Three JS

Une introduction

Rudi Giot - 9 novembre 2022

1. Introduction

1.1. Notions de base

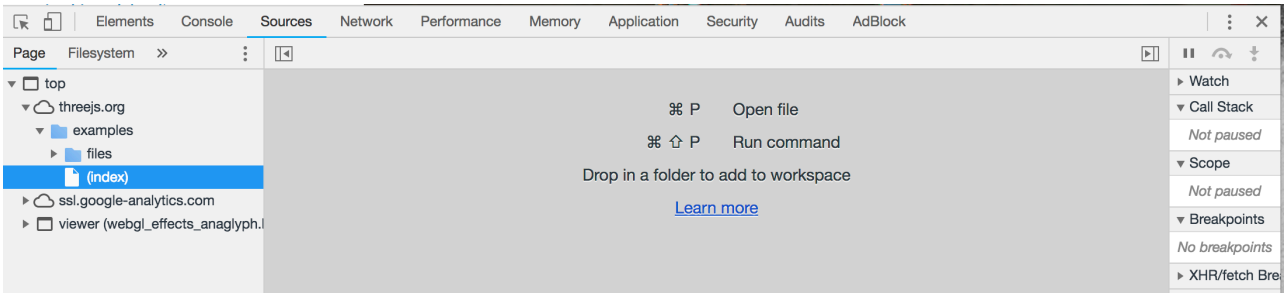
Les navigateurs sont maintenant tous capables d'afficher des scènes *3D* calculées en temps réel grâce au *WebGL*, qui utilise les capacités de vos cartes graphiques (*GPU*). Il est donc possible de réaliser des applications destinées à un navigateur en utilisant directement *WebGL*, mais cette tâche 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* pour vous aider à apprendre le langage et pour s'en inspirer et créer de nouvelles scènes en 3D :

- https://threejs.org/examples
- https://richardmattka.com/
- https://bruno-simon.com/
- http://letsplay.ouigo.com/
- https://chartogne-taillet.com/fr
- https://www.midwam.com/en

1.2. Programmer, tester et debugger le code

Nous vous conseillons d'utiliser *Visual Studio Code* (avec l'extension *Live Preview*) pour écrire vos programmes ainsi que *Chrome* ou *Firefox* comme navigateur de test, car ils offrent un debugueur (Menu -> Outils développeurs) 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 lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Basic Template</title>
    <link href="./style.css" rel="stylesheet">
    <script src="./libs/three.min.js"></script>
</head>
<body>
    <canvas class="webgl"></canvas>
    <script src="./js/script.js"></script>
</body>
</html>
```

Vous remarquerez qu'on préfère toujours séparer la partie *HTML*, du *CSS*, du code *JavaScript*.

Nous allons donc généralement créer trois fichiers distincts avec le contenu adapté : le fichier HTML ci-dessus, un fichier CSS et les fichier contenant e Script, ci-dessous.

```
body{
    margin: 0;
    overflow: hidden;
}
```

Le fichier style.css

```
const scene = new THREE.Scene();
…
```

Le fichier script.js

1.4. Installation de Three.JS

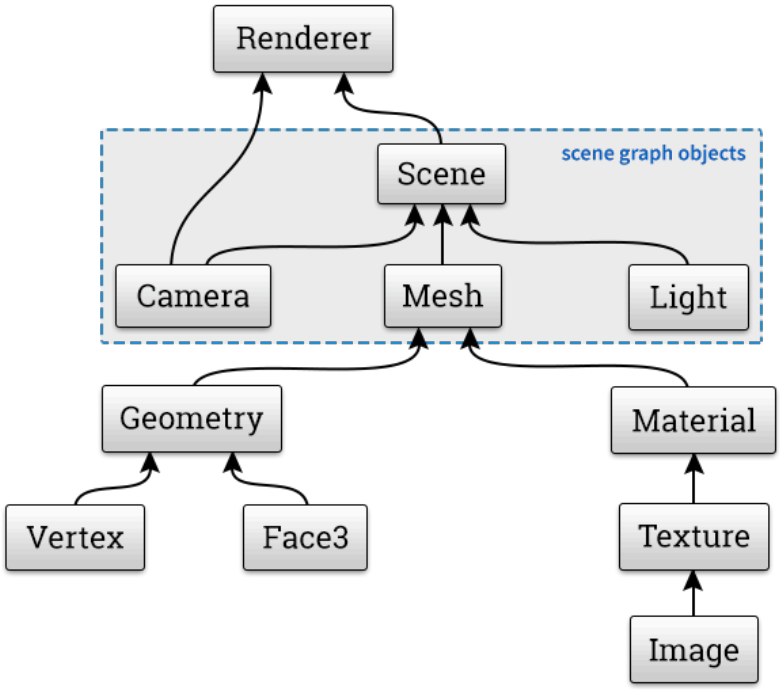
L'installation de *Three* est très simple, il suffit de télécharger la librairie https://github.com/mrdoob/three.js/releases et de la copier dans le répertoire que vous avez choisi (dans notre exemple ci-dessus le répertoire ./libs).

Nous vous conseillons de toujours créer un répertoire spécial pour chaque projet, il contiendra les fichiers *HTML*, *CSS* et *JavaScript* ainsi que les librairies que vous aurez utilisés. Les librairies changent en fonction de leur version, il est donc prudent de toujours conserver la bonne version des libraires qui ont servis à la réalisation d'un projet. Si vous préférez, vous pouvez aussi utiliser WebPack.

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 de base

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*) :

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

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

