Three JS

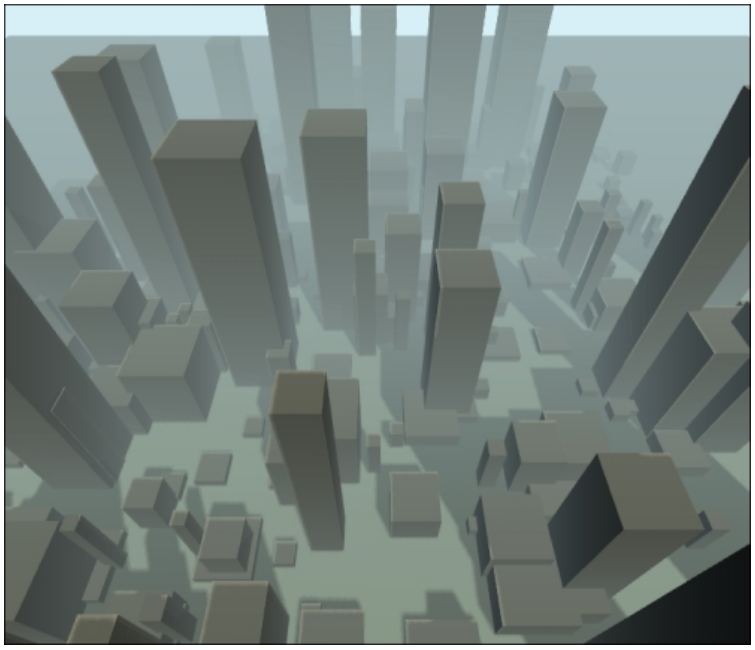
Une introduction

Rudi Giot - 11 octobre 2020

1. Introduction

1.1. Notions de base

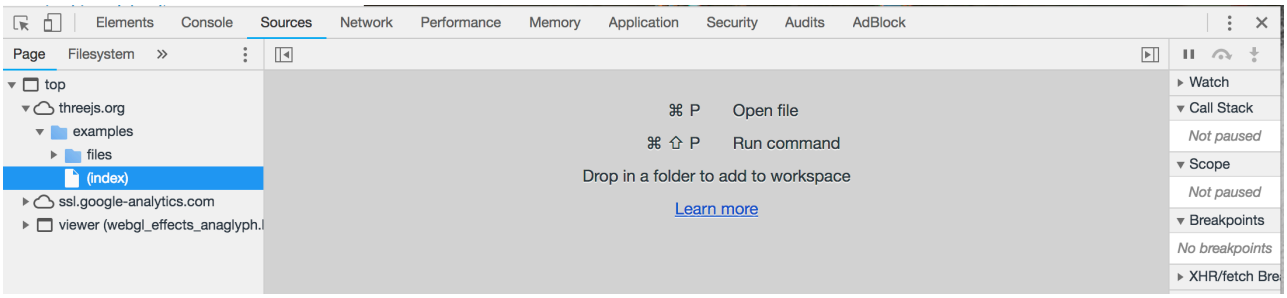
Les navigateurs sont maintenant tous capables d'afficher des scènes 3D calculées en temps réel grâce au *WebGL*, qui utilise les capacités de vos cartes graphiques (*GPU*). Il est donc possible de réaliser des applications destinées à un navigateur en utilisant directement *WebGL*, mais cette tâche est très ardue. *WebGL* est complexe à appréhender et nécessite un grand nombre de ligne de code pour n'afficher que très peu de choses. C'est là que *Three.js* entre en jeu en proposant d'utiliser *JavaScript* pour créer des scènes 3D complexes sans avoir à étudier le *WebGL* en détail.



Three.js (que nous noterons par la suite *Three*) fournit donc un grand nombre de fonctionnalité dans une librairie (*API*) riche, conviviale et documentée. Il existe un grand nombre d'exemples disponibles sur *Internet* (par exemple sur https://threejs.org/examples) pour vous aider à apprendre le langage et pour s'en inspirer et créer de nouvelles scènes en 3D.

1.2. Tester et debugger le code

Nous vous conseillons d'utiliser *Chrome* ou *Firefox* comme navigateur de test, car ils offrent un debugueur simple et pratique.



Fenêtre de déboguage dans Chrome

1.3. Le squelette HTML d'une application

La première chose que nous devons créer est un « squelette HTML vide » qui hébergera notre code Three (Javascript) dans nos futurs exemples et exercices :

