Si la page ne possède pas de canevas, alors il est possible de créer un `WebGLRenderer` sans passer de paramètres au constructeur. Voici un exemple de création d'un canevas de la taille de la fenêtre, et son ajout dans le HTML de la page :

```
1 import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
2
3 // On crée le render
4 let renderer = new THREE.WebGLRenderer();
5
6 // On règle la taille du canevas
7 renderer.setSize(window.innerWidth, window.innerHeight);
8
9 // On ajoute le canevas à la balise <body>
10 document.body.appendChild(renderer.domElement);
```

En revanche, si le canevas a déjà été préparé, il suffit de le passer en paramètre du constructeur :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 </head>
5 <body>
6 <canvas width="200" height="200" id="canvas3d"></canvas>
7 <script type="module">
8   import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
9
10   // La propriété canvas reçoit la balise <canvas> déclarée dans le HTML de la page
11   const renderer = new THREE.WebGLRenderer(
12     {
13       canvas: document.getElementById('canvas3d')
14     }
15   );
16 </script>
17 </body>
18 </html>
```

Dans ce cas, il ne faut pas oublier de calculer l'*aspect ratio* de la caméra en utilisant la largeur et la hauteur du canevas existant avant de générer l'image finale avec la méthode `render`, sinon l'image sera étirée :

```
1 let canvasElement = document.getElementById('canvas3d');
2 let scene = new THREE.Scene();
3 let camera = new THREE.PerspectiveCamera(75, canvasElement.width / canvasElement.height, 0.1,
4   1000);
5 let renderer = new THREE.WebGLRenderer(
6   {
7     canvas: canvasElement
8   }
9 );
10 renderer.render(scene, camera);
```

Pour le moment, cet exemple n'affiche qu'un écran noir, mais il servira de base aux futurs environnements 3D.

### Syntaxe À retenir

- Pour créer un environnement 3D, il faut créer une scène sur laquelle les objets vont être placés, une caméra qui va définir le point de vue et un *renderer* qui va créer l'image finale. Chaque élément placé sur la scène possède des coordonnées permettant de définir sa position sur les axes *x*, *y* et *z*.

### Complément

Introduction à three.js<sup>1</sup>

## V. Exercice : Appliquez la notion

### Question

[solution n°3 p.30]

Le code JavaScript suivant permet d'ajouter une sphère à une scène stockée dans une variable *scene* :

```
1 // Création des éléments
2 /* Votre code ici */
3
4 // Création et ajout de la sphère à la scène
5 const geometry = new THREE.SphereGeometry(2, 16, 16);
6 const material = new THREE.MeshBasicMaterial({color: 0x6600FF});
7 const sphere = new THREE.Mesh(geometry, material);
8 scene.add(sphere);
9
10 // Rendu de la scène
11 /* Votre code ici */
```

Sur une page HTML, créez un canevas d'une largeur de 300 et d'une hauteur de 300. Importez ensuite la librairie *three.js*, puis créez les éléments suivants en amont du code permettant de créer la sphère :

- Une scène, dans une variable nommée *scene*.
- Une caméra ayant un angle de vue de 80 et une distance d'affichage comprise entre 0.1 et 1000.
- Un *renderer* WebGL utilisant le canevas de la page HTML.

Attention : par défaut, la sphère et la caméra sont toutes les deux créées aux coordonnées (0, 0, 0), donc la sphère est en dehors de la distance d'affichage. Reculez donc suffisamment la caméra pour que la sphère puisse être dessinée.

Une fois la sphère créée, demandez au *renderer* de dessiner l'image finale.

## VI. Ajouter des éléments

### Objectifs

- Créer des objets 3D
- Ajouter des éléments à la scène
- Éclairer la scène

### Mise en situation

Pour le moment, la caméra n'a rien à filmer puisque la scène est vide. Dans cette partie, nous allons voir comment créer des objets 3D et les ajouter à la scène, ainsi que la gestion de l'éclairage.

<sup>1</sup> <https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene>

**Définition**   **Geometry**

La librairie three.js dispose de nombreuses formes tridimensionnelles basiques : des **cubes** (`BoxGeometry`), des **cylindres** (`CylinderGeometry`), des **sphères** (`SphereGeometry`), des **cônes** (`ConeGeometry`)...

Tous ces éléments sont des objets de type `Geometry`, qui représentent un ensemble de points dans l'espace. Ces points seront ensuite reliés entre eux pour créer les faces de l'objet final.

**Exemple**

Pour créer un objet en forme de boîte, il faut instancier une `BoxGeometry`. Par défaut, la forme est un cube de taille 1, mais le constructeur permet de définir sa largeur, sa hauteur et sa profondeur :

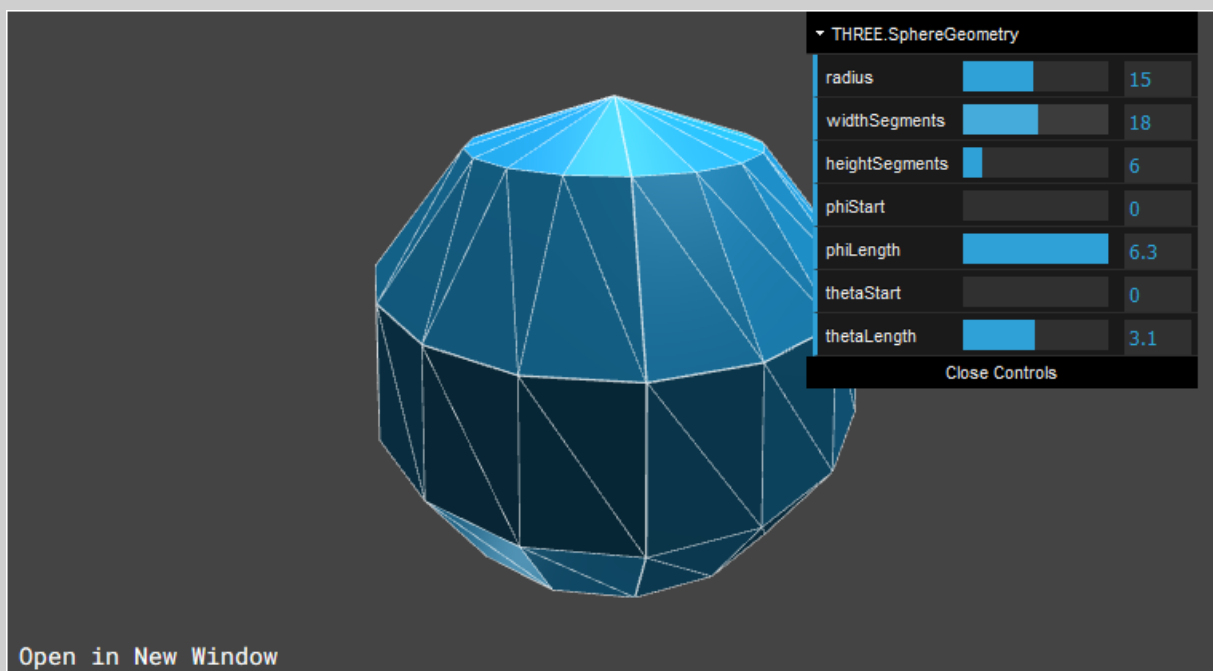
```
1 // Un rectangle de 1x2x1
2 let geometry = new THREE.BoxGeometry(1, 2, 1);
```

Pour créer une sphère, il faut instancier une `SphereGeometry`, qui prend un minimum de trois paramètres : le rayon de la sphère et le nombre de segments horizontaux et verticaux. Plus le nombre de segments est élevé, plus il y aura de triangles composant la sphère, ce qui aura pour effet de lui donner une forme plus arrondie. Attention cependant, un nombre trop élevé de triangles à rendre peut nuire aux performances de la scène.

