

THREEJS

Créé par [Alexandre Rivet](#)

INTRODUCTION

DÉFINITION

- Une API JavaScript permettant de faire des rendus graphiques avancés très rapidement dans un navigateur Web
- Utilise le GPU (Graphic Processing Unit) pour faire le rendu très rapidement
- Compatible ([caniuse](#)) avec la plupart des navigateurs web: Google Chrome, Safari, Firefox, Edge, etc.

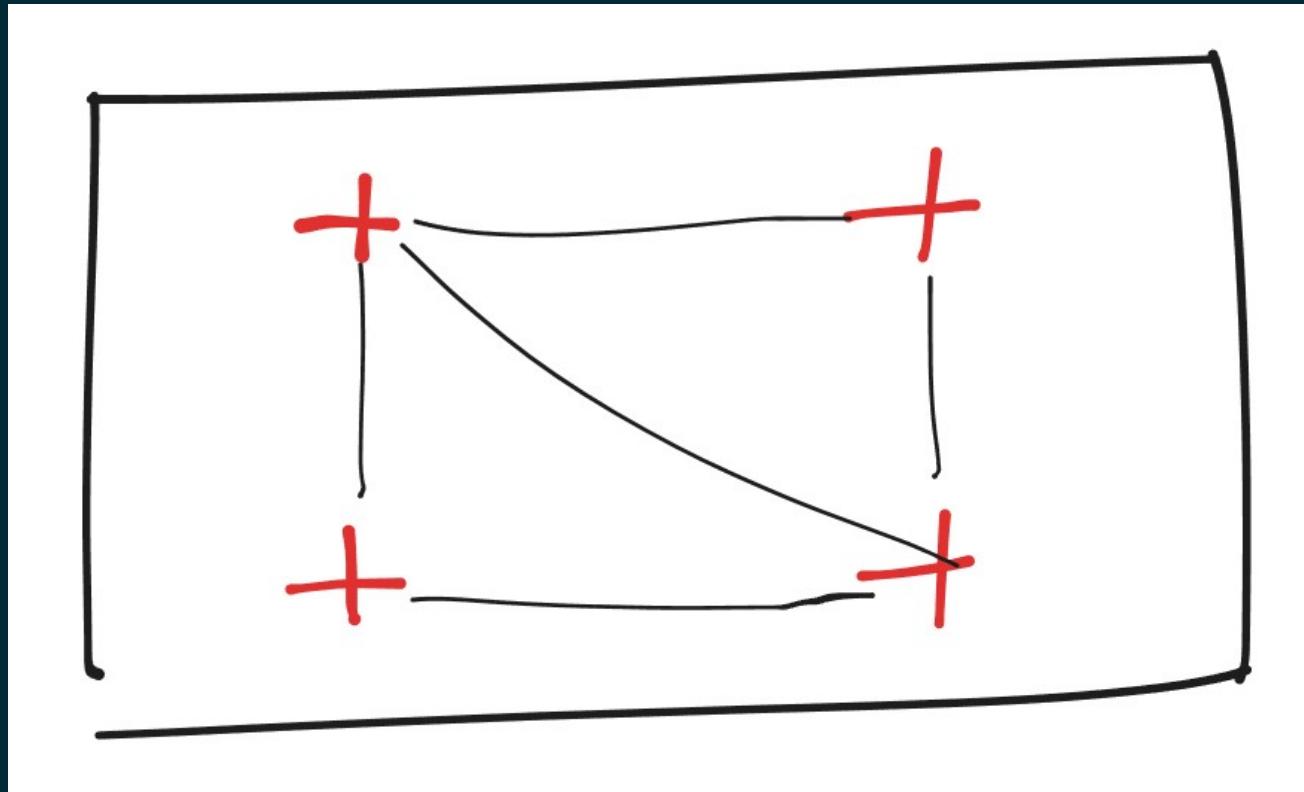
CPU VS GPU

- Le CPU peut faire des opérations complexes très rapidement les unes après les autres
- Le GPU est un peu plus lent mais peut faire des milliers d'opérations en parallèle

COMPRENDRE COMMENT RENDRE UN MODÈLE 3D (1/6)

- Un modèle 3D est composé de milliers de triangles. Chaque triangle est composé de 3 points
- Une fois que le modèle 3D est correctement placé et donc les points associés aussi, le GPU va dessiner chaque pixel à partir des triangles à l'écran
- Important de dire que le GPU va dessiner tout en une seule fois et c'est pourquoi cela est très rapide

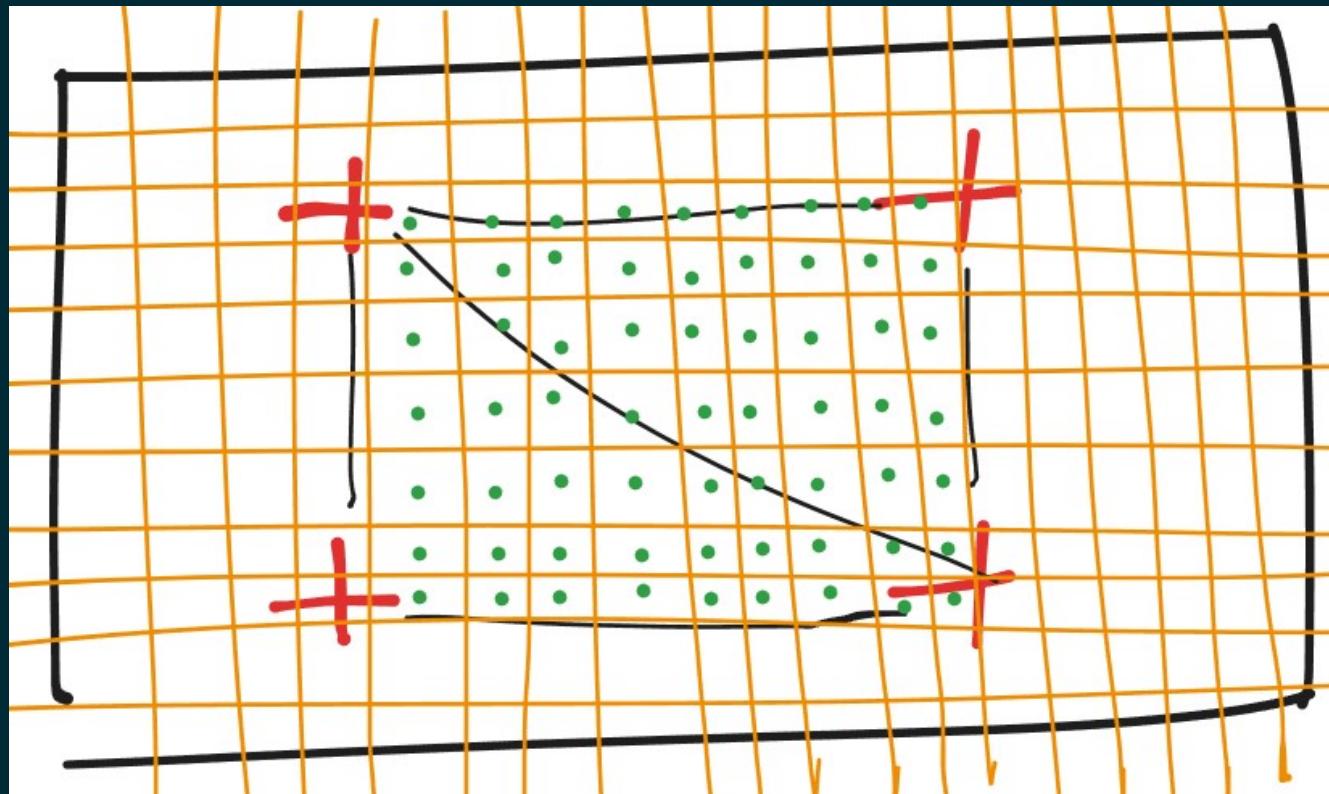
COMPRENDRE COMMENT RENDRE UN MODÈLE 3D (2/6)



COMPRENDRE COMMENT RENDRE UN MODÈLE 3D (3/6)

- Pour positionner les points à l'écran, colorier les pixels, tout cela va se passer dans ce qu'on appelle les shaders
- Pour rendre ça correctement, on va devoir envoyer des informations à ces shaders, comme la position des points, les infos de caméra, etc.
- Et le GPU choisira quels pixels colorier et comment

COMPRENDRE COMMENT RENDRE UN MODÈLE 3D (4/6)



COMPRENDRE COMMENT RENDRE UN MODÈLE 3D (5/6)



Le WebGL est donc assez complexe et pour dessiner le carré du dessus, il faudrait au moins une centaine de lignes de code.