```
const 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 cube dans cette scène:

```
const geometry = new THREE.BoxGeometry(1, 1, 1)
const material = new THREE.MeshBasicMaterial({ color: 0xff0000 })
const mesh = new THREE.Mesh(geometry, material)
scene.add(mesh)
```

Nous pouvons remarquer qu'il faut préalablement créer deux objets (de type *BoxGeometry* et *MeshBasicMaterial*) pour pouvoir ensuite créer le cube (*mesh*). Nous reviendrons plus loin en détail sur ces deux points.

Notez que les trois instructions pour positionner la caméra peuvent se réduire à une seule instruction :

```
camera.position.set(-30, 40, 30);
```

On peut appliquer le même principe aux *meshes*, pour la rotation, par exemple :

```
mesh.rotation.x = 0.1;

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

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

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 :

```
// Renderer
const renderer = new THREE.WebGLRenderer({
    canvas: document.querySelector('canvas.webgl')
})
renderer.setSize(800, 600);
renderer.render(scene, camera);
```

Attention le « ***canvas.webgl*** » fait référence au nom que vous lui avez donné dans le fichier HTM plus haut.

Vous pouvez maintenant assembler toutes ces lignes de codes dans les bons fichiers et essayez de visualiser le cube dans votre *browser*.

Exercice : Créez une scène avec quelques cubes de couleurs différentes.

2.3. Les caméras de base

Nous avons vu que dans une scène de base, il fallait ajouter au minimum une caméra. Nous allons voir quels sont les deux principaux types de caméras et leurs paramètres afin de visualiser différents rendus.

Les caméras peuvent être de plusieurs types :

- Array
- Perspective
- Cube
- Stereo
- Orthographic

Les deux caméras que nous allons utiliser dans un premier temps sont les *Perspective* et *Ortographic*.

Lorsque nous écrivons l'instruction :

```
const camera = new THREE.PerspectiveCamera(75, 800/600, 1, 1000)
```

Le premier paramètre (75) est l'angle (en degré) de vision vertical de la caméra. Au delà de 75 on peut avoir de la distorsion qui peut être gênante ou intéressante en fonction de l'application. En général il est fixé entre 45 et 75.

Le deuxième paramètre est l' « aspect ratio » qui pourrait être 16/9 pour un format télé, par exemple. On va souvent calculer ce rapport en divisant la largeur (en pixel) par la hauteur de notre fenêtre de rendu.

Les deux derniers sont les paramètres proches et lointains (*near* et *far*). Ils représentent respectivement la distance en dessous et au dessus de laquelle le moteur de rendu ne tiendra pas compte des objets qui s'y trouvent. N'utilisez pas de valeurs extrêmes pour ces paramètres, elles pourraient provoquer des temps de calcul important et aussi des *glitches*. En général, on utilisera raisonnablement des valeurs de 0.1 et 100.

Exercice : En utilisant la méthode *camera.position.length()* calculez la distance entre votre caméra et votre cube et modifiez en fonction de cette distance le paramètre lointain pour voir ses effets.

Nous pouvons maintenant essayer un autre type de caméra :

