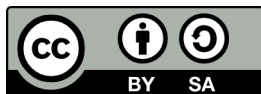


# Faire de la 3D dans un navigateur web avec three.js

v140714  
Brouillon

Ce texte est sous licence Creative Commons Attribution - Partage dans les Mêmes Conditions 4.0 International. Pour accéder à une copie de cette licence, merci de vous rendre à l'adresse suivante <http://creativecommons.org/licenses/by-sa/4.0/> ou envoyez un courrier à Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.



Le titulaire des droits autorise toute utilisation de l'œuvre originale (y compris à des fins commerciales) ainsi que la création d'œuvres dérivées, à condition qu'elles soient distribuées sous une licence identique à celle qui régit l'œuvre originale.

# Activité 1

## Introduction et installation

Three.js (<http://threejs.org/>) est un moteur 3D (recherchez ce qu'est un moteur 3D) basé sur WebGL (recherchez le terme WebGL sur internet). Il permet de construire et d'animer des scènes en 3D directement dans un navigateur web. Il a été créé par mrdoob.

Three.js n'est pas un logiciel (à la différence d'Unity3D par exemple), c'est une bibliothèque JavaScript. Vous allez devoir écrire du code dans un éditeur de texte.

Comme indiqué plus haut, three.js va vous permettre d'afficher des scènes en 3D dans un navigateur web, qui dit navigateur web, dit 3 éléments :

- du HTML
- du CSS
- du JavaScript

Le HTML sera très simple puisqu'il se résumera à une balise <div> (balise qui va permettre d'afficher le contenu 3D dans le navigateur). Le code CSS sera aussi très simple (et même non nécessaire !). Tout se jouera donc au niveau du JavaScript.

### ***À faire vous-même***

Dans votre espace de travail, créez un dossier nommé "app\_00". Placez-vous dans ce dossier "app\_00" et créez un répertoire dénommé "lib".

Créez à l'aide d'un éditeur de texte (par exemple Scite si vous êtes sous GNU/Linux) les 3 fichiers suivants (et les placer dans le répertoire "app\_00") :

#### index.html

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <title>Activités three.js</title>
  <link rel="stylesheet" href="css/style.css">
  <script src="lib/three.min.js"></script>
</head>
<body>
</body>
<script src="script.js"></script>
</html>
```

#### style.css

```
html, body, canvas {
  width: 100%;
  height: 100%;
  padding: 0;
  margin: 0;
  overflow: hidden;
}
```

#### script.js

```
// Ce fichier est pour l'instant vide
```

Voici ce que vous devriez obtenir :



lib



index.html



script.js

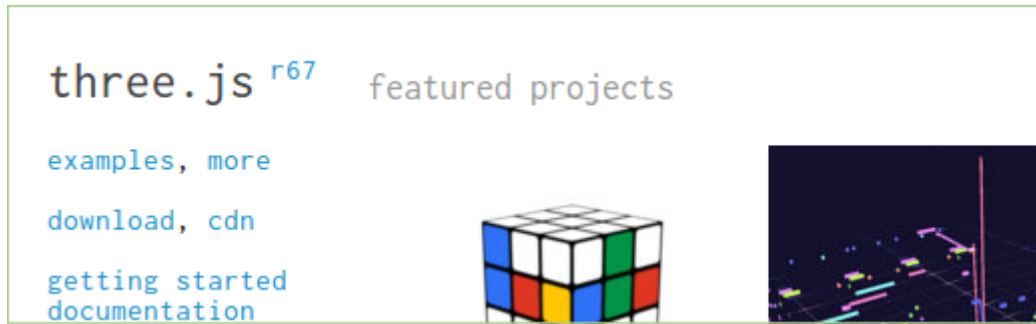


style.css

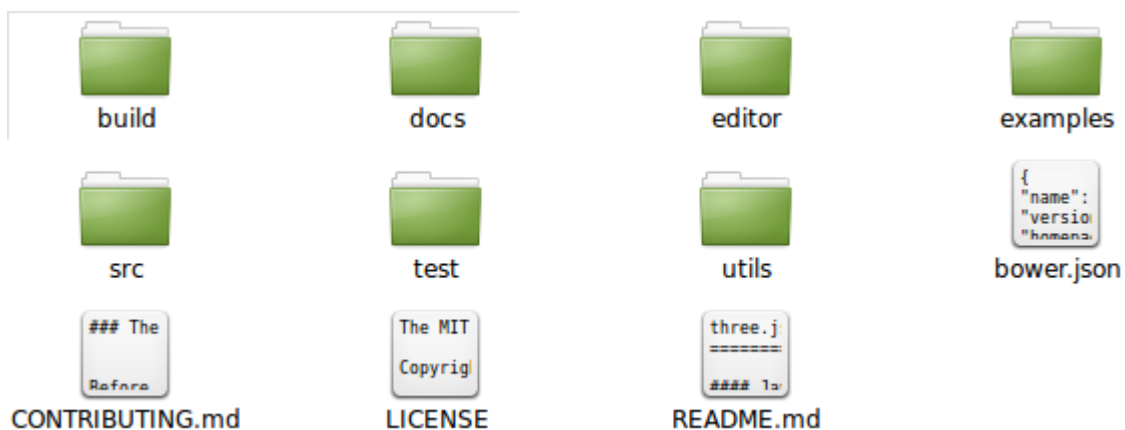
Rien de spécial dans ce code HTML, nous utilisons une feuille de style et nous "chargeons" la bibliothèque three.js (`<script src="lib/three.min.js"></script>`). Mais avant de pouvoir utiliser cette bibliothèque, il faut la télécharger.

### À faire vous-même

Sur le site <http://threejs.org/>, cliquez (dans le menu à gauche) sur download



Une archive zip d'une centaine de mega-octet devrait se télécharger. "Dézipper" cette archive "mrdoob-three.js-r67-1-gd3cb4e7.zip" (la fin du nom de l'archive devrait être différente pour vous, car elle correspond au numéro de version de la bibliothèque) et placez le dossier obtenu dans votre répertoire de travail. Placez-vous dans ce dossier (le dossier devrait avoir un nom qui correspond au nom de l'archive zip avec le ".zip" en moins). Voici ce que vous devriez obtenir :



Placez-vous dans le dossier build.



Contenu du dossier build

Récupérez (copier) le fichier "three.min.js" et placez-le dans le répertoire lib que vous avez créé précédemment.

Tout est prêt !

Dans les activités suivantes, pour créer une nouvelle application, il vous suffira d'effectuer un copier-coller du dossier "app\_00" et de renommer le dossier résultant de ce copier-coller en "app\_01", "app\_02", "app\_03", ...



## Activité 2

### Première scène

Nous allons dans cette activité créer notre première scène.

#### À faire vous-même

Créez (en faisant un copier-coller du dossier "app\_00") un nouveau dossier "app\_01", ouvrez, à l'aide d'un éditeur de texte le fichier script.js et saisissez le code ci-dessous dans ce fichier.

script.js

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.innerHeight, 0.1, 1000);
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild(renderer.domElement);
function render() {
    requestAnimationFrame(render);
    renderer.render(scene, camera);
}
render();
```

Décortiquons ce code ligne par ligne :

#### ligne 1

Nous commençons par créer un objet "scene" (`var scene = new THREE.Scene();`). Ici, rien de spécial à dire, l'objet "scene" est un objet de base de three.js, aucun programme utilisant three.js ne pourra se dispenser de la création de cet objet "scene". Notez que j'ai employé le terme d'objet, car nous avons bien affaire à un objet au sens programmation orientée objet. Il y a une petite différence par rapport aux objets que nous avons déjà rencontrés dans l'activité 14 du document "Apprendre à programmer...", en effet, ici, nous utilisons un constructeur (mot clé "new"). Nous n'avons pas besoin de nous attarder sur cette notion de constructeur, cela ne nous sera pas utile. Pour le reste, nous aurons, comme dans l'activité 14, un objet qui possédera des attributs et des méthodes (si nécessaire, n'hésitez pas à vous replonger dans cette activité 14).

#### ligne 2

Afin d'observer la scène que nous venons de créer à la ligne 1, nous devons créer une caméra (objet "camera"). Le constructeur de l'objet caméra (`THREE.PerspectiveCamera`) prend des paramètres (c'est ces paramètres qui vont, en quelque sorte, nous permettre de "régler" la caméra), voici leur signification :

- 75 => ce premier paramètre correspond au champ de vision de la caméra, plus ce nombre est grand, plus le champ de vision de la caméra est grand.
- "window.innerWidth/window.innerHeight" => le second paramètre correspond au ratio largeur-hauteur de l'image. Pour en savoir plus sur cette notion de ratio, voici l'article Wikipedia consacré à ce sujet : [http://fr.wikipedia.org/wiki/Format\\_d'image](http://fr.wikipedia.org/wiki/Format_d'image). Dans l'exemple qui nous intéresse ici, "window.innerWidth" correspond à la largeur de la fenêtre qui accueillera la scène et "window.innerHeight" correspond à sa hauteur. Sauf cas particulier, vous pourrez laisser ce paramètre tel quel.
- "0.1" et "1000" => les 3<sup>e</sup> et 4<sup>e</sup> paramètres correspondent respectivement, à la distance minimum et à la distance maximum de vision de la caméra. Ici aussi, tout du moins dans un premier temps, il sera inutile de modifier ces 2 paramètres.

#### ligne 3

Nous créons l'objet "renderer". Cet objet sera utilisé un peu plus loin dans le code afin de permettre le rendu (l'affichage) de la scène. Notez que c'est la bibliothèque WebGL ("`WebGLRenderer`") qui assurera ce rendu.

#### ligne 4

La méthode "setSize" de l'objet "renderer" permet de définir la taille de la fenêtre qui affichera la scène. Ici nous indiquons que la fenêtre de rendu devra avoir la même taille que la fenêtre qui affichera la scène. Si vous n'avez pas bien compris, ce n'est pas grave, ici aussi, dans la plupart des cas, vous n'aurez pas à modifier cette ligne.

#### ligne 5

`"document.body.appendChild(renderer.domElement);"` cette ligne permet d'intégrer la fenêtre de rendu dans la page HTML. Encore une fois, aucune raison de modifier cette ligne.

#### lignes 6, 7 et 8

`"function render()"` Nous avons ici la fonction "render", cette fonction est très importante, voici pourquoi :

Quand vous jouez à un jeu sur votre ordinateur (et que votre ordinateur manque de "puissance"), il arrive parfois que l'affichage saccade (on parle de "lag"), pourquoi ?

Il faut savoir que "l'ordinateur" doit, plusieurs dizaines de fois par seconde (le nombre d'images affichées par seconde est souvent désigné par l'acronyme FPS (Frames per second)), afficher une nouvelle image à l'écran.

Cela demande beaucoup de calculs (complexes) au microprocesseur central (CPU). Petite parenthèse, c'est d'ailleurs pour cela qu'aujourd'hui cette tâche est très souvent laissée à un microprocesseur spécialisé dans ce genre de calcul : le GPU (Graphics Processing Unit, ce microprocesseur spécialisé se trouve sur la carte graphique de votre ordinateur).

Quand ni le CPU, ni le GPU n'arrivent à afficher suffisamment d'images par seconde, votre jeu saccade.

En matière de programmation, il faut, une fois que la nouvelle image est prête à être affichée (après par exemple avoir bougé de quelques pixels le personnage principal), envoyer l'ordre au CPU d'afficher cette nouvelle image (après avoir fait tous les calculs nécessaires).

Dans three.js, cet "ordre" est envoyé grâce à la ligne `"renderer.render(scene, camera);"` (Appel de la méthode "render" de l'objet "renderer". Cette méthode prend 2 paramètres : la scène à rendre et la caméra à utiliser pour rendre cette scène).