COMPRENDRE COMMENT RENDRE UN MODÈLE 3D (6/6)

Manipuler l'API WebGL apporte quand même son lot de bénéfices comme:

- Mieux contrôler les instructions qui sont exécutées sur le GPU
- Qui dit mieux contrôler, dit aussi de pouvoir mieux optimiser car les performances sont cruciales dans les expériences WebGL

EXEMPLES DE RÉALISATION EN WEBGL

- makemepulse
- Bruno Simon
- Prometheus
- Lusion
- ThreeJS Editor

THREEJS À LA RESCOUSSE (1/2)

- Librairie JavaScript
- Plus haut niveau que le WebGL natif
- Crée par Ricardo Cabello aka Mr.doob ([website](#))

THREEJS À LA RESCOUSSE (2/2)

- Supportée par une grande communauté et subie des **mises à jour** tous les mois
- Une quantité de projets utilisant ThreeJS: [website](#)
- Une très bonne **documentation**

POURQUOI THREEJS PLUTÔT QU'UNE AUTRE ?

- La librairie la plus populaire pour les raisons énoncées précédemment
- Les autres librairies à explorer sont: [babylon.js](#), [PLAYCANVAS](#)

ENVIRONNEMENT DE TRAVAIL

EDITEURS

- Editeurs de textes simples: Sublime Text, Visual Studio Code (recommandé)
- Des environnements de développement intégrés (IDE): WebStorm, IntelliJ IDEA
- Editeurs en ligne: CodePen, JSFiddle

NAVIGATEURS

- Google Chrome (recommandé)
- Mozilla Firefox
- Safari
- Microsoft Edge

PREMIER PROJET

SERVEUR LOCAL & BUNDLER

Pour lancer notre expérience WebGL, nous allons avoir besoin d'un serveur local.

La principale raison est que certains navigateurs bloquent des fonctionnalités pour des raisons de sécurité.

Et pour pouvoir résoudre ce problème, la solution la plus simple est de passer par un bundler.

LE BUNDLER

Il existe pleins d'outils qui buildent le code comme:

- Webpack
- Vite
- Parcel
- ...

VITE

Webpack est l'outil le plus populaire mais ces derniers jours la tendance a l'air de tendre vers Vite et c'est ce que nous allons utiliser dans tout ce cours.

Il est plus rapide à s'installer, plus rapide à s'exécuter et tend à avoir moins de bug.

En bref bien plus agréable pendant le développement.

NODEJS (1/2)

Pour exécuter Vite, nous allons avoir besoin de Node.js
Pour savoir si Node.js est déjà installé ou même quelle version est déjà installée, vous pouvez lancer dans votre terminal:

```
node -v
```

- Commande non reconnue, donc Node.js non installé. Aller sur [Node.js](#) et télécharger la LTS puis l'installer.
- La version s'affiche, donc Node.js est bien installé. Vérifier qu'une version > 14.18 est installée pour fonctionner avec Vite

NODE.JS (2/2)

Créer un dossier qui contiendra notre expérience
WebGL et l'ouvrir sur VSCode (ou autre)
Une fois VS ouvert, ouvrir le terminal et lancer la
commande:

```
npm init -y
```

DÉPENDANCES

- Vite

```
npm i vite
```

- ThreeJS

```
npm i three
```

FICHIER HTML

Créer un fichier index.html à la racine du dossier

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ma page web</title>
  </head>
  <body>
    Ma page web
  </body>
</html>
```

LANCER VITE

Dans le fichier package.json, remplacer le script par défaut test par:

```
// ...
"scripts": {
  "dev": "vite"
},
// ...
```

Dans le terminal, exécuter la commande:

```
npm run dev
```

Et vous devriez pouvoir accéder à [la page web](#)

AJOUTER LE JAVASCRIPT (1/2)

Créer un fichier main.js à côté du fichier index.html
avec un console.log

```
console.log('Hello from JavaScript');
```

AJOUTER LE JAVASCRIPT (2/2)

Ajouter la balise script à la fin du body pour charger
votre fichier JavaScript depuis le fichier HTML

```
<script type="module" src="./main.js">
```

UTILISER THREEJS

Importer three dans le fichier main.js et afficher le résultat dans la console

```
import * as THREE from 'three';

console.log(THREE);
```

A l'intérieur, il y a la majorité des classes nécessaires pour créer une expérience WebGL

PREMIÈRE SCÈNE (1/14)

Pour afficher quelque chose à l'écran, nous allons avoir besoin de plusieurs choses:

- Une scène THREE.Scene et des objets dans cette scène
- Une caméra
- Un moteur de rendu

PREMIÈRE SCÈNE (2/14) - SCENE

C'est une boîte dans laquelle se trouvera les objets, les modèles 3D, les particules, les lumières, etc.

Pour créer la scène:

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

PREMIÈRE SCÈNE (3/14) - OBJETS & GÉOMÉTRIES

Les objets peuvent être de tous types: des géométries simples, des modèles 3D, des particules, des lumières, etc.

Nous allons commencer avec un cube bleu

PREMIÈRE SCÈNE (4/14) - OBJETS & GÉOMÉTRIES

On parle très souvent de **Mesh** qui est une combinaison de la géométrie (donc la forme) et d'un matériaux (à quoi il ressemble)

Dans Three, il existe pleins de géométries et matériaux différents.

Dans notre cas du cube bleu:

```
const boxGeometry = new THREE.BoxGeometry(1, 1, 1);
const boxMaterial = new THREE.MeshBasicMaterial({ color: 0x0000ff });
const boxMesh = new THREE.Mesh(boxGeometry, boxMaterial);
scene.add(boxMesh);
```

PREMIÈRE SCÈNE (5/14) - CAMÉRAS

La caméra est quelque chose de théorique mais qui n'est pas visible.

Quand on veut rendre une scène, il faut un point de vue depuis laquelle effectuer le rendu.

Comme les humains, dans une pièce, pour tous le monde il y a les mêmes objets, mais chacun les perçoit depuis ses yeux (notre caméra à nous).

PREMIÈRE SCÈNE (6/14) - CAMÉRAS

Il existe des notions très importantes sur les caméras:

- champs de vision: c'est l'angle d'ouverture de la caméra. Plus la valeur est grande, plus l'ouverture aussi mais cela peut résulter à des déformations. A l'inverse, un angle petit résultera sur une scène beaucoup plus zoomée. Dans ThreeJS cette valeur d'angle est exprimée en degrés.
- ratio d'aspect: C'est le rapport entre la largeur et la hauteur de la zone dans laquelle on effectue le rendu (le canvas dans notre cas)

PREMIÈRE SCÈNE (7/14) - CAMÉRAS

Pour créer une PerspectiveCamera:

```
const viewportSize = {  
    width: 800,  
    height: 600  
};  
  
const camera = new THREE.PerspectiveCamera(  
    75, // FOV  
    viewportSize.width / viewportSize.height // aspect ratio  
);  
// Les caméras sont aussi des objets, donc on les ajoute aussi à la scène  
scene.add(camera);
```

PREMIÈRE SCÈNE (8/14) - MOTEUR DE RENDU

Sans aucune surprise, c'est ce qui va nous permettre de faire le rendu. On demande simplement de faire le rendu de notre scène via une certaine caméra et il s'occupe du reste.

Le résultat de ce rendu doit être dessiner dans un canvas

Remplacer "Ma page web" par:

```
<canvas id="webgl"></canvas>
```

PREMIÈRE SCÈNE (9/14) - MOTEUR DE RENDU

Pour instancier le moteur de rendu:

```
const canvas = document.querySelector('canvas#webgl');
const renderer = new THREE.WebGLRenderer({
  canvas,
});
renderer.setSize(viewportSize.width, viewportSize.height);
```

Après cette étape, on ne verra rien à l'écran mais le canvas a été modifié par Three.

PREMIÈRE SCÈNE (10/14) - RENDU

Tout est prêt pour demander au moteur de rendu
d'effectuer le rendu:

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

PREMIÈRE SCÈNE (11/14) - RENDU



Après cette étape, on se serait attendu à voir quelque part notre cube bleu, mais on obtient qu'un écran noir.

PREMIÈRE SCÈNE (12/14) - RENDU

Quand on a créé nos objets (cube et caméra), on n'a jamais spécifié de position. Donc ils sont tous à la position par défaut, le centre de la scène.

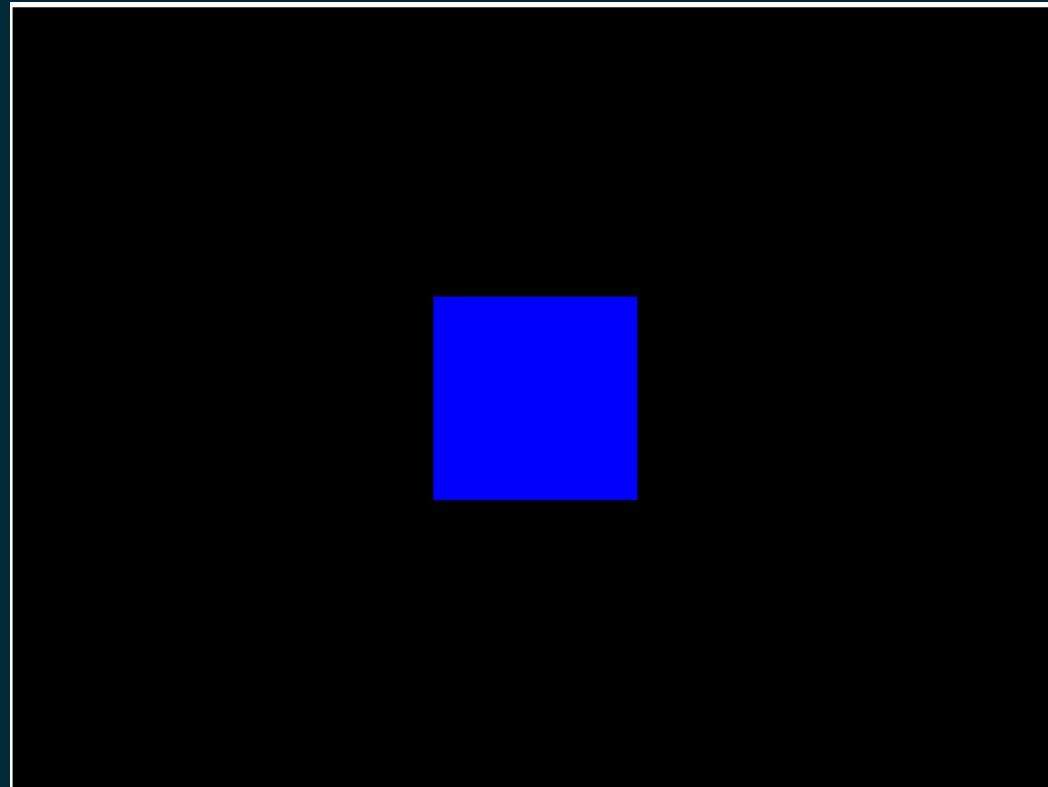
Donc finalement, la caméra est à l'intérieur du cube et notre matériaux n'est pas paramétré pour être vu de l'extérieur.

Pour résoudre ça, on va déplacer notre caméra:

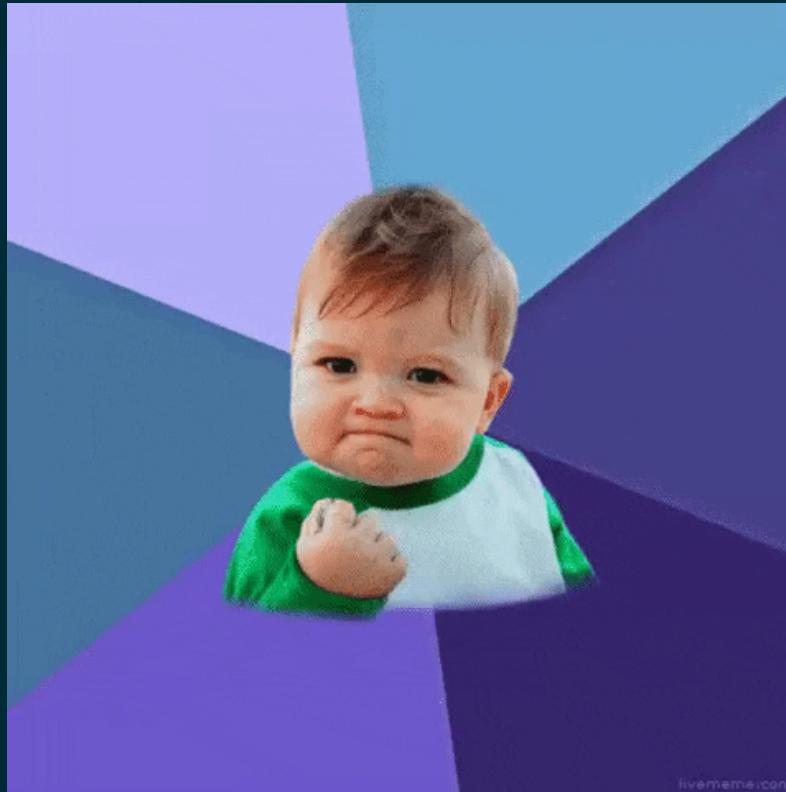
```
camera.position.z = 3;
```

PREMIÈRE SCÈNE (13/14) - RENDU

Voilà le résultat que l'on devrait obtenir après nos derniers changements



PREMIÈRE SCÈNE (14/14) - RENDU

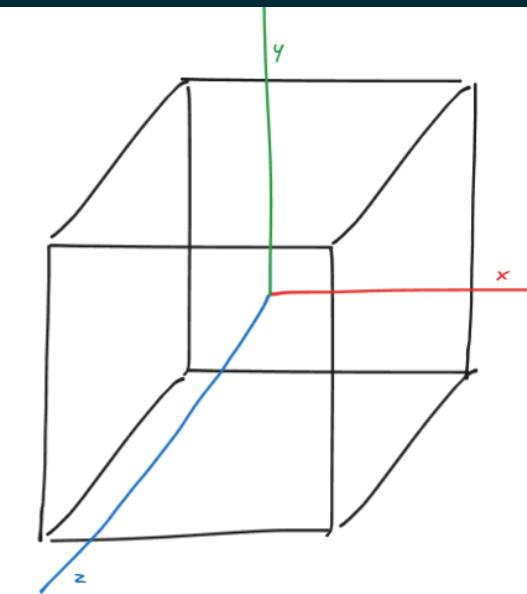


Et ça y est on a réussi à avoir notre premier rendu avec ThreeJS !

COMPRENDRE LA HIÉRARCHIE DES OBJETS

LE REPÈRE (1/2)

Un repère 3D est composé de 3 axes: X, Y, Z.
Il existe 2 types de repère: repère main gauche ou
repère main droite. La seule différence entre les 2 est le
sens de l'axe des Z



LE REPÈRE (2/2)

Il existe dans `three` `AxesHelper` pour aider à visualiser
le repère

```
const axesHelper = new THREE.AxesHelper();
scene.add(axesHelper);
```

TRANSFORMER UN OBJET

Il existe 4 propriétés pour transformer un objet

- position
- rotation
- scale
- quaternion

Toutes ces propriétés vont être compilés dans des matrices

POSITION (1/4)

La propriété position va permettre de déplacer un objet.

Elle possède 3 propriétés:

- x
- y
- z

La valeur par défaut est de (0, 0, 0)

POSITION (2/4)

Les valeurs qui sont utilisées pour déplacer les objets et donc l'unité est arbitraire. Il faut juste imaginer que 1 représente 1 unité et que cette unité peut représenter n'importe quoi en fonction de ce l'on est en train de développer.

POSITION (3/4)

Pour comprendre, mettre à jour la position du mesh:

```
boxMesh.position.x = 1;  
boxMesh.position.y = -1;  
boxMesh.position.z = -1;  
  
camera.position.y = 1;  
camera.position.x = 1;
```

POSITION (4/4)

position hérite de `Vector3` qui expose pleins de méthodes différentes et d'opérations mathématiques sur les vecteurs:

- `set(x, y, z)`
- `distanceTo(anotherVec3)`
- `angleTo(anotherVec3)`
- `dot(anotherVec3)`
- ...

SCALE (1/2)

La propriété `scale` va permettre de changer l'échelle d'un objet (sa taille).

Elle possède 3 propriétés:

- x
- y
- z

La valeur par défaut est de (1, 1, 1)

SCALE (2/2)

Pour comprendre, mettre à jour le scale du mesh:

```
boxMesh.scale.x = 2;  
boxMesh.scale.y = 0.5;  
boxMesh.scale.z = 0.25;
```

ROTATION (1/6)

Il existe 2 manière de mettre à jour la rotation sur un objet:

- rotation
- quaternion

Note: Mettre à jour l'une des deux propriétés met à jour l'autre automatiquement

ROTATION (2/6)

La propriété rotation va permettre de changer l'orientation d'un objet.

Elle possède 3 propriétés:

- x
- y
- z

La valeur par défaut est de (0, 0, 0)

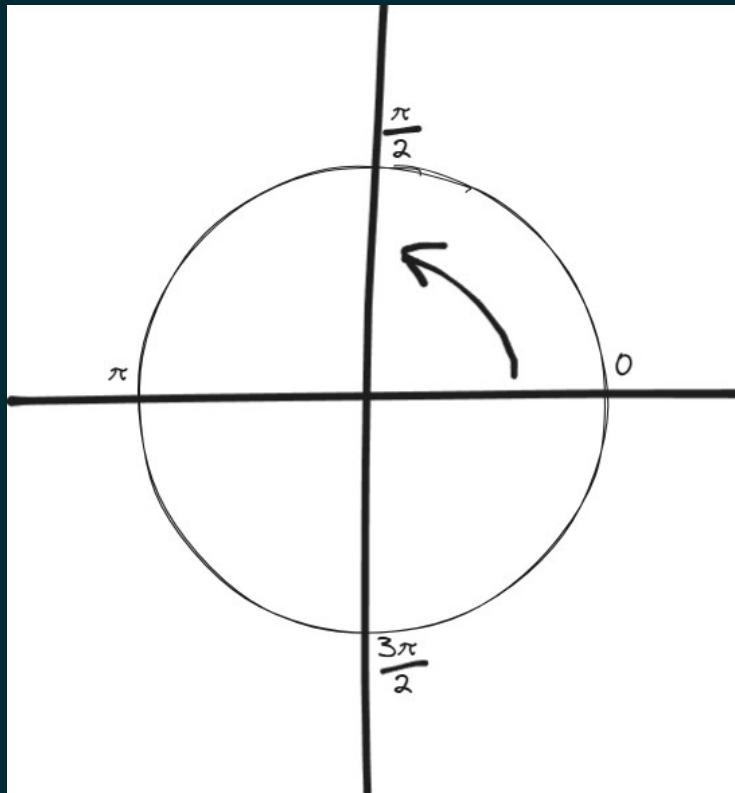
ROTATION (3/6)

rotation hérite de [Euler](#) qui expose quelques méthodes différentes et d'opérations mathématiques sur les angles d'Euler:

- `set(x, y, z, order)`
- `...`

ROTATION (4/6)

Les angles sont exprimées en radians et non en degrés.
Cela veut dire que les valeurs sont exprimées en
fonction de PI



ROTATION (5/6)

Pour comprendre, mettre à jour la rotation du mesh:

```
boxMesh.rotation.z = Math.PI * 0.25;
```

ROTATION (6/6)

Dans les angles d'Euler, on peut changer l'ordre des rotations grâce à la méthode `reorder`
En fonction de l'ordre des axes, le résultat de la rotation ne sera pas le même:

```
boxMesh.rotation.reorder('YXZ');
```

Note: Toujours changer l'ordre avant de changer la rotation elle-même sinon cela n'aura aucun effet

QUATERNION

On a vu que l'ordre des axes pouvaient devenir problématique. Et pour contourner ça, il existe le

Quaternion qui utilise d'autres formules.

Le quaternion représente aussi la rotation de manière plus mathématique et est utilisé dans de nombreux logiciels pour s'affranchir de pleins de problèmes liés aux angles d'Euler

On ne rentrera pas plus en détails dans les Quaternions

DIRIGER UN OBJET VERS UN AUTRE

Il arrive parfois qu'on souhaite faire en sorte qu'un objet regarde dans la direction d'un autre. Typiquement une caméra où on voudrait qu'elle regarde un objet précis.

A la place de devoir faire des calculs qui peuvent s'avérer complexes, il existe une méthode sur Object3D:

```
camera.lookAt(boxMesh.position);
```

LES GROUPES D'OBJETS

Il peut être contraignant de devoir bouger plusieurs objets en même temps si on doit le faire un par un. Il existe une manière de grouper les objets de sorte de pouvoir changer la position, le scale et la rotation du groupe qui affectera tous les objets à l'intérieur

```
const group = new THREE.Group();
scene.add(group);
// Et maintenant on peut faire group.add(Anything)
```

LES OBJETS

LES GÉOMÉTRIES (1/4)

Qu'est ce qu'une géométrie ?

- Composée de vertices (points 3D)
- Composée de faces (triangles)
- Utilisée dans les meshes
- Peut stocker d'autres informations (uv coordinates, normales, colors, etc.)

LES GÉOMÉTRIES (2/4)

Il existe pleins de géométries déjà prêtes à l'emploi (dont `BoxGeometry` que l'on a déjà croisé) et elles héritent toutes de la classe `BufferGeometry` qui expose pleins de méthodes pour manipuler la géométrie

LES GÉOMÉTRIES (3/4)

Voilà une petite liste non exhaustives des géométries disponibles:

- BoxGeometry
- CapsuleGeometry
- CircleGeometry
- ConeGeometry
- CylinderGeometry
- DodecahedronGeometry
- IcosahedronGeometry
- LatheGeometry
- OctahedronGeometry

LES GÉOMÉTRIES (4/4)

- PlaneGeometry
- RingGeometry
- ShapeGeometry
- SphereGeometry
- TetrahedronGeometry
- TorusGeometry
- TorusKnotGeometry
- TubeGeometry

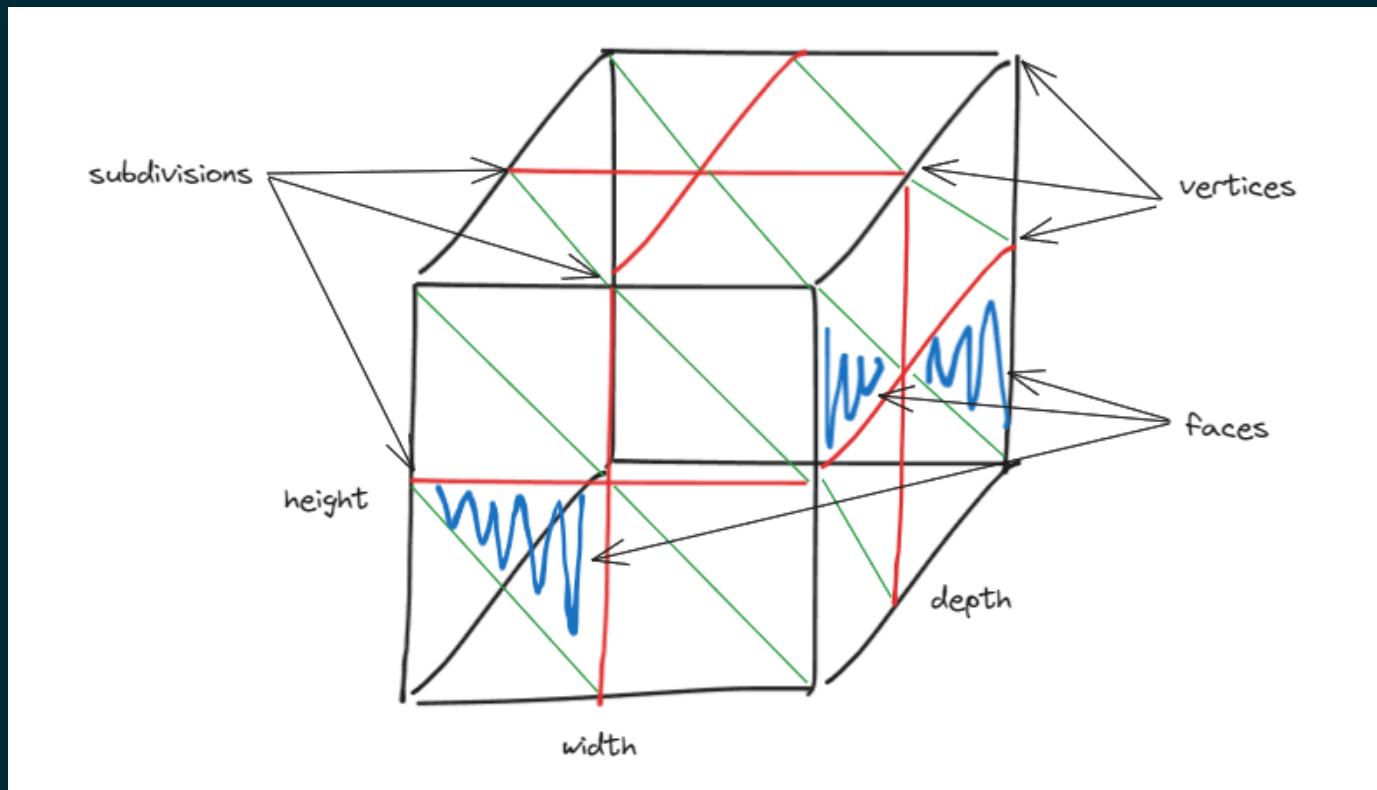
Note: Dans la documentation de chacun, vous trouverez tout ce qu'il faut pour jouer avec les paramètres de chacun

BOXGEOMETRY (1/3)

La Box Geometry peut prendre jusqu'à 6 paramètres:

- width: la taille sur l'axe des x
- height: la taille sur l'axe des y
- depth: la taille sur l'axe des z
- widthSegments: le nombre de sous divisions sur l'axe des x
- heightSegments: le nombre de sous divisions sur l'axe des y
- depthSegments: le nombre de sous divisions sur l'axe des z

BOXGEOMETRY (2/3)



BOXGEOMETRY (3/3)

Pour tester cela:

```
const boxGeometry = new THREE.BoxGeometry(1, 1, 1, 2, 2, 2);
```

A ce stade on ne voit aucune différence, mais on peut changer n'importe quel material de sorte de voir l'objet en mode filaire

```
const boxMaterial = new THREE.MeshBasicMaterial({  
    color: 0x0000ff,  
    // Rajouter cette propriété pour changer le mode de rendu filaire  
    wireframe: true  
});
```

GÉOMÉTRIE PERSONNALISÉE (1/9)

On a vu qu'il y avait donc une notion de vertices et de faces qui était à la base de tout géométrie.

Avant de rentrer dans les détails, il faut comprendre dans quoi sont stockées ces informations.

GÉOMÉTRIE PERSONNALISÉE (2/9)

On parle de `Float32Array`

- Un tableau typé: `Float32` et seulement ça à l'intérieur
- Un tableau avec une taille fixe
- C'est donc plus simple pour être utilisé par l'ordinateur

GÉOMÉTRIE PERSONNALISÉE (3/9)

Créer un tableau qui va contenir les points pour faire un triangle

```
const verticesArray = new Float32Array(9);
verticesArray[0] = 0;
verticesArray[1] = 0;
verticesArray[2] = 0;
verticesArray[3] = 0;
verticesArray[4] = 1;
verticesArray[5] = 0;
verticesArray[6] = 1;
verticesArray[7] = 0;
verticesArray[8] = 0;
```

GÉOMÉTRIE PERSONNALISÉE (4/9)

Transformer cet array en **BufferAttribute**

```
const verticesAttribute = new THREE.BufferAttribute(verticesArray, 3);
```

Le 3 justement est là pour dire combien on a de valeur par vertex, dans notre cas c'est 3 (pour x, y et z)

GÉOMÉTRIE PERSONNALISÉE (5/9)

Créer la géométrie avec BufferGeometry

```
const triangleGeometry = new THREE.BufferGeometry();
/*
* position est quelque chose de connue dans THREE
* important d'utiliser ce nom là pour que cela fonctionne
* avec les shaders de THREE
*/
triangleGeometry.setAttribute('position', verticesAttribute);
const triangleMaterial = new THREE.MeshBasicMaterial({
  color: 0xff0000,
  wireframe: true
});
const triangleMesh = new THREE.Mesh(triangleGeometry, triangleMaterial);
scene.add(triangleMesh);
```

Le 3 justement est là pour dire combien on a de valeur par vertex, dans notre cas c'est 3 (pour x, y et z)

GÉOMÉTRIE PERSONNALISÉE (6/9)

Dans le cas présent, on n'a pas eu à préciser les faces, THREE s'en est chargé pour nous. Mais il arrivera pleins de fois où on doit préciser les faces notamment dans le cas où un vertex est utilisé par plusieurs faces

GÉOMÉTRIE PERSONNALISÉE (7/9)

On va essayer de faire un carré à la place du triangle.

Pour faire un carré, on aura 4 vertices pour les 4 sommets de notre carré et il nous faut 2 triangles pour le faire

```
const squareVerticesArray = new Float32Array(12);
squareVerticesArray[0] = 0;
squareVerticesArray[1] = 0;
squareVerticesArray[2] = 0;
squareVerticesArray[3] = 0;
squareVerticesArray[4] = 1;
squareVerticesArray[5] = 0;
squareVerticesArray[6] = 1;
squareVerticesArray[7] = 1;
squareVerticesArray[8] = 0;
squareVerticesArray[9] = 1;
squareVerticesArray[10] = 0;
squareVerticesArray[11] = 0;
```

GÉOMÉTRIE PERSONNALISÉE (8/9)

On fait la même chose qu'avant

```
const squareVerticesAttribute = new THREE.BufferAttribute(  
    squareVerticesArray,  
    3  
);  
const squareGeometry = new THREE.BufferGeometry();  
/*  
 * position est quelque chose de connue dans THREE  
 * important d'utiliser ce nom là pour que cela fonctionne  
 * avec les shaders de THREE  
 */  
squareGeometry.setAttribute('position', squareVerticesAttribute);  
const squareMaterial = new THREE.MeshBasicMaterial({  
    color: 0x00ff00,  
    wireframe: true  
});  
const squareMesh = new THREE.Mesh(squareGeometry, squareMaterial);  
scene.add(squareMesh);
```

GÉOMÉTRIE PERSONNALISÉE (9/9)

Maintenant il faut rajouter la notion d'index pour créer les faces