```
const camera = new THREE.OrthographicCamera(- 1,1,1,-1,0.1,100)
```

La caméra orthographique ne tient pas compte de la perspective et son constructeur possède des paramètres différents :

- Les quatre premiers paramètres vont définir la fenêtre de rendu (gauche, droite, haut et bas)
- Les deux derniers paramètre sont *near* et *far* comme dans la camera en perspective

Exercice : Testez différents paramètres pour la caméra orthographique en changeant le ratio de votre fenêtre de visualisation pour voir les effets sur le cube.

Notez que vous pouvez à tout moment dans votre code changer la position ou la rotation de la caméra ou de votre cube en utilisant les propriétés de « *position* » ou de « *rotation* » :

```
mesh.position.x = 100;
camera.position.z = 20;

camera.rotation.z = 2;
```

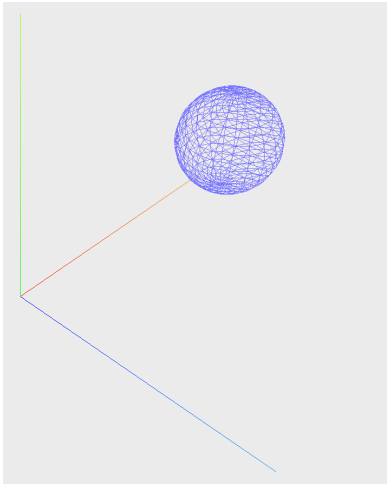
Notez égalementque vous pouvez modifier l'échelle d'un mesh avec les propriétés de « *scale* » :

```
mesh.scale.y = 3;
```

2.4. Les axes X, Y et Z

Quand on modifie la position et la rotation de la camera, on est souvent perdu au niveau du repère XYZ. Il est donc souvent utile à des fin de débuguage d'afficher les axes pour s'y retrouver :

```
// show axes in the screen
const axes = new THREE.AxesHelper(20);
scene.add(axes);
```



Le paramètre de la méthode *AxisHelper()* correspond à la longueur des axes et les couleurs représentent : X en rouge, Y en vert et Z en bleu.

2.5. 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() {
        mesh.position.x += 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 :

```
let step=0;
...
function render() {
            mesh.position.x = (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 en même temps, sur la caméra, changez leur vitesse, leur amplitude, …

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

2.6. La lumière

Il y a plusieurs types de source de lumière dans *THREE*, nous allons voir les types : *ambient*, *point*, *spot*, *directional* et *hemisphere*. Elles possèdent des caractéristiques différentes, utiles en fonction des applications que vous avez à réaliser. Vous pourrez les expérimenter dans vos projets et choisir celles qui sont la plus appropriées à votre application.

2.6.1. Spot Light

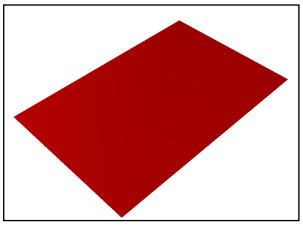
Les « *Spot Lights* » émettent de la lumière d'un seul point dans une direction donnée le long d'un cône dont la taille augmente en fonction de la distance. Ce type de lumière permet d'afficher des ombres. Testons ces lignes de code sur notre scène en ajoutant un objet *SpotLight* à la scène :

```
    const 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. Essayons, par exemple de placer un plan (de couleur rouge) dans notre scène :

```
const planeGeometry = new THREE.PlaneGeometry(30, 20, 1, 1);
const planeMaterial = new THREE.MeshLambertMaterial({
                color: 0xFF0000
            });
let plane = new THREE.Mesh(planeGeometry, planeMaterial);
```

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.



2.6.2. Ambient Light

L' « *ambiant light* » éclaire la scène globalement, tous les objets de manières égale. Ce type de lumière ne peut pas être utilisé pour afficher des ombres et elle n'a pas de direction.

```
let ambiColor = "#0c0c0c";
let ambientLight = new THREE.AmbientLight(ambiColor);
scene.add(ambientLight);
```

2.6.3. Point Light

Le « *Point Light* » est une lumière émise d'un seul point mais dans toutes les directions. On l'utilise, par exemple, pour simuler une ampoule, ce type de lumière permet d'afficher des ombres.