```
1 // Une sphère d'un rayon de 2, composée de 16 segments horizontaux et verticaux
2 let geometry = new THREE.SphereGeometry(2, 16, 16);
```

**Conseil**

La partie « Geometry » de la documentation, dans le menu de gauche, recense toutes les géométries utilisables. La page de chaque géométrie permet de retrouver les paramètres de son constructeur et dispose également d'une démonstration interactive permettant de modifier en direct chaque paramètre pour visualiser son rendu.

**Définition**   **Material**

Un ensemble de points n'est pas suffisant pour créer un objet 3D. En effet, un *renderer* a besoin de savoir à quoi doivent ressembler les faces de l'objet : leur couleur, mais aussi leur texture, la réaction qu'ils ont lorsque la lumière les frappe (réflexion et réfraction), etc. Toutes ces informations sont contenues dans un objet de type **Material**, qu'il faut appliquer à l'objet.

Three.js possède plusieurs `Material` de base, comme le `MeshBasicMaterial`, qui n'est pas affecté par la lumière ; le `MeshStandardMaterial`, qui permet de créer des objets plus réalistes et affectés par la lumière ; ou le `MeshToonMaterial`, qui permet de donner un effet « cartoon », avec des ombres ayant un nombre limité de graduations.

Pour créer un `Material`, il suffit d'instancier la classe correspondante. Le constructeur de chaque `Material` peut recevoir un objet permettant de le configurer. Les propriétés dépendent du type de `Material` créé.

### Exemple

Pour créer un `MeshStandardMaterial`, il faut utiliser le code suivant :

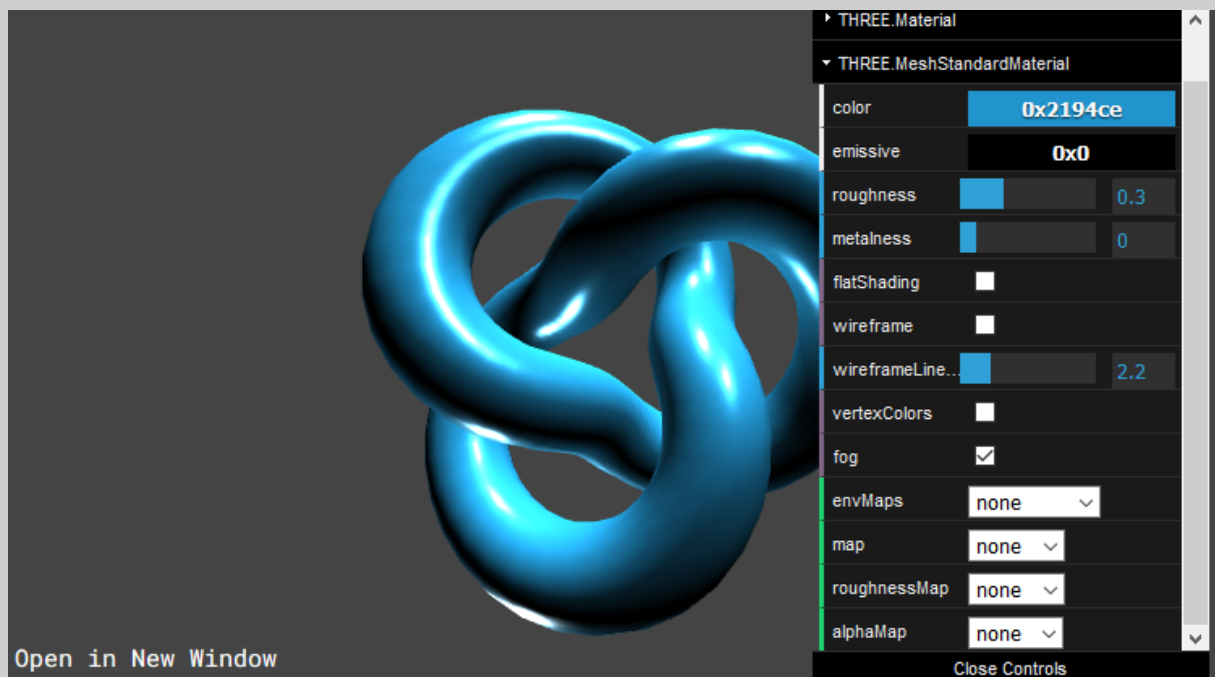
```
1 let material = new THREE.MeshStandardMaterial();
```

Il est possible de renseigner certains paramètres, comme la rugosité, qui permet d'avoir des objets plus ou moins réfléchissants :

```
1 let material = new THREE.MeshStandardMaterial({
2   roughness: 0.3,
3 });
```

### Conseil

Comme pour les `Geometry`, la partie `Material` présente tous les matériaux possibles, avec une démonstration interactive permettant de mieux comprendre l'effet des différents paramètres :



### Méthode Mesh

Un ensemble de `Geometry` et de `Material` est appelé **Mesh**. Ainsi, pour appliquer un matériau à un ensemble de points, il va falloir créer un objet `Mesh` dont le constructeur va prendre ces deux éléments en paramètres.

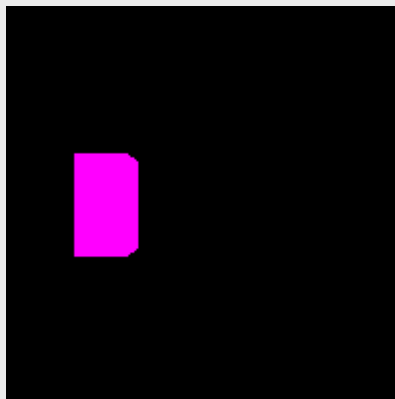
Le `Mesh` est l'élément qui sera ajouté à la scène en dernier, grâce à la méthode `add` de l'objet `Scene`. Les objets doivent être ajoutés à la scène avant le rendu de l'image finale, sans quoi ils n'y apparaîtront pas.

Par défaut, le `Mesh` est ajouté à la position (0, 0, 0), mais il est possible de modifier son emplacement grâce à sa propriété `position`. La méthode `set` permet de définir de nouvelles coordonnées, mais il est également possible de modifier indépendamment les propriétés `x`, `y` et `z`.

### Exemple

Le code suivant permet de créer une boîte de 1x2x1, de lui appliquer un matériau basique qui n'est pas affecté par la lumière, de créer un `Mesh` à la position (-2, 0, -3) et de l'ajouter à la scène :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 </head>
5 <body>
6 <canvas width="200" height="200" id="canvas"></canvas>
7 <script type="module">
8   import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
9
10  // Création de la scène, de la caméra et du renderer
11  let canvasElement = document.getElementById('canvas');
12  let scene = new THREE.Scene();
13  let camera = new THREE.PerspectiveCamera(70, canvasElement.width / canvasElement.height,
14  0.1, 1000);
15  camera.position.z = 3;
16  let renderer = new THREE.WebGLRenderer({
17    {
18      canvas: canvasElement
19    }
20  });
21
22  // Création de l'objet
23  let geometry = new THREE.BoxGeometry(1, 2, 1);
24  let material = new THREE.MeshBasicMaterial({color: 0xFF00FF});
25  let box = new THREE.Mesh(geometry, material);
26  box.position.set(-2, 0, -3);
27  // Ajout de l'objet à la scène
28  scene.add(box);
29
30  // Rendu de la scène
31  renderer.render(scene, camera);
32 </script>
33 </body>
34 </html>
```



### Méthode

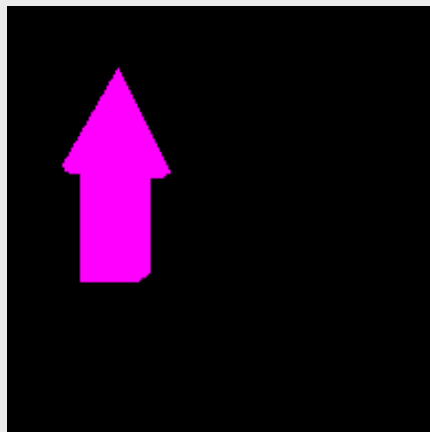
Il est également possible d'ajouter un `Mesh` à l'intérieur d'un autre en utilisant la méthode `add`. Dans ce cas, seul l'élément parent devra être ajouté à la scène et tous les sous-objets seront déplacés en même temps que l'objet parent. Cela permet de créer des structures complexes, composées de plusieurs formes, et de pouvoir les manipuler facilement.

À noter que, dans tous les cas, la position d'un objet dépend de son parent : la position des objets ajoutés à la scène se fait par rapport à l'origine de la scène, tandis que les sous-objets sont placés par rapport à leur objet parent.

### Exemple

Ajoutons un cône au-dessus de la boîte :

```
1 // Création de la boîte
2 let boxGeometry = new THREE.BoxGeometry(1, 2, 1);
3 let material = new THREE.MeshBasicMaterial({color: 0xFF00FF});
4 let box = new THREE.Mesh(boxGeometry, material);
5 // Ajout de la boîte à la scène
6 scene.add(box);
7 // On place la boîte aux coordonnées (-2, 0, -3) de la scène
8 box.position.set(-2, 0, -3);
9
10 // Création du cône
11 let coneGeometry = new THREE.ConeGeometry(1, 2);
12 let cone = new THREE.Mesh(coneGeometry, material);
13 // On ajoute le cône à la boîte
14 box.add(cone);
15 // On place le cône aux coordonnées (0, 2, 0) de la boîte.
16 // À noter qu'ici, on place le cône par rapport à la boîte et non pas à la scène.
17 cone.position.set(0, 2, 0);
```



### Méthode Lumières

Jusqu'à présent, seul le `MeshBasicMaterial` a été utilisé dans les exemples : les autres n'afficheront qu'un écran noir. Cela s'explique par le fait qu'il n'y a aucune source de lumière sur la scène : elle est plongée dans l'obscurité et l'utilisation d'un matériau qui réagit à la lumière ferait disparaître l'objet 3D.

Three.js possède plusieurs types de lumières à ajouter à la scène, comme `AmbientLight`, qui éclaire tous les objets de la scène de manière égale ; `SpotLight`, qui simule un projecteur ; ou `PointLight`, qui diffuse de la lumière dans toutes les directions à partir d'un point, comme un soleil ou une ampoule. Chaque lumière doit être ajoutée à la scène avec `add`.