```
const squareIndices = [  
    0, 1, 2,  
    0, 2, 3,  
];  
squareGeometry.setIndex(squareIndices);
```

LES CAMÉRAS

ARRAYCAMERA

L'ArrayCamera permet de rendre une scène à travers plusieurs caméras dans des zones spécifiques de rendu



STEREOCAMERA

La StereoCamera permet de rendre une scène à travers 2 caméras qui imite le comportement des yeux pour créer un effet de parallax



Note: Notamment utile quand on veut utiliser des casques VR

CUBECAMERA

La CubeCamera permet de rendre une scène dans 6 directions créer un effet de parallax

Note: Notamment utile pour rendre des texture d'environnement, pour de la réflexion ou des ombres

PERSPECTIVECAMERA (1/7)

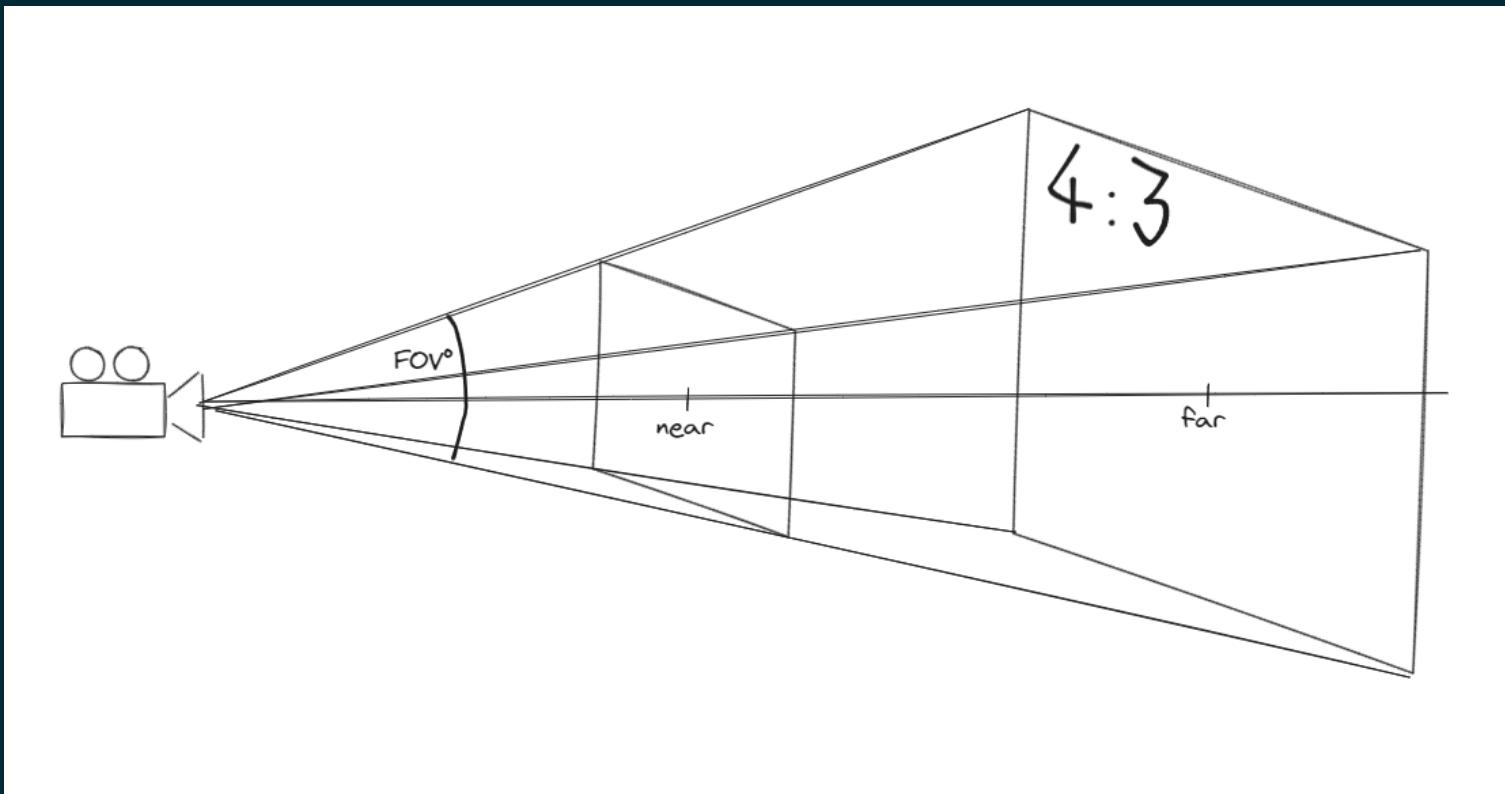
Comme son nom l'indique la PerspectiveCamera permet de rendre une scène avec de la perspective et donc garder la notion de profondeur

PERSPECTIVECAMERA (2/7)

On a vu pour créer une PerspectiveCamera qu'il fallait faire