```
const pointColor = "#ccffcc";
let pointLight = new THREE.PointLight(pointColor);
pointLight.distance = 100;
scene.add(pointLight);
```

Parfois, à des fins de debugging on va ajouter une petite sphère sur le point lumineux pour visualiser sa position et comprendre ses effets :

```
const pointSphere = new THREE.SphereGeometry( 0.5, 16, 8 );
pointLight.add(new THREE.Mesh( pointSphere,
        new THREE.MeshBasicMaterial({color: 0xccffcc})));
```

2.6.4. Directional Light

La « *Directional Light* » émet de la lumière dans une direction spécifique en se comportant comme si les rayons étaient parallèles, venant d'une source infiniment lointaine. Ce type de lumière est communément utilisé pour simuler la lumière du jour. Elle permet aussi d'afficher des ombres. Attention les paramètres de rotation n'affecte pas ce type de lumière.

```
let pointColor = "#ff5808";
let directionalLight = new THREE.DirectionalLight(pointColor);
directionalLight.position.set(-40, 60, -10);
directionalLight.intensity = 0.5;
scene.add(directionalLight);
```

2.6.5. Hemisphere Light

Les « *Hémisphère Lights* » se positionnent au dessus de la scène avec une couleur qui provoque un dégradé entre la couleur du ciel (donnée par le premier paramètre du constructeur) et la couleur du sol (le second paramètre). Elles ne provoquent pas d'ombres.

```
let hemiLight = new THREE.HemisphereLight(0x0000ff, 0x00ff00);
hemiLight.position.set(0, 500, 0);
scene.add(hemiLight);
```

2.7. 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 spécifier que nous voulons ce rendu à différents endroits dans le code :

```
renderer.shadowMap.enabled = true;
```

La première opération est de demander à *THREE* de réaliser ces calculs à travers le propriété *shadowMap.enabled* 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. Attention, les *meshes* doivent utiliser un matériel spécifique pour afficher les ombres, par exemple, de type *THREE.MeshLambertMaterial*.

2.8. 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 :

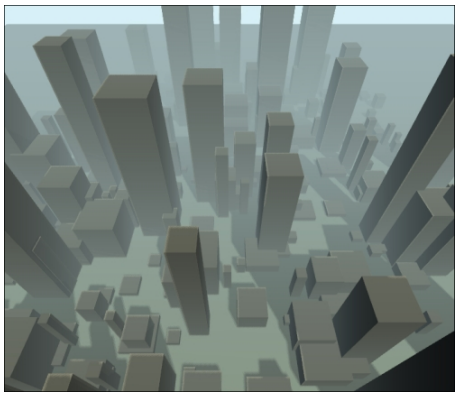
```
scene.fog=new THREE.Fog( 0xffffff, 0.015, 100 );
```

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* (que nous allons découvrir dans le chapitre suivant) en temps réel.

Exercice : Essayez de recréez une scène avec plusieurs cubes, des ombres et du brouillard pour avoir une ville « fantomatique » :



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 (*GUI*) qui va permettre d'ajuster les paramètres choisis. Il serait aussi aisé de se mouvoir dans la scène en utilisant la souris ou le clavier pour déplacer la caméra, par exemple.

3.1. La souris et le clavier

Pour récupérer les mouvements de la souris quand elle bouge, il faut créer un « *listener* » dans lequel on récupère les positions :

```
const cursor = { x: 0, y: 0 };
window.addEventListener('mousemove', (event) =>
{
    cursor.x = event.clientX / sizes.width - 0.5;
    cursor.y = event.clientY / sizes.height - 0.5;
    console.log(cursor.x, cursor.y);
})
```

Pour ensuite, les utiliser dans la fonction de la mise à jour avant le rendu :

```
    camera.position.x = cursor.x;
    camera.position.y = cursor.y;
```

Ou encore, pour un mouvement plus naturel :

```
camera.position.x = Math.sin(cursor.x * Math.PI * 2) * 2;
camera.position.z = Math.cos(cursor.x * Math.PI * 2) * 2;
camera.position.y = cursor.y * 3;
camera.lookAt(scene.position);
```

Une autre manière, plus rapide, mais moins paramétrable pour faire bouger la caméra avec la souris est d'utiliser une librairie externe. Par exemple, en ajoutant dans le fichier *HTML* de base :

