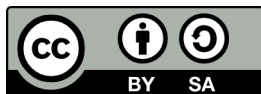


Faire de la 3D dans un navigateur web avec Babylon.js

v020214
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.

Notes de l'auteur

Ce document est à l'origine destiné aux élèves de seconde, du lycée G Fichet de Bonneville (Haute-Savoie), qui ont choisi de suivre l'enseignement d'exploration PSN (Pratique Scientifique et Numérique). Il est aussi utilisable en spécialité ISN.

Dans tous les cas, des connaissances en HTML, en JavaScript et en CSS sont indispensables avant d'aborder la lecture de ce document. Pour les débutants en programmation, 19 activités, consacrées au trio JavaScript, HTML et CSS, sont téléchargeables : http://www.webisn.byethost5.com/apprendre_prog_avec_JS.pdf

Ce document est en cours de rédaction, une nouvelle version (augmentée et corrigée) sera mise en ligne environ une fois par mois. Babylon.js est une librairie très récente, elle évolue très rapidement.

Si certaines personnes s'étonnent de certaines pratiques pédagogiques employées (notion de "direction" par exemple), je leur rappelle que les élèves de seconde, en début d'année, ne connaissent pas la notion de vecteur.

Je tiens à remercier les auteurs de Babylon.js pour leur formidable travail qui va permettre de mettre "la création 3D dans un navigateur web" à la portée de "tous".

Merci aussi à @Temechon pour son aide précieuse.

David Roche

1ère partie : les bases de BabylonJS

Chapitre 1 : les bases

Babylon.js (<http://www.babylonjs.com/>) 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 :

- David Catuhe
- Michel Rousseau
- Pierre Lagarde
- Sébastien Pertus
- David Rousset

Babylon.js n'est pas un logiciel (à la différence d'Unity3D par exemple), c'est une librairie JavaScript. Vous allez devoir écrire du code dans un éditeur de texte. Une documentation est en cours de rédaction (<http://www.sokrate.fr/documentation/babylonjs/index.html>) n'hésitez pas à la consulter.

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

- du HTML (ici HTML5 plus précisément)
- du CSS
- du JavaScript

Le HTML sera très simple puisqu'il se résumera à une balise <canvas> (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.

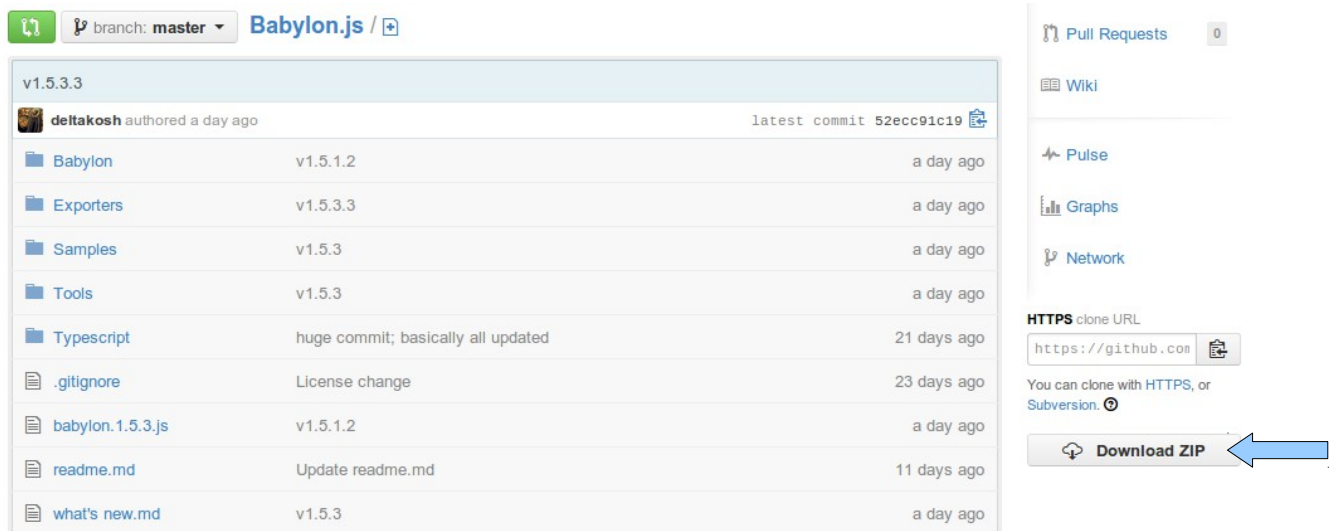
Vous trouverez ci-dessous le code JavaScript qui sera utilisé tout au long de ce document.

Code HTML (fichier index.html)

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <title>Titre de la page</title>
  <link rel="stylesheet" href="css/style.css">
  <script src="lib/babylon.1.5.3.js"></script>
</head>
<body>
  <canvas id="renderCanvas"></canvas>
</body>
<script src="javascript/script.js"></script>
</html>
```

Rien de spécial dans ce code HTML, nous utilisons une feuille de style et nous "chargeons" la librairie Babylon.js (`<script src="lib/babylon.1.5.3.js"></script>`). Cette librairie est à télécharger ici :

<https://github.com/BabylonJS/Babylon.js>



Cliquez sur le bouton Download ZIP, ce qui devrait avoir pour conséquence de débiter le téléchargement d'une archive dénommée (au moment de la rédaction de ce document) "Babylon.js-master.zip". Une fois le téléchargement terminé, "dézippez" cette archive. Vous devriez alors rapidement identifier le fichier "babylon.1.5.3.js" (attention, au moment de la rédaction de ce document, la librairie est en version 1.5.3 (`babylon.1.5.3.js`), il est fort probable qu'au moment de votre lecture, la version ne soit plus même. Veuillez à adapter le code HTML en conséquence : "`<script src='lib/babylon.X.X.X.js'></script>`" les X sont à remplacer par le bon numéro de version).

Vous avez sans doute remarqué que "`<script src='javascript/script.js'></script>`" se trouve à la fin du code HTML (juste avant la fermeture de la balise html). Cela permet de mettre en place la balise `<canvas>` avant l'exécution du JavaScript qui sera contenu dans le fichier `script.js`.

Le fichier CSS n'est pas très compliqué non plus :

code CSS (fichier `style.css`)

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

Je ne commenterai pas ce code, pour plus d'information, n'hésitez pas à faire des recherches sur internet.

Vous devriez avoir à votre disposition 3 fichiers : `index.html`, `style.css` et `babylon.X.X.X.js`

Créez un dossier "babylon-ex1", placez ensuite le fichier "index.html" et les dossiers "asset", "css", "lib" et "javascript" dans ce dossier "babylon-ex1".

Contenu du dossier "babylon-ex1"



Le fichier "style.css" sera placé dans le dossier "css", le fichier `babylon.X.X.X.js` sera placé dans le dossier "lib". Le dossier "asset" restera pour l'instant vide.

À partir de maintenant nous allons uniquement nous intéresser au fichier "script.js" qui sera (quand il existera) placé dans le dossier "javascript".

Au fur et à mesure que vous avancerez dans la lecture de ce document, il vous suffira de "copier-coller" le dossier "babylon-ex1" et de le renommer ("babylon-ex2", "babylon-ex3"....) à chaque fois que nous traiterons un nouvel exemple.

Cette organisation (dossiers+fichiers) est une proposition, vous pouvez en choisir une autre, si c'est le cas, n'oubliez pas de modifier le fichier "index.html" en conséquence.

Nous allons maintenant entrer dans le vif du sujet en écrivant le code minimum permettant de générer une scène 3D grâce à Babylon.js.

babylon-ex1 (fichier script.js)

```
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);
engine.runRenderLoop(function () {
    scene.render();
});
```

Nous commençons par créer un objet "canvas" (ligne 1) (utilisation de la méthode `getElementById` et utilisation de l'id de la balise `<canvas>` définie dans le fichier HTML)

Cet objet canvas est utilisé afin de créer l'objet nommé "engine" (ligne 2).

Arrêtons-nous quelques instants sur la structure de cette ligne 2 : le mot clé "new" permet de créer un nouvel objet (son utilisation n'est pas systématique, il existe d'autres façons pour créer un objet en JavaScript).

"BABYLON.Engine" : "Engine" est une méthode de l'objet "BABYLON", la structure du type "BABYLON.xxxxxx" signifie que nous utilisons une méthode ou un attribut propre à la bibliothèque Babylon.js.

Nous créons ensuite l'objet "scene" (ligne 3) qui accueillera les différents "acteurs" (caméra, lumière, objet) de notre scène 3D.

La structure sera toujours la même :

création objet "canvas" -> création objet "engine" -> création objet "scene" -> utilisation de l'objet "scene"

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 Babylon.js, cet "ordre" est envoyé grâce à la ligne `scene.render();` (appel de la méthode "render" de l'objet "scene").

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".

```
"engine.runRenderLoop(function () {scene.render();});" :
```

La méthode "runRenderLoop" de l'objet "engine" va permettre d'exécuter plusieurs fois par seconde la fonction qui lui a été passée en paramètre (`function () {scene.render();}`). Comme vous pouvez le constater, cette fonction ne porte pas de nom, on parle de fonction anonyme. Les fonctions anonymes sont monnaie courante en JavaScript, elles sont très souvent utilisées comme paramètre d'autres fonctions (ou méthodes).

Que fait cette fonction anonyme ?

Pas grand-chose, elle appelle la méthode "render".

En résumé, la méthode "runRenderLoop" va permettre d'exécuter plusieurs fois par seconde la méthode "render" (et donc d'afficher une nouvelle image).

Nous ne rentrerons pas dans les détails du fonctionnement de la méthode "runRenderLoop", mais sachez que cette méthode est "intelligente" et qu'elle permet donc d'obtenir les meilleurs résultats possible au niveau du FPS.

Pour l'instant votre scène 3D n'est pas encore "visionnable", il manque 3 éléments fondamentaux :

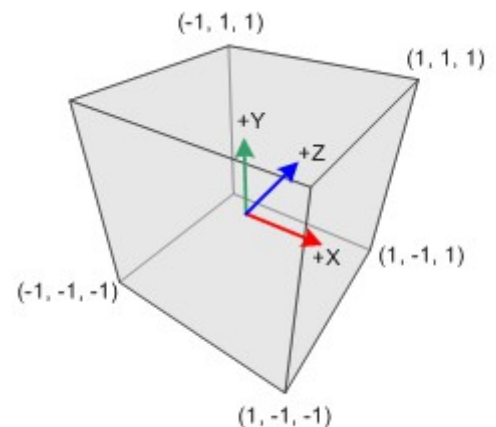
- une caméra : pour observer notre scène
- un éclairage : pour éclairer notre scène
- un (ou des) objet(s) : pour peupler notre scène

Chapitre 2 : Une première scène

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.

Pour indiquer une position dans l'espace, il est nécessaire de créer un objet de type "Vector3" (attention au faux ami, cet objet représente un point, pas un vecteur !) : `"new BABYLON.Vector3(x,y,z)"` avec, évidemment, x,y,z, les coordonnées du point dans l'espace.

Pour vous aider, voici une représentation du repère utilisé dans Babylon.js :



Commençons le "peuplement" de notre scène en plaçant notre éclairage.

Il existe différents types d'éclairage, mais, dans ce chapitre nous n'en verrons qu'un, le "point lumineux" : la source de lumière est un point, la lumière se propage dans toutes les directions depuis ce point.

Pour créer ce point lumineux, il faut utiliser la méthode "PointLight" :

```
"new BABYLON.PointLight(nom,position, scene);"
```

signification des 3 arguments de la méthode "PointLight" :

- nom : nom choisi pour le point lumineux (exemple : "pointLu")
- position : position du point lumineux (exemple : `new BABYLON.Vector3(0,10,20)`)
- scene : nom de la scène à éclairer (scène définie à l'aide de `"new BABYLON.Scene(engine);"`)

Deuxième élément à placer, la caméra qui "filmiera" notre scène.

Aussi il existe différents types de caméra et ici nous n'en verrons qu'un dans ce chapitre.

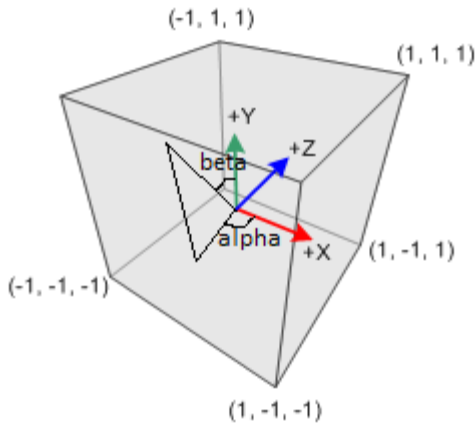
La caméra "ArcRotateCamera" est une caméra qui a la propriété de tourner autour du point qu'elle filme (cette rotation se fait à l'aide de la souris). La méthode "ArcRotateCamera" permet la création de ce type de caméra :

```
"new BABYLON.ArcRotateCamera(nom, angle_alpha, angle_beta, rayon, point_visé, scene);"
```

Ici aussi, passons en revue les différents paramètres de cette méthode :

- nom : nom choisi pour la caméra (exemple : "maCamera")
- angle_alpha : position de la caméra (voir le schéma ci-dessous)
- angle_beta : position de la caméra (voir le schéma ci-dessous)
- rayon : distance entre la caméra et le "point_visé"
- point_visé : position du point à viser (exemple : `new BABYLON.Vector3(0,0,0)` : la caméra vise ici le centre de la scène)

- scene : nom de la scène à filmer (scène définie à l'aide de "new BABYLON.Scene(engine);")



ATTENTION : les angles "alpha" et "beta" sont en radian et pas en degré (petit rappel : π radian \Rightarrow 180°)

Troisième élément : l'objet

Babylon.js vous permet de dessiner 4 types de mesh (objet 3D) : des cubes, des sphères, des cylindres, des plans et des tores (pour des meshes plus complexes, il faudra passer par un modèleur 3D comme blender,

nous verrons tout cela dans un prochain chapitre).

Pour cette première scène, nous allons dessiner un tore (pour les connaisseurs, c'est le donut d'Homer Simpson).

C'est la méthode "CreateTorus" qui sera utilisée pour dessiner notre tore :