Mais, vu que cet appel doit être effectué plusieurs fois par seconde (à chaque rendu d'image), il doit donc se trouver dans une boucle. Cette boucle est souvent appelée "boucle de jeu".

Comment fonctionne cette boucle de jeu dans three.js ?

La ligne `"requestAnimationFrame(render);"` permet d'appeler à intervalle de temps régulier la fonction se trouvant en paramètre, c'est à dire ici, la fonction "render".

La fonction "render" sera donc exécutée à intervalle de temps régulier (c'est donc cette fonction "render" qui jouera le rôle de "boucle de jeu") et que trouve-t-on dans cette fonction ?

Comme nous venons de le voir la ligne `"requestAnimationFrame(render);"` mais aussi la ligne `"renderer.render(scene, camera);"` qui permet de rendre la scène. Le rendu de la scène s'effectuera donc à intervalle de temps régulier (environ 60 fois par seconde si vous voulez avoir un jeu fluide).

#### ligne 9

Permet d'appeler la fonction "render" la première fois "manuellement". En effet, les appels suivants seront gérés par le "requestAnimationFrame" comme nous venons de le voir ci-dessus.

### ***À faire vous-même***

Testez "app\_01" en faisant un clic droit sur le fichier index.html et en choisissant "Ouvrir avec" Firefox ou Chrome (mais surtout pas Internet Explorer si vous êtes sur windows).

Comme vous pouvez le constater, rien ne s'affiche, à part un écran noir (si vous n'avez pas d'écran noir, c'est qu'il y a sans doute un problème). C'est normal, notre scène est vide, nous devons maintenant "peupler" notre scène.

## Activité 3

### Ajouter des objets à la scène

Dans cette activité nous allons donc "peupler" notre scène avec des objets. Three.js propose, par défaut, un grand nombre d'objets, mais il vous sera possible d'importer vos propres objets.

#### *À faire vous-même*

Créez un nouvel exemple (app\_02), toujours en faisant un copier-coller du répertoire "app\_00" (dans la suite je ne préciserai plus que vous devez faire un copier-coller du répertoire "app\_00").

Saisissez le code suivant dans le fichier script.js

#### script.js

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.innerHeight, 0.1, 1000);
camera.position.z=5;
var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
var sphGeo= new THREE.SphereGeometry(1, 32, 32);
var sphMat=new THREE.MeshBasicMaterial({color: 0xff0000});
var sph = new THREE.Mesh(sphGeo, sphMat);
scene.add(sph);
function render() {
    requestAnimationFrame(render);
    renderer.render(scene, camera);
}
render();
```

Testez ce code en faisant un clic droit sur le fichier index.html (comme dans l'activité précédente). Vous devriez voir apparaître une sphère rouge (enfin, pour l'instant cela ressemble plutôt à un cercle), si ce n'est pas le cas, vérifiez que vous n'avez pas fait d'erreur en saisissant le code.

Analysons ce code ligne par ligne :

#### lignes 1 et 2

rien de nouveau

#### ligne 3

"camera.position.z=5; " => nous modifions la coordonnée z de la caméra. Cela demande sans doute quelques explications :

Qui dit scène 3D, dit 3 coordonnées (x,y,z) pour les points. Vous n'avez sans doute pas l'habitude de "raisonner" dans l'espace (en 3D) et cela peut-être quelque peu déroutants au départ. Donc, pas "d'inquiétude si vous éprouvez des difficultés. L'origine de notre repère (point de coordonnées (0,0,0)) se trouve au centre de la scène. Par défaut, tout objet (caméra comprise) est positionné à l'origine du repère. En écrivant "camera.position.z=5;", nous plaçons notre caméra au point de coordonnées (0,0,5) (puisque nous modifions uniquement la coordonnée z).

#### lignes 4, 5 et 6

rien de nouveau

#### lignes 7, 8 et 9

Ces 3 lignes sont consacrées à la création de notre objet "sphère". Dans three.js, un objet est la combinaison de 2 choses :

- une "géométrie" qui donne la forme de l'objet, ici nous définissons donc un objet de type "sphère" (THREE.SphereGeometry), le constructeur (utilisation de "new") prend 3 paramètres : le premier paramètre correspond au rayon de la sphère (ici "1"), les 2<sup>e</sup> et 3<sup>e</sup> paramètres permettent de jouer sur la qualité du rendu de la sphère, plus ces 2 valeurs sont élevées, plus la sphère est "belle" (évidemment, plus vous augmentez la qualité, plus vous sollicitez le GPU).
- Un matériau, nous aurons l'occasion de revenir sur cette notion dans une prochaine activité, pour l'instant, nous utiliserons le matériau de base "MeshBasicMaterial". Le constructeur prend en paramètre un objet



JavaScript "{color: 0xff0000}", cet objet possède un attribut "color" qui permet de définir la couleur du matériau et donc la couleur de l'objet. La couleur "0xff0000" (rouge) est donnée au format hexadécimal, pour en savoir plus : [http://fr.wikipedia.org/wiki/Liste\\_de\\_couleurs](http://fr.wikipedia.org/wiki/Liste_de_couleurs).

Une fois la géométrie et le matériau définis, il nous reste à créer l'objet lui-même à l'aide du constructeur

"THREE.Mesh": "var sph = new THREE.Mesh(sphGeo, sphMat);". Vous aurez compris par vous-même que le 1<sup>er</sup> paramètre correspond à la géométrie de l'objet (ici une sphère), le 2<sup>de</sup> au matériau utilisé pour "fabriquer" notre sphère.

#### ligne 10

L'objet sphère a été créé, il faut maintenant l'ajouter à la scène : "scene.add(sph);"

Le reste ne diffère pas de l'exemple précédent.

#### ***À faire vous-même***

Créez un nouvel exemple (app\_03). Reprenez la structure de "app\_02" mais remplacez la sphère par une boîte (parallélépipède rectangle).

Pour vous aider : pour créer une géométrie "boîte" il faut utiliser le constructeur "THREE.BoxGeometry", ce constructeur prend 3 paramètres : la longueur, la largeur et la hauteur de la boîte.

La boîte devra être verte.

#### ***À faire vous-même***

Créez un nouvel exemple (app\_04). Vous devez afficher un tore de couleur bleue à la place de la boîte verte. Cette fois, pas d'aide directe, je vous donne la page de la documentation qui explique l'utilisation du constructeur

"THREE.TorusGeometry" : <http://threejs.org/docs/#Reference/Extras/Geometries/TorusGeometry> (petite information tout de même, ne vous préoccupez pas de trop des 3 derniers paramètres du constructeur).

## Activité 4

### Utiliser les axes du repère

Comme déjà dit dans l'activité précédente, vous allez devoir gérer un système de coordonnées en 3 dimensions, ce qui n'est pas toujours simple : où est l'axe x ? Où est l'axe y ? Et l'axe z ?  
Three.js nous propose d'afficher à l'écran les axes x, y et z.

#### À faire vous-même

Créez un nouvel exemple (app\_05) et saisissez le code suivant dans le fichier script.js

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75,window.innerWidth/window.innerHeight, 0.1, 1000);
camera.position.z = 5;
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild(renderer.domElement);
var axisHelper = new THREE.AxisHelper(5);
scene.add(axisHelper);
function render() {
    requestAnimationFrame(render);
    renderer.render(scene, camera);
}
render();
```

Testez cet exemple.

Au niveau, du code rien de compliqué, nous utilisons le constructeur "THREE.AxisHelper" pour créer le système d'axes. Ce constructeur prend un paramètre : la longueur de l'axe.

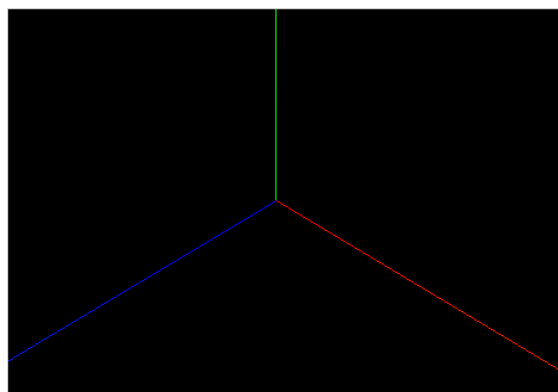
Vous devriez voir apparaître une ligne rouge et une ligne verte. La ligne rouge représente l'axe x et la ligne verte représente l'axe y. Mais où est l'axe z (représenté par une ligne bleue) ? Il est tout à fait normal de ne pas le voir, puisque la caméra est positionnée sur l'axe z "camera.position.z = 5; "

#### À faire vous-même

Dans un nouvel exemple (app\_06) utilisez "camera.position.x " et "camera.position.y " afin de déplacer la caméra. Le but étant de voir à l'écran l'axe z.

Très important : afin que la caméra vise le centre de la scène, vous allez devoir rajouter la ligne suivante à votre code : "camera.lookAt(scene.position);". Dans les exemples suivants, sauf indication contraire, la caméra devra toujours pointer vers le centre de la scène. Nous aurons l'occasion de revenir sur la méthode "lookAt" plus tard.

Par exemple, voici ce que vous devriez obtenir avec "camera.position.x = 5;","camera.position.y = 5;" et "camera.position.z = 5;".



Les axes sont des outils de débogage, une fois le débogage terminé, il ne faudra pas oublier de les supprimer.

## Activité 5

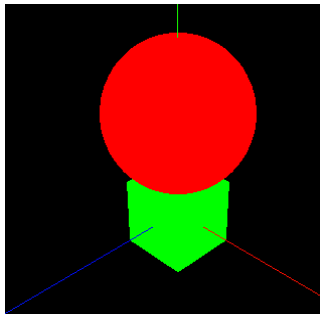
### Positions et rotations

#### À faire vous-même

Créez un nouvel exemple (app\_07). Placez la caméra à la position de coordonnées (5,5,5). Créez un cube vert (de côté 1) et placez-le à l'origine du repère. Dans la même scène, ajoutez une sphère rouge de rayon 1. La sphère devra se trouver juste au-dessus du cube. Vous pourrez utiliser le système d'axe pour vous aider.

Pour vous aider : comme pour la caméra, vous pouvez placer un objet dans la scène à l'aide des attributs "position.x", "position.y" et "position.z".

Vous devriez obtenir quelque chose qui ressemble à cela :



Petite astuce : sachez qu'il est possible de regrouper toutes les coordonnées sur une seule ligne avec la méthode "set".

Soit un objet "monObjet" au lieu d'écrire :

```
monObjet.position.x = 10 ;
```

```
monObjet.position.y = 20 ;
```

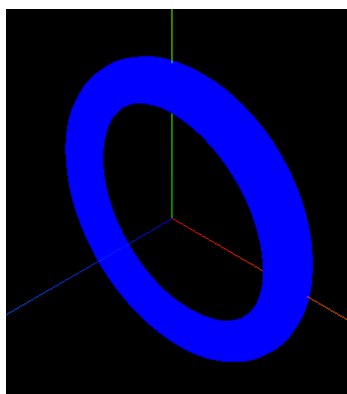
```
monObjet.position.z = 5 ;
```

vous pouvez écrire : `monObjet.position.set(10,20,5) ;`

La méthode "set" fonctionne aussi pour la caméra.