```
<script src="./libs/orbitcontrols.js"></script>
```

Et ensuite dans votre code d'initialisation :

```
const controls = new THREE.OrbitControls(camera,
                                                renderer.domElement);
controls.enableDamping = true;
```

et simplement dans la fonction de rendu :

```
controls.update();
```

Tout comme pour la souris, le plus simple pour contrôler les mouvements de la caméra avec le clavier est d'utiliser la librairie *OrbitControls* en spécifiant :

```
controls.listenToKeyEvents(window);
```

3.2. Performances

Le rendu de votre scène s'effectue au meilleur *framerate* possible. Il est souvent intéressant de vérifier ce paramètre. Pour ce faire, vous pouvez utiliser la librairie Stats :

```
<script src="../libs/stats.min.js"></script>
```

Ajouter dans votre code :

```
stats = new Stats();
stats.showPanel(0); // 0: fps, 1: ms, 2: mb, 3+: custom
document.body.appendChild( stats.dom );
```

Et ne pas oublier dans le *render()* :

```
stats.update();
```

3.3. GUI

Construire une interface graphique (*GUI*) avec *Three* est un projet très conséquent. Heureusement, certaines personnes y ont 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 un répertoire 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 :

```
let controls = new function () {
        this.rotationSpeed = 0.02;
    };
```

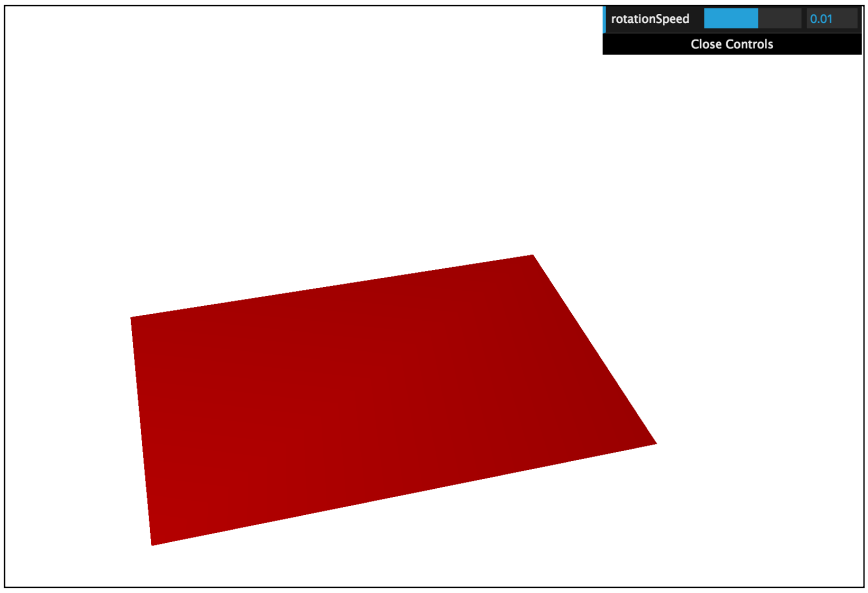
… et les utiliser dans la fonction de rendu :

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

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

```
let gui = new dat.GUI();
gui.add(controls, 'rotationSpeed', 0, 0.5);
```

Le résultat est 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*.

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.

Si vous voulez faire votre rendu en plein écran vous pouvez au moment de sa création faire :

```
renderer.setSize(window.innerWidth, window.innerWidth);
```

Mais il restera sans doute une petite marge que vous pouvez faire disparaître en réalisant un petit ajout au *CSS* :

```
body {
    margin: 0;
    overflow: hidden;
}
```

5. Les objets d'une scène

Nous avons déjà vu plus haut comment créer des cubes et des plans. Il existe un grand nombre d'autres objets (*geometry*) pré-existant dans *Three* : *Plane*, *Circle*, *Ring*, *Sphere*, *Cylinder*, *Torus*, *Polyhedron*, *Octahedron*, *TetraHedron*, *Text*, *Dodecahedron*, … Nous allons d'abord voir quelques objets standards, puis nous verrons comment faire nos propres *Geometry* à base de *Vertices* et finalement comment afficher du *Text*.

5.1. Les objets standards

Voici comment utiliser quelques *Geometry* standards prédéfinies dans THREE.