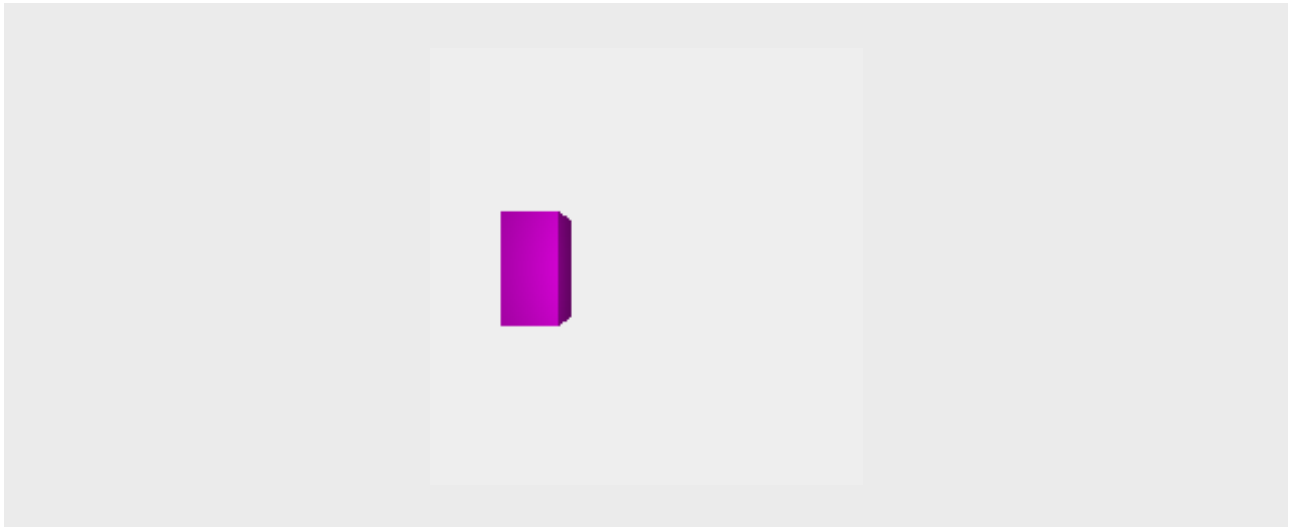


Il est également possible de définir la couleur du fond (par défaut, noir) grâce à la méthode `setClearColor` du `renderer`.

### Exemple

Reprenons l'exemple suivant en utilisant le `MeshStandardMaterial`, pour un rendu plus réaliste. Pour cela, il faut également ajouter une source de lumière, comme une `PointLight` : son constructeur prend en paramètre une couleur (ici, blanc), une intensité et une distance limite d'effet. Par souci de visibilité, la couleur est également changée pour un gris clair :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 </head>
5 <body>
6 <canvas width="200" height="200" id="canvas"></canvas>
7 <script type="module">
8   import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
9
10  // Création de la scène, de la caméra et du renderer
11  let canvasElement = document.getElementById('canvas');
12  let scene = new THREE.Scene();
13  let camera = new THREE.PerspectiveCamera(70, canvasElement.width / canvasElement.height,
14  0.1, 1000);
15  camera.position.z = 3;
16  let renderer = new THREE.WebGLRenderer({
17    {
18      canvas: canvasElement
19    }
20  });
21
22  // Création de l'objet
23  let geometry = new THREE.BoxGeometry(1, 2, 1);
24  let material = new THREE.MeshStandardMaterial({color: 0xFF00FF});
25  let box = new THREE.Mesh(geometry, material);
26  box.position.set(-2, 0, -3);
27  // Ajout de l'objet à la scène
28  scene.add(box);
29
30  // Ajout de la lumière blanche
31  let light = new THREE.PointLight( 0xFFFFFF, 1, 50 );
32  light.position.set(0, 0, 0);
33  scene.add(light);
34
35  // Rendu de la scène
36  renderer.setClearColor(0xEEEEEE);
37  renderer.render(scene, camera);
38 </script>
39 </body>
40 </html>
```



### Syntaxe À retenir

- Pour créer un objet 3D, il faut d'abord définir sa `Geometry`, puis son `Material`, afin de créer un `Mesh` qui pourra être ajouté à la scène grâce à la méthode `add`.
- Pour éclairer la scène, il faut également y ajouter un ou plusieurs objets `Light`. Il est également possible de définir la couleur de fond grâce à la méthode `setClearColor` du `renderer`.

### Complément

Un exemple de page de géométrie sur la documentation : la sphère<sup>1</sup>

Un exemple de page de matériau sur la documentation : le `MeshStandardMaterial`<sup>2</sup>

## VII. Exercice : Appliquez la notion

### Question

[solution n°4 p.31]

Un client souhaiterait créer un environnement 3D ayant pour thème « Vacances au bord de la plage » et aurait besoin de petits parasols en 3D. En vous aidant de la documentation officielle de `three.js`, créez une scène et ajoutez-y un parasol composé d'un **cylindre blanc**, pour faire le mât, et d'un **cône vert**, pour faire la toile. La lumière doit interagir avec chaque composant du parasol, et le mât doit refléter légèrement la lumière. Voici un exemple de parasol possible :



<sup>1</sup> <https://threejs.org/docs/#api/en/geometries/SphereGeometry>

<sup>2</sup> <https://threejs.org/docs/#api/en/materials/MeshStandardMaterial>

La taille et les couleurs exactes sont à votre convenance : faites vos propres parasols, tant qu'ils respectent les contraintes de base. Il faut également que le parasol soit simple à manipuler. En déplaçant le mât, la toile doit également se déplacer.

### Indice :

Aidez-vous de la documentation du cylindre<sup>1</sup> et de celle du cône<sup>2</sup>.

### Indice :

Pour que le mat reflète légèrement la lumière, il faut donner à son matériau une valeur de `roughness` inférieure à 1 : l'exemple utilise 0.3. Pour voir la différence, il faut zoomer sur le mât, avec une source de lumière assez proche.

## VIII. Animer des objets

### Objectifs

- Comprendre le concept de *render loop*
- Animer un objet 3D

### Mise en situation

Rendre une simple image, c'est déjà bien, mais animer en temps réel les différents objets 3D présents sur la scène, c'est encore mieux. Dans cette partie, nous allons voir comment réaliser des animations simples, comme changer la position d'un objet et comment le faire tourner sur lui-même.

#### Définition La render loop

Pour que les objets soient animés, il ne va pas falloir générer une seule image, comme jusqu'à présent, mais une succession d'images. En effet, une vidéo n'est qu'un ensemble d'images affichées très rapidement (appelées *frames*), ce qui a pour effet de donner l'illusion du mouvement. Pour créer une animation, il va falloir faire le rendu de la scène dans une boucle. Or, une simple boucle ne suffit pas : il faut également que chaque image soit rendue à un intervalle constant, sinon les animations ne seront pas fluides.

#### Méthode

Pour créer une *render loop*, il faut utiliser une fonction **réursive**, c'est-à-dire une fonction qui va s'appeler elle-même. Les navigateurs disposent nativement d'outils permettant de gérer les *render loop*, et l'un d'entre eux est la fonction `requestAnimationFrame`. Celle-ci prend en paramètre une fonction à appeler à chaque *frame* de notre animation : c'est elle qui va bouger les objets, puis faire le rendu de la scène avant d'attendre la prochaine *frame* (en appelant `requestAnimationFrame`).

Pour faire bouger un objet, il faut donc changer légèrement la propriété voulue (`position`, `rotation`...) à chaque tour de boucle.

```
1 const animate = function () {
2   // Modification des attributs des objets (position, rotation...)
3   // ...
4
5   // Rendu de la frame
6   renderer.render(scene, camera);
7
8   // Attente de la prochaine frame. La fonction animate sera appelée par la fonction
  requestAnimationFrame
9   requestAnimationFrame(animate); // On attend la prochaine frame
```

1 <https://threejs.org/docs/#api/en/geometries/CylinderGeometry>

2 <https://threejs.org/docs/index.html#api/en/geometries/ConeGeometry>

```
10 };
11
12 // On appelle une première fois la fonction pour rentrer dans la boucle
13 animate();
```

### Attention

Les méthodes de rotation prennent en paramètres des radians. Pour convertir des degrés en radians, il faut réaliser le calcul suivant :  $\text{degrés} * \text{Math.PI} / 180$ .

### Exemple

Voici le code permettant de créer une scène en plein écran avec un cône qui s'envole dans les airs en tournant :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <style>
5     body {
6       margin: 0;
7     }
8
9     canvas {
10      display: block;
11    }
12  </style>
13 </head>
14 <body>
15 <script type="module">
16   import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
17
18   // Création de la scène, de la caméra et du renderer
19   let scene = new THREE.Scene();
20   let camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1,
21   1000);
22   camera.position.z = 4;
23
24   let renderer = new THREE.WebGLRenderer();
25   renderer.setSize(window.innerWidth, window.innerHeight);
26   document.body.appendChild(renderer.domElement);
27
28   // Création du cône
29   let coneGeometry = new THREE.ConeGeometry(1, 2, 16);
30   let coneMaterial = new THREE.MeshStandardMaterial({color: 0xFFFF00});
31   let cone = new THREE.Mesh(coneGeometry, coneMaterial);
32   scene.add(cone);
33
34   // Lumière
35   let light = new THREE.PointLight(0xFFFFFF, 1, 50);
36   light.position.set(0, 0, 3);
37   scene.add(light);
38
39   // Couleur du fond de la scène
40   renderer.setClearColor(0xFFFFFF);
41
42   const animate = function () {
43     // Déplacement du cube sur l'axe des y
```

```

43     cone.position.y += 0.01;
44
45     // Rotation du cube sur l'axe des y
46     cone.rotation.y += 0.01;
47
48     // Rendu de la frame
49     renderer.render(scene, camera);
50
51     // Attente de la prochaine frame
52     requestAnimationFrame(animate);
53 };
54
55 // On appelle une première fois la fonction pour rentrer dans la boucle
56 animate();
57 </script>
58 </body>
59 </html>

```

### Méthode Animer les objets

Les propriétés **position** et **rotation** des `Mesh` permettent de faire bouger un objet sur un axe de la scène. Cependant, l'objet `Mesh` possède également des méthodes permettant d'effectuer des mouvements basés sur les axes locaux de l'objet, qui dépendent donc de la rotation de ce dernier.