```
"BABYLON.Mesh.CreateTorus(nom,diamètre, épaisseur, détail, scene, modifiable);"
```

- nom : nom choisi pour le mesh (ce nom peut être utile, notamment pour rechercher un objet à l'aide de la méthode "getMeshByName")
- diamètre : diamètre du tore
- épaisseur : épaisseur du tore
- détail : plus cette valeur est élevée, plus le tore sera "beau" (attention un "beau" tore demande plus de calcul et donc consomme plus de ressource CPU)
- scene : nom de la scène où notre tore sera dessiné
- modifiable : le mesh sera-t-il modifié ? (mettre true ou false)

À faire vous même

Après avoir étudié ce programme avec la plus grande attention, saisissez-le dans un fichier "script.js", enfin, testez-le en ouvrant le fichier index.html à l'aide d'un navigateur web (Internet Explorer 11, Chrome (version récente) ou Firefox (version récente)).

Pour vous aider :

vous devez savoir que la ligne "scene.activeCamera.attachControl(canvas);" permet à la caméra de tourner (modification des angles "alpha" et "beta") autour du "point_visé" (utilisation de la souris).

Math.PI correspond à la valeur de π (3,14159..)

Babylon-ex2 (script.js)

```
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);
var camera = new BABYLON.ArcRotateCamera("maCamera", 0, Math.PI/4, 10, new
BABYLON.Vector3(0, 0, 0), scene);

var light = new BABYLON.PointLight("pointLu", new BABYLON.Vector3(0, 0, 10), scene);
var tore = BABYLON.Mesh.CreateTorus("torus", 5, 1, 50, scene, false);
scene.activeCamera.attachControl(canvas);
engine.runRenderLoop(function () {
    scene.render();
});
```

À la création de la scène, par défaut, le centre du tore a pour coordonnées (0,0,0). De plus, toujours par défaut, il se trouve dans le plan (x,z).

Vous pouvez modifier les coordonnées du centre du tore, à l'aide de l'attribut position de l'objet de type Mesh, par exemple pour notre tore : "tore.position=new BABYLON.Vector3(0, 5, 5)"

Si vous désirez modifier une seule des trois coordonnées, vous pouvez utiliser, par exemple : "tore.position.z=5"

Il est aussi possible de faire tourner le tore :

`"tore.rotation.x=Math.PI"` provoque une rotation de π autour de l'axe x

`"tore.rotation.y=Math.PI/2"` provoque une rotation de $\pi/2$ autour de l'axe y

`"tore.rotation.z=Math.PI/4"` provoque une rotation de $\pi/4$ autour de l'axe z

Enfin, il est aussi possible de modifier la taille d'un mesh avec les attributs `scale.x`, `scale.y` et `scale.z` :

`"tore.scale.z=0.5"`

À faire vous même

Apporter des modifications au programme précédent (Babylon-ex2) en modifiant les coordonnées du centre du tore. Modifier ensuite la taille du tore à l'aide de l'attribut "scale". Enfin, une fois ces transformations effectuées, modifier la position de l'éclairage et de la caméra comme bon vous semble.

À faire vous même

ATTENTION : pour tout ce qui concerne la rotation, le tore possède ses propres axes x, y et z (on parle de repère local), se repère tourne en même temps que le tore, voici un exemple, qui, je l'espère, vous permettra de mieux appréhender cette notion de repère local.

Voici la situation avec le code Babylon-ex2 (l'axe y pointe vers nous).

Si nous effectuons une rotation de 90° autour de l'axe x, nous verrons alors la tranche du tore verticalement.

Que se passe-t-il si ensuite nous effectuons une rotation autour de l'axe z encore de 90° ? Si l'on y prend garde, on pourrait répondre :

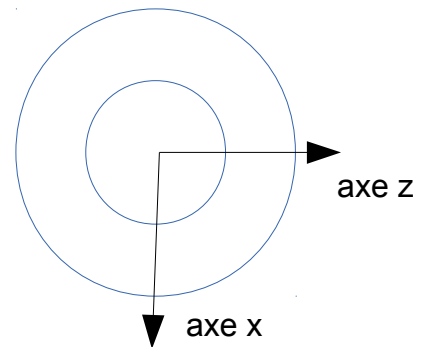
"rien, car dans cette situation z est un axe de symétrie".

Faites le test vous-même, alors ?

Pourquoi n'obtient-on pas le résultat "attendu" ?

Au moment de la rotation de 90° autour de l'axe x, l'axe z du tore a lui-même effectué une rotation de 90° (il pointe maintenant vers la feuille).

Si vous effectuez ensuite une rotation de 90° autour de l'axe z, il est donc logique de voir toujours la tranche du tore, mais cette fois-ci à l'horizontale.



Chapitre 3 : objets, caméras et éclairages

Les objets

Babylon.js propose, comme déjà indiqué, plusieurs autres mesh, voici les méthodes qui vous permettront d'afficher ces autres mesh :

Pour créer une boîte : `"BABYLON.Mesh.CreateBox(nom, taille, scene);"`

Pour créer une sphère : `"BABYLON.Mesh.CreateSphere(nom,détail,taille,scene);"`

Pour créer un cylindre : `"BABYLON.Mesh.CreateCylinder(nom, hauteur, diamètre haut, diamètre bas, détail, scene, modifiable);"` (possibilité de faire un cône)

Pour créer un plan : `"BABYLON.Mesh.CreatePlane(nom, taille, scene);"`

Tout ce que nous avons vu pour le tore (position, rotation et scale) est valable pour ces autres meshes.

À faire vous même

Voici un programme que vous permettra de dessiner un mug (une tasse) à l'écran, analysez ce code :

Babylon-ex3 (script.js)

```
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);
var camera = new BABYLON.ArcRotateCamera("maCamera", 0, Math.PI/2, 15, new
BABYLON.Vector3(0, 0, 0), scene);
var light = new BABYLON.PointLight("pointLumineux1", new BABYLON.Vector3(0, 0, 10),
scene);
var tore = BABYLON.Mesh.CreateTorus("torus", 4, 1, 50, scene, false);
var cyl=BABYLON.Mesh.CreateCylinder("nom", 7, 5, 5, 50, scene, false);
tore.rotation.z = Math.PI/2;
tore.position.z=2;

scene.activeCamera.attachControl(canvas);
engine.runRenderLoop(function () {
    scene.render();
});
```

Imaginons maintenant que nous désirions déplacer ce mug.

Il est, je pense, évident pour vous qu'il faut déplacer les 2 mesh qui constituent ce mug.

Dans ce cas précis, cela ne pose pas vraiment de problème, mais imaginez-vous avec un objet constitué d'une vingtaine de mesh !

Il existe une solution : les mesh ont un attribut "parent" qui permet pour un mesh donné de modifier l'origine de son repère.

```
mesh1.parent = mesh2
```

après cette ligne, pour mesh1 le point (0,0,0) ne sera plus le centre de la scène, mais le centre de mesh2.

Modifiez le code pour qu'un déplacement du cylindre entraine automatiquement le déplacement du tore (comme cela,



pour déplacer notre mug, il suffira de déplacer le cylindre).

Les éclairages (nous parlerons de la couleur des éclairages dans le prochain chapitre)

Babylon.js propose 3 types d'éclairages (il en existe 4, mais nous n'en verrons que 3 dans ce chapitre) :

- le point lumineux (point light)
- la lumière directionnelle (directional light)
- le spot (spot light)

La lumière directionnelle vous permet de simuler une source située à "l'infini", les rayons sont donc parallèles entre eux (un peu comme les rayons du soleil) : `"BABYLON.DirectionalLight(nom, direction+sens, scene);"`
la "direction+sens" est un "Vector3", étrange non ?

Au début de ce document, nous avons vu que "Vector3" représente un point dans l'espace 3D, c'est vrai, je ne vous ai pas menti.

Mais, ce que j'ai omis de vous dire, c'est que "Vector3" peut aussi être une direction et un sens dans ce même espace 3D. Je m'explique :

Pour définir une direction et un sens, un point ne suffit pas (ce n'est pas à vous que je vais apprendre que "par un point il passe une infinité de droite", donc impossible de donner une direction avec un seul point). 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 soit l'origine du repère (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) ?

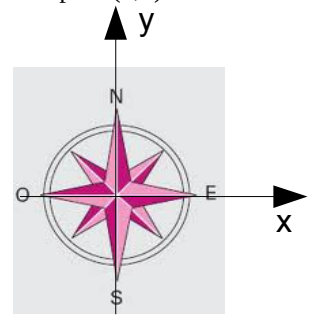
Réponse : direction : "Ouest-Est" sens : "vers l'est"

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 :

Avec quel "Vector3" représenter la direction et le sens de la flèche (1) ?

Réponse : (0,1,1)

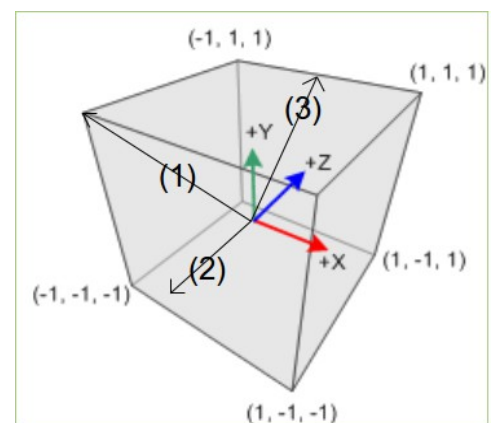
Avec quel "Vector3" représenter la direction et le sens de la flèche (2) ?

Réponse : (0,0,-1)

Avec quel "Vector3" représenter la direction et le sens 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). Cependant, par souci de clarté, je vous conseille d'utiliser uniquement 0, 1 et -1 dans un "Vector3" quand celui-ci sera destiné à indiquer une direction et un sens.



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 afin de vérifier si la direction et le sens que vous venez de définir à l'aide d'un "Vector3" est bien celui recherché.

Enfin, pour ceux qui ne l'ont pas encore traité en cours de mathématiques, cette "flèche" est un outil mathématique dénommé vecteur.

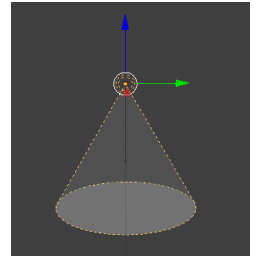
Le spot, comme son nom l'indique est un spot :

```
"BABYLON.SpotLight(nom, position, direction+sens, angleOuverture, puissance, scene);"
```

le nom, la position et la "direction+sens" ne devraient pas vous poser de problème l'angle correspond à l'angle d'ouverture du spot (en radian).

La puissance vous permet d'indiquer la puissance lumineuse émise par le spot.

ATTENTION, pour des raisons que je n'expliquerai pas ici, **plus** la valeur est importante **moins** l'intensité lumineuse sera forte.



À faire vous même

Écrire un ou plusieurs programme(s) vous permettant de tester ces différentes sources de lumière (soit seule, soit en les combinant)

Les caméras

Il nous reste à voir deux types de caméra : la "FreeCamera" et la "TouchCamera"

La "FreeCamera" est une caméra dite subjective (pour voir la définition d'une caméra subjective :

http://fr.wikipedia.org/wiki/Cam%C3%A9ra_objective_et_subjective), typiquement c'est la caméra la plus souvent utilisée dans le jeu qualifié de FPS (First-Person Shooter). Vous pouvez déplacer la caméra (translation et rotation) à l'aide de la souris (rotation) et du clavier (translation).

Son utilisation est très simple : `"new BABYLON.FreeCamera(nom, position, scene);"`

Position est un "Vector3"

La "TouchCamera" sera utile en cas d'utilisation d'une interface tactile (écran tactile). Nous développerons son utilisation dans un chapitre ultérieur.

À faire vous même

Écrire un programme se basant sur l'utilisation d'une "FreeCamera" à la place d'une "ArcRotateCamera". Vérifiez que vous êtes bien "libre de vos mouvements" (flèches du clavier pour les déplacements et souris pour les rotations).

Chapitre 4 : les matériaux et les textures

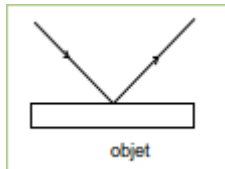
Pourquoi arrivez-vous à distinguer un bout de bois et un morceau de métal du premier coup d'œil ?

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 :

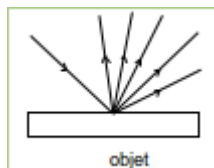
- diffuser la lumière
- réfléchir la lumière

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".

BabylonJS permet de créer ses propres matériaux et de choisir la proportion de réflexion et de diffusion. Vous allez pouvoir choisir la couleur de la lumière diffusée et la couleur de la lumière réfléchie.

Création d'un matériau avec BabylonJS

Définissons un matériau : `var materiau = new BABYLON.StandardMaterial(nom, scene);`

Une fois le matériau défini, vous allez pouvoir lui donner une couleur (ou plutôt des couleurs) :

- couleur de la lumière réfléchie :
`materiau.specularColor = new BABYLON.Color3(Rouge, Vert, bleu)`
- couleur de la lumière diffusée :
`materiau.diffuseColor = new BABYLON.Color3(Rouge, Vert, bleu)`

Pour définir une couleur (avec "BABYLON.Color3") il faut renseigner le canal rouge, le canal vert et le canal bleu. Pour plus d'informations sur la notion de synthèse additive (RVB), vous pouvez consulter (http://fr.wikipedia.org/wiki/Synth%C3%A8se_additive). Les valeurs "Rouge", "Vert" et "Bleu" sont, dans BabylonJS, comprises entre 0 et 1 (Color3(1,1,1) donnera du blanc).

Il est possible de choisir la "proportion" de lumière réfléchie, pour un matériau donné, en utilisant l'attribut "matériau.specularPower".

Il reste ensuite à associer le matériau nouvellement créé à un objet (attribut "material" du mesh) :
`mesh.material= matériau`

À faire vous même

Testez le code suivant et modifiez ensuite certains paramètres ("matériau.specularPower" par exemple) afin de mieux appréhender les notions vu ci-dessus. Testez plus particulièrement le cas où l'on a :

```
"matériau.diffuseColor = new BABYLON.Color3(0,0,0)"  
et "matériau.specularColor = new BABYLON.Color3(0.8,0.8,0.8)".
```

Babylon-ex4 (script.js)