```
const camera = new THREE.PerspectiveCamera(  
  75,  
  viewportSize.width / viewportSize.height,  
  1,  
  1000  
);
```

PERSPECTIVE CAMERA (3/7)



PERSPECTIVECAMERA (4/7)

Champs de vision

- Angle de vision vertical
- En degrés
- Aussi appelé FOV

PERSPECTIVECAMERA (5/7)

Aspect ratio

Largeur de la zone de rendu divisée par la hauteur.

*Note: Même principe que les pour les TVs avec les 4:3 ou
16:9*

PERSPECTIVECAMERA (6/7)

Plans avant et arrière

Des valeurs qui vont dire à quelle distance de la caméra les objets ont le droit d'être rendu.

C'est à dire que tout objet plus proche que le `near` ou plus loin que le `far` ne sera pas rendu et donc non visible

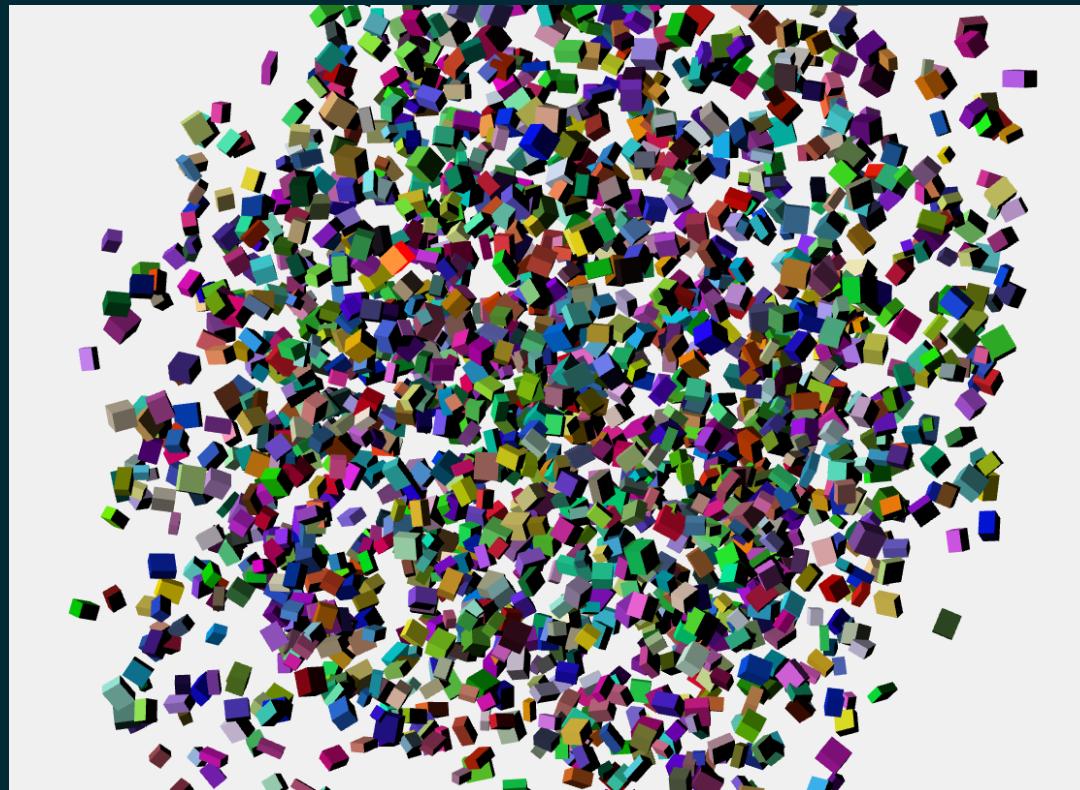
PERSPECTIVECAMERA (7/7)

Plans avant et arrière

Ne pas mettre une valeur de near trop petite et une valeur far trop élevée sinon cela risque d'engendrer ce qu'on appelle du z-fighting. C'est lorsque les faces de vos objets clignotent car le GPU n'arrive pas à réellement décider si la face est visible ou non.

ORTHOGRAPHICCAMERA (1/6)

L'OrthographicCamera permet de rendre une scène sans perspective. On perd toute notion de profondeur

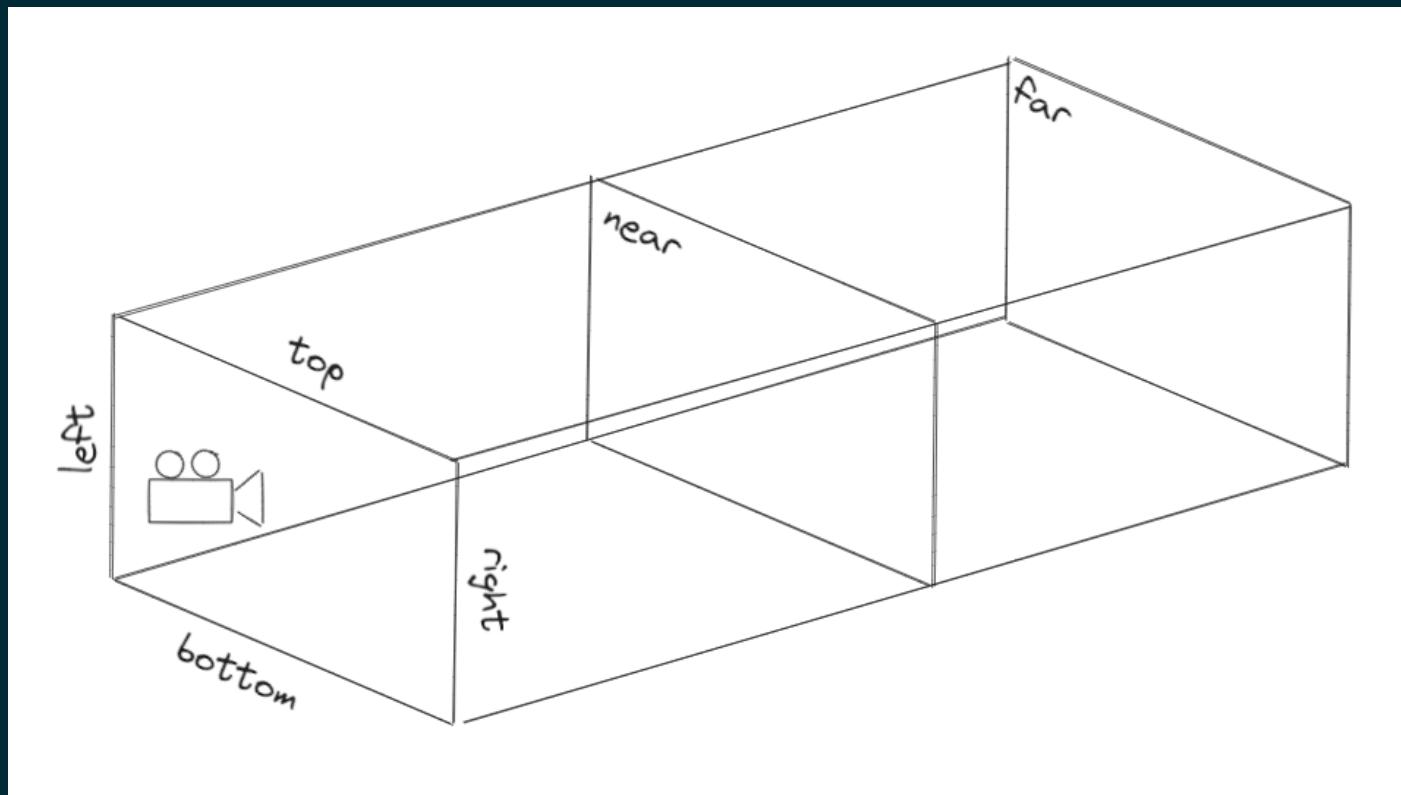


ORTHOGRAPHICCAMERA (2/6)

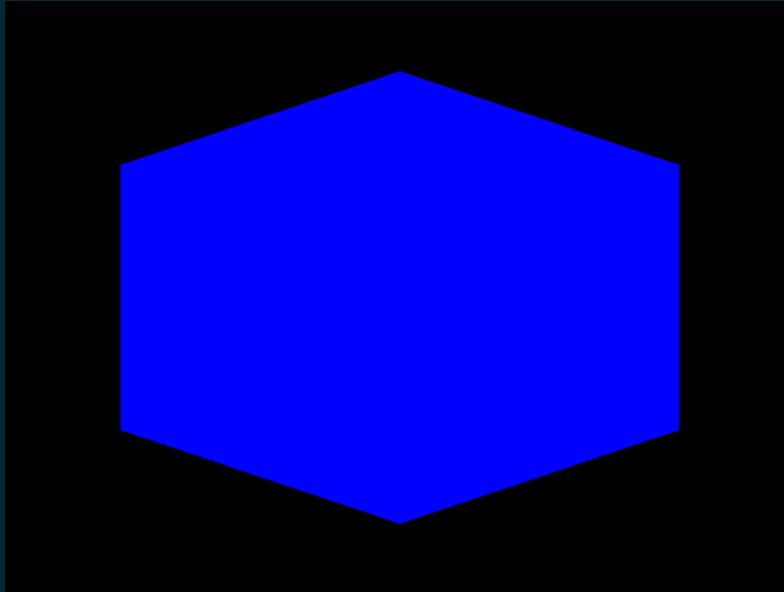
Pour créer une OrthographicCamera, les paramètres vont changer

```
const camera = new THREE.OrthographicCamera(  
    -1, // left  
    1, // right  
    1, // top  
    -1, // bottom  
    1,  
    1000  
)
```

ORTHOGRAPHIC CAMERA (3/6)



ORTHOGRAPHIC CAMERA (4/6)



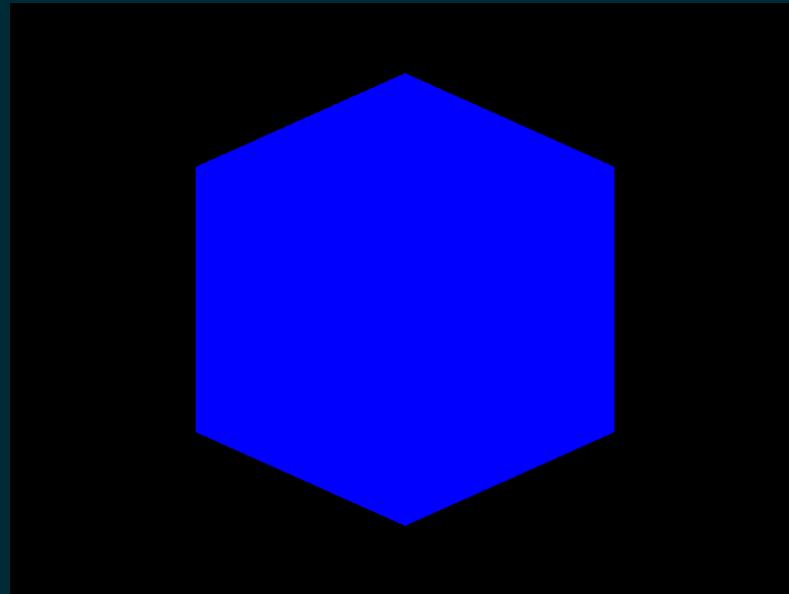
Le problème avec ces paramètres, c'est que notre cube semble s'être fait tassé et ne ressemble plus à un cube.

ORTHOGRAPHICCAMERA (5/6)

Pour corriger, cela on doit utiliser l'aspect ratio de la zone de rendu

```
const aspectRatio = viewportSize.width / viewportSize.height;
const camera = new THREE.OrthographicCamera(
    -1 * aspectRatio, // left
    1 * aspectRatio, // right
    1, // top
    -1, // bottom
    1,
    1000
);
```

ORTHOGRAPHIC CAMERA (6/6)



On récupère notre cube (mais toujours sans notion de profondeur)

ANIMATIONS

INTRODUCTION

Jusqu'à maintenant, on a rendu notre scène qu'une seule fois.

On va maintenant animer nos objets et pour voir la progression de ces animations, on va devoir faire de nouveau rendu de la scène à chaque mise à jour de la scène.

LE FRAMERATE

Nos écrans tournent à une fréquence de rafraîchissement qu'on appelle le framerate. La plupart des écrans aujourd'hui tournent en 60 images par seconde mais pas toujours.

Important de noter que les animations doivent paraître pareil peu importe le framerate.

REQUESTANIMATIONFRAME

On va avoir besoin d'une fonction qui s'occupe d'appeler ce que l'on veut à chaque frame. On va donc avoir besoin d'une fonction qui s'occupe de mettre à jour nos objets puis fait un rendu à chaque frame.

Cette fonction est `requestAnimationFrame`

TICK (1/5)

On va donc avoir besoin d'une fonction qui s'occupe de mettre à jour nos objets puis fait un rendu à chaque frame.