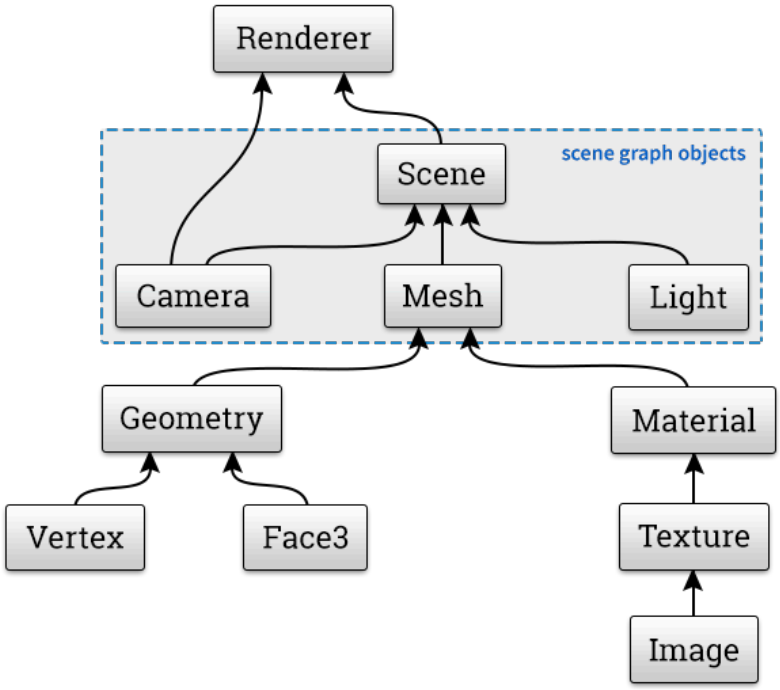
```
<!DOCTYPE html>
<html>
    <head>
        <title>Squelette HTML pour Three.js</title>
        <script src="../libs/three.js"></script>
        <style>
            body{
                margin: 0;
                overflow: hidden;
            }
        </style>
    </head>
    <body>
        <div id="WebGL-output"> </div>
        <script>

            init();

            function init() {

                // votre code Three.js ici

            };

        </script>
    </body>
</html>
```

Template HTML pour accueillir un code Three

2. Les composants de base

2.1. Arborescence des composants

Le composant principal dans *Three* est la « *Scene* ». ***THREE.Scene*** est une structure qui va contenir toutes les informations graphiques (caméra, objets, lumières, …) de la scène à afficher. Cette scène consiste en une ensemble de noeuds qui forment une structure arborescente.



Structure arborescente du « Renderer »[1]

Pour créer une « *scene* », nous avons au minimum besoin des trois composants essentiels :

- la « *Camera* » : qui détermine la zone de « rendu »
- les « *Meshes* » : qui représentent des formes tridimensionnelles (en français « maillages ») qui vont apparaitre dans la scène (des cubes, spheres, …)
- les « Lights » : qui éclairent la scène

[1] http://www.ux-republic.com/webgl-three-js/

2.2. Création d'une scène

Nous allons détailler le processus de création d'une scène pas à pas. Pour commencer, on créée un objet « *scene* » qui va contenir l'ensemble des autres objets (*Camera*, *Meshes* et *Lights*) :

```
var scene = new THREE.Scene();
```

On va ensuite créer et positionner une « *camera* » qui sera « notre point de vue » sur la scène :

```
var camera = new THREE.PerspectiveCamera(45, window.innerWidth /
                                window.innerHeight, 0.1, 1000);
camera.position.x = -30;
camera.position.y = 40;
camera.position.z = 30;
camera.lookAt(scene.position);
```

… et puis ajouter cet objet *camera* à notre objet *scene* :

```
        scene.add(camera);
```

Nous allons maintenant positionner un plan dans cette scène:

```
var planeGeometry = new THREE.PlaneGeometry(60, 40, 1, 1);
var planeMaterial = new THREE.MeshLambertMaterial({color:
0xffffff});
var plane = new THREE.Mesh(planeGeometry, planeMaterial);
plane.rotation.x = -0.5 * Math.PI;
plane.position.x = 0;
plane.position.y = 0;
plane.position.z = 0;
scene.add(plane);
```

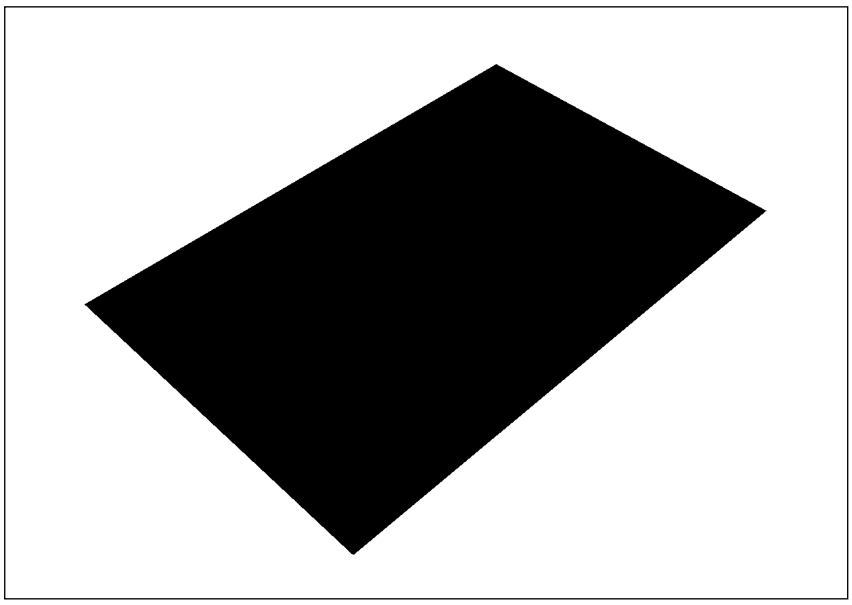
Nous pouvons remarquer qu'il faut préalablement créer deux objets (*PlaneGeometry* et *PlaneMaterial*) pour pouvoir ensuite créer le plan. Nous reviendrons plus loin en détail sur ces deux points.