Pour déplacer un `Mesh` sur un axe local, il est possible d'utiliser les méthodes `translateX`, `translateY` et `translateZ`, qui prennent en paramètre la distance à parcourir. Il est également possible de faire bouger l'objet sur un axe personnalisé grâce à la méthode `translateOnAxis`, qui prend en paramètres un objet `Vector3`, avec des propriétés `x`, `y` et `z` indiquant l'axe à suivre, et une distance.

Pour faire tourner un objet sur un axe local, il est possible d'utiliser `rotateX`, `rotateY` ou `rotateZ`, qui prennent en paramètre l'angle de rotation souhaité, en radians. `rotateOnAxis` permet également de faire tourner l'objet selon n'importe quel axe.

### Exemple

Cette fonction permet de faire s'envoler le cône de l'exemple précédent tout en le faisant tourner sur lui-même. Étant donné que l'objet n'a pas une rotation de base, l'animation est équivalente à la précédente.

```

1 const animate = function () {
2     // Déplacement et rotation du cube sur l'axe des y
3     cone.translateY(0.01);
4     cone.rotateY(0.01);
5
6     // Rendu de la frame
7     renderer.render(scene, camera);
8
9     // Attente de la prochaine frame
10    requestAnimationFrame(animate);
11 };

```

Cependant, en rajoutant `cone.rotation.x = 90 * Math.PI / 180;` au moment d'initialiser l'objet, afin qu'il soit tourné vers la caméra, les deux animations seront différentes : le premier exemple continue de faire monter l'objet (axe Y de la scène), tandis que le second fait avancer le cube vers la caméra (axe Y de l'objet, donc après la rotation).

### Méthode Supprimer des objets de la scène

Maintenant que nos objets peuvent se déplacer, il est possible que certains sortent du cadre et ne soient plus visibles. Or, même hors champ, ces objets continuent d'utiliser des ressources. Il est donc préférable de les supprimer en utilisant la méthode `remove` de l'objet `Scene`. La liste des objets de la scène est accessible depuis la propriété `children`.

### Exemple

Lorsque le cône est au-dessus d'une certaine limite, on le supprime :

```
1 const animate = function () {
2   // Si le cône est sur la scène...
3   if (scene.children.includes(cone)) {
4     // Déplacement et rotation du cube sur l'axe des y
5     cone.translateY(0.01);
6     cone.rotateY(0.01);
7
8     if (cone.position.y > 3) {
9       scene.remove(cone); // On supprime le cône
10    }
11  }
12
13  // Rendu de la frame
14  renderer.render(scene, camera);
15
16  // Attente de la prochaine frame
17  requestAnimationFrame(animate);
18 };
```

### Attention

Il est plus performant de changer la position d'un objet plutôt que de le supprimer, puis de le recréer. Si un objet est utilisé plusieurs fois, il est préférable de le déplacer hors champ en attendant de le réutiliser.

### Syntaxe À retenir

- Une animation est composée de plusieurs *frames*, c'est-à-dire d'images affichées successivement, suffisamment rapidement pour donner une impression de mouvement. Chaque *frame* est générée dans une *render loop*, c'est-à-dire une boucle qui va s'occuper de bouger les objets et de calculer la prochaine image.
- Pour manipuler des objets 3D, l'objet `Mesh` dispose de nombreuses méthodes, comme `translateX` ou `rotateX`. Pour supprimer un objet de la scène, il faut utiliser la méthode `remove`.

### Complément

L'objet `Object3D`, parent des `Mesh`<sup>1</sup>

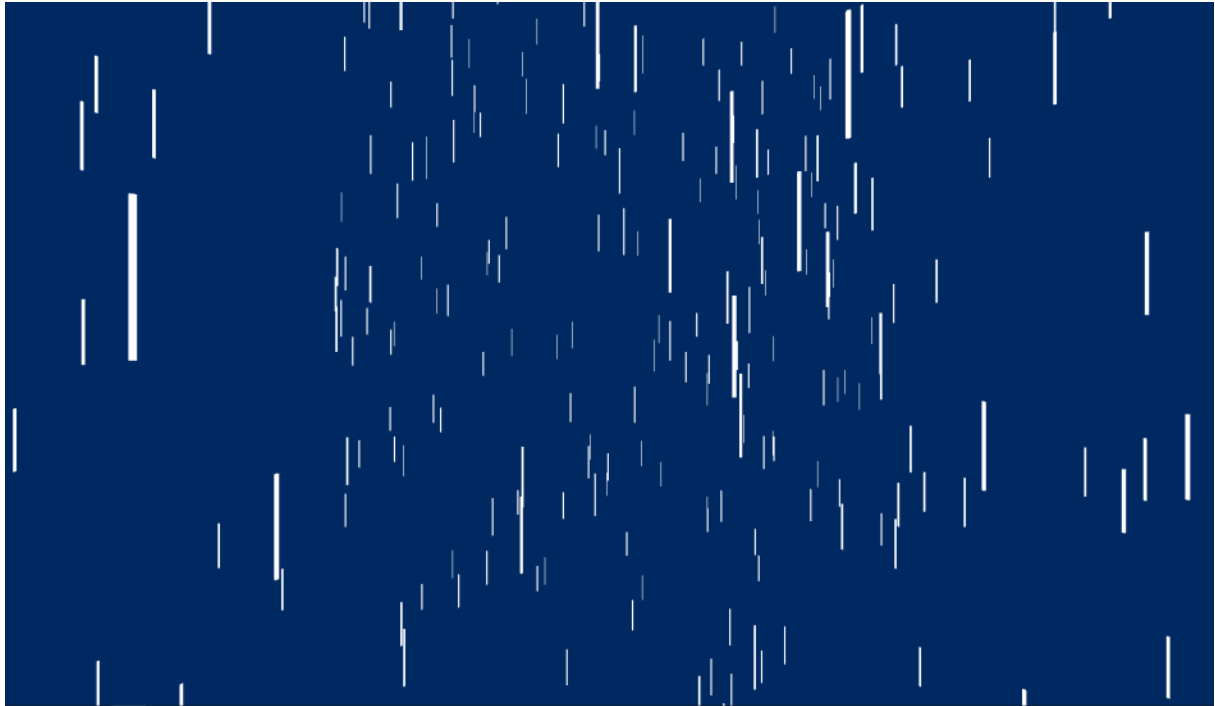
<sup>1</sup> <https://threejs.org/docs/#api/en/core/Object3D>

## IX. Exercice : Appliquez la notion

### Question

[solution n°5 p.32]

Pour cet exercice, vous allez devoir faire une scène de pluie nocturne. Sur une scène au fond bleu nuit, vous allez devoir créer une pluie diluvienne dont les gouttes d'eau seront représentées par des cylindres blancs utilisant un matériau qui n'est pas sensible à la lumière.



Toutes les gouttes d'eau devront apparaître hors champ, au-dessus de la caméra, à des coordonnées aléatoires (comprises entre 0 et 10 sur les axes des  $x$  et des  $z$ ) et tomber jusqu'à ce qu'elles atteignent 0 sur l'axe des  $y$ . Une fois atteint, la goutte devra être déplacée à de nouvelles coordonnées aléatoires, au-dessus de la caméra, et ainsi de suite.

Pour vous aider, voici une fonction JavaScript permettant de générer un nombre aléatoire entre 0 et 10 :

```
1 function getRandomNumber()  
2 {  
3     return Math.random() * Math.floor(10);  
4 }
```

#### Indice :

La position de départ des gouttes doit utiliser 3 coordonnées aléatoires, pour les axes  $x$ ,  $y$  et  $z$ . Cependant, il faut ajouter une valeur fixe à l'axe des  $y$  pour s'assurer que toutes les gouttes apparaissent hors champ, tout en conservant un décalage entre les gouttes.

#### Indice :

Il ne suffit que d'une `Geometry` et d'un `Material` pour créer autant de `Mesh` que l'on souhaite.

#### Indice :

Vous pouvez utiliser le tableau `scene.children` pour parcourir toutes les gouttes : il ne doit y avoir rien d'autre sur la scène (la lumière n'est pas utile, puisque les gouttes ne doivent pas y être sensibles).

## X. Essentiel

## XI. Auto-évaluation

### A. Exercice final

#### Exercice 1

[solution n°6 p.34]

Exercice

Quels sont les contextes de rendu utilisables en JavaScript ?

- ☐ DirectX
- ☐ WebGL
- ☐ OpenGL
- ☐ Canvas2D

Exercice

Pourquoi utiliser three.js plutôt que le WebGL ?

- ☐ Three.js est plus performant
- ☐ Three.js est plus simple à utiliser
- ☐ Three.js ne sert que d'introduction à la 3D, mais il faudra bien utiliser le WebGL sur les vraies applications

Exercice

À quoi sert le *render* ?

- ☐ À placer tous les éléments qui vont être affichés
- ☐ À définir le point de vue à partir duquel les éléments vont être affichés
- ☐ À générer l'image finale

Exercice

Qu'est-ce que le *clipping plane* ?

- ☐ Le champ de vision de la caméra
- ☐ La distance d'affichage des objets
- ☐ Le ratio largeur/hauteur de l'image finale

Exercice

Quelle géométrie permet de créer un cube dans three.js ?

- ☐ BoxGeometry
- ☐ CubeGeometry
- ☐ RectGeometry

Exercice



Quel est l'élément qui représente l'objet 3D présent sur la scène ?