#### À faire vous-même

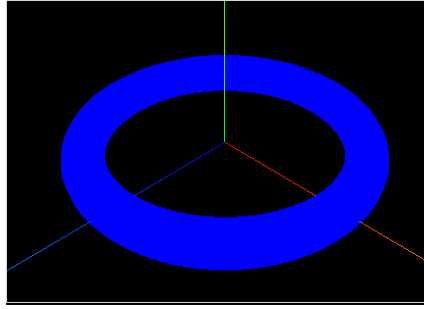
app\_08 : Toujours avec une caméra se trouvant au point de coordonnées (5,5,5), placez dans la scène un tore à l'origine du repère. Vous devriez obtenir quelque chose qui ressemble à ceci :



Il est possible de faire faire une rotation à un objet, par exemple, un `objet.rotation.x = Math.PI/2` entraînera une rotation de 90° de l'objet "objet" autour de l'axe x. Un `objet.rotation.z = Math.PI` entraînera une rotation de 180° de l'objet "objet" autour de l'axe z. Comme vous avez dû le remarquer, l'angle doit être en radian et surtout pas en degré (petit rappel :  $\pi$  radian = 180°). Pour obtenir directement la valeur de  $\pi$  en JavaScript, vous pouvez utiliser `Math.PI`.

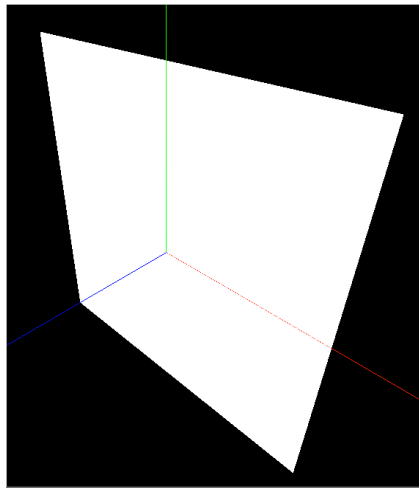
### À faire vous-même

Modifiez "app\_08" afin d'obtenir ceci :



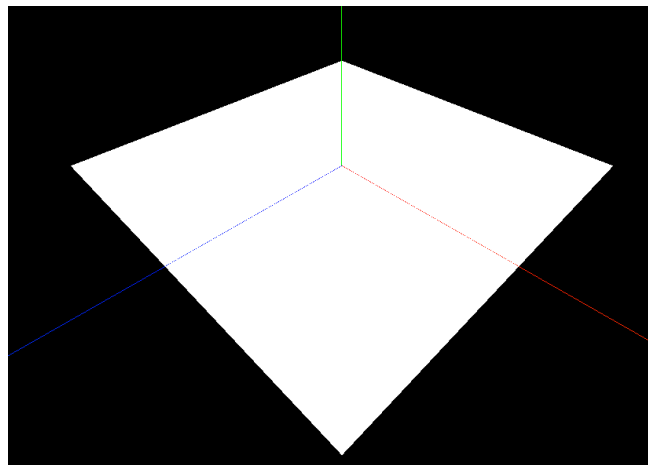
### À faire vous-même

app\_09 : Three.js propose de créer des plans (surfaces planes) à l'aide du constructeur "`THREE.PlaneGeometry`" (voir <http://threejs.org/docs/#Reference/Extras/Geometries/PlaneGeometry>), ce constructeur prend 2 paramètres : la largeur du plan et la hauteur de plan. Créez une scène composée d'un plan blanc carré de côté égal à 7. La caméra aura pour coordonnées (5,5,5). Vous devriez obtenir ceci :



### À faire vous-même

Modifiez app\_09 afin d'obtenir ceci :



Attention au signe de votre angle de rotation, car le "dessous" d'un plan est invisible.

Ce plan va jouer le rôle de "sol" dans les futurs exemples.

## Activité 6

### Les différents types de matériaux

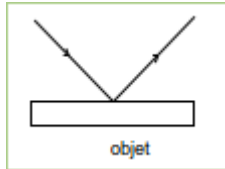
Pourquoi arrivez-vous à distinguer un bout de bois et un morceau de métal du premier coup d'œil ?

Parce qu'ils n'ont pas la même couleur bien sûr, mais surtout parce qu'ils ne renvoient pas la lumière de la même façon.

Un peu de "théorie" : quand vous éclairez un objet (non transparent), cet objet va :

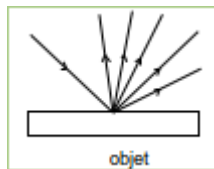
- diffuser la lumière
- réfléchir la lumière

réflexion de la lumière :



Un rayon lumineux qui arrive sur la surface d'un objet provoque la réémission d'un rayon dans une seule direction.

diffusion de la lumière :



Un rayon lumineux qui arrive sur la surface d'un objet provoque la réémission de rayons lumineux dans toutes les directions.

Dans de nombreux cas, les deux phénomènes cohabitent. Mais, par exemple, dans le cas du bois, la réflexion est presque totalement absente.

En revanche pour le métal, la réflexion domine souvent : plus la surface est lisse, plus elle est importante (une surface métallique parfaitement lisse donnera.....un miroir). Dans le cas de la réflexion, on utilise l'expression "spéculaire".

Three.js propose des matériaux "tout fait" mais il est aussi possible de créer ses propres matériaux. La gestion des matériaux est une "science" très complexe (pour vous en convaincre, recherchez sur internet ce qu'est un shader), nous ne ferons ici qu'effleurer le sujet : nous utiliserons donc que des matériaux "tout fait". Et même pour ces matériaux "tout fait", nous n'évoquerons que certaines de leurs propriétés.

Parmi les matériaux proposés par three.js, nous allons en étudier 2 : "MeshLambertMaterial" et

"MeshPhongMaterial" (pour utiliser un de ces 2 matériaux, il suffit de remplacer le constructeur

"MeshBasicMaterial" par "MeshLambertMaterial" ou "MeshPhongMaterial", exemple :

```
"var sphMat=new THREE.MeshPhongMaterial({color: 0x00dddd}) ;").
```

"MeshLambertMaterial" : Ce matériau est à utiliser que vous désirez un objet d'aspect mat (contraire de brillant), le constructeur prend en paramètre un objet JavaScript ("var sphMat=new THREE.MeshLambertMaterial({}) ;").

Voici une liste (non exhaustive) des attributs possibles de cet objet JavaScript :

- color : donne la couleur du matériau (exemple : color: 0x00dddd)
- ambient : Cette notion est assez difficile à expliquer, mais pour simplifier, on pourra dire que cet attribut définit la couleur d'un objet quand il se trouve à l'ombre, par défaut cette couleur est blanche (ambient : 0xffffffff).
- emissive : définit la couleur émise par un matériau (attention cela ne veut pas dire que votre objet est "source primaire" de lumière, c'est juste que cette couleur n'est pas modifiée par les sources de lumière extérieures), par défaut cette couleur est noire (emissive : 0x000000).
- transparent : définit si un objet est transparent (transparent : true) ou opaque (transparent : false), par défaut l'objet est opaque.
- opacity : si l'objet est transparent, opacity définit le degré de transparence. La valeur est comprise entre 0 (complètement transparent) et 1 (complètement opaque).

"MeshPhongMaterial" : Ce matériau est à utiliser que vous désirez un objet d'aspect brillant, le constructeur prend en paramètre un objet JavaScript (`"var sphMat=new THREE.MeshPhongMaterial({}) ;"`). Voici une liste (non exhaustive) des attributs possibles de cet objet JavaScript :

- color : même chose que ci-dessus
- ambient : même chose que ci-dessus
- emissive : même chose que ci-dessus
- transparent : même chose que ci-dessus
- opacity : même chose que ci-dessus
- specular : définit la couleur de la lumière émise par réflexion spéculaire ! Pour être un peu plus concret et un peu plus pratique, si cette couleur est identique à la couleur définie pour l'attribut "color", vous aurez alors un aspect plutôt métallique pour le matériau. Si la couleur définie pour cet attribut "specular" tire plutôt sur le gris, votre matériau aura un aspect "plastique".
- shininess : cet attribut définit la "brillance" de votre matériau, plus la valeur sera grande plus votre matériau paraîtra brillant (la valeur par défaut est de 30).

Toutes ces notions sont très difficiles à maîtriser, seuls des essais successifs et le tâtonnement vont permettre d'approcher le résultat escompté. À part pour les experts du domaine, c'est l'empirisme qui règne en maître sur ces notions.

Pour entrer dans les détails, une fois de plus, consultez la documentation officielle :

<http://threejs.org/docs/#Reference/Materials/MeshPhongMaterial>

### ***À faire vous-même***

app\_10 : Créez une scène composée d'une sphère (rayon 2 et coordonnées (0,0,0)) et d'une caméra (coordonnée (5,5,5)). Votre sphère devra être composée d'un matériau de type "MeshPhongMaterial".

Comme vous devez le constater, la sphère est complètement noire. Ceci est dû à l'absence de source lumineuse. En effet le matériau "MeshBasicMaterial" n'avait pas besoin de source lumineuse pour être visible, ce n'est pas le cas pour les matériaux "MeshPhongMaterial" et "MeshLambertMaterial".

## Activité 7

### Les éclairages

Three.js propose plusieurs types d'éclairage, nous allons en voir 3 :

La "lumière ambiante" qui permet d'éclairer tous les objets d'une scène de façon uniforme. Pour définir une lumière ambiante, il faut utiliser le constructeur "AmbientLight". Ce constructeur prend un paramètre, la couleur de la lumière. Exemple d'utilisation:

```
var lightAmb = new THREE.AmbientLight (0xffffff) ;
```

Le "point lumineux" est une source de lumière ponctuelle. Cette source émet de la lumière à partir d'un point dans toutes les directions de l'espace. Pour créer un "point lumineux" vous devez utiliser le constructeur "PointLight". Ce constructeur prend 3 paramètres : la couleur de la lumière, l'intensité de la lumière, la distance au-delà de laquelle l'intensité de la lumière devient nulle. Exemple :

```
var lightPoi = new THREE.PointLight (0xffffff, 1, 100) ;
```

Pour placer votre point lumineux dans votre scène, il faudra utiliser la méthode set (exemple :  
"lightPoi.position.set (30,40,50) ;")

Le "spot" est comme son nom l'indique un spot. Le constructeur ("SpotLight") peut prendre jusqu'à 5 paramètres, mais nous n'en verrons ici que 4 :

- la couleur de la lumière
- l'intensité de la lumière
- la distance au-delà de laquelle l'intensité de la lumière devient nulle
- l'angle d'ouverture du spot (par défaut cet angle est de  $\pi/3$  et il est au maximum de  $\pi/2$ )

Exemple : "var lightSpot = new THREE.SpotLight (0xffffff, 1, 100, Math.PI/4) ;"

Ensuite, vous devez placer votre spot à l'aide de la méthode set ("lightSpot.position.set (30,40,50) ;").

Les objets de type "SpotLight" possède plusieurs attributs, en voici un :

target => permet de définir la "cible" du spot . Par défaut le spot est dirigé vers le centre de la scène. Si par exemple vous voulez éclairer un objet que vous avez nommé "sph", il suffira d'écrire "lightSpot.target=sph".

Pour en savoir plus sur les différents attributs des objets de type "SpotLight", consulter la documentation officielle :

<http://threejs.org/docs/#Reference/Lights/SpotLight>

À ne pas oublier : vos objets de type "lumière" devront comme, tous les objets, être ajoutés à la scène en utilisant la méthode "add" (exemple : "scene.add(lightSpot) ;").

#### **À faire vous-même**

app\_11 : Créez une scène composée d'une sphère (rayon 2 et coordonnées (0,5,0)) et d'une caméra (coordonnée (5,5,5)). Votre sphère devra être composée d'un matériau de type "MeshPhongMaterial". Ajoutez 3 éclairages à la scène : 1 "AmbientLight", 1 "PointLight" et 1 "SpotLight". Vous choisirez, comme bon vous semble, les paramètres de ces éclairages (couleurs, positions, intensités...). Seule contrainte, le "SpotLight" devra éclairer la sphère.