Une fois la « *scene* » complète (elle contient le strict minimum : une caméra et un objet), il faut lui appliquer un « *renderer WebGL* » pour l'afficher :

```
var renderer = new THREE.WebGLRenderer();
renderer.setClearColor(0xFFFFFF);
renderer.setSize(window.innerWidth, window.innerHeight);
document.getElementById("WebGL-output")
                                .appendChild(renderer.domElement);
renderer.render(scene, camera);
```

Vous remarquerez que l'instruction *document.getElementById("**WebGL-output**")* fait référence au * * qui se trouve en tête de notre squelette de page *HTML*.

Le résultat (fichier *Exemple1.html*) affiché dans le browser n'est pas très spectaculaire, mais c'est un bon point de départ pour venir y ajouter de nouveaux objets et pour aborder les notions de lumières.



Remarque : Il y a deux types de caméras : l'*orthographic* (tous les éléments ont une taille indépendante des distances, souvent utilisé dans les jeux 2D) et la *perspective* (la plus naturelle, qui affiche un objet plus petit s'il est éloigné). Vous pouvez expérimenter ces deux types en fonction de vos projets et visualiser leurs particularités.

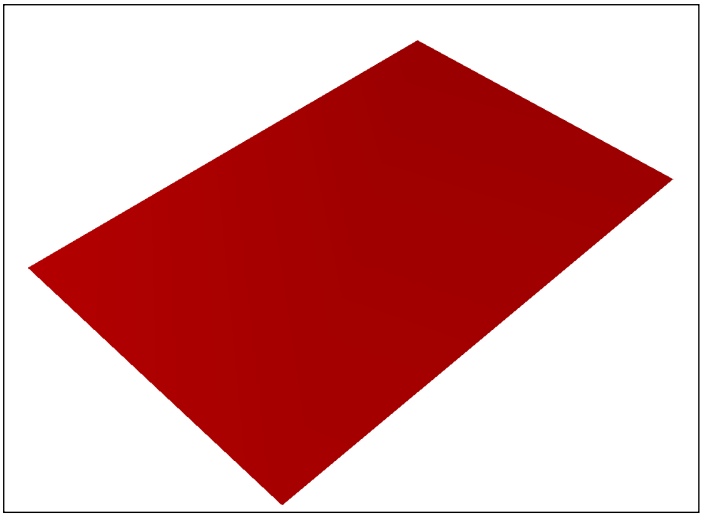
2.3. La lumière

Nous allons maintenant éclairer notre scène en ajoutant un objet *spotLight* à la scène :

```
var spotLight = new THREE.SpotLight(0xffffff);
spotLight.position.set(-40, 60, -10);
scene.add(spotLight);
```

Grâce a cette lumière qui éclaire notre scène, nous allons pouvoir, par exemple, faire apparaitre la couleur de nos objets. Essayez par exemple de modifier la couleur du plan :

```
var planeMaterial = new THREE.MeshLambertMaterial(
                                                {color: 0xFF0000});
```



Vous voyez alors (dans l'*Exemple2.html*) apparaitre le plan en rouge alors que, sans la lumière, le plan apparaissait toujours complètement noir. Nous verrons plus loins dans ce cours des notions plus approfondies concernant la lumière. Il existe, en effet, différents types de lumières : *ambiant*, *spot*, *directions*, *point*, *hemisphere*, *area* et *lensflare*. Elles possèdent des caractéristiques différentes utiles en fonction des applications que vous avez à réaliser. Vous pourrez les expérimenter dans vos projets et choisir celle qui est la plus appropriée à votre application.

Il y a plusieurs types de lumières, on aurait pu avoir un résultat similaire (si vous observez bien, ça n'est pas identique) avec une lumière d'ambiance :

```
scene.add(new THREE.AmbientLight(0xFFFFFF));
```

2.4. Animation

Pour animer la scène, par exemple, en faisant tourner le plan selon l'axe vertical (en z) nous allons faire appel au « moteur de rendu » de manière régulière pour « rafraîchir » la scène à chaque *frame*. Pour cela, nous allons créer une fonction de rendu qui s'appellera elle-même à chaque frame:

```
function render() {
        plane.rotation.z += 0.01;
        requestAnimationFrame(render);
        renderer.render(scene, camera);
    }
```

Il ne restera plus qu'à appeler cette fonction en fin de programme:

```
render();
```

On peut observer le résultat dans l'*exemple4.html*. Mais avec un peu de trigonométrie, on peut assez facilement avoir des mouvements plus variés :

```
var step=0;
...
function render() {
        plane.rotation.z = (Math.cos(step));
        requestAnimationFrame(render);
        renderer.render(scene, camera);
    }
```

Ce code est disponible dans l'*exemple4c.html*. Expérimentez d'autres animations, sur d'autres axes, sur plusieurs axes, changez leur vitesse, leur amplitude, …

3. Interface utilisateur

En pratique, il n'est pas toujours évident d'avoir, à priori, une bonne position pour les objets, caméra ou lumières. Il est encore moins évident de bien choisir la valeur des paramètres d'une animation, la vitesse de rotation de notre plan, par exemple. Toutes ces valeurs peuvent être réglées directement en « temps réel » si nous ajoutons une interface utilisateur qui va permettre d'ajuster les paramètres choisis. Construire cette interface graphique avec *Three* est un projet très conséquent. Heureusement, certaines personnes y ont déjà travaillé (https://workshop.chromeexperiments.com/examples/gui/#1--Basic-Usage) et nous allons donc exploiter leur code qui est ouvert et libre de droit.