- ☐ Geometry
- ☐ Material
- ☐ Renderer
- ☐ Mesh

Exercice

Qu'est-ce que la *render loop* ?

- ☐ La boucle qui permet d'ajouter tous les éléments dans la scène
- ☐ La boucle qui permet de générer toutes les images d'une animation
- ☐ La boucle qui permet d'importer tous les éléments de la librairie

Exercice

Que fait la fonction `requestAnimationFrame` ?

- ☐ Elle appelle une fonction de *callback* au moment où la prochaine *frame* doit être générée
- ☐ Elle génère la *frame* courante
- ☐ Elle récupère toutes les *frames* pour en faire une animation

Exercice

Comment déplacer un objet sur l'axe des *x* de la scène ?

- ☐ La méthode `rotateX`
- ☐ La méthode `translateX`
- ☐ La propriété `position.x`

Exercice

Comment récupérer tous les objets présents sur la scène ?

- ☐ La méthode `getChildren`
- ☐ La méthode `get`
- ☐ La propriété `children`

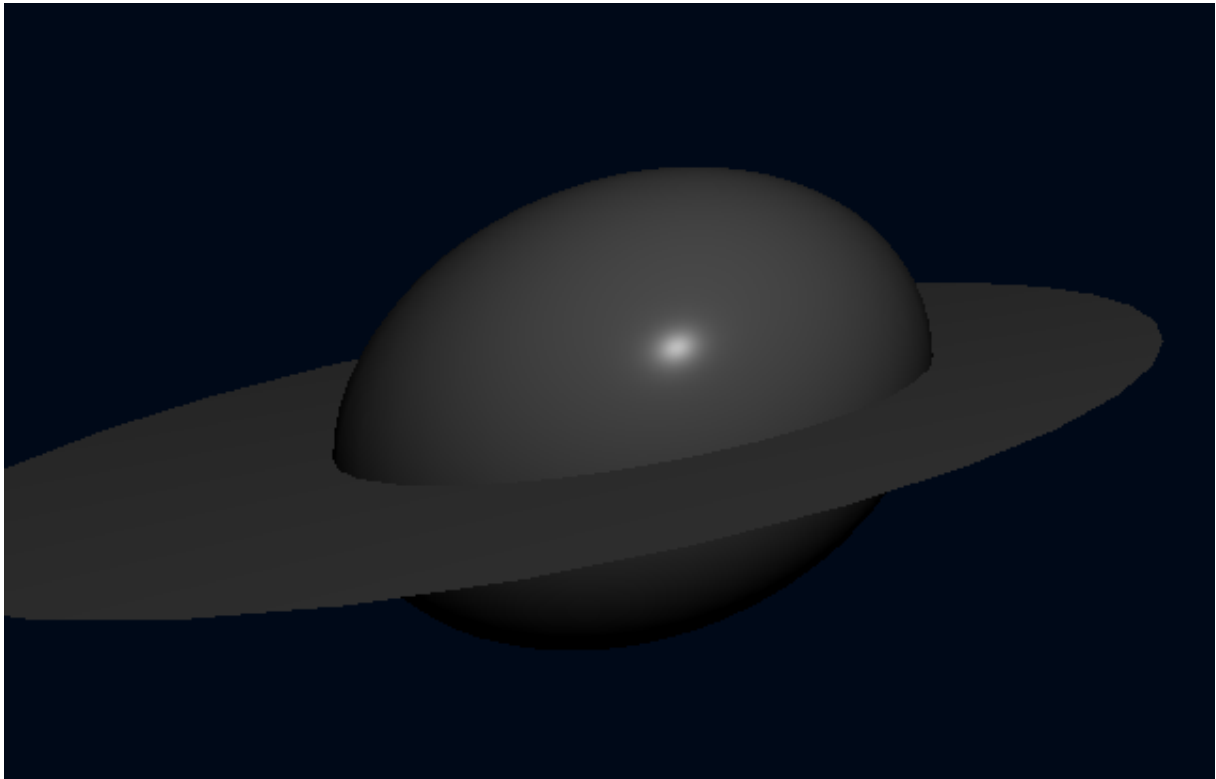
## B. Exercice : Défi

Dans cet exercice final, vous allez devoir poser les bases d'un jeu vidéo en 3D, dans lequel le joueur doit piloter un vaisseau spatial en évitant les astéroïdes. Le jeu ne sera pas complètement fonctionnel à la fin de cet exercice, mais la base des éléments graphiques sera présente.

### Question 1

[solution n°7 p.36]

Le joueur va piloter un vaisseau alien typique, composé d'une sphère (`SphereGeometry`) et d'un anneau (`RingGeometry`). Il devra être gris, se refléter à la lumière et évoluer dans un univers bleu foncé. Voici un exemple de vaisseau attendu :



En vous aidant de vos nouvelles connaissances et de la documentation de `three.js`, créez toutes les composantes d'une scène, puis ajoutez-y un vaisseau similaire à celui au-dessus. Attention : par défaut, les `RingGeometry` ne sont visibles que d'un côté, il est conseillé de les faire tourner avec un angle de 270 degrés pour qu'ils apparaissent comme sur l'exemple.

Afin de faciliter les prochaines étapes, l'anneau doit bouger en même temps que la sphère.

#### Indice :

N'oubliez pas de placer une lumière pour que le vaisseau soit visible, sinon il apparaîtra entièrement noir.

### Question 2

[solution n°8 p.37]

Il est temps d'animer le vaisseau. Pour cela, voici une classe JavaScript permettant de gérer les commandes utilisateur :

```
1 class UFOInput {
2   constructor() {
3     this.currentInput = null;
4     window.onkeydown = this.updateCurrentInput.bind(this);
5     window.onkeyup = this.nullifyCurrentInput.bind(this);
6   }
7
8   updateCurrentInput(event)
9   {
10    if (event.code === 'ArrowLeft') {
11      this.currentInput = 'LEFT';
12    } else if (event.code === 'ArrowRight') {
13      this.currentInput = 'RIGHT';
14    }
15  }
16 }
```

```

15     }
16
17     nullifyCurrentInput()
18     {
19         this.currentInput = null;
20     }
21
22     getCurrentInput()
23     {
24         return this.currentInput;
25     }
26 }

```

Pour l'utiliser, il faut créer une instance de la classe `UFOInput` et utiliser la méthode `getCurrentInput`, qui retourne *LEFT* si le joueur appuie sur la flèche de gauche, *RIGHT* s'il appuie sur la flèche de droite, et *null* si aucune des deux touches n'est appuyée.

Utilisez cette classe pour déplacer le vaisseau dans la direction choisie par l'utilisateur. Le vaisseau ne pourra se déplacer qu'horizontalement.

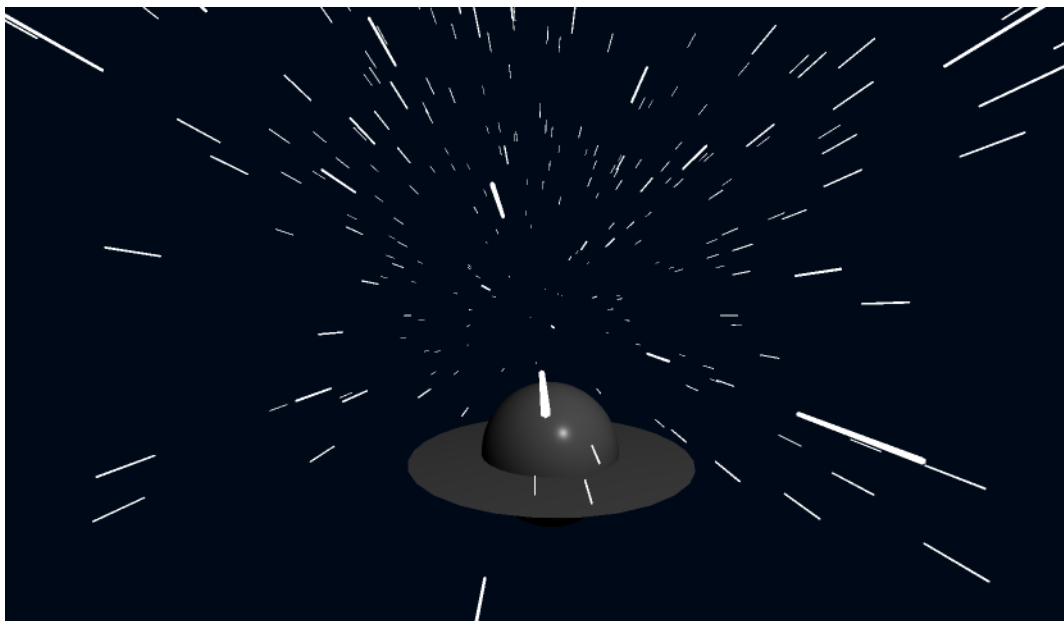
### Indice :

Il va falloir créer une *render loop* et vérifier la touche appuyée à chaque *frame*.