#### **À faire vous-même**

Reprenez l'exemple précédent (app\_11) et modifiez les différents paramètres (position de la sphère, positions des sources lumineuses, type de matériau utilisé pour la sphère, les différentes couleurs...).

## Activité 8

### Animer la scène

Si vous désirez animer les objets qui se trouvent dans une scène, il ne faut jamais oublier qu'une animation est une succession très rapide d'image fixe (comme pour un dessin animé). Pour "animer" un objet, il suffit de modifier un des paramètres de l'objet à chaque image. Comment modifier un paramètre à chaque image ? En plaçant cette modification dans la fonction "render()".

#### *À faire vous-même*

Analysez, saisissez puis testez l'exemple suivant (app\_12) :

##### script.js

```
var vAngX=0.02;
var vAngY=0.03;
var scene = new THREE.Scene();
var camera = new
THREE.PerspectiveCamera(75,window.innerWidth/window.innerHeight,0.1,1000);
camera.position.set(0,0,300)
camera.lookAt(scene.position);
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild(renderer.domElement);
var axisHelper = new THREE.AxisHelper(70);
scene.add(axisHelper);
var boxGeo= new THREE.BoxGeometry(50,50,50);
var boxMat=new THREE.MeshLambertMaterial({color: 0xff00ff});
var box = new THREE.Mesh(boxGeo,boxMat);
scene.add(box);
var pointLight = new THREE.PointLight( 0xffffffff);
pointLight.position.set(60,60,60);
scene.add(pointLight);
var spotLight = new THREE.SpotLight( 0xffffffff);
spotLight.position.set(-300,0,0);
scene.add(spotLight);
function render() {
    requestAnimationFrame(render);
    box.rotation.y=box.rotation.y+vAngY
    box.rotation.x=box.rotation.x+vAngX
    renderer.render(scene, camera);
}
render();
```

#### **Introduire la notion de temps**

Dans les jeux vidéos (ou dans les simulations), il est souvent indispensable d'introduire la notion de temps. Tout se passe comme si nous déclenchions un chronomètre au début de l'exécution du programme, pour cela il suffit d'introduire une variable "temps" et d'incrémenter cette variable temps à chaque image.

Si l'on part du principe que nous avons un FPS de 30 images par seconde, il faut donc incrémenter notre variable temps de 1/30 de seconde à chaque image.

#### *À faire vous même*

Saisissez, analysez et testez l'exemple (app\_13) suivant :

##### script.js

```
var temps=0;
var vAng=1;
var scene = new THREE.Scene();
var camera = new
THREE.PerspectiveCamera(75,window.innerWidth/window.innerHeight,0.1,1000);
camera.position.set(0,0,300)
camera.lookAt(scene.position);
var renderer = new THREE.WebGLRenderer();
```



```

renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild(renderer.domElement);
var axisHelper = new THREE.AxisHelper(70);
scene.add(axisHelper);
var boxGeo= new THREE.BoxGeometry(50,50,50);
var boxMat=new THREE.MeshLambertMaterial({color: 0xff00ff});
var box = new THREE.Mesh(boxGeo,boxMat);
scene.add(box);
var pointLight = new THREE.PointLight( 0xffffffff);
pointLight.position.set( 60, 60, 60 );
scene.add(pointLight);
var spotLight = new THREE.SpotLight( 0xffffffff);
spotLight.position.set(-300,0,0);
scene.add(spotLight);
function render() {
    requestAnimationFrame(render);
    temps=temps+1/30
    box.rotation.y=vAng*temps;
    renderer.render(scene, camera);
}
render();

```

L'introduction d'une variable temps va nous permettre d'utiliser ce que l'on appelle en physique des équations horaires. Par exemple, pour la rotation d'un objet, on trouve l'équation horaire suivante :  $\alpha = \omega.t$  (avec  $\alpha$  l'angle,  $\omega$  la vitesse angulaire en radian par seconde et  $t$  le temps). Si vous êtes observateur, cela devrait vous rappeler quelque chose.

### Gros problème de temps

Nous sommes parties du principe que le nombre d'images par seconde sera de 30, or, rien n'est moins sûr, si par exemple votre scène se complexifie, le FPS risque de chuter : quelle en serait alors la conséquence pour le "chronomètre interne" de notre programme ?

Le temps doit s'écouler toujours "à la même vitesse" quel que soit le nombre d'images par seconde. Nous voici confrontés à un véritable problème : la méthode employée dans l'exemple 13 n'est donc pas satisfaisante.

Heureusement, three.js fournit un système de chronomètre "clé en main" :

Dans un premier, il faut créer un objet de type clock : `var chrono=new THREE.Clock();`

Ensuite, vous pouvez déclencher le chrono grâce à l'instruction suivante : `chrono.start();`

Afin, la méthode "getElapsedTime()" de l'objet "chrono" vous renverra le temps écoulé (en seconde) depuis le déclenchement du chronomètre (exemple : `"temps=chrono.getElapsedTime()"`).

### À faire vous même

app\_14 : En repartant de l'exemple précédent (app\_13), modifiez le code afin de profiter du chronomètre proposé par three.js.

### À faire vous même

app\_15 : vous allez écrire un programme permettant de simuler l'orbite (circulaire) d'une planète autour d'une étoile. Tous les éléments visuels (texture(s), lumière(s), position de la caméra) sont laissés à votre libre choix.

Pour vous aider :

Soit le point A de coordonnées  $x$  et  $y$

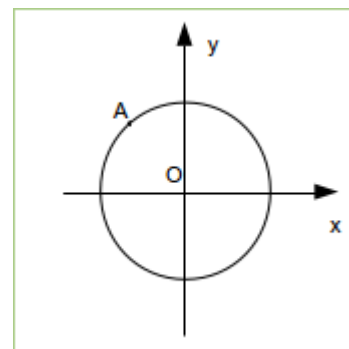
À décrira un cercle ayant pour rayon  $r$  et pour centre O si à tout instant :

$$x = r \cdot \sin(\omega.t)$$

$$y = r \cdot \cos(\omega.t)$$

avec  $\omega$  la vitesse angulaire (vitesse de rotation en radian par seconde)

et  $t$  la variable temps.



Et si notre étoile ne se trouve pas au point de coordonnées (0,0,0), les équations de la trajectoire de la planète deviennent plus complexes ?

Oui, un peu, mais, à la limite, ce n'est pas vraiment un problème, car threejs nous propose une solution très simple à mettre en œuvre.

En effet, il est possible de modifier l'origine du repère pour un objet donnée :

Quand vous écrivez `scene.add(planete) ;`, non seulement vous ajoutez l'objet "planete" à la scène, mais vous définissez aussi le système de coordonnées qui sera utilisé par l'objet que vous venez d'ajouter. Dans ce cas précis (utilisation de "scene"), l'origine du système de coordonnées se trouvera au centre de notre scène. Imaginons que nous avons déjà ajouté un objet "etoile" (à des coordonnées autres que (0,0,0)). Si nous ajoutons maintenant l'objet "planete" avec un `etoile.add(planete) ;` à la place d'un `scene.add(planete) ;`, pour la planète, l'origine de son repère ne sera pas le centre de la scène, mais le centre de l'objet "etoile".

### ***À faire vous même***

app\_16 : en reprenant l'exemple précédent (app\_15), placez l'étoile, non plus au centre de la scène, mais aux coordonnées (150,0,0) (positionnez la caméra en conséquence).

### ***À faire vous même***

app\_17 : reprenez l'exemple précédent (app\_15). Votre système sera maintenant constitué d'une étoile, d'une planète, qui orbitera autour de l'étoile et d'un satellite (ex : la Lune) qui orbitera autour de la planète.

## Activité 9

### Gestion des événements (clavier)

Dans cette activité nous allons nous intéresser à l'interaction "utilisateur-machine". Pour assurer cette interaction, JavaScript met à notre disposition les "listeners". Ces "listeners", vont "écouter" (ou plutôt surveiller) les périphériques d'entrées (clavier et souris par exemple, nous verrons la souris un peu plus tard).

Pour mettre en place un listener, nous utiliserons la méthode "addEventListener" de l'objet "window" (l'objet "window" est l'objet de base en JavaScript, tous les autres objets descendent de l'objet "window" ). La méthode "addEventListener" prend 2 paramètres :

- l'événement à surveiller
- une fonction de callback

Qu'est-ce qu'une fonction de callback ?

Une fonction de callback est une fonction qui sera exécutée seulement après un « événement » donné, pas avant. Souvent, les fonctions de callback sont des fonctions anonymes. La fonction de callback passée en paramètre de la méthode "addEventListener" sera exécutée seulement quand l'événement surveillé par le listener sera déclenché (appui sur une touche du clavier, clic de souris....)

Voici quelques événements qu'il est possible de surveiller grâce au listener :

"keydown" : appui (sans relâcher) sur une touche

"keyup" : relâcher une touche

"keypress" : appuyer puis relâcher sur une touche

Voici un exemple d'utilisation de "addEventListener" :

```
window.addEventListener ("click", function(){
    ****fonction de callback ici****
});
```

#### ***À faire vous même***

Écrire un exemple (app\_18) qui, grâce à la méthode "alert" (fenêtre surgissante), affichera à l'écran "Hello World !" en cas de "simple clic" sur le bouton gauche de la souris.

La fonction de callback peut prendre un paramètre que l'on nomme souvent "event". Ce paramètre est un objet qui contient des informations sur l'événement qui vient d'être déclenché.

Par exemple, il est possible, en cas d'utilisation des événements "keydown" et "keyup" de connaître le code de la touche du clavier qui a été actionnée par l'utilisateur avec "event.keyCode" (attention, visiblement cela ne fonctionne pas avec l'événement "keypress") :

```
window.addEventListener ("keydown", function(event){
    alert (event.keyCode)
});
```

Pour connaître les valeurs des codes des touches, consultez cette page (voir le paragraphe "3.3. Key CodeValues") :

<http://unixpapa.com/js/key.html>

#### ***À faire vous même***

Écrire un exemple (app\_19) qui affichera (méthode "alert") : "touche A" si vous appuyez sur la touche A et "autre touche que la touche A" si vous appuyez sur une autre touche que la touche A.

#### ***À faire vous même***

app\_20 : saisissez, analysez et testez le code suivant :

script.js

```
var chrono=new THREE.Clock();
var temps;
var vAng=3;
var keys={left:0,right:0}
```

```

window.addEventListener('keydown',function(event){
    if (event.keyCode==37){
        keys.left=1;
    }
    if (event.keyCode==39){
        keys.right=1;
    }
});
window.addEventListener('keyup',function(event){
    if (event.keyCode==37){
        keys.left=0;
    }
    if (event.keyCode==39){
        keys.right=0;
    }
});
var scene = new THREE.Scene();
var camera = new
THREE.PerspectiveCamera(75,window.innerWidth/window.innerHeight,0.1,1000);
camera.position.set(0,0,300)
camera.lookAt(scene.position);
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild(renderer.domElement);
var axisHelper = new THREE.AxisHelper(70);
scene.add(axisHelper);
var boxGeo= new THREE.BoxGeometry(50,50,50);
var boxMat=new THREE.MeshLambertMaterial({color: 0xff00ff});
var box = new THREE.Mesh(boxGeo,boxMat);
scene.add(box);
var pointLight = new THREE.PointLight( 0xffffffff);
pointLight.position.set( 60, 60, 60 );
scene.add(pointLight);
var spotLight = new THREE.SpotLight( 0xffffffff);
spotLight.position.set(-300,0,0);
scene.add(spotLight);
chrono.start();
animation=function(){
    deltatemps=chrono.getDelta();
    if (keys.left==1){
        box.rotation.y=box.rotation.y-vAng*deltatemps;
    }
    if (keys.right==1){
        box.rotation.y=box.rotation.y+vAng*deltatemps;
    }
}
function render() {
    requestAnimationFrame(render);
    animation();
    renderer.render(scene, camera);
}
render();

```

Pour vous aider dans votre analyse :

- Nous créons un objet "keys" (2 attributs "left" et "right"). Si l'utilisateur appuie (sans la relâcher) sur la flèche gauche, l'attribut keys.left prend la valeur 1. Si l'utilisateur relâche cette touche, nous avons alors keys.left=0.
- Tout ce qui concerne l'animation du cube a été placé dans une fonction "animation". Cette fonction "animation" étant appelée depuis la fonction "render", la fonction "animation" est donc appelée à chaque image. L'idée est de rendre le code plus clair en évitant de surcharger la fonction "render".
- Nous avons remplacé "chrono.getElapsedTime()" par "chrono.getDelta()". La méthode "getDelta()" donne le temps écoulé depuis le dernier appel à cette même méthode "getDelta()". Donc, vu la position dans le programme (dans la fonction "animation"), "chrono.getDelta()" donne le temps écoulé entre 2 images successives.