```
var canvas = document.getElementById("renderCanvas");  
var engine = new BABYLON.Engine(canvas, true);  
var scene = new BABYLON.Scene(engine);  
var camera = new BABYLON.ArcRotateCamera("maCamera", 0, Math.PI/2, 15, new  
BABYLON.Vector3(0, 0, 0), scene);  
var light = new BABYLON.PointLight("pointLumineux1", new BABYLON.Vector3(0, 0, 10),  
scene);  
var tore = BABYLON.Mesh.CreateTorus("torus", 4, 1, 50, scene, false);  
var cyl=BABYLON.Mesh.CreateCylinder("nom", 7, 5, 5, 50, scene, false);  
tore.rotation.z = Math.PI/2;  
tore.position.z=2;  
  
var matériau = new BABYLON.StandardMaterial("mat1", scene);  
matériau.specularColor = new BABYLON.Color3(0.5, 0,0);  
matériau.diffuseColor = new BABYLON.Color3(0,0,0.5);  
matériau.specularPower=20;  
cyl.material=matériau;  
tore.material=matériau;  
  
scene.activeCamera.attachControl(canvas);  
engine.runRenderLoop(function () {  
    scene.render();  
});
```

N'avez-vous pas la légère impression que notre mug à quelque chose de métallique ? Augmentez la valeur de "specularPower", quelle est votre impression ?

J'ai conscience que tout cela est complexe.

Certaines parties d'un objet peuvent "briller" et d'autres non. La couleur de la lumière renvoyée par les parties qui brillent sera plutôt gérée par le "specularColor" et les parties mates (non brillante) seront plutôt gérées par "diffuseColor". La position de la caméra et de l'éclairage ont aussi une grande importance (surtout dans le cas de la réflexion puisque la lumière est réfléchie dans une seule direction).

Au niveau du matériau, vous pouvez aussi jouer sur d'autres facteurs :

Ambiant color : la "couleur ambiante" est la couleur d'un objet quand celui-ci se trouve dans l'ombre (quand il n'est pas directement éclairé. Pour définir cette "couleur ambiante" :

```
matériau.ambientColor = new BABYLON.Color3(r,v,b);
```

Emissive color : la "couleur émissive" est la couleur de la lumière émise par un objet quand ce dernier émet sa propre lumière (en absence de toute autre source de lumière). Pour définir cette "couleur émissive" :

```
matériau.emissiveColor = new BABYLON.Color3(r,v,b);
```

À faire vous même

Écrire un programme qui vous permettra de réinvestir toutes les notions vues jusqu'à présent dans ce chapitre

Toutes ces notions sont très difficiles à appréhender, surtout pour des novices. Si vous avez du mal à obtenir ce que vous désirez, rien de plus normal, persévérez en modifiant les différents paramètres l'un après l'autre. L'expérimentation est sans aucun doute la meilleure méthode, au moins au début.

Les textures

En plus des différentes couleurs, il est possible d'appliquer sur un objet une (ou des) texture(s).

Qu'est-ce qu'une texture ?

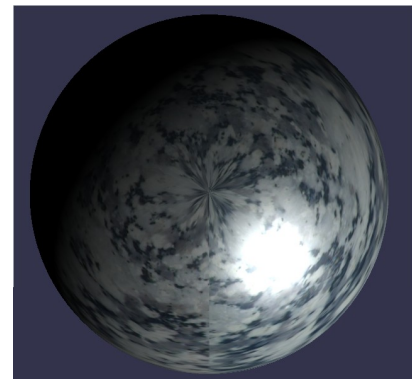
J'aurai tendance à dire un papier peint que l'on applique sur un objet.

Ce "papier peint" est une image (jpeg, png,...) que l'on va "plaquer" sur notre objet (mesh).

Pour ajouter une texture à un matériau ("maTexture.jpg" est l'image qui nous servira de texture) :

```
var mat = new BABYLON.StandardMaterial("matériau_1", scene);  
mat.diffuseTexture = new BABYLON.Texture("maTexture.jpg", scene);
```

Attention, dans l'exemple ci-dessus, l'image doit se trouver dans le même dossier que notre fichier "index.html".



À faire vous même

Téléchargez une image sur internet et utilisez cette image pour appliquer une texture sur l'objet de votre choix.

Comme pour la couleur, il existe différents types de texture :

- la texture "diffuseTexture" vu ci-dessus
- la texture "ambientTexture"
- la texture "emissiveTexture"
- la texture "specularTexture" (attention cette texture n'apparaîtra qu'aux endroits où la lumière est réfléchie)

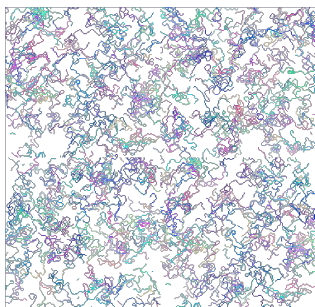
À faire vous même

Testez ces différents types de texture.

Texture et transparence

Certaines images possèdent ce que l'on appelle un canal alpha, ces images gèrent la transparence. Les pixels de l'image qui auront une couleur prédéterminée apparaîtront transparents.

Voici un exemple :

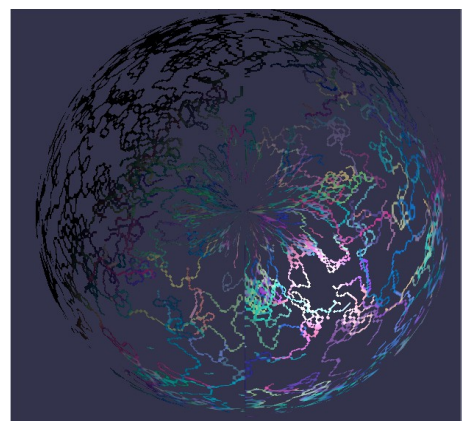


Pour cette image (au format png), tous les pixels blancs seront considérés comme transparents.

Voici le résultat de l'application de cette texture sur une sphère :

Pour activer la transparence, il suffit de passer l'attribut "diffuseTexture.hasAlpha" à true :

```
mat.diffuseTexture.hasAlpha = true;
```



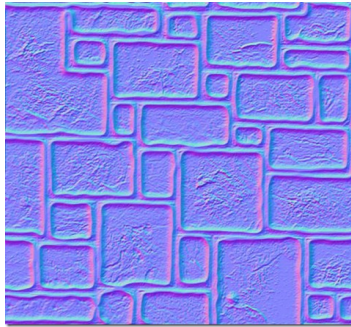
À faire vous même

Télécharger une image gérant la transparence et l'utiliser afin d'appliquer une texture (avec transparence) sur l'objet de votre choix.

Bump texture

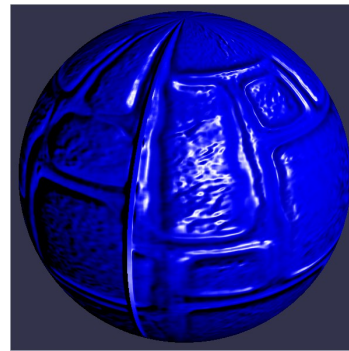
Le Bump Mapping (placage de relief) est une technique permettant de simuler la présence de relief sur un objet grâce à une texture (pour en savoir plus sur le Bump Mapping : http://fr.wikipedia.org/wiki/Placage_de_relief)

Voici l'image de départ :



texture téléchargée sur <https://github.com/BabylonJS/Babylon.js/wiki/14-Advanced-Texturing>

et voici le résultat de son application sur une sphère :



Voici le code permettant d'obtenir ce résultat :

```
var mat = new BABYLON.StandardMaterial("matSp", scene);  
mat.diffuseColor = new BABYLON.Color3(0, 0, 1);  
mat.bumpTexture = new BABYLON.Texture("text_bump.png", scene);
```

Placage des textures

Il ne faut pas perdre de vue qu'une texture est une image plane (donc en 2 dimensions). L'application (on parle de projection) de cette image sur un objet en 3 dimensions (sauf si votre objet est un plan) entraîne forcément des déformations au niveau de la texture.

BabylonJS propose de nombreux outils permettant de gérer avec précision la projection d'une texture sur un objet, cependant, j'ai choisi de ne pas aborder ce sujet (très complexe) ici.

En effet, dans la plupart des cas, le placage de textures complexes se fera à l'aide du logiciel Blender (voir le dernier chapitre de ce document).

Chapitre 5 : les animations

Jusqu'à présent les différents objets que nous avons créés étaient statiques, mettons maintenant un peu de mouvement dans nos scènes.

La première méthode d'animation que nous allons aborder ici a de nombreux points communs avec les méthodes d'animations utilisées en CSS3.

La création d'une animation se fait en 3 étapes :

- utilisation de la méthode BABYLON.Animation :

```
var anim1= new BABYLON.Animation (nom, paramètre concerné par l'animation, nbre de
FPS maxi, type du paramètre concerné par l'animation,évolution du paramètre
concerné par l'animation)
```

- création d'un tableau contenant la description de l'animation
- attribution de l'animation à l'objet concerné

Prenons tout de suite un exemple :

À faire vous même

Saisissez, analysez et testez l'exemple suivant :

Babylon-ex5 (script.js)

```
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);

var camera = new BABYLON.ArcRotateCamera("Camera",Math.PI/4, Math.PI/2, 30, new
BABYLON.Vector3(0, 0, 0), scene);
var light = new BABYLON.PointLight("Omnidir", new BABYLON.Vector3(10, 0, 10), scene);
var maBox = BABYLON.Mesh.CreateBox("box1", 10, scene);
camera.attachControl(canvas);

var animationBox = new
BABYLON.Animation("boxAnim","scaling.z",30,BABYLON.Animation.ANIMATIONTYPE_FLOAT,BABYLON.
Animation.ANIMATIONLOOPMODE_CYCLE);
var keys = [];
keys.push({
    frame: 0,
    value: 1
});
keys.push({
    frame: 25,
    value: 0.75
});
keys.push({
    frame: 40,
    value: 0.5
});
```

```

keys.push({
    frame: 65,
    value: 0.75
});
keys.push({
    frame: 75,
    value: 0.5
});
keys.push({
    frame: 100,
    value: 0.9
});
animationBox.setKeys(keys);
maBox.animations.push(animationBox);
scene.beginAnimation(maBox, 0, 100, false);

engine.runRenderLoop(function () {
    scene.render();
});

```

Quelques commentaires sur ce code afin de faciliter votre analyse :

```

"var animationBox = new
BABYLON.Animation("boxAnim", "scaling.z", 30, BABYLON.Animation.ANIMATIONTYPE_FLOAT, BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE); "

```

- dans cette animation, nous allons faire varier "scaling.z"
- nous rechercherons 30 images/seconde
- la valeur que nous faisons varier ("scaling.z") est de type "FLOAT" (nombre flottant)
- Si notre animation est "jouée en boucle", nous repartons toujours de la valeur initiale (ici une taille de 10) : "LOOPMODE_CYCLE"

Il existe d'autres "types" (en plus de "FLOAT") : la variable de l'animation peut-être un "Vector3"

```

("BABYLON.Animation.ANIMATIONTYPE_VECTOR3") ou un "Quaternion"
("BABYLON.Animation.ANIMATIONTYPE_QUATERNION").

```

N.B. Nous n'aborderons pas la notion de "Quaternion" dans ce document, c'est une notion mathématique trop complexe pour des élèves de lycée.

"LOOPMODE_RELATIVE" et "LOOPMODE_CONSTANT" (utilisé à la place de "LOOPMODE_CYCLE") permettent de :

- pour "LOOPMODE_RELATIVE" : si l'animation est "jouée en boucle", l'animation repartira, non pas de la valeur initiale, mais de la dernière valeur atteinte au cours de la précédente animation.
Dans notre exemple, nous terminons l'animation avec un "scaling.z=0.9" (voir ci-dessous pour l'explication), notre cube a donc un largeur selon z de 9 (90% de la valeur initiale : 10). La 2^e fois que l'animation sera jouée, la valeur initiale sera de 9, à la fin du 2^e cycle, la taille du cube sera de 90% de 9.....
Au fur et à mesure notre cube va donc voir sa taille diminuée. Si vous avez du mal à comprendre, remplacer le "BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE" de l'exemple 5 par "BABYLON.Animation.ANIMATIONLOOPMODE_RELATIVE".
- pour "LOOPMODE_CONSTANT" : à la fin du premier cycle d'animation, la valeur restera constante (même si normalement l'animation devait se "jouer en boucle"). N'hésitez pas, là aussi, à remplacer "BABYLON.Animation.ANIMATIONLOOPMODE_CYCLE" par "BABYLON.Animation.ANIMATIONLOOPMODE_CONSTANT"

```

"var keys = []"

```

Nous créons un tableau qui va contenir les différentes phases de notre animation.

```

"keys.push({ frame: 25, value: 0.75 }); "

```

Chaque "keys.push" permet de rajouter un objet JavaScript au tableau "keys" défini ci-dessus.

Ces objets ont 2 attributs : "frame" et "value". "{ frame: 25, value: 0.75 }" signifie qu'à la 25^e image après le début de l'animation, "scaling.z" sera égale à 0.75

Si vous analysez l'exemple 5 vous remarquerez que, par exemple, entre l'image n°25 et l'image n°40, nous n'apportons aucune indication sur la valeur de "scaling.z" : c'est BabylonJS qui se chargera de compléter les valeurs manquantes afin de passer "en douceur" d'un "scaling.z=0.75" (image n°25) à un "scaling.z=0.5" (image n°40). Les images pour

lesquelles on définit une valeur de `"scaling.z"` sont appelées "images clés" (d'où le nom du tableau).

`"animationBox.setKeys(keys);` " une fois le tableau `"keys"` rempli, la méthode `"setKeys"` permettra de "lier" ce tableau avec l'animation définie plus haut.

Pour lancer l'animation, un simple `"scene.beginAnimation(maBox, 0, 100, false)"` suffit. Les paramètres de la méthode `"beginAnimation"` sont :

- le nom de l'objet qui sera animé
- le numéro de la première image
- le numéro de la dernière image
- l'animation est jouée en boucle (`true/false`)

À faire vous même

Écrivez un programme permettant d'avoir à l'écran une sphère. Animez ensuite cette sphère (variation de son rayon)

Pour vous aider :

- paramètre à faire varier : `"scaling"`
- `"scaling"` est de type `Vector3`

Autre méthode possible pour gérer les animations

La méthode présentée ci-dessus peut, dans certains cas, se révéler trop limitée. Afin d'avoir une plus grande latitude dans la gestion des animations, BabylonJS vous donne la possibilité de modifier les paramètres d'un objet à chaque image en utilisant la méthode `"registerBeforeRender"` (méthode de l'objet `"scene"`). Cette méthode prend un seul paramètre : une fonction qui sera appelée avant le rendu de chaque image.

À faire vous même

Saisissez, analysez et testez l'exemple suivant :

Babylon-ex6 (script.js)

```
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);

var camera = new BABYLON.ArcRotateCamera("Camera",0, Math.PI/4, 15, new
BABYLON.Vector3(0, 0, 0), scene);
var light = new BABYLON.PointLight("Omnidir", new BABYLON.Vector3(10, 0, 10), scene);
var maBox= new BABYLON.Mesh.CreateBox("box_1",5,scene);

scene.registerBeforeRender(function() {
    maBox.rotation.y=maBox.rotation.y+0.05;
});

engine.runRenderLoop(function () {
    scene.render();
});
```

Ce code ne présente aucune difficulté : à chaque image la valeur de `"maBox.rotation.y"` augmente de 0,05 radian.

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 suivant :

Babylon-ex7 (script.js)

```
var temps=0;
var vitesseRotation=2;
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);

var camera = new BABYLON.ArcRotateCamera("Camera",0, Math.PI/4, 15, new
BABYLON.Vector3(0, 0, 0), scene);
var light = new BABYLON.PointLight("Omnidir", new BABYLON.Vector3(10, 0, 10), scene);
var maBox= new BABYLON.Mesh.CreateBox("box_1",5,scene);

scene.registerBeforeRender(function() {
    temps=temps+(1/30)
    maBox.rotation.y=vitesseRotation*temps;
});

engine.runRenderLoop(function () {
    scene.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 α l'angle, ω 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 7 n'est donc pas satisfaisante.