### Question 3

[solution n°9 p.38]

Pour le moment, le vaisseau a l'air de faire du surplace. Pour ajouter un effet de vitesse, créez des « étoiles », c'est-à-dire des cylindres se déplaçant vers la caméra :



Pour cela, vous allez devoir faire apparaître des cylindres à des positions aléatoires et les faire avancer sur l'axe *z* dans une zone de 10x10. Pour donner un effet de profondeur, la coordonnée de départ sur l'axe des *z* devra être reculée de 10. Voici une fonction permettant de gérer un nombre aléatoire entre 0 et 10 :

```

1 function getRandomNumber()
2 {
3     return Math.random() * Math.floor(10);
4 }

```

Les cylindres doivent être blancs et ne doivent pas refléter la lumière. Une fois qu'une étoile a suffisamment avancé et qu'elle passe hors champ, il faudra la replacer à une nouvelle coordonnée aléatoire pour donner l'illusion que le vaisseau ne s'arrête jamais. Pour que la galaxie soit peuplée, insérez 500 étoiles dans l'univers.

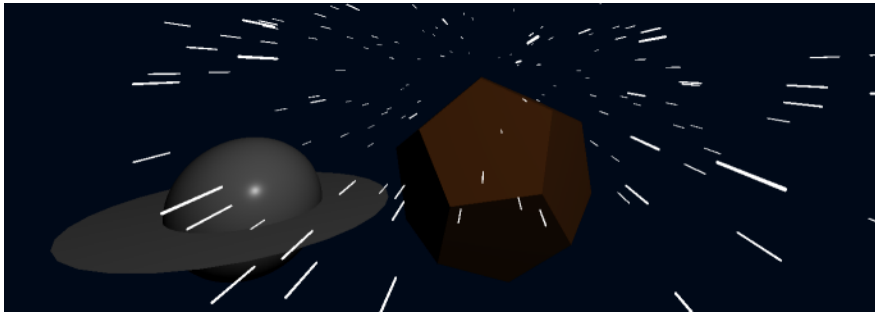
**Indice :**

Attention, contrairement à l'exercice précédent sur les gouttes d'eau, il ne faudra pas récupérer tous les éléments de la scène pour les faire avancer, mais seulement les étoiles. Pour cela, il faudra stocker les étoiles dans un tableau.

**Question 4**

[solution n°10 p.39]

Pour finir, rajoutons un astéroïde que le vaisseau devra éviter. Cet astéroïde sera représenté par un `DodecahedronGeometry` de couleur marron qui réagit à la lumière. Il devra se déplacer vers la caméra, de la même manière que les étoiles, mais en tournant sur lui-même. Il ne doit y avoir qu'un seul météore, qui sera remplacé une fois hors champ.



Le vaisseau ne pouvant se déplacer qu'horizontalement, l'astéroïde devra apparaître sur la même coordonnée  $y$  que le vaisseau. De plus, pour que le joueur puisse avoir le temps de réagir, le météore devra apparaître à la coordonnée  $-40$  et se rapprocher de la caméra sur l'axe des  $z$ . Sa coordonnée  $x$ , elle, doit être aléatoire.

**Solutions des exercices**

## p.7 Solution n°1

Pour importer la librairie, il faut ajouter la ligne `import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';` au début du code :

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Three.js</title>
5   <style>
6     body {
7       margin: 0;
8     }
9
10    canvas {
11      display: block;
12    }
13  </style>
14 </head>
15 <body>
16 <script type="module">
17   // Import de three.js
18   import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
19
20   // Affichage de la forme
21   let color = 0xFF0000;
22   let xRotation = 0.005;
23   let yRotation = 0.005;
24   let cameraZPosition = 3;
25
26   const scene = new THREE.Scene();
27   const camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight,
0.1, 1000);
28   camera.position.z = cameraZPosition;
29
30   const renderer = new THREE.WebGLRenderer();
31   renderer.setSize(window.innerWidth, window.innerHeight);
32   document.body.appendChild(renderer.domElement);
33
34   const geometry = new THREE.BoxGeometry();
35   const material = new THREE.MeshBasicMaterial({color: color});
36   const forme = new THREE.Mesh(geometry, material);
37   scene.add(forme);
38
39   const animate = function () {
40     requestAnimationFrame(animate);
41     forme.rotation.x += xRotation;
42     forme.rotation.y += yRotation;
43     renderer.render(scene, camera);
44   };
45
46   animate();
47 </script>
48 </body>
49 </html>

```

La forme affichée à l'écran est un cube.

## p. 8 Solution n°2

Voici la déclaration des variables permettant d'effectuer les modifications demandées :

```
1 let cubeColor = 0x005500;
2 let yRotation = 0.005;
3 let xRotation = yRotation * 2;
4 let cameraZPosition = 7;
```

Bien évidemment, votre réponse peut être correcte même si vous n'avez pas ces valeurs exactes. Le plus important est d'avoir réussi à trouver une nuance de vert, d'avoir compris comment éloigner la caméra et d'avoir manipulé les valeurs de la rotation.

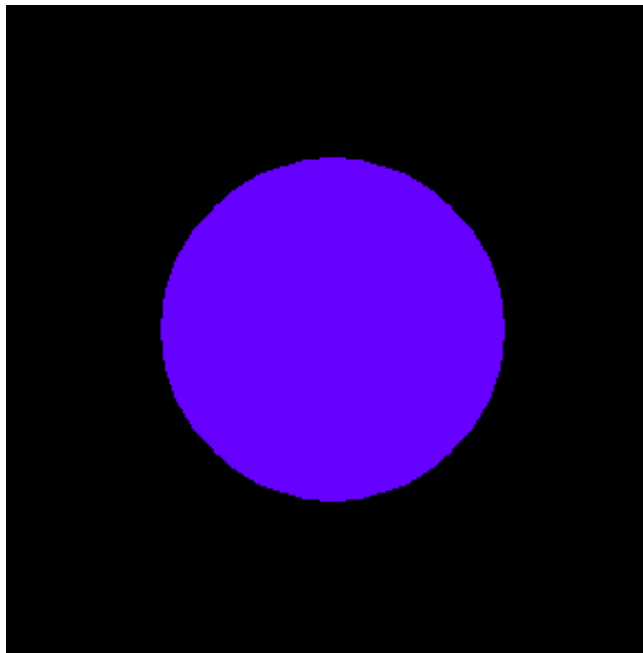
## p. 12 Solution n°3

La scène, la caméra et le *renderer* doivent être créés avant le code permettant de créer la sphère. Attention, l'*aspect ratio* de la caméra doit se baser sur le canevas existant.

Le code doit se terminer par l'appel à la méthode `render` :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 </head>
5 <body>
6 <canvas width="300" height="300" id="canvas3d"></canvas>
7 <script type="module">
8   import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
9
10  let canvasElement = document.getElementById('canvas3d');
11  let scene = new THREE.Scene();
12  let camera = new THREE.PerspectiveCamera(80, canvasElement.width / canvasElement.height,
13    0.1, 1000);
14  camera.position.z = 5;
15  let renderer = new THREE.WebGLRenderer(
16    {
17      canvas: canvasElement
18    }
19  );
20  const geometry = new THREE.SphereGeometry(2, 16, 16);
21  const material = new THREE.MeshBasicMaterial({color: 0x6600FF});
22  const sphere = new THREE.Mesh(geometry, material);
23  scene.add(sphere);
24  renderer.render(scene, camera);
25 </script>
26 </body>
27 </html>
```

Le rendu final est le suivant :



#### p. 18 Solution n°4

Après avoir créé la scène, la caméra et le *renderer*, il faut créer un objet de type *CylinderGeometry* et un autre de type *ConeGeometry*. Le mât doit être le parent de la toile. Pour positionner le cône au-dessus du cylindre, il faut utiliser la méthode *set* de la propriété *position*.

Les deux objets doivent utiliser un *MeshStandardMaterial*, et celui du mât doit avoir une *roughness* strictement inférieure à 1. Pour que les objets ne soient pas entièrement noirs, n'oubliez pas d'ajouter une source de lumière, comme une *PointLight*.

Le script complet permettant de générer l'exemple est :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 </head>
5 <body>
6 <canvas width="200" height="200" id="canvas"></canvas>
7 <script type="module">
8   import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
9
10  // Création de la scène, de la caméra et du renderer
11  let canvasElement = document.getElementById('canvas');
12  let scene = new THREE.Scene();
13  let camera = new THREE.PerspectiveCamera(70, canvasElement.width / canvasElement.height,
14    0.1, 1000);
15  camera.position.z = 4;
16  let renderer = new THREE.WebGLRenderer({
17    {
18      canvas: canvasElement
19    }
20  });
21  // Création du mât
```

```

22 let poleGeometry = new THREE.CylinderGeometry(0.1, 0.1, 2, 8);
23 let poleMaterial = new THREE.MeshStandardMaterial({
24     color: 0xFFFFFF,
25     roughness: 0.3
26 });
27 let pole = new THREE.Mesh(poleGeometry, poleMaterial);
28 scene.add(pole);
29
30 // Création de la toile
31 let hessianGeometry = new THREE.ConeGeometry(1.2, 1, 8);
32 let hessianMaterial = new THREE.MeshStandardMaterial({color: 0x55FF55});
33 let hessian = new THREE.Mesh(hessianGeometry, hessianMaterial);
34 hessian.position.set(0, 1.5, 0);
35 pole.add(hessian);
36
37 // Lumière
38 let light = new THREE.PointLight(0xFFFFFF, 1, 50);
39 light.position.set(0, 0, 3);
40 scene.add(light);
41
42 // Rendu de la scène
43 renderer.setClearColor(0x87CEFA); // Bleu ciel au bord de la plage
44 renderer.render(scene, camera);
45 </script>
46 </body>
47 </html>

```

### p. 23 Solution n°5

La première étape est de définir la position de départ de chaque goutte. Pour cela, il faut créer une fonction prenant en paramètre une goutte et qui règle sa position aléatoirement : chaque coordonnée devra recevoir le résultat de la fonction `getRandomNumber()`, donnée dans l'énoncé. L'axe des y devra cependant être augmenté de 10 unités afin que les gouttes apparaissent hors champ. Cette fonction sera appelée au moment de l'initialisation des gouttes, mais aussi pour remplacer les gouttes qui sont tombées hors champ.