Pourquoi utiliser `chrono.getDelta()` à la place de `chrono.getElapsedTime()` ?

Pourquoi avoir écrit `box.rotation.y=box.rotation.y+vAng*deltatemps;` au lieu de `box.rotation.y=vAng*temps;` ?

### ***À faire vous même***

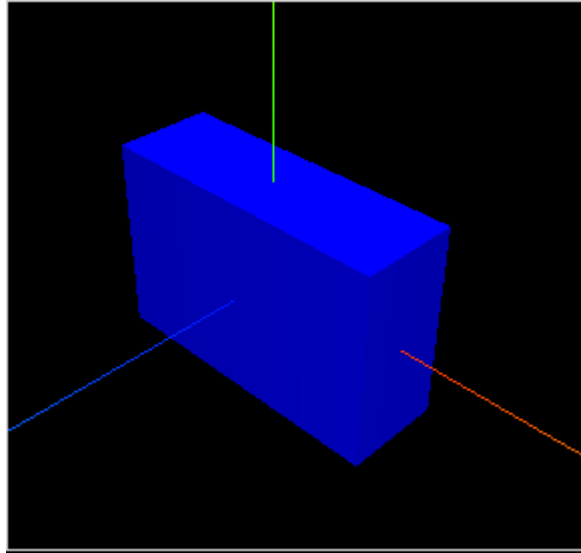
app\_21 : écrire le programme suivant : si l'utilisateur appuie sur la touche "Entrée" (keyCode 13), le cube devra se mettre en rotation. Un nouvel appui sur la touche "Entrée" arrêtera la rotation.

## Activité 10

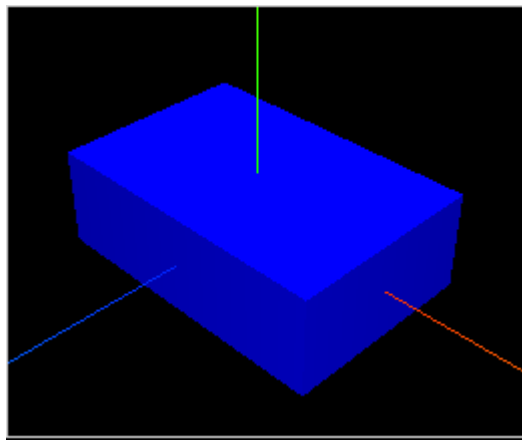
### repère local et repère global

Pour l'instant, nous n'avons pas eu à faire de rotations successives, prenons l'exemple suivant :

Voici un objet de type box (dimensions (3,2,1)) :

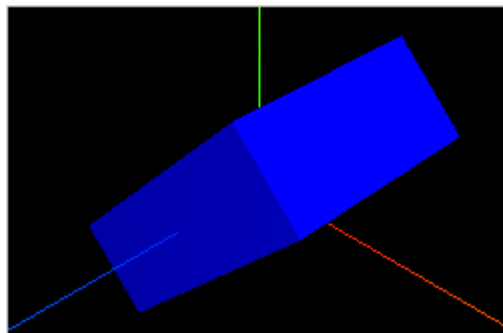


Faisons faire une première rotation à notre objet : `"box.rotation.x=Math.PI/2;"`, voici le résultat,



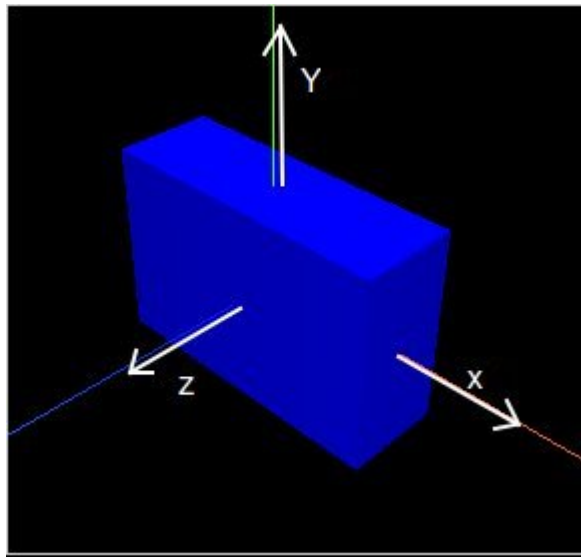
et il correspond à ce qui était attendu.

2<sup>e</sup> rotation : `"box.rotation.y=Math.PI/4;"` (en conservant bien sûr la première), le résultat :

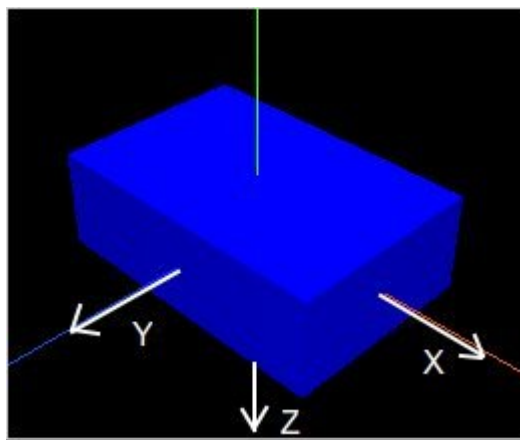


Ici, il y a un problème, cela donne l'impression que la deuxième rotation c'est faite autour de l'axe z et pas autour de l'axe y ! Mais pourquoi ?

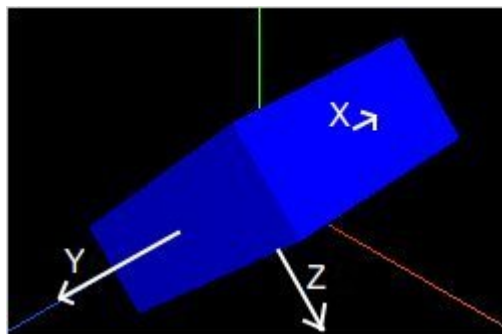
Chaque objet 3D possède son propre système d'axes (que nous appellerons  $(X',Y',Z')$ ), au départ (tant qu'aucune rotation n'a été effectuée), les axes propres à l'objet (on parlera de repère local) sont alignés avec les axes liés à la scène (on parlera de repère global).



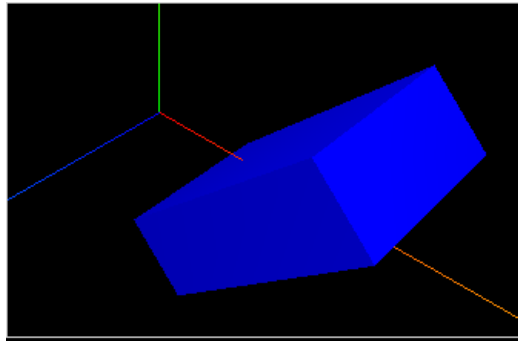
En cas de rotation de l'objet, les axes du repère global ne "bougent" pas alors que les axes liés à l'objet subissent la rotation.



Sachant que la rotation proposée par threejs (exemple: `box.rotation.y=Math.PI/4;`) concerne le repère local de l'objet qui subit la rotation et pas le repère global, nous obtenons donc :



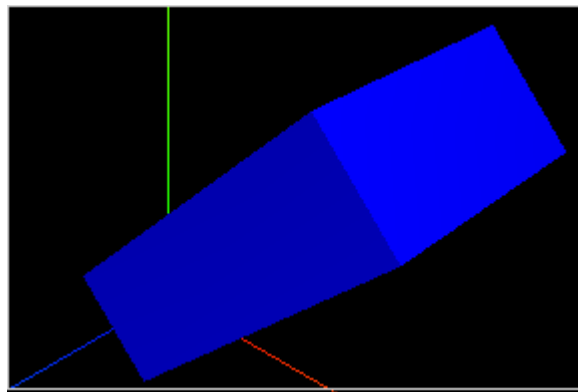
En résumé, en cas de rotation (avec `box.rotation`) il faut raisonner dans le repère local, en revanche, si vous déplacez l'objet avec par exemple un `box.position.x=2;`, le déplacement se fait par rapport au repère global.



Et si l'on désire effectuer un déplacement par rapport au repère local ?

Il faudra utiliser la méthode `translateX` (ou `translateY` ou encore `translateZ`). Cette méthode prend en paramètres la distance que vous voulez faire parcourir à votre objet.

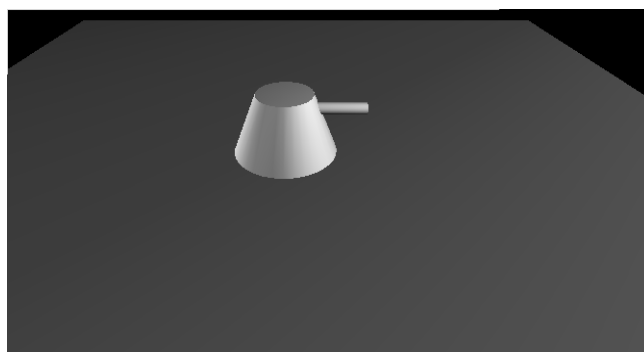
Par exemple, en remplaçant `box.position.x=2;` par `box.translateX(2);`, vous obtiendrez :



### ***À faire vous même***

app\_22 : Voici les différents "éléments" que votre scène devra incorporer :

- un "tank" (voir la capture d'écran ci-dessous)
- un sol (1 plan)



Les flèches "gauche" et "droite" permettront au "tank" d'effectuer une rotation sur lui même. La flèche "haut" permettra au tank d'avancer (toujours avec le canon vers l'avant), la flèche "bas" permettra au tank de reculer.

Pour vous aider : le "tank" est composé de 2 cylindres (pour créer des cylindres, voir la documentation officielle : <http://threejs.org/docs/#Reference/Extras.Geometries/CylinderGeometry>).



### ***À faire vous même***

app\_23 : Reprenez l'app\_22, mais remplacez le "tank" par un simple cube. Le système de déplacement du cube devra être identique à celui du "tank". En plus, en cas d'appui sur la touche espace, le cube devra "sauter" de façon réaliste :

- lors de la phase de montée : la vitesse devra diminuer, lors de la phase de descente : la vitesse devra augmenter
- le saut ne sera possible que si le cube est au sol au moment de l'appui sur la touche espace
- aucune modification de la trajectoire (à l'aide des flèches) ne sera possible durant le saut

Voici quelques conseils pour vous aider :

- créez une variable "timeJump" qui donnera le temps écoulé depuis le début du saut (depuis que l'utilisateur a appuyé sur la touche espace). Cette variable devra être remise à zéro au début de chaque saut.
- Créez une variable "isGround", qui sera true si le "cube" est au sol et false si le "cube" est "en l'air".
- Voici l'équation qui donne l'altitude (h) du cube durant le saut :  
$$h = -A \times \text{timeJump} \times \text{timeJump} + B \times \text{timeJump} + C$$
  
A et B étant des constantes qu'il faudra ajuster en fonction du résultat que vous voulez obtenir, C correspond à la longueur de l'arête du cube divisée par 2.