Cette librairie « *dat.GUI* » est relativement simple à mettre en oeuvre. Il faut d'abord la télécharger et la placer dans le répertoire racine de notre serveur HTTP. Il faut ensuite y faire référence dans l'entête de notre « squelette HTML » :

```
<script type="text/javascript" src="../libs/dat.gui.js"></script>
```

Ensuite, il faut définir les paramètres qui seront « modifiables » lors de l'exécution de notre code :

```
        this.rotationSpeed = 0.01;
```

… et les utiliser dans la fonction de rendu :

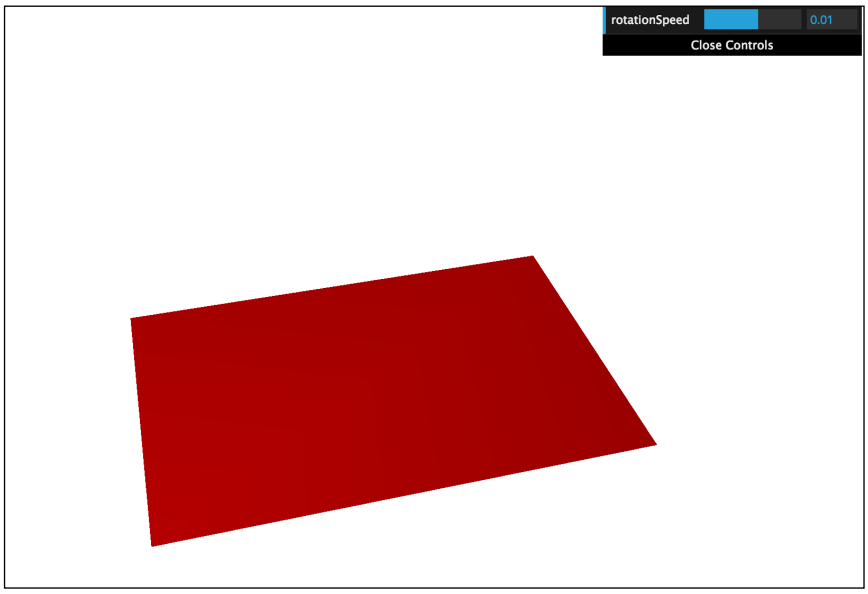
```
        plane.rotation.z += rotationSpeed;
```

Pour terminer il faut créer l'objet correspondant à l'interface graphique et y ajouter les différents widgets/contrôles :

```
        var gui = new dat.GUI();
        gui.add(this, 'rotationSpeed', -0.1, 0.1);
```

Le résultat doit être le suivant (avec le plan qui tourne autour d'un axe perpendiculaire et centré sur lui-même) :

Le code de cet exemple est disponible dans l'*Exemple4.html*.



On peut réaliser la même chose de manière plus structurée avec une fonction qui construit et gère l'interface :

```
function buildGui() {

    gui = new dat.GUI();

    var params = {
          'light color': spotLight.color.getHex(),
                              …
                    }

    gui.addColor( params, 'light color' ).onChange( function ( val )
          {
                spotLight.color.setHex( val );
                } );

    gui.open();
}
```

Remarque : pour changer la couleur du plan , il suffit d'écrire :

```
plane.material.color.set(val);
```

4. Redimensionnement de la scène

Lorsqu'on redimensionne le browser la caméra ne s'adapte pas automatiquement à la nouvelle taille de la fenêtre du navigateur. On doit donc ajouter une fonction qui sera appelée lorsque l'utilisateur réalise cette opération. Il s'agit d'un événement *onResize* sur lequel nous devons mettre un *EventListener* :

```
window.addEventListener('resize', onResize, false);
```

Cette ligne de code doit être ajoutée dans la fonction *init()*. Il faut ensuite déclarer la fonction qui sera appelée à chaque occurence de cet événement et y mettre à jour la *camera* et le *renderer* :

```
function onResize() {
    camera.aspect = window.innerWidth / window.innerHeight;
    camera.updateProjectionMatrix();
    renderer.setSize(window.innerWidth, window.innerHeight);
}
```

Le problème avec ce code, c'est qu'il fait appel aux objets *scene, camera* et *renderer* qui sont définis dans une autre fonction (*init*). Nous allons donc devoir rendre ces variables globales pour qu'elles soient accessibles dans toutes les fonctions que nous écrirons par la suite:

```
var camera;
var scene;
var renderer;

function init() {
...
window.addEventListener('resize', onResize, false);
...
}

function onResize() {
...
}
```

Vous pouvez visualiser le résultat de ce code dans le fichier *Exemple4b.html*.

5. Les objets d'une scène

5.1.Le cube

Pour créer un *mesh* de type cube dans notre scène nous allons d'abord devoir créer (comme pour le plan) une *Geometry* (de type *Box* dans ce cas précis) et un *Material*. Nous pouvons ensuite créer un objet cube que nous positionnons et ajoutons à la scène :

```
var cubeGeometry = new THREE.BoxGeometry(4,4,4);
var cubeMaterial = new THREE.MeshLambertMaterial({color:0xffff00});
var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);
cube.position.y = 3;
scene.add(cube);
```

Le résultat (fichier *Exemple6.html*) n'est que moyennement convaincant. En effet, le cube apparait bien au dessus du plan mais notre cerveau pourrait l'interpréter comme étant posé sur le plan. Il manque en fait des ombres pour mieux « comprendre » la scène.