```

1 function setToRandomPositionAboveScreen(drop)
2 {
3     drop.position.set(getRandomNumber(), 10 + getRandomNumber(), getRandomNumber());
4 }

```

Il faut ensuite créer les gouttes. Chaque goutte est un cylindre qui n'est pas sensible à la lumière : il faut donc utiliser la `CylinderGeometry` et le `MeshBasicMaterial` pour créer toutes les Mesh de nos gouttes.

```

1 // Création des gouttes
2 let dropNumber = 500;
3 let coneGeometry = new THREE.CylinderGeometry(0.01, 0.01, 0.5, 6);
4 let coneMaterial = new THREE.MeshBasicMaterial({color: 0xFFFFFF});
5 for (let i = 0; i < dropNumber; i++) {
6     let drop = new THREE.Mesh(coneGeometry, coneMaterial);
7     setToRandomPositionAboveScreen(drop)
8     scene.add(drop);
9 }

```

Maintenant que les gouttes sont créées, il faut les faire tomber. Pour cela, il suffit de parcourir tous les enfants de la scène et de diminuer leur position sur l'axe des y. Si la position est inférieure à 0, il faut remplacer la goutte aléatoirement.



```

1 const animate = function () {
2   for (let i in scene.children) {
3     scene.children[i].translateY(-0.1); // position.y fonctionne aussi
4     if (scene.children[i].position.y < 0) {
5       setToRandomPositionAboveScreen(scene.children[i]);
6     }
7   }
8   // Rendu de la frame
9   render.render(scene, camera);
10
11   // Attente de la prochaine frame
12   requestAnimationFrame(animate);
13 };

```

Une fois la caméra placée et le reste des éléments créé (scène, *render*...), le code final est :

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <style>
5     body {
6       margin: 0;
7     }
8
9     canvas {
10      display: block;
11    }
12  </style>
13 </head>
14 <body>
15 <script type="module">
16   import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
17
18   // Création de la scène, de la caméra et du render
19   let scene = new THREE.Scene();
20   let camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1,
21     1000);
22   camera.position.z = 10;
23   camera.position.y = 5;
24   camera.position.x = 5;
25
26   let renderer = new THREE.WebGLRenderer();
27   renderer.setSize(window.innerWidth, window.innerHeight);
28   document.body.appendChild(renderer.domElement);
29
30   function getRandomNumber()
31   {
32     return Math.random() * Math.floor(10);
33   }
34
35   function setToRandomPositionAboveScreen(drop)
36   {
37     drop.position.set(getRandomNumber(), 10 + getRandomNumber(), getRandomNumber());
38   }
39
40   // Création des gouttes
41   let dropNumber = 500;
42   let dropGeometry = new THREE.CylinderGeometry(0.01, 0.01, 0.5, 6);
43   let dropMaterial = new THREE.MeshBasicMaterial({color: 0xFFFFFF});

```

```

43   for (let i = 0; i < dropNumber; i++) {
44       let drop = new THREE.Mesh(dropGeometry, dropMaterial);
45       setToRandomPositionAboveScreen(drop);
46       scene.add(drop);
47   }
48
49   // Rendu de la scène
50   renderer.setClearColor(0x002962);
51
52   const animate = function () {
53       for (let i in scene.children) {
54           scene.children[i].translateY(-0.1); // position.y fonctionne aussi
55           if (scene.children[i].position.y < 0) {
56               setToRandomPositionAboveScreen(scene.children[i]);
57           }
58       }
59       // Rendu de la frame
60       renderer.render(scene, camera);
61
62       // Attente de la prochaine frame
63       requestAnimationFrame(animate);
64   };
65
66   // On appelle une première fois la fonction pour rentrer dans la boucle
67   animate();
68 </script>
69 </body>
70 </html>


```

## Exercice p. 24 Solution n°6

### Exercice

Quels sont les contextes de rendu utilisables en JavaScript ?


- ☐ DirectX
- ☒ WebGL
- ☐ OpenGL
- ☒ Canvas2D

 En JavaScript, on peut utiliser Canvas2D et WebGL. DirectX et OpenGL sont des contextes de rendu utilisés, entre autres, sur Windows.

### Exercice

Pourquoi utiliser three.js plutôt que le WebGL ?

- ☐ Three.js est plus performant
- ☒ Three.js est plus simple à utiliser
- ☐ Three.js ne sert que d'introduction à la 3D, mais il faudra bien utiliser le WebGL sur les vraies applications

 D'une manière générale, utiliser une librairie permet de simplifier l'utilisation d'un outil. Three.js ne fait pas exception, puisqu'elle permet de manipuler le WebGL très simplement.


**Exercice**

---

À quoi sert le *renderer* ?

- ☐ À placer tous les éléments qui vont être affichés
- ☐ À définir le point de vue à partir duquel les éléments vont être affichés

☒ À générer l'image finale


 Les éléments sont placés sur une scène et le point de vue est défini par une caméra. Le *renderer* permet de générer l'image de la scène, capturée par la caméra.

**Exercice**

---

Qu'est-ce que le *clipping plane* ?

- ☐ Le champ de vision de la caméra
- ☒ La distance d'affichage des objets
- ☐ Le ratio largeur/hauteur de l'image finale


 La caméra possède deux propriétés permettant de régler l'affichage des objets : `nearClippingPlane` et `farClippingPlane`. Seuls les objets placés entre ces deux plans seront affichés : les autres seront considérés trop proches ou trop éloignés.

**Exercice**

---

Quelle géométrie permet de créer un cube dans `three.js` ?

- ☒ `BoxGeometry`
- ☐ `CubeGeometry`
- ☐ `RectGeometry`


 Un cube est simplement une boîte dont les côtés sont de taille égale : il faut donc utiliser `BoxGeometry` pour les créer.

**Exercice**

---

Quel est l'élément qui représente l'objet 3D présent sur la scène ?


- ☐ `Geometry`
- ☐ `Material`
- ☐ `Renderer`
- ☒ `Mesh`

 La `Geometry` ne fait que définir un ensemble de points, et le `Material` précise comment afficher les éléments, mais c'est bien le `Mesh`, l'union de ces deux éléments, qui est ajouté sur la scène.

**Exercice**


---

Qu'est-ce que la *render loop* ?

- ☐ La boucle qui permet d'ajouter tous les éléments dans la scène
- ☒ La boucle qui permet de générer toutes les images d'une animation
- ☐ La boucle qui permet d'importer tous les éléments de la librairie
-  Une animation est composée de plusieurs images, appelées *frames*. Chaque *frame* est générée par la *render loop*.


### Exercice

Que fait la fonction `requestAnimationFrame` ?

- ☒ Elle appelle une fonction de *callback* au moment où la prochaine *frame* doit être générée
- ☐ Elle génère la *frame* courante
- ☐ Elle récupère toutes les *frames* pour en faire une animation
-  La fonction `requestAnimationFrame` permet d'attendre jusqu'à la génération de la prochaine *frame*. Cela permet de créer des animations fluides en gardant un intervalle régulier entre les images.


### Exercice

Comment déplacer un objet sur l'axe des *x* de la scène ?

- ☐ La méthode `rotateX`
- ☐ La méthode `translateX`
- ☒ La propriété `position.x`
-  La propriété `position.x` permet de déplacer un objet sur l'axe des *x* de la scène. Il ne faut pas la confondre avec la méthode `translateX`, qui permet de déplacer un objet sur l'axe des *x* local à l'objet, qui dépend donc de la rotation de l'objet à déplacer.

### Exercice

Comment récupérer tous les objets présents sur la scène ?

- ☐ La méthode `getChildren`
- ☐ La méthode `get`
- ☒ La propriété `children`
-  La propriété `children` permet de récupérer tous les éléments présents sur la scène.