# Activité 11

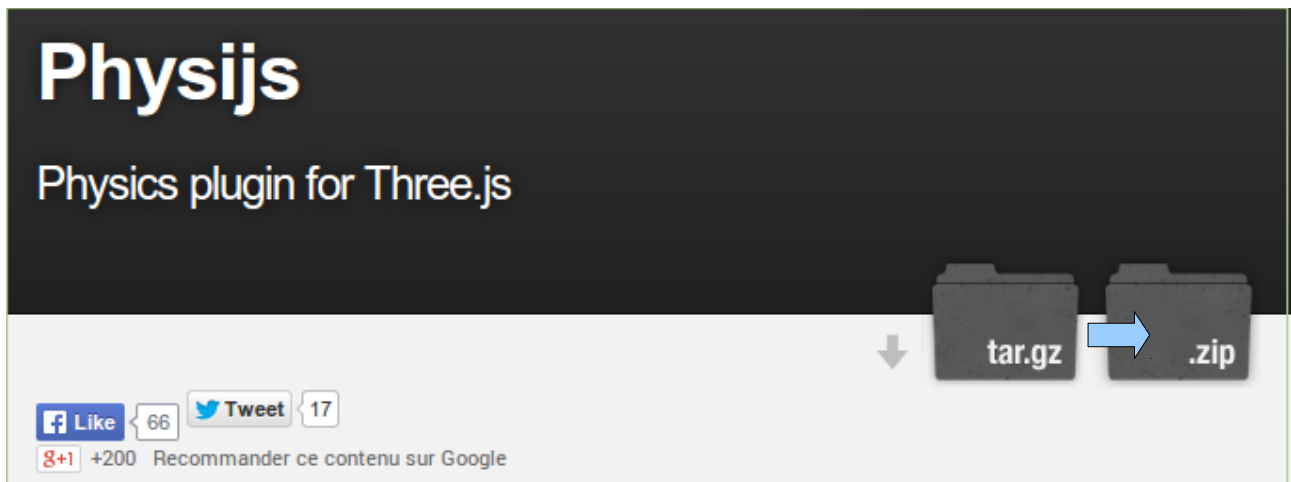
## Un peu de physique (préalable)

Chandler Prall a créé une extension qui permet de gérer la gravitation, les collisions et bien d'autres choses (ce que l'on appelle généralement la "physique" dans les jeux vidéo). Cette extension se nomme Physijs :

<http://chandlerprall.github.io/Physijs/>

### À faire vous même

app\_24 : Téléchargez l'archive sur le site de l'auteur (<http://chandlerprall.github.io/Physijs/>) en cliquant sur ".zip"



Une fois l'archive téléchargée, dézippez le contenu, ouvrez ensuite le dossier obtenu, vous devriez obtenir les fichiers et dossier suivants :



Copier-coller les fichiers "physi.js" et "physijs\_worker.js" dans le dossier "lib" de l'app\_24.

Allez dans le dossier "examples" :

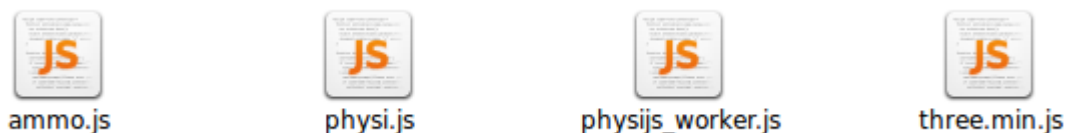


Ouvrez le dossier "js"



Copier-coller le fichier "ammo.js" toujours dans votre dossier "lib"

Voici le contenu du dossier "lib" de l'app\_23 :



## À faire vous même

Il est nécessaire, pour utiliser physijs, de modifier le fichier "index.html" (toujours de l'app\_24) :

index.html

```
<!doctype html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <title>Activités three.js</title>
    <link rel="stylesheet" href="style.css">
    <script src="lib/three.min.js"></script>
    <script src="lib/physi.js"></script>
</head>
<body>
</body>
<script src="script.js"></script>
</html>
```

Nous avons uniquement ajouté une ligne : "<script src="lib/physi.js"></script>"

Physijs utilise, pour cause de recherche de bonnes performances, les "web workers" (pour les curieux, voici un excellent article de David Rousset sur la question : <http://blogs.msdn.com/b/davrous/archive/2011/07/08/introduction-aux-web-workers-d-html5-le-multithreading-version-javascript.aspx>).

Cette utilisation des web workers n'est pas sans conséquence sur notre façon de tester nos programmes. Jusqu'à présent, nous devions juste cliquer sur le fichier "index.html" et le tour était joué.

Pour des raisons de sécurité, que je ne détaillerai pas ici, cette méthode ne fonctionnera plus avec les web workers et donc avec physijs.

Différentes méthodes vous sont proposées dans cet article : <https://github.com/mrdoob/three.js/wiki/How-to-run-things-locally>

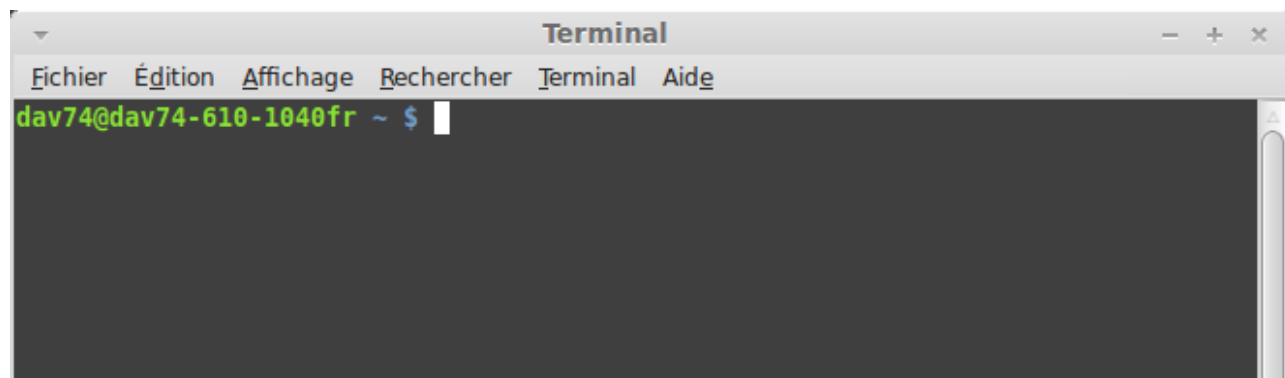
Voici une description de la méthode qui a ma préférence (mais vous pouvez utiliser les autres).

ATTENTION : Visiblement tout ce que suit est inutile si vous utilisez Firefox. Avec le navigateur de chez Mozilla, vous pouvez continuer à directement cliquer sur le fichier index.html (si cela ne fonctionne pas, suivez alors la procédure décrite ci-dessous).

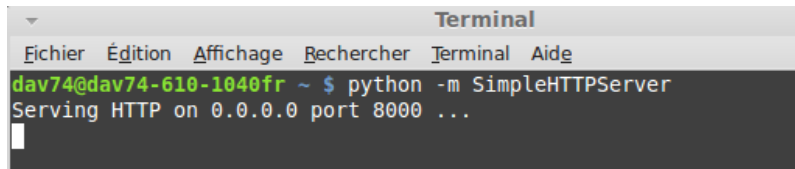
## À faire vous même

L'installation de Python est indispensable, si Python n'est pas installé, installez-le.

Ouvrez ensuite une console (un terminal), voici ce que cela donne sous Linux :

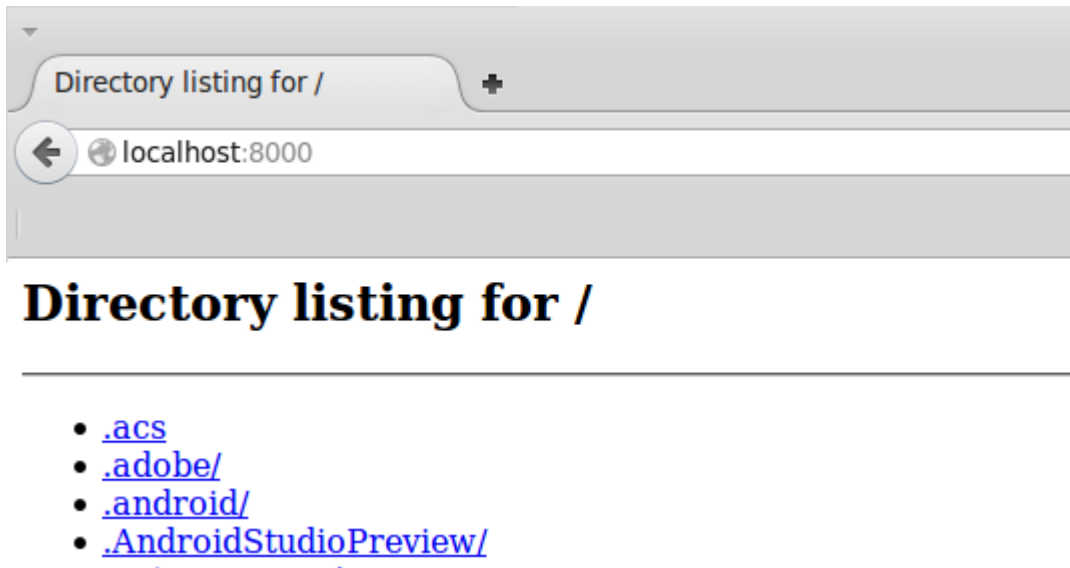


Dans la console, tapez "python -m SimpleHTTPServer" si vous utilisez Python 2.X ou "python -m http.server" si vous utilisez Python 3.X



```
Terminal
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
dav74@dav74-610-1040fr ~ $ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

Sans fermer la console, ouvrez votre navigateur favori et tapez dans la barre d'adresse : "localhost:8000"



Cliquez sur les différents dossiers de l'arborescence (demandez de l'aide si vous avez des difficultés) afin de trouver votre dossier de travail (celui qui contient tous vos exemples).

- [app\\_18/](#)
- [app\\_19/](#)
- [app\\_20/](#)
- [app\\_21/](#)
- [app\\_22/](#)
- [app\\_23/](#)

Cliquez alors sur "app\_24", ceci devrait permettre de lancer l'exemple app\_24.

Quand vous créez un nouvel exemple (par exemple app\_25), il suffira d'utiliser le bouton de "retour en arrière" du navigateur pour vous retrouver avec la liste des exemples. Un simple clic sur app\_25 vous permettra de tester le nouvel exemple. Inutile donc de reprendre tout le processus (attention à ne pas fermer la console ou le navigateur par mégarde).

Il existe une autre méthode, plus simple à mettre en œuvre, mais plus risquée au niveau de la sécurité (notamment si vous utilisez votre navigateur pour surfer sur le web) : la modification des paramètres du navigateur. Cette méthode est décrite ici : <https://github.com/mrdoob/three.js/wiki/How-to-run-things-locally> (à utiliser en dernier recours si vraiment vous n'arrivez pas à mettre en œuvre la méthode décrite ci-dessus). Si vous modifiez les paramètres du navigateur, vous pourrez continuer à procéder comme auparavant, un simple clic sur le fichier "index.html" suffira.

## Activité 12

### Un peu de physique (passons à l'action)

#### À faire vous même

Reprenez l'app\_24 et saisissez le code suivant :

script.js