Heureusement, les créateurs de BabylonJS ont pensé à tout : ils proposent une méthode renvoyant le nombre d'images par seconde : `"BABYLON.Tools.GetFps()"`

À faire vous même

Modifiez le code de l'exemple 7 en utilisant la méthode `"BABYLON.Tools.GetFps()"` afin d'avoir un "écoulement du temps" indépendant du nombre d'images par seconde.

Miniprojet

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 :

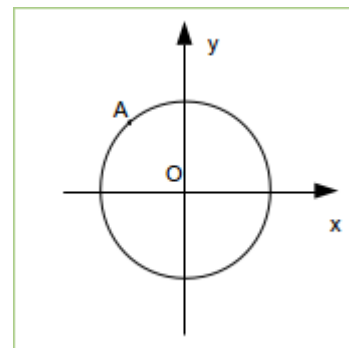
Soit le point A de coordonnées x et y

À décrire 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 ω la vitesse angulaire (vitesse de rotation en radian par seconde)
et t la variable temps.



Pour aller plus loin :

En recherchant des informations sur internet, créer un simulateur de système solaire avec les 8 planètes. Vous respecterez les périodes de révolution et les proportions (taille) des planètes.

Chapitre 6 : Gestion des événements (clavier+souris)

Dans ce chapitre 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).

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 :

"click" : clic du bouton gauche de la souris

"dblclick" : double-clic du bouton gauche de la souris

"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 programme 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", "keyup" et "keypress" de connaître le code de la touche du clavier qui a été actionnée par l'utilisateur avec "event.keyCode" :

```
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 programme 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.

Voici une méthode qui va vous permettre de programmer l'utilisation des touches, cette méthode est une méthode parmi beaucoup d'autres, n'hésitez pas à en changer si elle ne vous convient pas.

À faire vous même

Saisissez, analysez et testez le code suivant :

Babylon-ex8 (script.js)

```
var time=0;
var vitesseRotation=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 canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);

var camera = new BABYLON.ArcRotateCamera("Camera",0, Math.PI/4, 15, new
BABYLON.Vector3(0, 0, 0), scene);
var light = new BABYLON.PointLight("Omnidir", new BABYLON.Vector3(15, 0, 10), scene);
var maBox= new BABYLON.Mesh.CreateBox("box_1",2,scene);

scene.registerBeforeRender(function(){
    time=time+(1/BABYLON.Tools.GetFps());
    if (keys.left==1){
        maBox.rotation.y=vitesseRotation*time;
    }
    if (keys.right==1){
        maBox.rotation.y=-vitesseRotation*time;
    }
});

engine.runRenderLoop(function () {
    scene.render();
});
```

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.

À faire vous même

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.

Cliquez sur les objets

Il est possible de détecter les clics de souris sur les objets présents à l'écran. Le listener doit avoir la structure suivante :

```
window.addEventListener('click',function(event){
    var pickResult=scene.pick(event.clientX, event.clientY);
});
```

"pickResult" est un objet contenant des informations sur le mesh qui a "subi" le clic de souris :

- "pickResult.hit" est égal à "true" si le clic a été effectué lorsque le pointeur de la souris se trouvait sur un mesh et "false" dans le cas contraire.
- "pickResult.distance" donne la distance entre la caméra et le point sur lequel l'utilisateur vient de cliquer (type float)

- "pickResult.pickedMesh" si le clic a été effectué lorsque le pointeur de la souris se trouvait sur un mesh, "pickResult.pickedMesh" correspond à ce mesh (type "BABYLON.Mesh")
- "pickResult.pickedPoint" si l'utilisateur clique sur un mesh, donne les coordonnées (dans le repère local du mesh) du point cliqué. (type "Vector3" ou Null si aucun objet)

À faire vous même

Saisissez, analysez (que va faire ce programme ? Quel est l'intérêt de la variable "rotCube" ?) et testez l'exemple suivant

Babylon-ex9 (script.js)

```
var time=0;
var vitesseRotation=3;
var rotCube=0;
window.addEventListener('click',function(event){
    var pickResult=scene.pick(event.clientX, event.clientY);
    if (pickResult.hit){
        if (rotCube==0){
            rotCube=1;
        }
        else{
            rotCube=0
        }
    }
});
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);

var camera = new BABYLON.ArcRotateCamera("Camera",0, Math.PI/4, 50, new
BABYLON.Vector3(0, 0, 0), scene);
var light = new BABYLON.PointLight("OmniDir", new BABYLON.Vector3(15, 0, 10), scene);
var maBox= new BABYLON.Mesh.CreateBox("box_1",2,scene);

scene.registerBeforeRender(function(){
    time=time+(1/BABYLON.Tools.GetFps());
    if (rotCube==1){
        maBox.rotation.y=vitesseRotation*time;
    }
});

engine.runRenderLoop(function () {
    scene.render();
});
```

À faire vous même

Voici un code qui permet d'afficher 5 cubes. Ces cubes sont tous les 5 en rotation :

Babylon-ex10 (script.js)

```
var time=0;
var vitesseRotation=3;
var tabBox=[];

var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);

var camera = new BABYLON.ArcRotateCamera("Camera",0, Math.PI/4, 50, new
BABYLON.Vector3(0, 0, 0), scene);
var light = new BABYLON.PointLight("OmniDir", new BABYLON.Vector3(15, 0, 10), scene);
//Mise en place des 5 cubes
for (var i=0;i<=5;i++){
    var mabox=new BABYLON.Mesh.CreateBox("box",2,scene);
    mabox.position.z=-5+i*5;
```

```

        tabBox.push(mabox);
    }

    scene.registerBeforeRender(function() {
        time=time+(1/BABYLON.Tools.GetFps());
        //animation des 5 cubes
        for (var i=0;i<=5;i++){
            tabBox[i].rotation.y=vitesseRotation*time;
        }
    });

    engine.runRenderLoop(function () {
        scene.render();
    });

```

L'idée est relativement simple :

- on crée 5 cubes que l'on place au fur et à mesure de leur création dans un tableau ("tabBox.push(mabox)"), non sans avoir oublié au préalable de modifier leur position ("mabox.position.z=-5+i*5; "), en utilisant une boucle
- toujours à l'aide d'une boucle for, on parcourt le tableau "tabBox" afin d'animer chacun des cubes "tabBox[i].rotation.y=vitesseRotation*time;"

Après avoir analysé (et compris...) l'exemple 10, modifier ce dernier afin qu'un clic de souris sur un cube provoque sa mise en rotation (un second clic provoquant son arrêt). Bien évidemment , les 5 cubes devront être indépendants les uns des autres.

Pour vous aider :

- vous pourrez créer un tableau "tabBoxState" qui nous permettra de connaître l'état du cube i (si tabBoxState[i]=0, le cube i sera immobile, si tabBoxState[i]=1, le cube i sera en rotation)
- si le clic de souris concerne le cube i, alors "tabBox[i]==pickResult.pickedMesh" renverra true.

Chapitre 7 : créer une scène

Nous avons, pour l'instant, travaillé sur des objets, nous allons, dans ce chapitre, nous intéresser à l'environnement dans lequel nous placerons les objets.

Pour le moment nos objets "flottent dans l'air", nous allons, notamment dans un souci de réalisme, donner un support (un "sol" ou un "plancher") à nos objets.

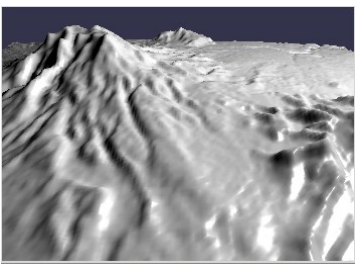
Il est bien évidemment possible d'utiliser un plan (createPlan) : attention, lors de sa création, par défaut, le plan se trouve dans le plan xy, la face visible du plan est dirigée vers les z négatifs.

HeightMap

Il est possible de rendre notre scène beaucoup plus réaliste en créant un sol avec des reliefs. Nous allons utiliser une heightMap (carte d'élévations). Une heightMap est une image en niveau de gris : à chaque pixel de l'image, on associe une valeur comprise entre 0 et 255, 0 correspond à un pixel noir, 255 correspond à un pixel blanc. Plus la valeur associée à un pixel sera importante, plus le point correspondant aura une altitude élevée dans notre scène 3D. Voici, ci-contre, un exemple de heightMap trouvé sur internet



et voici le résultat dans BabylonJS



Pour réaliser ce genre de relief, il nous faudra utiliser la méthode "CreateGroundFromHeightMap".

```
var ground = BABYLON.Mesh.CreateGroundFromHeightMap(nom,url_image,largeur,longueur,niveau_details,altitude_mini,altitude_maxi, scene, modifiable);
```

url_image : image utilisée pour le heightMap

largeur : largeur du "sol"

longueur : longueur du "sol"

niveau_details : niveau de détail

altitude_mini : altitude correspondant à un point "noir"

altitude_maxi : altitude correspondant à un point "blanc"

modifiable : le "sol" est-il modifiable (true/false)

Comme tous les mesh, il est possible d'associer un matériau à votre "sol".

À faire vous même

Après avoir récupéré un "heightMap" sur internet, créez une scène avec un sol.

SkyBox

L'idée de la SkyBox : notre "monde" sera un cube (un grand cube), nous placerons les éléments constituant notre monde (mesh, caméra, lumière...) à l'intérieur de ce cube. Les faces internes du cube seront "tapissées" d'image représentant le ciel : normalement, grâce aux outils mis en place dans BabylonJS, l'illusion devrait être "parfaite".

La mise en place de la SkyBox est simple :

```
var skybox = BABYLON.Mesh.CreateBox("skyBox", 100.0, scene);
var skyboxMaterial=new BABYLON.StandardMaterial("skybox",scene) ;
skyboxMaterial.backFaceCulling= false ;
skybox.material=skyboxMaterial ;

skyboxMaterial.diffuseColor = new BABYLON.Color3(0,0,0);
skyboxMaterial.specularColor = new BABYLON.Color3(0,0,0);

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

Quelques remarques sur ce code :

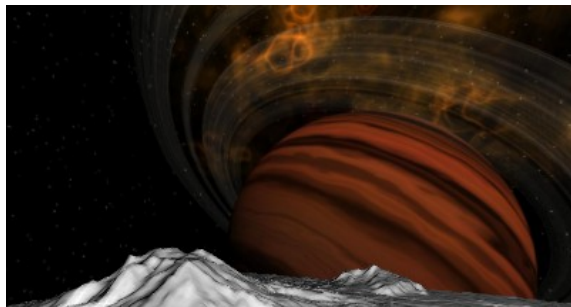
- "backFaceCulling" : permet d'éviter de calculer le rendu d'une partie invisible d'un objet. Nous devons désactiver cette fonction pour utiliser la skyBox.
- Il est important de ne pas émettre de "diffuseColor" et de "specularColor".

Vous devez télécharger les images pour la skybox, vous trouverez sans doute votre bonheur ici : <http://www.3delyvisions.com/skf1.htm>

Il faut une image par faces du cube, il vous faut donc 6 images. Ces images devront se nommer : "skybox_nx.jpg", "skybox_ny.jpg", "skybox_nz.jpg", "skybox_px.jpg", "skybox_py.jpg", "skybox_pz.jpg"

Une fois les fichiers renommés, il faudra les placer dans le répertoire "skybox" (que vous aurez créé au préalable).

En combinant le "heightmapping" et le "skyboxing", il est possible, en quelques lignes de code, d'obtenir ce genre "d'univers"



Le brouillard

Par défaut le brouillard (fog en anglais) est désactivé, nous avons :

```
scene.fogMode = BABYLON.Scene.FOGMODE_NONE (pas de brouillard)
```

Pour activer le brouillard il suffit de changer la valeur de l'attribut fogMode de l'objet scene :

```
scene.fogMode = BABYLON.Scene.FOGMODE_EXP
```

Il existe 2 autres possibilités qui pourront remplacer "BABYLON.Scene.FOGMODE_EXP" :

- "BABYLON.Scene.FOGMODE_EXP2" (la visibilité diminuera plus rapidement qu'avec "BABYLON.Scene.FOGMODE_EXP")
- "BABYLON.Scene.FOGMODE_LINEAR" (la visibilité diminuera moins rapidement qu'avec "BABYLON.Scene.FOGMODE_EXP")

Si vous utilisez "BABYLON.Scene.FOGMODE_EXP" ou "BABYLON.Scene.FOGMODE_EXP2", il est possible de modifier l'intensité du brouillard : "scene.fogDensity = 0.01" (plus la valeur sera grande moins la visibilité sera

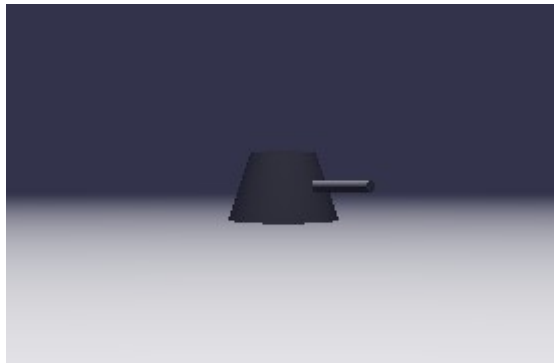
importante).

Il est également possible de jouer sur la "couleur" du brouillard : `"scene.fogColor = new BABYLON.Color3(0.9, 0.9, 0.85)"`

Miniprojet

Voici les différents "éléments" que votre scène devra incorporer :

- un "tank" (voir la capture d'écran ci-dessous)
- un sol (1 plan)
- du brouillard (pour "cacher" les limites du plan)



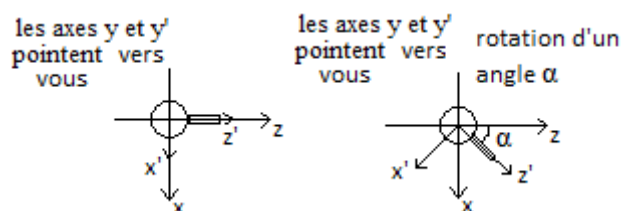
Les flèches "gauche" et "droite" permettront au "tank" d'effectuer une rotation sur lui même. La flèche "haut" permettra au tank d'avancer (toujours avec le canon vers l'avant).