Si vous voulez faire bouger (animer) le cube, vous pouvez (comme on l'a déjà fait avec le plan) modifier chacune des coordonnées de sa position :

```
cube.position.x=1;
cube.position.y=1;
cube.position.z=1;
```

On peut faire la même chose en une seule ligne de code :

```
cube.position.set(1,1,1);
```

Ou encore :

```
cube.position.set(10,3,1);
```

On peut appliquer le même principe aux autres propriétés, la rotation, par exemple :

```
cube.rotation.x = 0.1;

cube.rotation.set(0.1, 0, 0);

cube.rotation = new THREE.Vector3(0.1, 0, 0);
```

Attention : vous remarquerez que ces trois dernières lignes de code réalisent la même opération.

Exercice : Créez une scène avec plusieurs plans et plusieurs cubes et animer l'ensemble de manière originale.

Il existe un grand nombre d'objets (*geometry*) pré-existant dans *Three* : *Circle*, *Ring*, *Sphere*, *Cylinder*, *Torus*, *Polyhedron*, *Octahedron*, *TetraHedron*, *Text*, *Dodecahedron*, … Nous verrons plus tard la *Sphere* et le *Text* en détail mais pour les autres, testez-les en vous reportant à la documentation.

Exemples :

```
var cylinderGeometry = new THREE.CylinderGeometry( 5, 5, 20, 32 );
var cylinderMaterial = new THREE.MeshLambertMaterial(
                                                { color: 0xFF00FF });
var cylinder = new THREE.Mesh( cylinderGeometry, cylinderMaterial );
scene.add( cylinder );
```

```
var geometry = new THREE.TorusGeometry( 10, 3, 16, 100 );
var material = new THREE.MeshBasicMaterial( { color: 0xffff00 } );
var torus = new THREE.Mesh( geometry, material );
scene.add( torus );
```

Pour le reste voir la rubrique *Geometries* de la documentation :
https://threejs.org/docs/#api/en/geometries/BoxGeometry

5.2. Les ombres

Le rendu d'ombres est une opération très couteuse en terme de temps de calcul pour un ordinateur. Il est donc par défaut désactivé. Nous allons donc devoir explicitement spécifié que nous voulons ce rendu à différents endroits dans le code :

```
renderer.setClearColor(new THREE.Color(0xEEEEEE, 1.0));
renderer.setSize(window.innerWidth, window.innerHeight);
renderer.shadowMapEnabled = true;
```

La première opération est de demander à THREE de réaliser ces calculs à travers le propriété *shadowMapEnabled* qui par défaut est à *false*. Mais ça n'est pas suffisant, il faut aussi définir quel objet provoque l'ombre (*castShadow*) et quel objet reçoit l'ombre (*receiveShadow*). Dans notre cas, nous voulons que le cube provoque une ombre sur le plan, le code est donc le suivant :

```
plane.receiveShadow = true;
...
cube.castShadow = true;
...
```

Il faut pour finir spécifier quelle lumière va provoquer l'ombre :

```
spotLight.castShadow = true;
```

Vous pouvez dès lors visualiser dans l'*exemple7.html* l'ombre projetée sur le plan par la lumière sur le cube.

5.3.La sphère

Pour créer un *mesh* de type sphère dans notre scène nous allons d'abord devoir créer (comme pour le plan et le cube) une *Geometry* (de type *Sphere*) et un *Material*. Vous remarquerez qu'ici nous spécifions que *wireframe* sera initialisé à *true* (ce la permet une visualisation en « fil de fer »). Nous pouvons ensuite créer un objet cube que nous positionnons et ajoutons à la scène :

```
var sphereGeometry = new THREE.SphereGeometry(4, 20, 20);
var sphereMaterial = new THREE.MeshBasicMaterial({color: 0x7777ff,
wireframe: true});
var sphere = new THREE.Mesh(sphereGeometry, sphereMaterial);
sphere.position.x = 20;
sphere.position.y = 4;
sphere.position.z = 2;
scene.add(sphere);
```

Vous pouvez facilement ajouter ce code dans votre programme pour visualiser cette sphère. La solution se trouve dans l'*exemple8.hml*.

Exercice : Faire rebondir une balle indéfiniment sur le sol (plan), en utilisant la fonction *Cosinus* vue plus haut et sur le rythme de la musique choisie.

5.4. Ajout et suppression d'objets

Nous savons comment ajouter un objet (cube ou sphère) à la scène. Si par la suite, dans notre application, nous souhaitons le supprimer ou changer une de ses propriétés il est prudent de le nommer (l'identifier). Pour cela nous ajouterons simplement la ligne suivante à la construction de chacune des occurrences d'objet :

```
cube.name = "cubeNumero" + scene.children.length;
```

Nous pourrions évidemment choisir une autre méthode d'identification mais cette dernière est simple et fonctionnelle. A partir de ça, nous pourrons utiliser la fonction :

```
myObject = scene.getObjectByName(name)
```

… pour récupérer un objet identifié par son « *name* » et lui appliquer des modifications ou bien même le détruire :

```
scene.remove(myObject);
```

Il est également utile de savoir que dans *Three* les enfants de la scène sont stockés dans une liste. Nous pouvons donc, par exemple, supprimer le dernier objet ajouté à la scène en utilisant le code suivant :

```
var allChildren = scene.children;
var lastObject = allChildren[allChildren.length-1];
if (lastObject instanceof THREE.Mesh) {
    scene.remove(lastObject);
}
```

Vous remarquerez qu'avant de supprimer l'objet nous vérifions si celui-ci est bien un *Mesh*, de manière à éviter de supprimer une lumière ou une caméra qui sont également des enfants de la scène.