```
Physijs.scripts.worker = 'lib/physijs_worker.js';
Physijs.scripts.ammo = './ammo.js';

var scene = new Physijs.Scene;
var camera = new
THREE.PerspectiveCamera(75,window.innerWidth/window.innerHeight,0.1,1000);
camera.position.set(0,7,20)
camera.lookAt(scene.position);
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild(renderer.domElement);

var pointLight = new THREE.PointLight(0xffffff);
pointLight.position.set( 40, 40, 40 );
scene.add(pointLight);
var spotLight = new THREE.SpotLight( 0xffffff );
spotLight.position.set(0,50,0);
scene.add(spotLight);

var boxGeo= new THREE.BoxGeometry(4,4,4);
var mat=new THREE.MeshLambertMaterial({color: 0xeeeeee});
var box = new Physijs.BoxMesh(boxGeo,mat);
scene.add(box);

function render() {
    requestAnimationFrame(render);
    scene.simulate();
    renderer.render(scene, camera);
}
render();
```

Testez cet exemple (app\_24). Si le cube tombe sous l'effet de la gravitation, c'est que le moteur physique fonctionne correctement.

L'analyse du code nous montre les nouveautés :

Les 2 premières lignes ("Physijs.scripts.worker = 'lib/physijs\_worker.js'; " et "Physijs.scripts.ammo = './ammo.js'; ") sont obligatoires. La première ligne permet d'utiliser les web workers, la deuxième permet d'utiliser ammojs. Physijs n'est pas un moteur physique, c'est ammojs le moteur physique. Physijs facilite juste l'utilisation du couple ammo.js + threejs.

À la troisième ligne, "var scene = new THREE.Scene();" est remplacé par "var scene = new Physijs.Scene;".

Un peu plus loin, nous utilisons pour définir le cube "var box = new Physijs.BoxMesh(boxGeo,mat);" à la place de "var box = new THREE.Mesh(boxGeo,mat);".

Quand nous voulons que le moteur physique gère un objet, ce dernier crée un objet invisible autour de l'objet à gérer. Cet objet invisible est appelé rigidbody. Le moteur physique ne gère que les rigidbodies.

En précisant "Physijs.BoxMesh" nous indiquons au moteur physique que le rigidbody qui sera associé à notre cube sera de type box (c'est-à-dire que le rigidbody sera lui même un cube).

Avec des objets plus complexes, la forme du rigidbody ne correspondra pas toujours à la forme de l'objet, cela pourra entraîner des bugs : notre vaisseau spatial n'a visiblement pas été touché par le missile, mais pourtant il a explosé ! Ceci est simplement dû au fait que le rigidbody associé au vaisseau spatial n'a pas exactement la même forme que le vaisseau spatial lui-même et ce qui compte pour savoir si l'on a été touché, c'est la collision entre le rigidbody du missile et le rigidbody du vaisseau, d'où les problèmes.

Pourquoi ne pas créer un rigidbody qui aura exactement la forme du vaisseau ?

Plus le rigidbody est complexe, plus les calculs sont compliqués, plus le CPU sera sollicité. Il faut donc trouver un

compromis "justesse" du rigidbody, sollicitation du CPU (notez que je parle de CPU, mais dans certains cas, le GPU peut aussi effectuer les calculs nécessaires au moteur physique).

Dans la fonction "render" nous avons ajouté une ligne `scene.simulate();`. C'est cette ligne qui permet au moteur physique de refaire les calculs à chaque image.

## Gravitation

Nous allons maintenant pouvoir gérer la gravitation. Mais avant cela, une petite parenthèse mathématique s'impose :

Pour définir une direction et un sens, un point ne suffit pas. Comment faire alors ?  
Afin de simplifier les choses, raisonnons en 2D (le principe est le même en 3D) :

Pour définir une direction, il faut 2 points, mais partons du principe qu'un de ces 2 points est l'origine du repère (coordonnées (0,0)). Avec un seul point, nous pouvons alors définir une direction et un sens : imaginons un point de coordonnée (0,1), nous aurons alors une direction "Sud-Nord" et le sens "vers le nord".

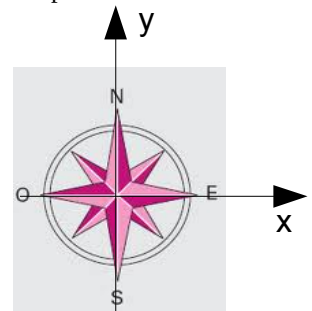
Un point de coordonnée (0, -1) indiquera la même direction "Sud-Nord" mais pas le même sens, ici on ira "vers le sud".

Quels sont la direction et le sens donnés par un point de coordonnées (1,0) ?

Réponse : direction : "Ouest-Est" sens : "vers l'est"

Un peu plus difficile : Quels sont la direction et le sens donnés par un point de coordonnées (1,1) ?

Réponse : direction : "SudOuest-NordEst" sens : "vers le NordEst"



Quels sont la direction et le sens donnés par le point de coordonnées (1,-1) ?

Réponse : direction : "NordOuest-SudEst" sens : "vers le SudEst"

En 3D le principe est le même : on prend toujours comme point de départ l'origine du repère (0,0,0) on définira une direction et un sens par une flèche. Par souci de simplification, dans la suite, je parlerai des coordonnées d'une flèche.

## À faire vous même

Quelles sont les coordonnées de la flèche (1) ?

Réponse : (0,1,1)

Quelles sont les coordonnées de la flèche (2) ?

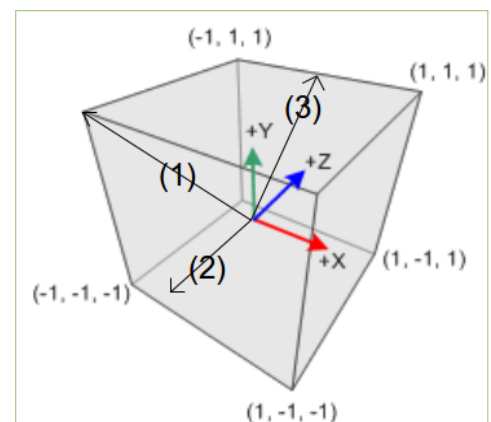
Réponse : (0,0,-1)

Quelles sont les coordonnées de la flèche (3) ?

Réponse : (0,1,1)

J'ai, à chaque fois, utilisé 0, 1 et -1 au niveau des coordonnées. Il est possible d'utiliser n'importe quelle valeur (à la place de 1 et de -1).

Plus les valeurs utilisées seront importantes, plus la flèche sera grande (une flèche de coordonnées (0,2,0) sera 2 fois plus grande qu'une flèche de coordonnées (0,1,0)).



À chaque fois que vous aurez besoin de définir une direction et un sens, il faudra procéder de cette façon (en vous ramenant à l'origine du repère). Une fois votre "flèche" définie, vous pourrez, mentalement, la déplacer où bon vous semble.

Enfin, pour ceux qui ne l'ont pas encore traité en cours de mathématiques, cette flèche est un outil mathématique dénommé vecteur. Un vecteur possède : une direction, un sens et une norme (qui correspond à la longueur du vecteur).

Dans Threejs, un vecteur sera représenté par ses coordonnées :

```
var monVecteur = new THREE.Vector3(0,1,1)
```

représente un vecteur dénommé "monVecteur" qui a pour coordonnées (0,1,1).

L'intensité de pesanteur souvent symbolisée par la lettre  $g$ , vaut sur Terre, en moyenne,  $10 \text{ N.kg}^{-1}$  (en faite c'est plutôt  $9,8 \text{ N.kg}^{-1}$ ). Sur la Lune, nous avons, en moyenne,  $g_L = 1,6 \text{ N.kg}^{-1}$ . L'intensité de la pesanteur intervient dans certaines expressions classiques de la physique :

- $P = m \times g$  (avec  $P$  le poids en Newton d'un objet de masse  $m$  (en Kg))
- $v = \sqrt{2 \times g \times h}$  avec  $v$  la vitesse (en  $\text{m.s}^{-1}$ ) d'un objet après une chute libre (sans frottements) d'une hauteur  $h$  (en m)

Vous pouvez constater qu'un objet tombe "plus vite" sur la Terre que sur la Lune ( $g$  plus grand que  $g_L$ ). En revanche la masse n'intervient pas dans la vitesse de chute : une plume et un marteau tombent exactement à la même vitesse (dans le cas où les frottements de l'air sont nuls). Pour vous en convaincre, voici la vidéo de l'expérience réalisée par Neil Armstrong sur la Lune : [https://www.youtube.com/watch?v=5C5\\_dOEyAfk](https://www.youtube.com/watch?v=5C5_dOEyAfk)

En faite, l'intensité de pesanteur est représentée, non pas par un simple nombre, mais par un vecteur. La direction est le sens du vecteur donne la direction (verticale) et le sens (vers le bas) d'un objet en train de chuter. Sur Terre, si l'on ne tient pas compte des particularités locales, nous pouvons dire que le vecteur intensité de pesanteur a pour coordonnées  $(0, -10, 0)$  (si je reprends le système de coordonnées de Threejs avec l'axe  $y$  pointant vers le haut).

Dans Threejs, par défaut, l'intensité de la pesanteur a pour coordonnées  $(0, -10, 0)$ , mais il est très facile de la modifier grâce à la méthode "setGravity" de l'objet "scene". Cette méthode prend un seul paramètre : le vecteur intensité de pesanteur.

Exemple : `"scene.setGravity(new THREE.Vector3(0, -20, 0))"` permet de doubler l'intensité de pesanteur par rapport à la valeur terrestre.

### ***À faire vous même***

Modifiez "app\_24" pour que le cube tombe 2 fois plus vite.

Pour vous aider : petit rappel  $\Rightarrow v = \sqrt{2 \times g \times h}$  , attention à la présence de la racine carrée (il ne suffit pas de doubler  $g$  pour doubler la vitesse).

### ***À faire vous même***

Modifiez "app\_24" pour que le cube "tombe" désormais vers le haut !

Il est aussi possible de modifier la masse d'un objet (par défaut un objet à une masse qui est proportionnelle à son volume : plus un objet est gros, plus sa masse est importante), en ajoutant un paramètre au constructeur "Physijs.BoxMesh". Par exemple `"var box = new Physijs.BoxMesh(boxGeo, mat, 0);"` permettra d'avoir une masse nulle pour l'objet "box". Mettre la masse d'un objet à zéro peut être utile si vous ne voulez pas qu'un objet soit soumis à la gravitation.

### ***À faire vous même***

Modifiez "app\_24" pour que le cube ne soit plus soumis à la gravitation.

## Activité 13

### Encore de la physique

Si vous laissez tomber une "balle rebondissante" par terre, elle va rebondir (logique vu son nom ;-)). Si maintenant vous laissez tomber une balle de tennis, elle va aussi rebondir, mais moins que la "balle rebondissante". Ceci est tout simplement dû au fait qu'elles ne sont pas constituées du même type de matériau : la balle rebondissante est constituée d'un matériau plus élastique que la balle de tennis. Physijs permet de tenir compte de ces différents types de matériau.

Voici la procédure à suivre pour créer un de ces matériaux (reprenons et modifions l'exemple donné dans l'app\_24) :