Pour vous aider : le "tank" est composé de 2 cylindres.

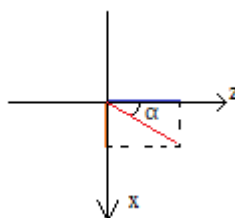
Reste maintenant à voir le difficile problème du déplacement.

N'oubliez pas qu'il existe 2 types de repère : le repère global O, x, y, z (associé à la scène) et le repère local O', x', y', z' (associé au "tank" : il se déplacera en même temps que le tank). Il est très important de réaliser que le repère local tournera en même temps que le "tank" (le canon du "tank" sera toujours confondu avec l'axe z' (mais pas avec l'axe z)).

tank vue du dessus



Après la rotation d'un angle α , l'utilisateur appuie sur la flèche "haut", les coordonnées du "tank" (dans le repère global O, x, y, z) vont donc devoir être modifiées, mais comment ?



Si le tank avance de 1 (ligne rouge), de combien doit augmenter la coordonnée z (ligne bleue) ? Et de combien doit augmenter la coordonnée x (ligne orange) ?

Nous allons devoir faire un peu de trigonométrie.



J'espère que vous avez remarqué que nous avons un triangle rectangle : la ligne rouge est l'hypoténuse, la ligne bleue est le côté adjacent et le côté orange est le côté opposé. Donc :

$\cos \alpha = \text{ligne bleue} / \text{ligne rouge}$ et $\sin \alpha = \text{ligne orange} / \text{ligne rouge}$

donc si le tank avance de 1 (ligne rouge) alors la coordonnée z devra augmenter de $\cos \alpha$ et la coordonnée x de $\sin \alpha$

Évidemment, quelque soit la position du tank et quelques soit l'angle α (dans BabylonJS : `tank.rotation.y`), ce raisonnement est valable.

Comment cela va se traduire dans BabylonJS ?

Partons du principe qu'en cas d'appui sur le "flèche haut", à chaque image, le tank avance de 1. À chaque image il faudra donc modifier les coordonnées (globales) comme suit :

`tank.position.x = tank.position.x + Math.sin(tank.rotation.y)`

`tank.position.z = tank.position.z + Math.cos(tank.rotation.y)`

À vous de jouer...

Chapitre 8 : un peu de physique

BabylonJS nous permet de gérer la gravitation et les collisions (ce que l'on appelle généralement la "physique" dans les jeux vidéo).

Depuis la version 1.8.0, BabylonJS intègre le moteur physique cannonJS, cette intégration permet de rendre les scènes de manière extrêmement réaliste...revers de la médaille, la gestion de la "physique" par cannonJS est assez gourmande en matière de puissance de calcul et demande donc une machine assez "costaude".

En général, ce n'est pas vraiment le cas des ordinateurs dont nous disposons dans les établissements du secondaire, voilà pourquoi j'ai divisé ce chapitre en 2 parties : la première partie sera consacrée à la gestion de la "physique" sans cannonJS (beaucoup plus limiter, mais aussi beaucoup moins "gourmande"), la deuxième partie abordera cette même gestion de la physique, mais cette fois en utilisant les outils proposés par cannonJS.

La "physique" dans BabylonJS sans cannonJS

Un seul type d'objets peut se voir soumis à la gravitation : les caméras.

Vous activerez la "gravitation" pour la caméra avec les 3 lignes suivantes :

```
scene.gravity = new BABYLON.Vector3(0, -1, 0);
camera.applyGravity = true;
camera.checkCollisions = true;
```

"scene.gravity" : vous activez la gravité dans votre scène. Le "BABYLON.Vector3" représente le vecteur intensité de la pesanteur, habituellement noté g .

Si la notion de vecteur ne vous est pas familière, sachez que vous devez avoir un "Vector3" avec les coordonnées x et z à 0 et la coordonnée y négative (plus cette coordonnée y sera "négative" et plus votre objet "tombera vite" (physiquement c'est un peu plus complexe, mais ce document n'est pas un cours de physique), la caméra tombera moins vite avec un "Vector3(0, -1, 0)" qu'avec un "Vector3(0, -10, 0)").

Si vous voulez simuler la gravité terrestre, il faudra avoir "Vector3(0, -9.8, 0)".

Évidemment il est possible de simuler un monde avec des lois physiques complètement différentes, par exemple avec une caméra qui "tombera" à l'horizontale : "Vector3(0, 0, 2)" ou une caméra qui "tombera" vers le haut :

"Vector3(0, 2, 0)".

Je précise que tout ceci est exact si votre "sol" se trouve dans le plan xOz et que les " y " positifs sont vers le haut.

"camera.applyGravity" : permet d'activer la gravitation pour votre caméra

"camera.checkCollisions" : permet d'activer les collisions pour la caméra (nous aurons l'occasion de revenir sur les collisions un peu plus bas).

À faire vous même

Babylon-ex10 (script.js)

Saisissez, analysez (que va faire ce programme ? Quel est l'intérêt de la variable "rotCube" ?) et testez l'exemple suivant

```
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);
var camera = new BABYLON.FreeCamera("FreeCamera", new BABYLON.Vector3(0, 30, 0), scene);
camera.attachControl(canvas);
var light = new BABYLON.PointLight("pointLumineux1", new BABYLON.Vector3(100, 100, 0), scene);
var plan=BABYLON.Mesh.CreatePlane("sol",200,scene);
plan.rotation.x=Math.PI/2;
```

```

var cube=BABYLON.Mesh.CreateBox("boite",5.0,scene)
cube.position.z=40;
cube.position.y=2;
var mat= new BABYLON.StandardMaterial("matCube",scene);
mat.diffuseColor = new BABYLON.Color3(1,0,0);
cube.material=mat;
scene.gravity = new BABYLON.Vector3(0, -1, 0);
camera.applyGravity = true;
camera.checkCollisions = true;
engine.runRenderLoop(function () {
    scene.render();
});

```

Pour que la caméra (FreeCamera) tombe, il faut très légèrement "bouger" (à l'aide des flèches).

Vous constatez "avec horreur" que vous traversez le sol et que vous poursuivez votre chute "éternellement" (cauchemars récurrent pour de nombreuses personnes).

À faire vous même

Reprenez l'exemple précédent (Babylon-ex10) et ajouter la prise en charge des collisions pour le sol (votre sol deviendra "consistant") => "plan.checkCollisions = true".

Vous devez également ajouter les lignes suivantes :

"camera.ellipsoïde = new BABYLON.Vector3(1, 1, 1)" : permet de donner un volume à votre caméra (un objet avec un volume égal à zéro (!?) ne peut pas entrer en collision avec un autre objet)

"scene.collisionsEnabled = true" : permet de rendre les collisions actives dans la scène

Vérifiez que maintenant vous ne passez plus au travers du sol.

Si maintenant vous vous déplacez, vous remarquerez sans doute qu'il vous est possible de traverser le "cube rouge".

À faire vous même

Modifiez le code précédent (celui basé sur Babylon-ex10) afin d'activer la collision entre la caméra et le "cube rouge".

Collision entre deux objets

Il est possible de détecter la collision entre 2 objets grâce à la méthode "intersectsMesh" :

```
objet1.intersectsMesh(objet2,true);
```

Si "objet1" entre en contact avec "objet2" (ou vis versa) la méthode "intersectsMesh" renverra true (sinon elle renverra false). Quand le deuxième paramètre de la méthode est à true, la détection des collisions est plus précise (mais aussi plus gourmande en terme de calculs).

À faire vous même

En vous inspirant de l'exemple "Babylon-ex10", créer une scène avec une sphère (sphere) rouge qui tombe vers le sol. Une fois au sol (plan), la sphère devra changer de couleur.

Attention pour éviter d'avoir des problèmes, veuillez à rajouter, juste avant la ligne "engine.runRenderLoop", les 2 lignes suivantes :

```

sphere.computeWorldMatrix(true);
plan.computeWorldMatrix(true);

```

Ces 2 lignes "obligent" BabylonJS à effectuer certains calculs avant le rendu de la première image.

À faire vous même

Variante de l'exemple précédent : la sphère devra disparaître 2 secondes après avoir touché le sol.

Pour vous aider :

La méthode "dispose" ("sphere.dispose()") permet de supprimer un objet (ici l'objet "sphere").

Il est possible de tester l'existence d'un objet avec la méthode "scene.getMeshByName("sph")", "sph" est le nom de l'objet testé.

La méthode "setTimeout" vous permet d'exécuter une fonction après un certain délai :

```

window.setTimeout(function() {
    //cette fonction sera exécutée après un délai de 5 secondes
},5000)

```


Pour plus d'informations sur l'utilisation de setTimeout : <https://developer.mozilla.org/fr/docs/DOM/window.setTimeout>

Miniprojet

Reprenez le dernier "**A faire vous même**" et faites les modifications suivantes :

votre scène devra comporter 10 sphères, la position des sphères devra être aléatoire (en restant dans certaines limites : les sphères devront se trouver au-dessus du sol).

Les vitesses de chute des différentes sphères devront, elles aussi, être aléatoires (différentes pour chaque sphère).

Pour aller plus loin :

Il est aussi possible de simuler plus fidèlement la chute des sphères. En effet, les objets ne tombent pas à vitesse constante (si on ne tient pas compte des frottements).

Voici l'équation qui donne l'altitude d'un objet en fonction du temps t :

$$y = - \frac{1}{2} * g * t^2 + y_0$$

avec y l'altitude de l'objet (y est positif), g est l'intensité de la pesanteur (normalement $9,8 \text{ N.Kg}^{-1}$, mais dans BabylonJS, je vous conseille de prendre une valeur inférieure), y_0 est l'altitude de l'objet à $t = 0$

La "physique" dans BabylonJS avec cannonJS

Avant de pouvoir utiliser cannonJS il va vous falloir récupérer le fichier cannon.js sur le dépôt github de BabylonJS : <https://github.com/BabylonJS/Babylon.js>

Une fois ce fichier récupéré, placez-le dans le dossier lib et ajoutez, la ligne suivante à votre code HTML (fichier index.html) : `<script src="lib/cannon.js"></script>`

Le contenu de votre fichier index.html devrait ressembler à cela :

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <title>Titre de la page</title>
  <link rel="stylesheet" href="css/style.css">
  <script src="lib/cannon.js"></script>
  <script src="lib/babylon.1.8.0.js"></script>
</head>
<body>
  <canvas id="renderCanvas"></canvas>
</body>
<script src="javascript/script.js"></script>
</html>
```

Nous allons maintenant pouvoir profiter de la puissance du moteur physique cannonjs

La première chose à faire est d'activer le moteur physique :

```
scene.enablePhysics();
```

"scene" correspondant à l'objet de type Scene créé avec `var scene = new BABYLON.Scene(engine);`

Si vous désirez que les objets présents dans votre scène soient soumis à l'interaction, il faut définir un vecteur "intensité de la pesanteur" :

```
scene.setGravity(new BABYLON.Vector3(0,-10,0));
```

Le `"Vector3(0,-10,0)"` correspond bien à un vecteur vertical, dirigé vers le bas, d'intensité 10 N.Kg^{-1} (à la place du classique $9,8 \text{ N.Kg}^{-1}$)

Ensuite, il suffit de créer un objet classique, par exemple une sphère :

```
var sph=BABYLON.Mesh.CreateSphere("sphere",20,4,scene);
```

Si vous testez votre programme tel quel, il ne se passera rien de spécial, il faut ajouter ce que l'on appelle un "rigidbody" à notre objet :

le moteur physique n'applique pas les lois physiques directement à un objet de type "mesh", il les applique à un objet spécial que l'on appelle donc un "rigidbody".

Il suffit d'associer un "rigidbody" à un "mesh" pour avoir l'impression que les lois physiques s'appliquent à notre objet de type "mesh" (vous pouvez voir le "rigidbody" comme un objet invisible entourant l'objet de type "mesh" avec lequel il a été associé).

```
sph.setPhysicsState({ impostor: BABYLON.PhysicsEngine.SphereImpostor, mass: 2, friction: 5, restitution: 0 });
```

La méthode "setPhysicsState" permet de créer un "rigidbody" et de l'associer à un objet de type "mesh" (ici l'objet "sph").

Cette méthode prend en paramètre un objet JavaScript.

Le premier attribut de cet objet JavaScript est "impostor", il permet de définir la géométrie du "rigidbody". BabylonJS vous propose des "impostor" clé en main :

- pour les sphères ("BABYLON.PhysicsEngine.SphereImpostor")
- pour les cubes ("BABYLON.PhysicsEngine.BoxImpostor")
- pour les plans ("BABYLON.PhysicsEngine.PlaneImpostor")

d'autres "impostors clé en main" devraient être proposés dans les versions ultérieures de BabylonJS.

Le deuxième attribut, "mass", correspond...à la masse de l'objet

Le troisième attribut, "friction", permet de définir l'intensité des forces de frottements qui s'exerceront sur l'objet.

Le quatrième attribut, "restitution", est lié à la capacité d'un objet à rebondir : plus la valeur est grande, plus l'objet aura tendance à rebondir.

Les 2 derniers attributs ne sont pas obligatoires.

À faire vous même

Créez une scène comportant une sphère. Vous devrez "activer" la gestion de la physique par cannonJS. La sphère devra "obéir" aux lois physiques (gravitation avec une intensité de la pesanteur $g=10 \text{ N.Kg}^{-1}$)

À faire vous même

Vous avez sans doute remarqué que la sphère, précédemment créée, chute sans jamais s'arrêter. Nous allons donc rajouter un "sol".

Nous n'allons pas utiliser un plan pour notre sol, mais un cube très aplati :

```
var ground=BABYLON.Mesh.CreateBox("sol",100,scene);
ground.scaling.y=0.001
```

Pour jouer son rôle de sol, notre "cube aplati" doit posséder un rigidbody :