```
const sphereGeometry = new THREE.SphereGeometry(4, 20, 20);
const sphereMaterial = new THREE.MeshBasicMaterial({
                          color: 0x7777ff,
                          wireframe: true});
const sphere = new THREE.Mesh(sphereGeometry, sphereMaterial);
scene.add(sphere);
```

Vous remarquerez, dans cet exemple, le *wireframe* est à *true* ce qui permet une visualisation en « fil de fer ».

De la même manière vous pouvez essayer :

```
THREE.CylinderGeometry( 5, 5, 20, 32);
THREE.OctahedronGeometry(3)
THREE.TorusGeometry( 10, 3, 16, 100)
THREE.TetrahedronGeometry(3)
THREE.TorusKnotGeometry(3, 0.5, 50, 20)
THREE.IcosahedronGeometry(4)
THREE.RingGeometry( 1, 5, 32)
```

Pour d'autres *Geometries* vous pouvez vous référer à la documentation :
https://threejs.org/docs/#api/en/geometries/BoxGeometry

Il existe aussi des librairies complémentaires, telles que :

- *ParametricGeometries.js*
- *ConvexGeometry.js*

… qui permettent d'accéder à toutes sortes de formes prédéfinies.

5.2. *Vertices*, *geometries* et *meshes*

Nous venons de voir plusieurs *geometries* qui sont déjà prêtes à être utilisées. 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*.

Nous allons donc pouvoir maintenant définir un cube sans utiliser la *Geometry box* fournie par *Three*. Définissons d'abord un tableau avec les huit sommets (*vertices*) :

```
Let 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) :

```
Let 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* :

```
const 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* :

```
const material = new THREE.MeshBasicMaterial({color: 0x7777ff);
const 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*) :

```
Let 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.

5.3. 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*.

Vous devez ensuite utiliser les deux librairies :

```
    <script src="../libs/loaders/FontLoader.js"></script>
    <script src="../libs/TextGeometry.js"></script>
```

Pour charger la *font* et créer sa géométrie :

```
let loader = new THREE.FontLoader();
loader.load('helvetiker_bold.typeface.json', function (font) {
   let textGeo = new THREE.TextGeometry("THREE.JS", {
                    font: font,
                    size: 30,
                    height: 5,
                    curveSegments: 12,
                    bevelThickness: 1,
                    bevelSize: 1,
                    bevelEnabled: true
                });
textGeo.computeBoundingBox();
let textMaterial = new THREE.MeshPhongMaterial({
                    color: 0xff0000,
                    specular: 0xffffff
                });
let 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*).

5.4. Ajout, modification 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 des cubes (position aléatoire et couleur aléatoire) et à chaque *beat* d'une musique, faites tourner tous les cubes ensemble.

5.5. Les textures

Nous avons vu que chaque mesh reçoit une texture à sa création. Ces textures peuvent être de plusieurs types : couleur (*albedo*), *alpha*, *height*, *normal*, *ambient occlusion*, *metalness*, *roughness*.

5.5.1. Loader

Pour utiliser ces textures, il faut d'abord charger une image correspondant à cette texture. Avec *THREE* et *javascript*, on peut utiliser le *loading manager* :

```
const loadingManager = new THREE.LoadingManager()
loadingManager.onStart = () => {
    console.log('loading started')
}
loadingManager.onLoad = () => {
    console.log('loading finished')
}
loadingManager.onProgress = () => {
    console.log('loading progressing')
}
loadingManager.onError = () => {
    console.log('loading error')
}
const textureLoader = new THREE.TextureLoader(loadingManager)
```

Il faut ensuite faire appel au *loader* pour charger les images :

```
const colorTexture = textureLoader.load('/textures/door/color.jpg')
const alphaTexture = textureLoader.load('/textures/door/alpha.jpg')
const heightTexture = textureLoader.load('/textures/door/
height.jpg')
const normalTexture = textureLoader.load('/textures/door/
normal.jpg')
const ambientOcclusionTexture = textureLoader.load('/textures/door/
ambientOcclusion.jpg')
const metalnessTexture = textureLoader.load('/textures/door/
metalness.jpg')
const roughnessTexture = textureLoader.load('/textures/door/
roughness.jpg')
```

Pour pouvoir ensuite les utiliser :