## p. 26 Solution n°7

Voici le code nécessaire pour reproduire l'exemple :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <style>
5     body {
6       margin: 0;
7     }
8
9     canvas {
```

```

10         display: block;
11     }
12 </style>
13 </head>
14 <body>
15
16 <script type="module">
17     import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
18
19     // Création de la scène, de la caméra et du renderer
20     let scene = new THREE.Scene();
21     let camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1,
22 1000);
23     camera.position.z = 6;
24     camera.position.y = 2;
25     camera.position.x = 5;
26
27     let renderer = new THREE.WebGLRenderer();
28     renderer.setSize(window.innerWidth, window.innerHeight);
29     document.body.appendChild(renderer.domElement);
30
31     // Création du vaisseau
32     let sphereGeometry = new THREE.SphereGeometry(1, 32, 32); // La sphère
33     let ringGeometry = new THREE.RingGeometry(1, 2, 30); // L'anneau
34     let material = new THREE.MeshStandardMaterial({color: 0xAAAAAA, roughness: 0.3});
35     // Les deux éléments ont le même Material
36     let sphere = new THREE.Mesh(sphereGeometry, material);
37     let ring = new THREE.Mesh(ringGeometry, material);
38     ring.rotateX(270 * Math.PI / 180); // Conversion degrés/radians
39     sphere.add(ring);
40     scene.add(sphere);
41
42     // Le vaisseau réagit à la lumière, il faut donc en placer une
43     let light = new THREE.PointLight(0xFFFFFF, 0.5, 100);
44     light.position.set(10, 10, 10);
45     scene.add(light);
46
47     // Rendu de la scène
48     renderer.setClearColor(0x000918);
49     renderer.render(scene, camera);
50 </script>
51 </body>
52 </html>

```

### p. 26 Solution n°8

Pour clarifier le code, il est préférable de créer une fonction qui va s'occuper de bouger le vaisseau :

```

1 function moveShip(input, ship)
2 {
3     if (input.getCurrentInput() === 'LEFT') {
4         ship.translateX(-0.1);
5     } else
6     if (input.getCurrentInput() === 'RIGHT') {
7         ship.translateX(0.1);
8     }
9 }

```

Il suffit ensuite d'instancier la classe `UFOInput`, puis de créer une *render loop* pour y appeler la fonction `moveShip` avant le rendu :

```
1 let ufoInput = new UFOInput();
2
3 const animate = function () {
4     moveShip(ufoInput, sphere);
5
6     // Rendu de la frame
7     renderer.render(scene, camera);
8
9     // Attente de la prochaine frame
10    requestAnimationFrame(animate);
11 };
12
13 // Ne pas oublier d'appeler la fonction de rendu une première fois
14 animate();
```

### p. 27 Solution n°9

La première étape est de créer une fonction qui va placer une étoile à son point de départ aléatoire, mais reculé de 10 sur l'axe des z :

```
1 function setStarToRandomPosition(star)
2 {
3     star.position.set(getRandomNumber(), getRandomNumber(), getRandomNumber() - 10);
4 }
```

Il faut ensuite créer les étoiles et les stocker dans un tableau afin d'y avoir accès plus tard :

```
1 // Création des étoiles
2 let starNumber = 500;
3 let starGeometry = new THREE.CylinderGeometry(0.01, 0.01, 0.5, 6);
4 let starMaterial = new THREE.MeshBasicMaterial({color: 0xFFFFFF});
5 let stars = [];
6
7 for (let i = 0; i < starNumber; i++) {
8     let star = new THREE.Mesh(starGeometry, starMaterial);
9     star.rotateX(90 * Math.PI / 180);
10    setStarToRandomPosition(star);
11    stars.push(star);
12    scene.add(star);
13 }
```

Enfin, il faut créer la fonction qui va faire avancer toutes les étoiles, et vérifier leur position. Si leur position sur l'axe des z dépasse la limite passée en paramètre, alors on les replace. Cette limite sera la coordonnée z de la caméra.

```
1 function moveStars(stars, limit)
2 {
3     for (let i in stars) {
4         stars[i].position.z += 0.15;
5         if (stars[i].position.z > limit) {
6             setStarToRandomPosition(stars[i]);
7         }
8     }
9 }
```

Il ne reste plus qu'à appeler cette fonction dans la *render loop*, en lui donnant les coordonnées de la caméra en limite d'affichage :

```
1 const animate = function () {
2   moveShip(ufoInput, sphere);
3   moveStars(stars, camera.position.z);
4
5   // Rendu de la frame
6   renderer.render(scene, camera);
7
8   // Attente de la prochaine frame
9   requestAnimationFrame(animate);
10 };
```

#### p. 28 Solution n°10

Comme pour les étoiles, il faut une fonction permettant de placer l'astéroïde. Cette fonction doit prendre en compte la position du vaisseau :

```
1 function setMeteorToRandomPosition(meteor, shipYcoord)
2 {
3   meteor.position.set(getRandomNumber(), shipYcoord, -40);
4 }
```

On crée le météore en lui attribuant une position aléatoire :

```
1 // Création du météore
2 let meteorGeometry = new THREE.DodecahedronGeometry(2);
3 let meteorMaterial = new THREE.MeshStandardMaterial({color: 0x8B4513});
4 let meteor = new THREE.Mesh(meteorGeometry, meteorMaterial);
5 setMeteorToRandomPosition(meteor, sphere.position.y);
6 scene.add(meteor);
```

On crée ensuite la méthode lui permettant de se déplacer, avec un effet de rotation :

```
1 function moveMeteor(meteor, limit, shipYcoord)
2 {
3   meteor.position.z += 0.15;
4   meteor.rotateZ(0.01);
5   if (meteor.position.z > limit) {
6     setMeteorToRandomPosition(meteor, shipYcoord);
7   }
8 }
```

Enfin, on appelle la fonction dans la *render loop* :

```
1 const animate = function () {
2   moveShip(ufoInput, sphere);
3   moveStars(stars, camera.position.z);
4   moveMeteor(meteor, camera.position.z, sphere.position.y);
5
6   // Rendu de la frame
7   renderer.render(scene, camera);
8
9   // Attente de la prochaine frame
10  requestAnimationFrame(animate);
11 };
```

Le code complet et final du jeu est le suivant (en ayant placé la classe `UFOInput` dans un fichier `UFOInput.js`) :

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <style>
5     body {
6       margin: 0;
7     }
8
9     canvas {
10      display: block;
11    }
12  </style>
13 </head>
14 <body>
15 <script src="UFOInput.js"></script>
16 <script type="module">
17   import * as THREE from 'https://unpkg.com/three@0.122.0/build/three.module.js';
18
19   // Création de la scène, de la caméra et du renderer
20   let scene = new THREE.Scene();
21   let camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight, 0.1,
1000);
22   camera.position.z = 6;
23   camera.position.y = 2;
24   camera.position.x = 5;
25
26   let renderer = new THREE.WebGLRenderer();
27   renderer.setSize(window.innerWidth, window.innerHeight);
28   document.body.appendChild(renderer.domElement);
29
30   // Création du vaisseau
31   let sphereGeometry = new THREE.SphereGeometry(1, 32, 32); // La sphère
32   let ringGeometry = new THREE.RingGeometry(1, 2, 30); // L'anneau
33   let material = new THREE.MeshStandardMaterial({color: 0xAAAAAA, roughness: 0.3});
34   // Les deux éléments ont le même Material
35   let sphere = new THREE.Mesh(sphereGeometry, material);
36   let ring = new THREE.Mesh(ringGeometry, material);
37   ring.rotateX(270 * Math.PI / 180);
38   sphere.add(ring);
39   scene.add(sphere);
40
41   let light = new THREE.PointLight(0xFFFFFF, 0.5, 100);
42   light.position.set(10, 10, 10);
43   scene.add(light);
44
45   // Rendu de la scène
46   renderer.setClearColor(0x000918);
47
48
49   function moveShip(input, ship)
50   {
51     if (input.getCurrentInput() === 'LEFT') {
52       ship.translateX(-0.1);
53     } else
54     if (input.getCurrentInput() === 'RIGHT') {
55       ship.translateX(0.1);
56     }
57   }

```



```

57     }
58
59     function getRandomNumber()
60     {
61         return Math.random() * Math.floor(10);
62     }
63
64     function setStarToRandomPosition(star)
65     {
66         star.position.set(getRandomNumber(), getRandomNumber(), getRandomNumber() - 10);
67     }
68
69     // Création des étoiles
70     let starNumber = 500;
71     let starGeometry = new THREE.CylinderGeometry(0.01, 0.01, 0.5, 6);
72     let starMaterial = new THREE.MeshBasicMaterial({color: 0xFFFFFF});
73     let stars = [];
74
75     for (let i = 0; i < starNumber; i++) {
76         let star = new THREE.Mesh(starGeometry, starMaterial);
77         star.rotateX(90 * Math.PI / 180);
78         setStarToRandomPosition(star);
79         stars.push(star);
80         scene.add(star);
81     }
82
83     function moveStars(stars, limit)
84     {
85         for (let i in stars) {
86             stars[i].position.z += 0.15;
87             if (stars[i].position.z > limit) {
88                 setStarToRandomPosition(stars[i]);
89             }
90         }
91     }
92
93     function setMeteorToRandomPosition(meteor, shipYcoord)
94     {
95         meteor.position.set(getRandomNumber(), shipYcoord, -40);
96     }
97
98     // Création du météore
99     let meteorGeometry = new THREE.DodecahedronGeometry(2);
100    let meteorMaterial = new THREE.MeshStandardMaterial({color: 0x8B4513});
101    let meteor = new THREE.Mesh(meteorGeometry, meteorMaterial);
102    setMeteorToRandomPosition(meteor, sphere.position.y);
103    scene.add(meteor);
104
105    function moveMeteor(meteor, limit, shipYcoord)
106    {
107        meteor.position.z += 0.15;
108        meteor.rotateZ(0.01);
109        if (meteor.position.z > limit) {
110            setMeteorToRandomPosition(meteor, shipYcoord);
111        }
112    }
113
114    let ufoInput = new UFOInput();

```

```
115
116     const animate = function () {
117         moveShip(ufoInput, sphere);
118         moveStars(stars, camera.position.z);
119         moveMeteor(meteor, camera.position.z, sphere.position.y);
120
121         // Rendu de la frame
122         renderer.render(scene, camera);
123
124         // Attente de la prochaine frame
125         requestAnimationFrame(animate);
126     };
127
128     // Ne pas oublier d'appeler la fonction de rendu une première fois
129     animate();
130 </script>
131 </body>
132 </html>
```

Bien entendu, le jeu n'est clairement pas fini : il manque la gestion des collisions, des sons (bien qu'il n'y ait pas de sons dans l'espace), un menu et une musique épique, mais c'est un bon début !