Une autre manière pour « passer en revue » les différents objets de la scène est d'utiliser :

```
scene.traverse(function(obj) {
    if (obj instanceof THREE.Mesh && obj != plane ) {
          obj.rotation.x+=0.01;
          obj.rotation.y+=0.01;
          obj.rotation.z+=0.01;
}
```

Dans cet exemple, on applique une rotation à tous les Mesh de la scène (sauf le plan). Cette technique est intéressante quand on a une grande quantité d'objets à l'écran à animer ensemble.

Exercice : Réalisez une scène dans laquelle vous ajoutez un cube (à une position aléatoire et avec une couleur aléatoire) à chaque *beat* d'une musique. Faites tourner tous les cubes ensemble.

5.5. Le brouillard

Le principe du brouillard est simple : plus un objet est éloigné de la caméra et plus il sera « caché » par le brouillard. Pour activer cette fonction il suffit d'ajouter la ligne suivante :

```
scene.fog=new THREE.Fog( 0xffffff, 0.015, 100 );
```

Les paramètres spécifiés permettent de choisir la couleur du brouillard (blanc dans notre exemple: 0xFFFFFF) ainsi que sa proximité (0.015 dans notre exemple) et son éloignement (100 dans notre cas). Ces deux paramètres permettent en fait de régler la densité du brouillard. Il existe une méthode alternative pour définir un brouillard :

```
scene.fog=new THREE.FogExp2( 0xffffff, 0.01 );
```

Dans ce cas la densité de brouillard augmente de manière exponentielle plutôt que linéairement, ce qui visuellement est sans doute plus proche de la réalité. Mais c'est à vous d'essayer les deux méthodes et de choisir ce qui correspond le mieux avec votre projet. Vous pouvez expérimenter ces méthodes et valeurs de paramètres à partir de l'*Exemple9.html* ou de l'*Exemple9b.html* fait exactement la même chose que le précédent mais en permettant de modifier les valeurs de deux paramètres avec la *GUI* en temps réel.

5.6. *Vertices*, *geometries* et *meshes*

Il existe dans *Three* plusieurs *geometries* qui sont déjà prêtes à être utilisées. Nous avons vu les *sphere* et *box* plus haut. Une *geometry* consiste en un ensemble de points dans l'espace en trois dimensions qui connectés entre eux forment des faces. Par exemple, un cube possède huit sommets qui sont définis chacun par leurs trois coordonnées en *x*, *y* et *z*. Chacun de ces points est appelé un *vertex*, une ensemble de vertex est appelé *vertices* (le pluriel de *vertex*).

Remarque : *vertice* n'est pas un mot et *vertexes* non plus.

Un cube possède également douze côtés (*edges*) et six faces (*sides*) carrées. Avec *Three* on ne peut manipuler que des triangles (composés à partir de trois *vertices*). Pour élaborer la face d'un cube, un carré, il faut donc deux triangles (*faces*). Attention au vocabulaire : une face en français dans le cas du cube est un carré, en anglais, dans le domaine de la 3D, la « *face* » représente un triangle, l'élément à la base de toutes les *geometries*.

Question pour voir si vous avez bien compris : combien de *faces* possèdent un cube ? Réponse : Un cube possède six faces carrées et donc douze *faces*.

On va donc pouvoir maintenant définir un cube sans utiliser le *box* fournit par *Three*. Nous allons d'abord définir les huit sommets (*vertices*) :

```
var vertices = [
    new THREE.Vector3(1,3,1),
    new THREE.Vector3(1,3,-1),
    new THREE.Vector3(1,-1,1),
    new THREE.Vector3(1,-1,-1),
    new THREE.Vector3(-1,3,-1),
    new THREE.Vector3(-1,3,1),
    new THREE.Vector3(-1,-1,-1),
    new THREE.Vector3(-1,-1,1)
];
```

… et ensuite les douze *faces* à partir de ces huit *vertices* (numérotés de 0 à 7 dans le tableau) :

```
var faces = [
    new THREE.Face3(0,2,1),
    new THREE.Face3(2,3,1),
    new THREE.Face3(4,6,5),
    new THREE.Face3(6,7,5),
    new THREE.Face3(4,5,1),
    new THREE.Face3(5,0,1),
    new THREE.Face3(7,6,2),
    new THREE.Face3(6,3,2),
    new THREE.Face3(5,7,0),
    new THREE.Face3(7,2,0),
    new THREE.Face3(1,3,4),
    new THREE.Face3(3,6,4),
];
```

Il ne reste plus qu'à créer une *geometry* à partir des *faces* :

```
var geom = new THREE.Geometry();
geom.vertices = vertices;
geom.faces = faces;
geom.computeFaceNormals();
```

… et utiliser cette *geometry* pour créer notre maillage (*mesh*) et enfin l'ajouter à notre *scene* :

```
var material = new THREE.MeshBasicMaterial({color: 0x7777ff);
var myCube = new THREE.Mesh(geom,material);
scene.add(myCube);
```

Il est évidemment plus commode, dans ce cas là, d'utiliser l'objet *box* pré-existant dans *Three*. Cependant, à partir de ce code, vous allez pouvoir façonner votre *mesh* à votre guise. Essayez par exemple de modifier les valeurs de deux ou trois coordonnées des *vertices* et observez comment évolue votre maillage dans l'*Exemple10.html*.