```
var boxGeo= new THREE.BoxGeometry(4,4,4);
var mat=new THREE.MeshLambertMaterial({color: 0xeeeeee});
var matPhy= Physijs.createMaterial (mat,0.8,0.7);
var box = new Physijs.BoxMesh(boxGeo,matPhy);
scene.add(box);
```

Par rapport à l'exemple précédent (app\_24) nous créons "matPhy" à l'aide de la méthode "Physijs.createMaterial". Cette méthode "Physijs.createMaterial" prend 4 paramètres :

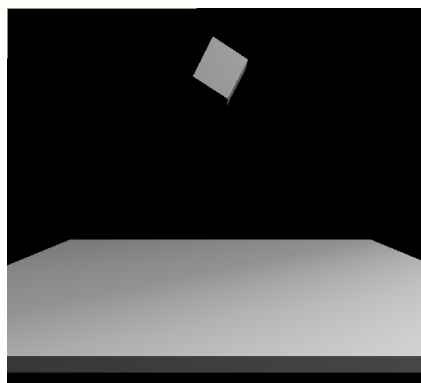
- "mat" => le matériau tel que nous le définissons auparavant (avec "MeshLambertMaterial" ou "MeshPhongMaterial")
- "0.8" => correspond aux frottements générés par le matériau
- "0.7" => correspond à l'élasticité du matériau (la "balle rebondissante" a une élasticité supérieure à la balle de tennis).

Ensuite, pour la création de "box", le deuxième paramètre du constructeur "Physijs.BoxMesh" devra être "matPhy" à la place de "mat".

Il faut bien que vous compreniez qu'ici "mat" représente uniquement l'aspect du matériau (couleur, brillance,...) alors que "matPhy" représente l'aspect (comme pour "mat"), mais aussi les propriétés physiques du matériau.

#### ***À faire vous même***

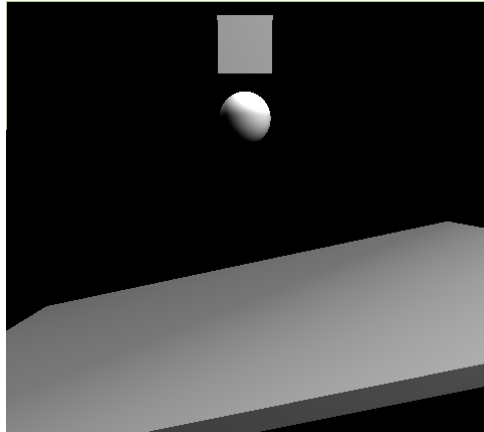
app\_25 : Créez 2 objets de type box : un de ces 2 objets sera identique au cube de l'exemple précédent (app\_24), l'autre devra jouer le rôle de sol. Le moteur physique gèrera ces 2 objets. Le cube devra, au départ, se trouver au-dessus du sol (voir l'image ci-dessous). Évidemment, le sol ne devra pas tomber en même temps que le cube. Vous choisirez les paramètres liés aux frottements et à l'élasticité des matériaux comme bon vous semble.





### *À faire vous même*

app\_26 : Reprenez l'exemple précédent en ajoutant une sphère (utilisation de "Physijs.SphereMesh") et en inclinant le sol.



### *À faire vous même*

Modifiez l'app\_26 en faisant varier les frottements au niveau du cube et de la sphère.

### *À faire vous même*

app\_27 : Votre scène sera composée d'un sol (utilisation d'un objet de type "box") horizontal et de 50 objets (n'hésitez pas à réduire ce nombre si votre machine n'arrive pas à suivre) qui seront soit des cubes, soit des sphères (la répartition cube-sphère devra être aléatoire). La position d'origine de chaque objet devra aussi être aléatoire (mais tous les objets devront se trouver au-dessus du sol). Tous les objets seront gérés par le moteur physique.



Pour obtenir des couleurs aléatoires :

```
var mat=new THREE.MeshLambertMaterial({color: Math.floor(Math.random()*0xffffffff)});
```

# Activité 14

## Toujours de la physique (collisions et forces)

### Les collisions

Il est possible de détecter les collisions entre objets à l'aide d'un listener (utilisation de la méthode "addEventListener"). La fonction de callback sera exécutée en cas de collision.

Exemple : soit un objet cube, voici le listener qui gèrera la collision du cube avec un autre objet de la scène :

```
sol.addEventListener('collision',function(obj){
    alert("collision avec le cube");
});
```

La fonction anonyme (function(obj){ alert("collision avec le cube"); }) sera exécutée si le cube entre en collision avec un autre objet. Le paramètre "obj" correspond à l'objet entré en collision avec le cube.

### À faire vous même

app\_28 : Analysez et testez le code suivant :

script.js

```
Physijs.scripts.worker = 'lib/physijs_worker.js';
Physijs.scripts.ammo = './ammo.js';

var scene = new Physijs.Scene;
var camera = new
THREE.PerspectiveCamera(75,window.innerWidth/window.innerHeight,0.1,1000);
camera.position.set(0,10,50)
camera.lookAt(scene.position);
var renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild(renderer.domElement);

var pointLight = new THREE.PointLight( 0xffffff );
pointLight.position.set( 40, 40, 40 );
scene.add(pointLight);
var spotLight = new THREE.SpotLight( 0xffffff );
spotLight.position.set(0,50,0);
scene.add(spotLight);

var cubeGeo= new THREE.BoxGeometry(4,4,4);
var mat=new THREE.MeshLambertMaterial({color: 0xeeeeee});
var matCube= Physijs.createMaterial (mat,0.5,0.9);
var cube = new Physijs.BoxMesh(cubeGeo,matCube);
cube.position.y=20;
cube.rotation.z=Math.PI/3;
cube.name="le cube";
scene.add(cube);

var solGeo= new THREE.BoxGeometry(50,1,50);
var matSol= Physijs.createMaterial (mat,0,0);
var sol = new Physijs.BoxMesh(solGeo,matSol,0);
sol.name="le sol"
scene.add(sol);

sol.addEventListener('collision',function(obj){
    alert("collision entre "+this.name+" et "+obj.name);
});

function render() {
    requestAnimationFrame(render);
    scene.simulate();
    renderer.render(scene, camera);
}
render();
```

Quelques indications pour vous aider dans votre analyse.

Les objets possèdent un attribut "name" qui permet de donner un nom particulier à un objet. Par exemple, ici, l'objet "sol" aura pour nom "le sol".

Le "this" de la fonction anonyme associée au listener "collision" correspond à l'objet auquel le listener est associé (dans notre cas l'objet "sol"). L'utilisation de "this" dans ce cas précis peut paraître superflue, mais vous pourriez en avoir besoin dans l'avenir. Avec "this" et avec "obj" vous avez donc les 2 objets concernés par la collision.

### ***À faire vous même***

app\_29 : En repartant de l'app\_26, modifiez le code pour qu'en cas de collision "sphère vs cube" les 2 protagonistes de la collision disparaissent.

Pour vous aider : si vous voulez faire disparaître l'objet "cube", il suffira d'écrire : `scene.remove(cube)`.

### **Appliquer une force sur un objet**

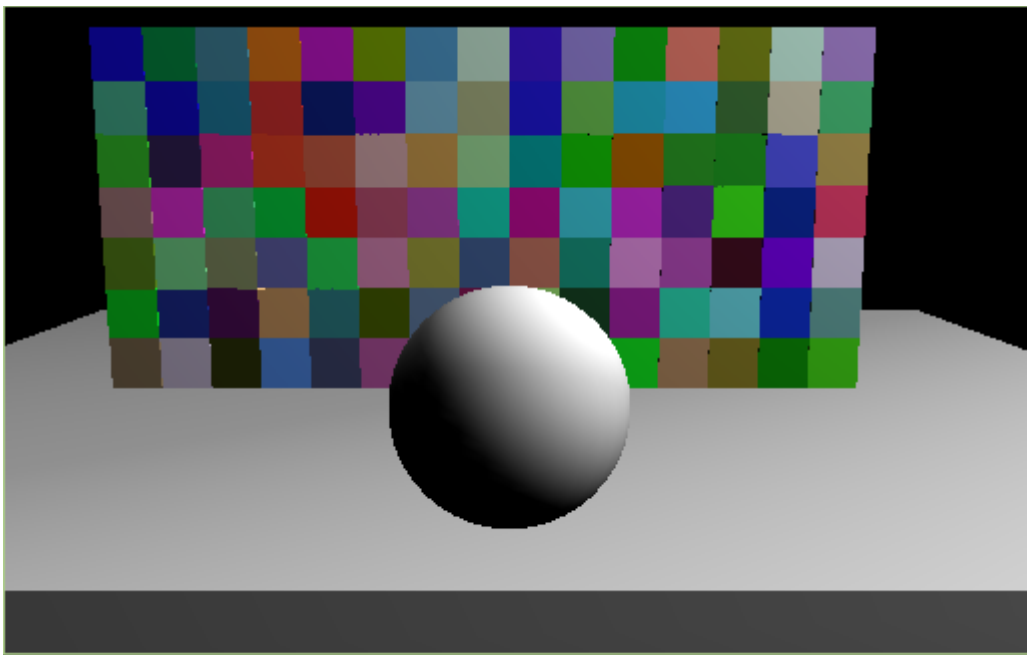
Une force a une direction, un sens et une valeur aussi appelée intensité (et aussi un point d'application, mais nous reviendrons plus tard sur cet aspect des choses) : une force est donc modélisée par un vecteur.

Pour appliquer une force sur objet (le point d'application étant le centre de l'objet), physijs, une fois de plus, simplifie bien les choses, par exemple :

`"cube.applyCentralImpulse(new THREE.Vector3(0,600,0)) ;`" appliquera au centre de l'objet "cube", une force verticale, dirigée vers le haut et d'intensité 600 (unité inconnue).

### ***À faire vous même***

app\_30 : Créez une scène avec un sol (type "box"), une sphère et un mur (constitué de cube)



L'appui sur la barre d'espace doit propulser la sphère vers le mur. L'idée étant de casser le mur. N'hésitez pas à faire varier les différents paramètres (intensité de la force exercée sur la sphère, frottements, masse des briques, masse de la sphère,...).

# Annexe 1 : Les extensions threejs

Threejs est une série d'extensions pour threejs créée par Jerome Etienne ([@jerome\\_etienne](#) sur twitter, son blog (en anglais) : <http://learningthreejs.com/>) un développeur français, spécialiste du WebGL et plus particulièrement de threejs. <http://www.threejsgames.com/extensions/>

Allez sur le site, vous découvrirez des extensions qui vous seront d'une grande utilité pour vos futurs projets.

## Annexe 2 : Améliorer les performances

Voici quelques conseils donnés par l'auteur de `physijs` afin d'améliorer les performances. L'idée est de "découpler" le rendu des images et les calculs liés au moteur physique.

Remplacer `"var scene = new Physijs.Scene;"`

par `"var scene = new Physijs.Scene({ fixedTimeStep: 1 / 120 });"` (on impose au moteur physique de réactualiser les calculs tous les 1/120 seconde, plus la valeur est petite, plus la simulation est correcte, mais plus elle est "gourmande" en terme de calculs).

`"scene.simulate();" ne doit plus être appelée depuis la fonction "render" mais depuis un listener créé exprès pour l'occasion :`

```
"scene.addEventListener( 'update', function() {  
    scene.simulate(undefined,1);  
});"
```

Nous avons aussi ajouté des paramètres à la méthode `"simulate"`, mais je n'aborderai pas cette question ici.

Il ne faudra pas oublier d'appeler la méthode `"simulate"` une première fois, sinon la simulation physique ne démarrera pas. Les 2 dernières lignes de votre programme seront donc :

```
"scene.simulate(undefined,1);  
render();"
```

À titre personnel, je n'ai pas trouvé de différence majeure en appliquant cette méthode, mais la différence serait peut-être plus flagrante avec une scène plus complexe.

to do list :

- threeex domevent
- threeex md2character
- ....