```
ground.setPhysicsState({ impostor: BABYLON.PhysicsEngine.BoxImpostor, mass: 0, friction: 5, restitution: 0 });
```

Vous remarquerez que la masse de notre "sol" est égale à 0 : ceci permet d'éviter que notre sol chute vers le bas.

Mettez en place une scène avec un sol et une sphère située à quelques mètres au-dessus du sol.

Vous devriez constater la chute de la sphère vers le sol (et éventuellement son rebond sur ce dernier).

Modifiez les différents paramètres du "rigidbody" (du sol et de la sphère) afin de vous familiariser avec ces différents attributs.

À faire vous même

Créer une scène avec un sol, des sphères et des cubes. Vous mettrez en place des "rigidbody" pour tous ces objets.

Admirez la qualité de la simulation des chocs entre les différents éléments de votre scène. Ici aussi, n'hésitez pas à modifier les paramètres des "rigidbody".

Attention : Quand vous désirez placer des objets au sol, prenez garde à la disposer très légèrement au-dessus du sol. En effet, si les 2 "rigidbody" (celui de l'objet et du sol) s'interpénètrent, cela provoque un bug d'affichage (vous ne verrez pas votre objet).

Application d'une force sur un objet

En physique on modélise la force qu'exerce un objet A sur un autre objet B par un objet mathématique vecteur que l'on dénomme "vecteur force".

Ce "vecteur force" possède 4 caractéristiques :

- sa direction
- son sens
- son intensité (valeur de la force en Newton)
- son point d'application

BabylonJS (avec l'aide de cannonJS), nous permet d'appliquer une force à un objet : la méthode utilisée est

"applyImpulse"

```
obj.applyImpulse(vecteurForce,pointApplicationForce);
```

avec :

obj => l'objet sur lequel la force est appliquée

vecteurForce => c'est un vecteur donc un objet "new BABYLON.Vector3(...)"

pointApplication => le point d'application de la force (donc ici aussi un objet "new BABYLON.Vector3(...)"

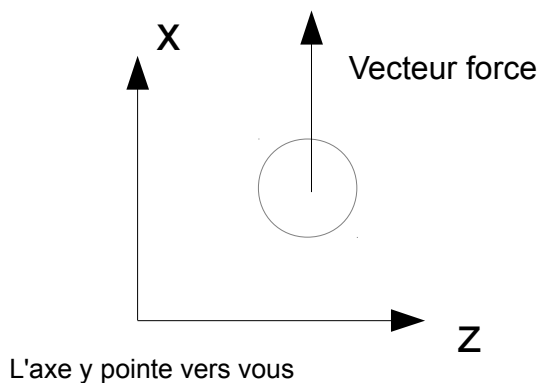
ATTENTION

"vecteurForce" et "pointApplication" devront être donnés dans le repère global (pas dans le repère local associé à l'objet concerné par la force). La non-prise en compte de cette information peut entraîner des comportements, à priori, très étranges lors de l'application d'une force sur un objet

conséquence, si vous désirez appliquer une force au centre d'un objet il vous faudra utiliser les coordonnées de l'objet (obj.position), voici un exemple avec un objet de type Sphere :

```
"sph.applyImpulse(new BABYLON.Vector3(15,0,0), sph.position);" aura pour conséquence :
```

Scène vue du dessus



Attention, le point d'application est très important, un autre choix de point d'application n'aurait pas du tout les mêmes conséquences : vous pourriez avoir, par exemple, une rotation de l'objet au lieu d'une translation (déplacement).

Autre point qu'il faut avoir à l'esprit : la force est appliquée pendant une durée correspondant à `1/BABYLON.Tools.GetFps()` seconde, c'est-à-dire la durée qui s'écoule entre 2 images.

À faire vous même

Écrire un mini "jeu" de bowling avec une boule qui renverse des quilles (les quilles pourront être des "cubes allongés en hauteur").

Chapitre 9 : importation depuis Blender

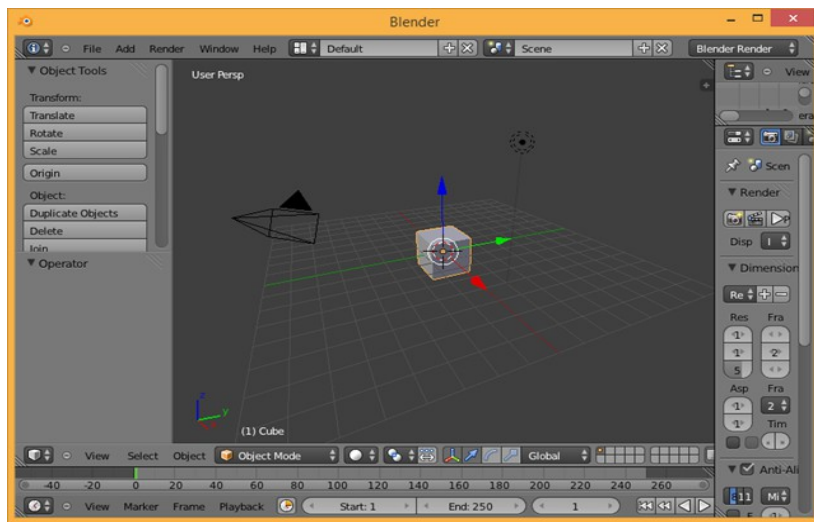
La création d'objets (mesh) et l'application de textures complexes directement dans BabylonJS peuvent très rapidement se montrer difficiles. Il est beaucoup plus "simple" de créer des objets complexes en utilisant un modèleur 3D comme le très connu Blender (libre et gratuit) : <http://www.blender.org/>

Je n'ai pas l'intention d'écrire ici un cours sur l'utilisation de Blender, le sujet est bien trop vaste. De plus, vous trouvez très facilement une foultitude de cours et de tutoriaux consacrés à l'utilisation de ce logiciel :

<http://fr.openclassrooms.com/informatique/cours/debutez-dans-la-3d-avec-blender>

http://www.youtube.com/watch?v=_DYSSi_HRCU

<http://fr.tuto.com/blender/tuto-blender-gratuit.htm>

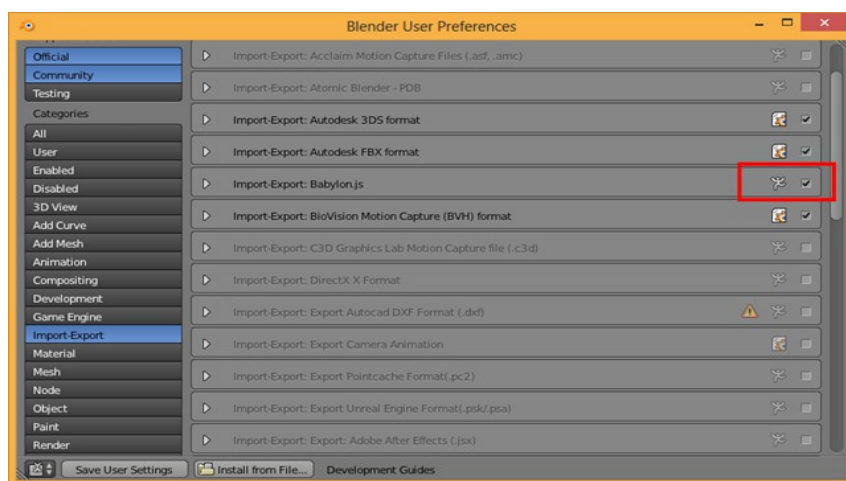


Les développeurs de BabylonJS ont créé un outil d'export permettant d'importer des objets ou des scènes complètes de Blender vers BabylonJS. Commençons par installer cet outil dans Blender :

Télécharger l'outil `io_export_babylon.py` : <https://github.com/BabylonJS/Babylon.js/tree/master/Exporters/Blender>

Placer ce fichier dans le répertoire `Blender\2.XX\scripts\addons`

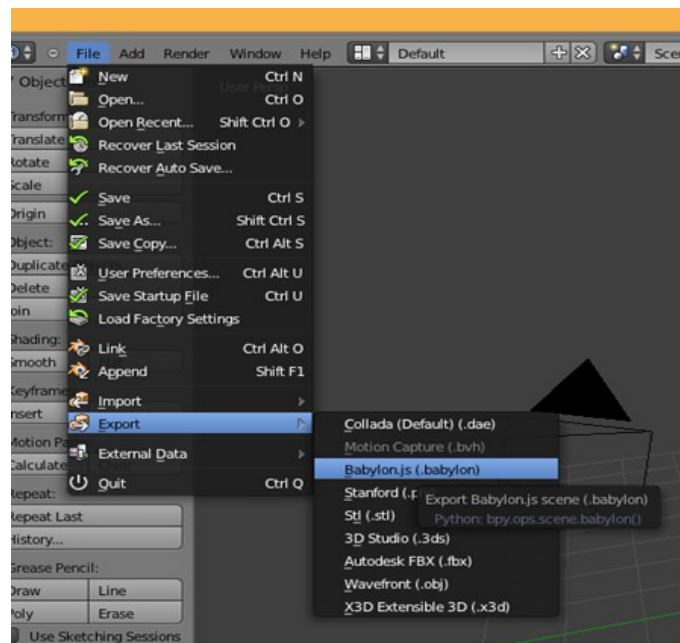
Lancer Blender et placez-vous dans `File/User Preferences/Addons`, cliquer ensuite sur la catégorie `Import-Export`



Cocher la case Import-Export Babylon.js

Importation d'objets (mesh)

Après avoir terminé votre scène, aller dans File/Export



Une fois Export Babylon.js scene (.babylon) sélectionné, vous devriez pouvoir sauvegarder un fichier possédant une extension .babylon. C'est ce fichier qui sera utilisé par BabylonJS.

À faire vous même

Étudiez l'exemple 11

Pour vous aider :

Après la mise en place (classique) de votre scène, la méthode "ImportMesh" vous permet d'importer les meshes (en utilisant un fichier de type .babylon).

Voici les paramètres de la méthode "ImportMesh" :

- le premier paramètre correspond au dossier contenant votre scène, dans la plupart des cas, une chaîne vide ("") fera parfaitement l'affaire.
- le deuxième paramètre vous permet d'indiquer le dossier qui contient le fichier importé depuis Blender (.babylon) => ici ""asset/""
- le troisième paramètre correspond au nom du fichier importé (ici "monFichier.babylon"). Ce fichier a été obtenu à partir d'une scène Blender contenant une tête de singe
- le quatrième paramètre est le nom de votre scène
- le cinquième paramètre est très important, c'est une fonction de callback

Cette fonction de callback (fonction qui sera exécutée quand les objets auront tous été "chargés" dans votre scène) prend 2 paramètres : un tableau contenant les objets importés (`newMeshes`) et un tableau contenant les systèmes de particules importés (`particleSystems`) (je ne développerai pas la notion de "système de particules" ici).

La fonction de callback appelle la fonction `game` (en passant en paramètre le tableau "`newMeshes`"). Cette fonction, qui est donc appelée une fois que les différents meshes ont été chargés (il ne faut pas que le rendu de la scène commence avant que ce chargement soit terminé), contient le reste de notre code.

Il est important de remarquer la ligne "`singe=newMeshes[0]`" qui permet de "récupérer" notre mesh "singe".

Babylon-ex11 (script.js)

```
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
```

```

var scene = new BABYLON.Scene(engine);
var singe;
var camera = new BABYLON.ArcRotateCamera("Camera",Math.PI/2,Math.PI/2, 15, new
BABYLON.Vector3(0, 0, 0), scene);
var light0 = new BABYLON.PointLight("Omnidir", new BABYLON.Vector3(60, 0,60), scene);
camera.attachControl(canvas);
BABYLON.SceneLoader.ImportMesh("", "asset/", "monFichier.babylon", scene, function
(newMeshes, particleSystems) {
    game(newMeshes)
});
var game=function(newMeshes){
    singe=newMeshes[0];
    singe.rotation.x=-Math.PI/2;
    singe.rotation.z=Math.PI;
    engine.runRenderLoop(function () {
        scene.render();
    });
}

```

NB : si, dans Blender, vous avez appliqué une texture à votre objet, le fichier image ayant été utilisée comme texture devra se trouver dans le même dossier que le fichier .babylon.

Avant de pouvoir visualiser le résultat de cet exemple, vous allez devoir, pour une raison que je ne développerai pas ici, mettre en place un "serveur web local" sur l'ordinateur que vous utilisez pour développer votre application BabylonJS. Si vous êtes capable de mettre en place ce genre de serveur en "local" (à l'aide de Wamp ou autres) les lignes qui suivent ne vous intéresseront pas.

En revanche, si vous n'avez aucune connaissance sur ce sujet, je vous propose de mettre en place un "serveur web local" programmé en JavaScript à l'aide de nodeJS (ExpressJS plus précisément).

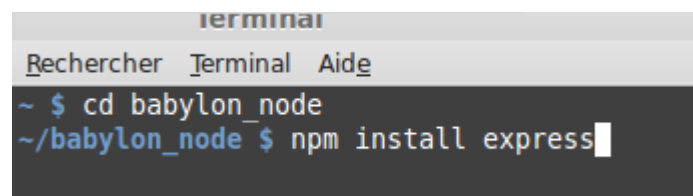
Pas question d'entrer dans les détails, les explications qui suivent seront purement "fonctionnelles" (si ce sujet vous intéresse, je vous propose de lire un document rédigé par mes soins disponible ici :

<http://www.webisn.byethost5.com/jquery>)

Commencez par installer nodeJS : <http://nodejs.org/>

Créez un dossier (par exemple babylon_node)

Ouvrez une console (terminal sous windows) et placez-vous dans le dossier que vous venez de créer (n'hésitez pas à demander de l'aider à votre enseignant). Une fois dans le dossier, tapez "npm install express" dans la console.



Vous devriez voir apparaître un dossier "node_modules"

Toujours dans le dossier babylon_node, créez un dossier "app" et un fichier "server.js" :

Vous devriez avoir, dans le dossier "babylon_node" la structure suivante :



Le dossier "app" contiendra les dossiers et fichiers suivants :



Nous avons donc exactement la même structure qu'auparavant.

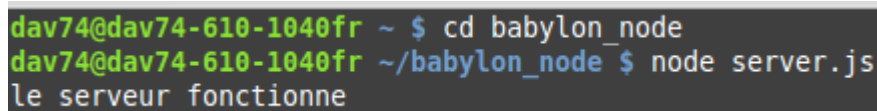
Voici le contenu du fichier server.js :

```
var express = require('express');
var app = express();
app.use('/javascript', express.static(__dirname + '/app/javascript'));
app.use('/lib', express.static(__dirname + '/app/lib'));
app.use('/css', express.static(__dirname + '/app/css'));
app.use('/asset', express.static(__dirname + '/app/asset'));
app.get('/', function (req, res) {
    res.sendFile(__dirname + '/app/index.html');
});
app.listen(8080);
co,sole.log("Le serveur fonctionne")
```

Pas question pour moi de commenter ce code, encore une fois, le but ici est uniquement de mettre en place un "serveur web".

Il nous reste maintenant à lancer le serveur :

Après avoir ouvert la console, placez-vous dans le dossier babylon_node et tapez "node server.js"