Exercice : Réalisez une pyramide à partir du code de l'exemple précédent. Solution dans l'*Exemple10b.html*.

Si vous désirez modifier votre mesh en cours d'exécution du code, c'est un peu compliqué. En effet, à priori les *meshes* sont créés en début de programme et ne se déforment pas en cours d'exécution, sauf si on le demande explicitement à *Three* dans la fonction de rendu :

```
            mesh.geometry.verticesNeedUpdate = true;
            mesh.geometry.computeFaceNormals();
```

A partir de là, on peut dans cette même fonction, modifier les valeurs des *vertices* et ainsi déformer notre *mesh* en temps réel à partir de contrôle ajouté dans l'interface graphique (*Exemple11.html*) :

```
var controlPoints = [];
    controlPoints.push(addControl(3, 5, 3));
    controlPoints.push(addControl(3, 5, 0));
    controlPoints.push(addControl(3, 0, 3));
    controlPoints.push(addControl(3, 0, 0));
    controlPoints.push(addControl(0, 5, 0));
    controlPoints.push(addControl(0, 5, 3));
    controlPoints.push(addControl(0, 0, 0));
    controlPoints.push(addControl(0, 0, 3));

var gui = new dat.GUI();

for (var i = 0; i < 8; i++) {
            folder = gui.addFolder('Vertex ' + (i + 1));
            folder.add(controlPoints[i], 'x', -10, 10);
            folder.add(controlPoints[i], 'y', -10, 10);
            folder.add(controlPoints[i], 'z', -10, 10);
        }

...

var vertices = [];
for (var i = 0; i < 8; i++) {
    vertices.push(new THREE.Vector3(controlPoints[i].x,
                        controlPoints[i].y, controlPoints[i].z));
            }

mesh.geometry.vertices = vertices;
```

Exercice : Créez une forme originale et déformez-là au rythme d'une musique.

5.7. Le texte

Pour afficher du texte, il faut d'abord choisir la *font*. Le problème est que cette *font* doit être convertie dans un format *JSON*. Le plus simple est donc de partir d'une *font* qui a déjà été convertie dans ce format et de s'en servir. Sinon vous devrez aller sur un site du genre : http://gero3.github.io/facetype.js/ pour y réaliser votre conversion et pouvoir utiliser votre *font* avec *THREE*.

```
var loader = new THREE.FontLoader();
            loader.load('helvetiker_bold.typeface.json', function
(font) {
                var textGeo = new THREE.TextGeometry("THREE.JS", {
                    font: font,
                    size: 30,
                    height: 5,
                    curveSegments: 12,
                    bevelThickness: 1,
                    bevelSize: 1,
                    bevelEnabled: true
                });
                textGeo.computeBoundingBox();

                var textMaterial = new THREE.MeshPhongMaterial({
                    color: 0xff0000,
                    specular: 0xffffff
                });

                var mesh = new THREE.Mesh(textGeo, textMaterial);

                scene.add(mesh);
```

Exercice : A partir de l'*Exemple12.html* essayez de faire tourner le texte sur son axe central et de changer le texte quand il est sur la tranche (solution dans l'*Exemple12b.html*).

6. Créer un clip vidéo

6.1. Introduction

Le but de ce chapitre est de créer un « cas d'utilisation » de Three. Nous allons créer à partir d'une musique ou une chanson, que vous pouvez choisir, un clip vidéo en 3D qui « colle » à votre morceau. La première étape consiste donc à réaliser ce choix et à analyser les moments clés de la partition, là où il y a des changement brusque, des rupture de rythme, des crescendo, … Ensuite il va falloir synchroniser des actions dans *Three* (mouvement de caméra, changement de couleur, redimensionnement d'objets, …) avec une ligne du temps. Nous allons donc voir dans les paragraphes suivants comment lire un morceau mp3 dans un *browser* et comment synchroniser *Three* avec un *Timer*.

6.2. Lecture de musique dans un *Browser*

Pour ajouter un fichier *mp3* dans une page *HTML*, il suffit d'insérer ce code à votre page :

```
<body>
    <audio autoplay>
        <source src="fichierAudio.mp3" type="audio/mp3">
    </audio>
```

6.3. Synchronisation de musique sur un *Timer*

Pour synchroniser la musique avec une animation, nous pourrions utiliser les *Timers* disponibles en *JavaScript*. Pour générer un *Timer* et lancer une fonction à un moment déterminé on peut écrire le code suivant :

```
window.setTimeout(function, milliseconds);
```

Malheureusement cette technique sur le long terme ne fonctionne pas bien car en fonction des performances du *Browser* et de l'application qu'il exécute/interprète (surtout avec du code *Three*/*WebGL*) le temps défini en milli-secondes est plus ou moins bien respecté. Si vous voulez synchroniser des événements de manière très précise, il est prudent d'utiliser une autre technique.

Il suffit, par exemple, au lancement de l'application de sauvegarder l'heure du système :

```
var startTime = new Date().getTime();
```

… et de s'y référer à chaque *frame* pour déclencher un ou plusieurs événements en fonction du temps « système » écoulé.