```
const material = new THREE.MeshBasicMaterial({ map: colorTexture })
```

5.5.2. Transformation

On peut à travers divers attributs modifier les propriétés de la texture, par exemple :

```
colorTexture.repeat.x = 2;
colorTexture.repeat.y = 3;
colorTexture.wrapS = THREE.RepeatWrapping;
colorTexture.wrapT = THREE.RepeatWrapping;
```

On peut modifier la manière dont le motif se répète :

```
colorTexture.wrapS = THREE.MirroredRepeatWrapping;
colorTexture.wrapT = THREE.MirroredRepeatWrapping;
```

En ajoutant éventuellement un offset :

```
colorTexture.offset.x = 0.5;
colorTexture.offset.y = 0.5;
```

Ou en le faisant tourner :

```
colorTexture.rotation = Math.PI * 0.25;
colorTexture.center.x = 0.5;
colorTexture.center.y = 0.5;
```

Vous pouvez aussi regarder les différentes valeurs possibles pour :

```
colorTexture.magFilter = THREE.LinearFilter;
colorTexture.generateMipmaps = false;
colorTexture.minFilter = THREE.NearestFilter;
```

Vous pourrez trouver des textures pour vous entrainer à ces adresses :

- poliigon.com
- 3dtextures.me
- arroway-textures.ch

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 jouer un fichier *mp3* dans une page *HTML*, il faut d'abord créer une « interaction » avec la page. En effet, pour éviter que des sites *Internet* vous agressent avec de la musique non désirée, il faut que l'utilisateur interagisse avec la page (via un bouton, par exemple) pour pouvoir lire un fichier musical. Nous avons donc d'abord créer un *button* :

```
<button type="button" onclick="init()">Play Music</button>
```

Et ensuite le code qui permet de lire le fichier audio :

```
<script type="text/javascript">
    function init(){
          let audio = new Audio();
          audio.preload = "auto";
          audio.src = "yourPathToYourMusic.mp3";
          audio.play();
    }
</script>
```

6.3. Analyse de musiques en *Javascript*

Pour automatiser une partie de l'animation sur la musique, on peut analyser les fréquences importantes dans le morceau joué, en temps réel, et s'en servir pour modifier certains paramètres de notre animation. Pour créer cet analyseur, nous allons d'abord créer un « contexte audio » :

```
const AudioContext = window.AudioContext || window.webkitAudioContext;
const audioContext = new AudioContext();
```

Ensuite créer un « *noeud audio* » et un « *noeud analyseur* », suite au *play()* :

```
let audio = new Audio();
audio.preload = "auto";
audio.src = "PathToYourMusic.mp3";
audio.play();
let audioSourceNode = audioContext.createMediaElementSource(audio);

analyserNode = audioContext.createAnalyser();
analyserNode.fftSize = 256;
dataArray = new Uint8Array(analyserNode.frequencyBinCount);
```

Et finir par créer les connexions audio :

```
audioSourceNode.connect(analyserNode);
analyserNode.connect(audioContext.destination);
```

Il ne reste qu'à utiliser les données dans la fonction de rendu :

```
analyserNode.getByteFrequencyData(dataArray);
for (let i = 0; i < analyserNode.frequencyBinCount; i++) {
            console.log(dataArray[i]);
            }
 plane.rotation.z = dataArray[5]/500;
```

Exercice : Réaliser un analyseur de fréquences

6.4. 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 :

```
let 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*) :

```
let startTime;

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

function render() {
    let 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*.

6.5. 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 :
- Chercher le BPM du morceau
- Découper temporellement le morceau en différentes « ambiances » , séquences
- Analyser le morceau en repérant précisément les montées/descentes, moments calmes/agités, forts/faibles, … faire un dessin …
- Première scène avec un cube
- Animer le cube sur le BPM

Deuxième journée :
- Lumières, Ombres et Brouillard
- 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 ») et synchroniser sur le BPM.

Troisième journée :
- Les formes supplémentaires
- Les meshes persos
- Modification d'objets pour animation
- Redimensionnement de la fenêtre
- FFT à appliquer aux animation

Quatrième journée :
- Finaliser le projet
- Présentation des projets finaux