```
dav74@dav74-610-1040fr ~ $ cd babylon_node
dav74@dav74-610-1040fr ~/babylon_node $ node server.js
le serveur fonctionne
```

si tout se passe bien, vous devriez avoir dans la console le message "le serveur fonctionne"

Il vous reste à ouvrir votre navigateur préféré et à taper, dans la barre d'adresse "localhost:8080" ; votre scène devrait alors apparaître.

Importer une scène complète

Il est possible d'importer une scène complète (avec les mesh, les lumières et les caméras) depuis Blender : le rendu dans votre navigateur sera (normalement) identique que le rendu dans Blender.

À faire vous même

Étudiez l'exemple 12

Pour vous aider :

La méthode "Load" permet de charger la scène. Cette méthode prend 4 paramètres :

- "asset/" le dossier qui contient le fichier .babylon
- "monFichier.babylon" le nom du fichier .babylon
- "engine" l'objet de type engine
- "function (newScene)" la fonction de callback

La fonction de callback lance une seconde fonction de callback ("executeWhenReady") qui est exécutée quand la scène est prête à être affichée (cette fonction de callback exécute notre fonction game). Le paramètre "newScene" est un objet de type "scene".

Vous remarquerez que nous avons créé ni scène, ni caméra et ni éclairage, tout est inclus dans notre fichier .babylon

Babylon-ex12 (script.js)

```
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
BABYLON.SceneLoader.Load("asset/", "monFichier.babylon", engine, function (newScene) {
    newScene.executeWhenReady(function () {
        game(newScene);
    });
});
var game=function(scene){
    scene.activeCamera.attachControl(canvas);
    engine.runRenderLoop(function() {
        scene.render();
    });
}
```

Afficher une scène créée dans Blender, dans un navigateur c'est déjà pas mal, mais il est possible de faire encore "mieux" : manipuler les objets de la scène.

À faire vous même

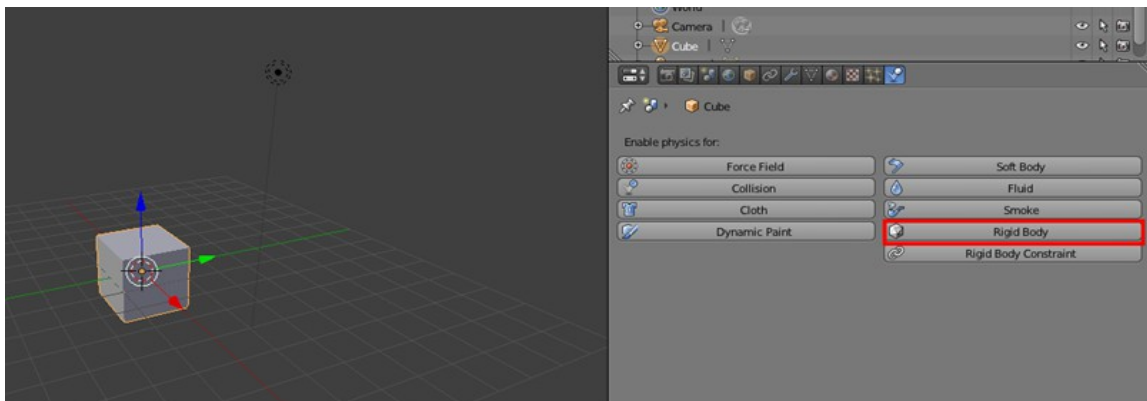
Étudiez l'exemple 13

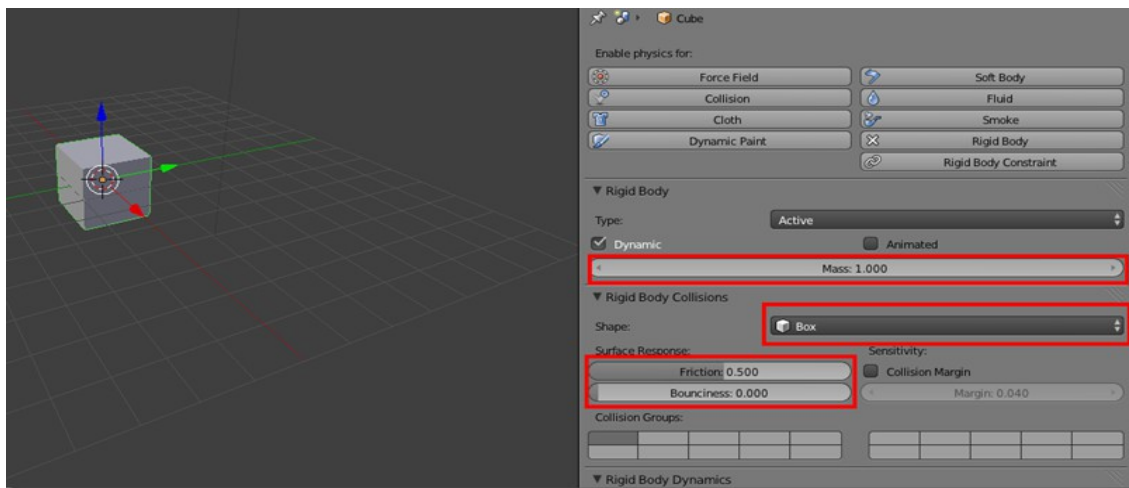
Pour vous aider : les mesh qui "peuplent" notre scène sont accessibles grâce au tableau "scene.meshes"

Babylon-ex13 (script.js)

```
var canvas = document.getElementById("renderCanvas");
var engine = new BABYLON.Engine(canvas, true);
BABYLON.SceneLoader.Load("asset/", "monFichier.babylon", engine, function (newScene) {
    newScene.executeWhenReady(function () {
        game(newScene);
    });
});
var game=function(scene){
    var time=0;
    scene.activeCamera.attachControl(canvas);
    var tabMesh=scene.meshes;
    singe=tabMesh[0];
    scene.registerBeforeRender(function(){
        singe.rotation.y=0.5*time;
        time=time+(1/BABYLON.Tools.GetFps());
    });
    engine.runRenderLoop(function() {
        scene.render();
    });
}
```

Blender permettant de définir des "rigidbody" (voir le chapitre sur la physique), il est possible d'exporter les caractéristiques physiques d'un objet dans un fichier .babylon





Attention : Blender ne permet pas de définir une masse à 0. Cependant, si vous mettez dans Blender une valeur inférieure ou égale à 0,001, elle sera considérée comme nulle dans BabylonJS.

2e partie : pour aller plus loin

Chapitre 10 : Programmation orientée objet

Nous allons provisoirement « abandonner » BabylonJS pour nous intéresser à la programmation orientée objet en JavaScript et développer les concepts vu dans le document "Apprendre la programmation avec JavaScript".

Vous devez déjà savoir qu'un objet possède des attributs et des méthodes, mais je préfère rappeler ici quelques bases en reprenant l'exemple de la voiture :

Une voiture a une marque et une couleur (entre autres) et bien "marque" et "couleur" seront des attributs de notre objet JavaScript voiture.

Une voiture peut accélérer, freiner, tourner..... et bien "freiner", "accélérer" et "tourner" seront des méthodes de notre objet JavaScript voiture.

Avant d'aller plus loin, faisons une parenthèse. Si parmi vous certains ont déjà eu l'occasion de rencontrer la notion d'objet, notamment en travaillant sur des langages comme le Java ou le C++, il est important que vous sachiez que JavaScript est un langage orienté objet par prototype. Ce mot prototype change beaucoup de choses, je vous invite donc à "oublier" (provisoirement) ce que vous avez déjà appris sur les objets (classe, instance,...), nous allons tout reprendre à zéro

Comment "fabrique"-t-on un objet ?

Nous en avons déjà vu une dans le document "Apprendre la programmation avec JavaScript" :

```
maVoiture = {  
    marque : 'Peugeot',  
    couleur : 'rouge',  
    annee : 2012,  
    accelere : function(){  
        document.write('La voiture accélère') ;  
        document.write('<br/>') ;  
    }  
}
```

Je vous rappelle que "marque", "couleur" et "annee" sont des attributs, "accelere" est une méthode.

Dans la suite de ce chapitre nous allons étudier une autre façon de créer des objets.

Avant de pouvoir créer l'objet en tant que tel, nous allons devoir construire un moule. À partir de ce moule, nous pourrons fabriquer autant d'objets que nécessaire. En POO (programmation orientée objet), ce moule s'appelle un constructeur. Un constructeur est une méthode qui porte le même nom que votre "classe" (dans notre cas Voiture). Le nom du constructeur doit commencer par une majuscule.

Un exemple avec, pour commencer, uniquement des attributs (nous verrons les méthodes plus loin)

```
function Voiture (marque, couleur) {  
    this.marque = marque;  
    this.couleur = couleur ;  
}
```

Vous pouvez constater que notre méthode "Voiture" (notre constructeur d'objet de type voiture) peut posséder des arguments (marque et couleur). Ces arguments seront renseignés au moment de la création de nos objets voitures (souvenez-vous que pour l'instant nous avons uniquement fabriqué le moule).

Passons maintenant au contenu de ce constructeur. Vous avez sans doute remarqué le "this.couleur" et le "this.marque".

"this" est un mot clé très important en POO, il désigne l'objet courant, celui que l'on est en train d'utiliser. Oui, je sais, c'est un peu compliqué, nous reviendrons dessus un peu plus tard.

Maintenant que notre "moule" (constructeur) est prêt, passons à la création de notre premier objet.

```
function Voiture (marque, couleur) {
    this.marque = marque;
    this.couleur = couleur ;
}
var voit_1 = new Voiture ("Fiat", "rouge");
```

Nous avons créé une "variable" (voit_1) contenant un objet de type Voiture, nous avons donc créé un objet voit_1 qui est de type voiture. Pour créer cet objet, nous avons utilisé le mot clé "new" suivi du nom du constructeur (Voiture). Comme pour n'importe quelle fonction, les paramètres "Fiat" et "rouge" sont mis dans la parenthèse qui suit le nom du constructeur.

Nous pouvons créer un deuxième objet de type voiture :

```
function Voiture (marque, couleur) {
    this.marque = marque;
    this.couleur = couleur ;
}
var voit_1 = new Voiture ("Fiat", "rouge");
var voit_2 = new Voiture ("Renault", "verte");
```

Nous avons maintenant 2 "voitures" à notre disposition : une de marque Fiat et de couleur rouge et une de marque Renault et de couleur verte.

Avoir créé ces 2 objets c'est bien, mais comment les utiliser ?

Il est relativement simple d'accéder aux attributs d'un objet :

nom_de_l'objet.nom_de_l'attribut

La notation pointée (c'est son nom) n'est pas la seule possibilité, mais c'est la seule que nous verrons ici.

Dans notre exemple pour accéder à l'attribut couleur de l'objet voit_1, il faudra écrire :

voit_1.couleur

de même, pour accéder à l'attribut marque de l'objet voit_2 :

voit_2.marque

Voici un exemple :

```
function Voiture (marque, couleur) {
    this.marque = marque;
    this.couleur = couleur ;
}
var voit_1 = new Voiture ("Fiat", "rouge");
document.write("Je viens de créer une nouvelle voiture, elle est ",voit_1.couleur,". Sa marque est ", voit_1.marque);
var voit_2 = new Voiture ("Renault", "verte");
document.write("Je viens de créer une nouvelle voiture, elle est ",voit_2.couleur,". Sa marque est ", voit_2.marque);
```

résultat

Je viens de créer une nouvelle voiture, elle est rouge. Sa marque est Fiat

Je viens de créer une nouvelle voiture, elle est verte. Sa marque est Renault

Par souci de simplification je n'ai pas codé les retours à la ligne. Si vous testez ces exemples avec le code tel quel, il n'y aura pas de retour à la ligne.

Il est bien évidemment possible de modifier l'attribut d'un objet (comme n'importe quelle variable) :

```
function Voiture (marque, couleur) {
    this.marque = marque;
    this.couleur = couleur ;
}
var voit_1 = new Voiture ("Fiat", "rouge");
document.write("Je viens de créer une nouvelle voiture, elle est ",voit_1.couleur,". Sa marque est ", voit_1.marque);
var voit_2 = new Voiture ("Renault", "verte");
document.write("Je viens de créer une autre voiture, elle est ",voit_2.couleur,". Sa marque est ", voit_2.marque);
```

```
voit_1.couleur="jaune";
document.write("Je viens de repeindre ma ",voit_1.marque, ", elle est maintenant ",voit_1.couleur);
```

résultat

Je viens de créer une nouvelle voiture, elle est rouge. Sa marque est Fiat
Je viens de créer une autre voiture, elle est verte. Sa marque est Renault
Je viens de repeindre ma Fiat, elle est maintenant jaune

Il est maintenant possible de mieux comprendre l'utilité du mot clé this. Au moment de la création de l'objet voit_1, le mot clé this a systématiquement été remplacé par "voit_1" (même chose pour "voit_2"). Cela correspond bien à la "définition" que je vous ai donnée un peu au-dessus, "this correspond à l'objet courant" ("this=voit_1" quand on crée ou qu'on utilise voit_1 et "this=voit_2" quand on crée ou qu'on utilise l'objet voit_2).

Avant de parler des méthodes, nous devons nous intéresser aux prototypes :

En JavaScript, tous les objets sont liés à un autre objet appelé prototype. Chaque objet possède son propre prototype. Si vous créez un objet de type Voiture, le prototype de Voiture sera accessible tout simplement en écrivant « Voiture.prototype ».

À quoi sert le prototype ?

Question complexe, j'aurais l'occasion d'y répondre un peu plus loin, quand nous aborderons la notion d'héritage. Pour l'instant, vous devez uniquement connaître la procédure à suivre pour créer une méthode :

La méthode « n'appartiendra » pas directement à l'objet, mais « appartiendra » à son prototype (ne surtout pas perdre de vue qu'un prototype est un objet comme un autre, il peut donc posséder des attributs et des méthodes), pour définir une méthode « accelere » utilisable par les objets de type voiture, il faudra écrire :

```
Voiture.prototype.accelere=function() {...}
```

Voici un exemple

```
function Voiture (marque, couleur) {
    this.marque = marque;
    this.couleur = couleur ;
}
Voiture.prototype.accelere = function () {
    document.write ("Votre voiture de marque ",this.marque," accélère");
}
var voit_1 = new Voiture ("Fiat", "rouge");
voit_1.accelere();
```

résultat

Votre voiture de marque Fiat accélère

Un autre exemple