Dans l'exemple suivant on désire appliquer une rotation du plan toutes les secondes (fichier *Exemple5.html*) :

```
function render() {

            nowTime = new Date().getTime();
            if (nowTime>startTime+1000) {
                plane.rotation.z += 0.1;
                startTime = nowTime;
            }

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

Cependant, comme on vérifie ce décalage au *framerate* (approximativement toutes les 20ms) en ré-initialisant le *startTime* à chaque *frame,* on introduit systématiquement un petit décalage compris entre 0 et 20ms à chaque boucle. Concrètement après une minute on peut déjà avoir une dérive d'une seconde. Il est donc toujours préférable de faire référence au *startTime*, sans le ré-initialiser. Pour créer un métronome sans dérive, on fera donc plutôt (fichier *Exemple5b.html*) :

```
function init() {
   var startTime = new Date().getTime();
   }

function render() {
           nowTime = new Date().getTime();
           ellapsedTime = nowTime - startTime;
           if (ellapsedTime=>1000 && ellapsedTime<2000) {
                plane.rotation.z += 0.1;
           }

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

Cette méthode permet, par exemple, de synchroniser parfaitement (avec un décalage inférieur à 20ms, ce qui est imperceptible pour un humain) une animation avec une musique. Nous allons le montrer avec l'exemple suivant qui va lire un fichier mp3 et exécuter un code qui consistera en une animation synchronisée sur le BPM (Beat Per Minute) du morceau. Une solution se trouve dans l'*Exemple5c.html*.

6.4. Sources d'inspirations

Quand vous avez compris comment synchroniser des éléments de l'animation avec le timer, il ne vous reste plus qu'à créer votre scène et animations. Mais si vous voulez aller plus vite, vous pouvez aller voir dans les nombreux exemples proposés par le site officiel https://threejs.org/examples et choisir celui qui vous plait le plus pour accompagner votre morceau choisi. Par exemple :

https://threejs.org/examples/#webgl_geometry_extrude_splines
https://threejs.org/examples/#webgl_animation_keyframes
https://threejs.org/examples/#webgl_camera_cinematic
https://threejs.org/examples/#webgl_clipping_intersection
https://threejs.org/examples/#webgl_geometry_hierarchy
https://threejs.org/examples/#webgl_geometry_hierarchy2
https://threejs.org/examples/#webgl_geometry_minecraft
https://threejs.org/examples/#webgl_geometry_normals
https://threejs.org/examples/#webgl_geometry_shapes
https://threejs.org/examples/#webgl_geometry_text
https://threejs.org/examples/#webgl_geometry_text_shapes
https://threejs.org/examples/#webgl_gpgpu_birds
https://threejs.org/examples/#webgl_gpgpu_protoplanet
https://threejs.org/examples/#webgl_octree

A intégrer :

http://localhost/Three/book/Exemples/chapter-08/01-grouping.html

http://localhost/learning-threejs/chapter-05/02-basic-2d-geometries-circle.html comme pacman !

http://localhost/learning-threejs/chapter-05/08-basic-3d-geometries-torus-knot.html

http://localhost/learning-threejs/chapter-07/03-basic-point-cloud.html

http://localhost/learning-threejs/chapter-07/06-rainy-scene.html

http://localhost/learning-threejs/chapter-07/07-snowy-scene.html

http://localhost/learning-threejs/chapter-07/10-create-particle-system-from-model.html (cocher as particle … vraiment excellent)

http://localhost/learning-threejs/chapter-09/03-animation-tween.html

http://localhost/learning-threejs/chapter-09/05-fly-controls-camera.html

http://localhost/learning-threejs/chapter-10/11-video-texture-alternative.html

http://localhost/learning-threejs/chapter-11/04-shaderpass-simple.html

http://localhost/learning-threejs/chapter-12/06-audio.html

Planning des exercices :

Première journée :
- Template vide Ex0
- Un plan Ex1
- Un plan avec un spot de lumière Ex2
- Ajouter une animation du plan Ex3
- Ajouter une animation sur la caméra Ex3b et Ex3c
- Easing avec cos() sur les positions/animations du plan Ex4c
- Easing plus complexes avec https://easings.net/fr
- GUI pour rotation et couleur du plan Ex4
- Timing Ex5

Réaliser une première séquence d'événements qui s'enchainent à des moments particuliers (réaliser une « Machine à états »)

Analyser le morceau de musique choisi en repérant précisément le timing des séquences clés, des montées/descentes, moments calmes/agités, forts/faibles, …

Adapter la petite séquence d'événement au morceau de musique

Deuxième journée :
- Cube Ex6
- Position de la caméra controlée par GUI Ex6b
- Changer échelle d'un Cube Ex6a (revoir Easing)
- Réaliser une ville aléatoire Ex4g_bis (sans tableau)
- Améliorer l'exercice en stockant dans un tableau les buildings et les animer (scale, position, couleur, …)
- Essayer les autres primitives (Cylindre, Torus, …)
- Lire un fichier audio et récupérer la FFT (MusicV0 et V1)
- Utiliser les éléments de la FFT dans votre morceau de musique
- Ajouter des éléments dans votre projet (cubes, torus, …)
- Combiner la séquence de la veille avec la FFT vue aujourd'hui

Troisième journée :
-

Quatrième journée :
- Finaliser le projet