```
function Voiture (marque, couleur) {
    this.marque = marque;
    this.couleur = couleur ;
}
Voiture.prototype.accelere = function () {
    document.write ("Votre voiture de marque ",this.marque," accélère") ;
}
var voit_1 = new Voiture ("Fiat", "rouge");
var voit_2 = new Voiture ("Renault", "verte");
voit_1.accelere();
voit_2.accelere();
```

résultat

Votre voiture de marque Fiat accélère
Votre voiture de marque Renault accélère

Remarquez bien l'utilisation du this dans la méthode accelere (). Comme pour les attributs, le this va successivement être « remplacé » par voit_1 puis par voit_2.

Revenons sur le processus mis en œuvre ici :

Au moment de l'utilisation de la méthode `accelere` (par exemple «`voit_1.accelere();`»), le moteur JavaScript va chercher une méthode `accelere` dans l'objet Voiture (il ne faut pas perdre de vue que `voit_1` est un objet de type Voiture). Ne trouvant pas de méthode `accelere` dans l'objet Voiture, il va étendre sa recherche au prototype de l'objet Voiture. Il va, dans cet exemple, trouver la méthode recherchée dans ce prototype. S'il ne l'avait pas trouvée, il aurait poursuivi sa recherche dans le prototype du prototype de l'objet voiture..... Au bout d'un moment, il serait arrivé au prototype de l'objet Object (objet de base du JavaScript) et si une fois de plus sa recherche s'était révélée infructueuse, il aurait alors retourné une erreur (du type « méthode non définie »). Tout ce processus utilise ce que l'on appelle « la chaîne des prototypes ».

Je précise qu'il est possible de définir les méthodes directement dans le constructeur de l'objet Voiture, mais ce procédé est déconseillé (utilisation de beaucoup de mémoire) et je n'en parlerai pas ici.

À faire vous même

Voici un exemple, relativement difficile, que vous allez étudier très attentivement. Avant même de tester ce programme, avez-vous une idée du résultat de son exécution ?

Babylon-ex14 (script.js)

```
var canvas = document.getElementById("renderCanvas");
var time=0;
var tabSph=[];
var tabEnnemi=[];
var nbrEn=5;
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);
var camera = new BABYLON.ArcRotateCamera("cam", Math.PI/2, 1.5*Math.PI/4, 50, new
BABYLON.Vector3(0,0,0), scene);
camera.attachControl(canvas);
var light = new BABYLON.PointLight("pointLumineux1", new BABYLON.Vector3(100, 100, 0),
scene);
var plan=BABYLON.Mesh.CreatePlane("sol",200,scene);
plan.rotation.x=Math.PI/2;
var mat=new BABYLON.StandardMaterial("mat",scene);
mat.diffuseColor = new BABYLON.Color3(1,0,0);
window.addEventListener('click',function(event){
    var pickResult=scene.pick(event.clientX, event.clientY);
    if (pickResult.hit){
        for (var i=0;i<tabSph.length;i++){
            if (tabSph[i]==pickResult.pickedMesh){
                tabEnnemi[i].touche(i)
            }
        }
    }
});
for (var i=0;i<nbrEn;i++){
    tabSph.push(BABYLON.Mesh.CreateSphere("sph"+i,20,2,scene));
}
function Ennemi(ptsVie,posX,posY,posZ){
    this.ptsVie=ptsVie;
    this.posX=posX;
    this.posY=posY;
    this.posZ=posZ;
    this.vitesse=0.1*Math.random()+0.2;
}
Ennemi.prototype.NouvPos=function(i){
    tabSph[i].position.x=this.posX;
    tabSph[i].position.y=this.posY;
    tabSph[i].position.z=this.posZ;
}
Ennemi.prototype.deplaEn=function(){
    if (this.posY<2 || this.posY>15){
        this.vitesse=-this.vitesse
    }
    this.posY=this.posY+this.vitesse
}
```

```

Ennemi.prototype.touche=function(i){
    this.ptsVie=this.ptsVie-1
    if (this.ptsVie==1){
        tabSph[i].material=mat;
    }
    if (this.ptsVie<=0){
        tabSph[i].dispose();
        tabSph.splice(i,1);
        tabEnnemi.splice(i,1);
    }
}

for (var i=0;i<nbrEn;i++){
    tabEnnemi.push(new Ennemi(3,6*i,5,0));
}

scene.registerBeforeRender(function(){
    time=time+(1/BABYLON.Tools.GetFps());
    for (var i=0;i<tabSph.length;i++){
        tabEnnemi[i].NouvPos(i);
        tabEnnemi[i].deplaEn();
    }
});
engine.runRenderLoop(function () {
    scene.render();
});

```

Testez le programme, vos prévisions étaient-elles justes ?

Chapitre 11 : L'héritage en JavaScript

L'héritage est une notion fondamentale en POO. Pour une fois l'héritage porte bien son nom, imaginons un objet A qui possède un attribut attrA et une méthode methA. Soit maintenant un objet B qui lui possède un attribut attrB et une méthode methB. Comme vous allez le voir, l'idée d'héritage est relativement simple : Si B hérite de A, alors l'utilisateur d'un objet de type B aura à sa disposition attrB et methB (ça c'est normal !), mais il pourra aussi utiliser attrA et methA. Si vous avez compris cela, vous avez compris le principe de l'héritage en POO.

Dans les langages orientés objet « classiques » (C++ ou Java), l'héritage est relativement simple à mettre en œuvre. En JavaScript, les choses sont plus complexes :

Il existe plusieurs façons de faire de l'héritage en JavaScript, mais nous allons ici en voir qu'une. Ce n'est pas la simplicité qui a guidé mon choix, j'ai choisi de vous exposer ici la méthode respectant (selon moi!) le plus « l'esprit JavaScript ».

Pour commencer, nous allons étudier un cas avec des constructeurs vides et des objets possédant uniquement des méthodes (pas d'attributs). Pour que l'objet de type B puisse utiliser la méthode issue de l'objet A, nous allons devoir utiliser la méthode create de l'objet Object (Object.create()).

Que fait cette méthode ?

Cette méthode permet de créer un objet, le prototype de cet objet correspondant à l'objet passé en paramètre de la méthode create. Rien compris ? Voici un exemple :

```
var monObj1=Object.create(monObj2)
```

Nous créons un objet monObj1, cet objet aura pour prototype monObj2.

Pour faire de l'héritage, nous allons faire comme suit :

```
var B.prototype=Object.create(A.prototype)
```

Nous créons (plutôt nous recréons) le prototype de l'objet B. Le prototype de l'objet B aura pour prototype le prototype de l'objet A (oui, je sais, c'est compliqué).

Voici un exemple :

```
//constructeur de A (vide, mais nécessaire)
function A() {
}
//nous créons la méthode maMethodeA
A.prototype.maMethodeA = function() {
    document.write("je suis une méthode de l'objet A")
};
function B() {
}
//l'héritage est là !
B.prototype = Object.create(A.prototype);
var monObjetB = new B();
//nous utilisons la méthode maMethodeA avec un objet de type B
monObjetB.maMethodeA();
```


résultat

je suis une méthode de l'objet A

Parcourons la chaîne des prototypes pour l'objet de type B :

- l'objet de type B (monObjetB) ne possède pas de méthode maMethodeA, passons à la suite
- le prototype de l'objet de type B ne possède pas de méthode maMethodeA, suivant
- le prototype du prototype de l'objet de type B (c'est-à-dire le prototype de l'objet de type A) possède la méthode recherchée, fin de la recherche

Il est évidemment possible de définir des méthodes propres à l'objet B. Attention cependant à bien définir ces méthodes après la ligne permettant l'héritage (`«B.prototype=Object.create(A.prototype);»`). Sinon votre méthode sera « écrasée ».

Un exemple

```
//constructeur de A (vide, mais nécessaire)
function A() {
}
//nous créons la méthode maMethodeA
A.prototype.maMethodeA = function() {
    document.write("je suis une méthode de l'objet A")
};
function B() {
}
//l'héritage est là !
B.prototype = Object.create(A.prototype);
//création de la méthode spécifique à B
B.prototype.maMethodeB=function() {
    document.write("je suis une méthode de l'objet B")
};
var monObjetB = new B();
//nous utilisons la méthode maMethodeA avec un objet de type B
monObjetB.maMethodeA();
monObjetB.maMethodeB();
```

résultat

je suis une méthode de l'objet A

je suis une méthode de l'objet B

Parfois le constructeur de l'objet B doit appeler le constructeur de l'objet A (par exemple pour que l'objet B puisse profiter des attributs définis dans le constructeur de l'objet A). Pour effectuer cet appel, il est indispensable d'utiliser la méthode `call`.

Prenons un exemple

```
//constructeur de A
function A() {
    document.write("le constructeur de A a bien été appelé");
    this.monAttrA="je suis un attribut de A";
}
//nous créons la méthode maMethodeA
A.prototype.maMethodeA = function() {
    document.write("je suis une méthode de l'objet A")
};
function B() {
    //cette ligne permet d'appeler le constructeur de A
    A.call(this);
}
//l'héritage est là !
B.prototype = Object.create(A.prototype);
//création de la méthode spécifique à B
B.prototype.maMethodeB=function() {
    document.write("je suis une méthode de l'objet B")
};
var monObjetB = new B();
//nous utilisons la méthode maMethodeA avec un objet de type B
monObjetB.maMethodeA();
```

```
monObjetB.maMethodeB();
document.write(monObjetB.monAttrA);
```

résultat

le constructeur de A a bien été appelé
je suis une méthode de l'objet A
je suis une méthode de l'objet B
je suis un attribut de A

Le this dans «A.call(this);» est très important, il permet d'utiliser l'attribut de l'objet A avec un objet de type B (le this en paramètre de la méthode call permet de passer une référence à l'instance en cours).

Oui, je sais, cela devient très compliqué, mais rassurez-vous, vous n'avez pas vraiment besoin de comprendre le « pourquoi du comment », vous devez juste retenir que si vous avez besoin d'appeler le constructeur de l'objet parent (ici A) depuis le constructeur de l'objet enfant (B) il faudra utiliser call comme indiqué ci-dessus.

Et si mon constructeur parent prend des paramètres, comment faire ?

Il faut alors utiliser le deuxième paramètre de la méthode call

un exemple

```
function A(color) {
    this.monAttrA=color;
}
A.prototype.maMethodeA = function() {
    document.write(this.monAttrA)
};
function B(couleur, nombre) {
    //cette ligne permet d'appeler le constructeur de A et de lui passer un paramètre
    A.call(this,couleur);
    this.nbr=nombre
}
B.prototype = Object.create(A.prototype);
var monObjetB = new B("vert",12);
document.write("couleur : ",monObjetB.monAttrA,"", nombre : ",monObjetB.nbr);
```

Je vous laisse étudier cet exemple, normalement il est assez compréhensible.

Une petite mise en garde : la méthode create est issue d'ECMAScript5, elle fonctionnera donc uniquement avec des navigateurs récents. Il existe des solutions pour contourner le problème (polyfill)
voir ici : https://developer.mozilla.org/enUS/docs/JavaScript/Reference/Global_Objects/Object/create

À faire vous même

Voici un exemple, relativement difficile, que vous allez étudier très attentivement. Avant même de tester ce programme, avez-vous une idée du résultat de son exécution ?

Babylon-ex15 (script.js)

```
var canvas = document.getElementById("renderCanvas");
var time=0;
var tabSph=[];
var tabBox=[];
var tabEnnemi=[];
var nbrEn=5
var engine = new BABYLON.Engine(canvas, true);
var scene = new BABYLON.Scene(engine);
var camera = new BABYLON.ArcRotateCamera("cam", Math.PI/2, 1.5*Math.PI/4, 50, new
BABYLON.Vector3(0,0,0), scene);
var light = new BABYLON.PointLight("pointLumineux1", new BABYLON.Vector3(100, 100, 0),
scene);
var plan=BABYLON.Mesh.CreatePlane("sol",200,scene);
plan.rotation.x=Math.PI/2;
var mat=new BABYLON.StandardMaterial("mat",scene);
mat.diffuseColor=new BABYLON.Color3(1,0,0);
var keys={left:0,right:0,up:0,down:0}
```

```

window.addEventListener('keydown', function(event) {
    if (event.keyCode==37) {
        keys.left=1;
    }
    if (event.keyCode==39) {
        keys.right=1;
    }
    if (event.keyCode==38) {
        keys.up=1;
    }
    if (event.keyCode==40) {
        keys.down=1;
    }
});
window.addEventListener('keyup', function(event) {
    if (event.keyCode==37) {
        keys.left=0;
    }
    if (event.keyCode==39) {
        keys.right=0;
    }
    if (event.keyCode==38) {
        keys.up=0;
    }
    if (event.keyCode==40) {
        keys.down=0;
    }
});

for (var i=0;i<nbrEn;i++){
    tabSph.push(BABYLON.Mesh.CreateSphere("sph"+i,20,2,scene));
}
tabBox.push(BABYLON.Mesh.CreateBox("box",2,scene));
tabBox[0].material=mat;

function Perso(posX,posY,posZ) {
    this.posX=posX;
    this.posY=posY;
    this.posZ=posZ;
}
Perso.prototype.NouvPos=function(i,tabMesh) {
    tabMesh[i].position.x=this.posX;
    tabMesh[i].position.y=this.posY;
    tabMesh[i].position.z=this.posZ;
}
function Ennemi(posX,posY,posZ) {
    Perso.call(this,posX,posY,posZ);
    this.vitesse=0.1*Math.random()+0.2;
}
Ennemi.prototype=Object.create(Perso.prototype);
Ennemi.prototype.deplaEn=function() {
    if (this.posY<2 || this.posY>15){
        this.vitesse=-this.vitesse
    }
    this.posY=this.posY+this.vitesse
}
function Player(posX,posY,posZ) {
    Perso.call(this,posX,posY,posZ);
    this.vitesse=1;
}
Player.prototype=Object.create(Perso.prototype);
Player.prototype.deplaPl=function() {
    if (keys.up==1) {
        this.posZ=this.posZ-this.vitesse;
    }
    if (keys.down==1) {
        this.posZ=this.posZ+this.vitesse;
    }
    if (keys.left==1) {
        this.posX=this.posX+this.vitesse;
    }
}

```

```

    }
    if (keys.right==1){
        this.posX=this.posX-this.vitesse;
    }
}
for (var i=0;i<nbrEn;i++){
    tabEnnemi.push(new Ennemi(6*i,5,0));
}
joueur=new Player(0,1,0);
scene.registerBeforeRender(function(){
    time=time+(1/BABYLON.Tools.GetFps());
    for (var i=0;i<tabSph.length;i++){
        tabEnnemi[i].NouvPos(i,tabSph);
        tabEnnemi[i].deplaEn();
    }
    joueur.NouvPos(0,tabBox);
    joueur.deplaPl();
});
engine.runRenderLoop(function () {
    scene.render();
});

```

Testez le programme, vos prévisions étaient-elles justes ?