```
const tick = () => {
    console.log('tick');

    requestAnimationFrame(tick);
};

tick();
```

TICK (2/5)

On va déplacer la fonction de rendu dans le ticker et on peut donc rajouter tout ce que l'on souhaite avant de demander le rendu

```
const tick = () => {
    boxMesh.rotation.y += 0.01;

    renderer.render(scene, camera);

    requestAnimationFrame(tick);
};

tick();
```

TICK (3/5)

D'un ordinateur à l'autre le résultat peut ne pas être le même, car avec un taux de rafraîchissement plus élevé, le fonction tick va être appelé plus de fois par seconde qu'un autre sur un autre ordinateur.

TICK (4/5)

Pour corriger, on utilise toujours la notion de temps écoulé entre 2 frames pour moduler les valeurs.

```
const clock = new THREE.Clock();
const tick = () => {
    // Time between 2 ticks
    const delta = clock.getDelta();
    // Time from the beginning of the application
    const elapsed = clock.getElapsedTime();

    boxMesh.rotation.y += delta;
    boxMesh.position.x = Math.cos(elapsed * Math.PI);

    renderer.render(scene, camera);

    requestAnimationFrame(tick);
};

tick();
```

TICK (5/5)

Petit point d'attention sur la clock de THREE, la fonction `getDelta` fonctionne de manière particulière si vous utilisez aussi la fonction `getElapsedTime`.

ANIMATIONS PLUS AVANCÉES

Vous voudrez sans doute avoir des animations plus poussées.

Il existe des librairies qui permettent de gérer ça pour nous:

- GSAP
- animejs
- Theatre.js

PLEIN ÉCRAN & REDIMENSIONNEMENT

PLEIN-ÉCRAN (1/2)

On va commencer par dire à notre canvas de prendre la taille de la fenêtre:

```
const viewportSize = {  
    width: window.innerWidth,  
    height: window.innerHeight  
};
```

PLEIN-ÉCRAN (2/2)

À ce moment-là, on a toujours les marges de base d'une page web, on va changer le style css pour avoir notre canvas dans toute la page.

```
* {
  margin: 0;
  padding: 0;
}

html,
body {
  overflow: hidden;
}

#webgl {
  position: absolute;
  top: 0;
  left: 0;
  outline: none;
}
```

REDIMENSIONNEMENT (1/3)

On a notre canvas en plein écran mais malheureusement, si on redimensionne la fenêtre, le canvas ne suivra pas.

REDIMENSIONNEMENT (2/3)

Pour écouter le resize de la fenêtre, on peut s'abonner
à l'événement que propose la fenêtre

```
window.addEventListener('resize', () => {  
  console.log('resize');  
});
```

REDIMENSIONNEMENT (3/3)

Une fois abonné, on peut mettre à jour le renderer et la caméra:

```
window.addEventListener('resize', () => {
    viewportSize.width = window.innerWidth;
    viewportSize.height = window.innerHeight;

    camera.aspect = viewportSize.width / viewport.height;
    camera.updateProjectionMatrix();

    renderer.setSize(viewportSize.width, viewportSize.height);
});
```

RATIO DE RÉSOLUTION PIXEL (1/5)

Vous observerez peut être des rendus soit un peu flous ou soit avec des effets d'escalier qu'on appelle aliasing. Si tel est le cas c'est sans doute dû au fait que vous êtes sur un écran qui a un ratio de résolution pixel supérieur à 1.

RATIO DE RÉSOLUTION PIXEL (2/5)

Cela correspond au nombre de pixels physiques sur l'écran pour une unité de pixel en CSS.

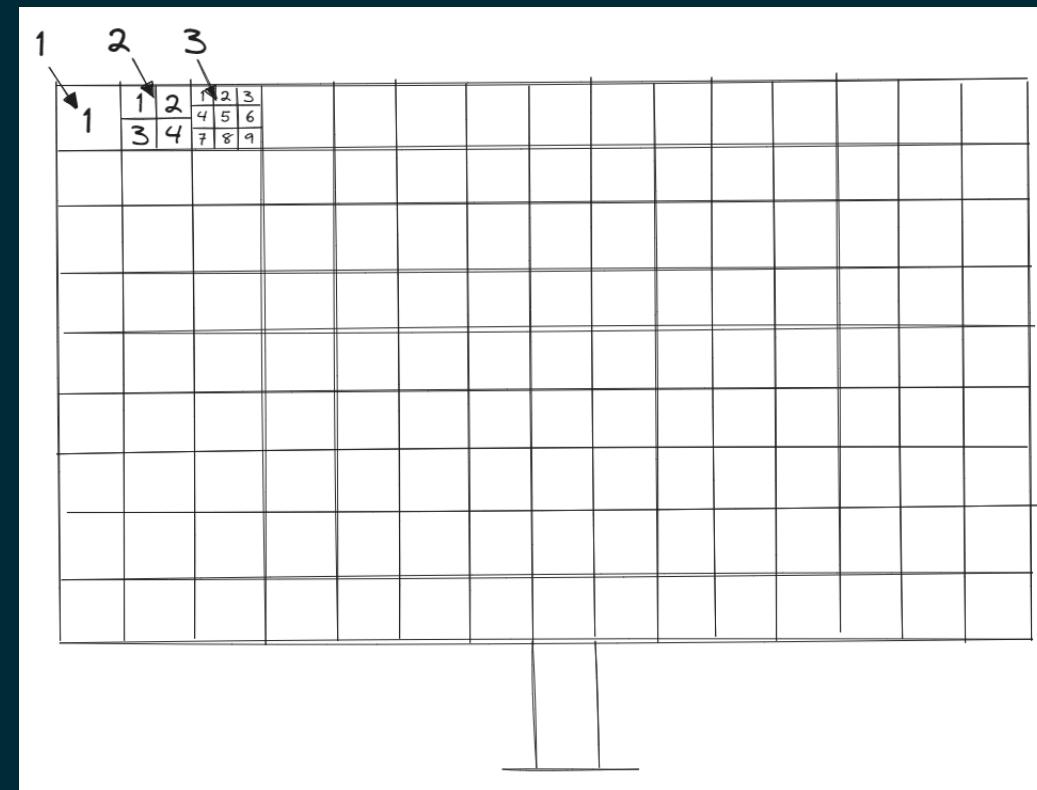
RATIO DE RÉSOLUTION PIXEL (3/5)

Pendant des années, tous les écrans avaient un ratio de résolution pixel à 1 et si on s'approchait de l'écran, on pouvait commencer à percevoir ces pixels.

Apple y a vu une grande amélioration et a commencé à construire des écrans avec un ratio de résolution pixel à 2.

Maintenant, de nombreux constructeurs les ont rejoints avec même des ratios à 3 ou plus.

RATIO DE RÉSOLUTION PIXEL (4/5)



RATIO DE RÉSOLUTION PIXEL (5/5)

Pour obtenir le ratio de résolution pixel courant, on peut faire `window.devicePixelRatio`

Et donc on peut mettre à jour notre renderer avec:

```
renderer.setPixelRatio(window.devicePixelRatio);
```

Mettez une limite à cette valeur, sinon ça fera des rendus excessifs et les performances seront dégradées

LES CONTRÔLES

MANIPULER LA CAMÉRA (1/4)

Pour l'instant notre point de vue est très statique et on va apprendre à le contrôler avec la souris

MANIPULER LA CAMÉRA (2/4)

D'abord on a besoin d'avoir les coordonnées de la souris dans le repère de notre canvas

```
const cursor = {
  x: 0,
  y: 0,
};
window.addEventListener('mousemove', (evt) => {
  cursor.x = (evt.clientX / viewportSize.width) * 2 - 1;
  cursor.y = 1 - (evt.clientY / viewportSize.height) * 2;
});
```

MANIPULER LA CAMÉRA (3/4)

Ensuite on peut exploiter ces valeurs pour bouger la caméra

```
const tick = () => {
    camera.position.x = cursor.x;
    camera.position.y = cursor.y;
    camera.lookAt(boxMesh.position);

    renderer.render(scene, camera);

    requestAnimationFrame(tick);
};
```

MANIPULER LA CAMÉRA (4/4)

Pour faire tourner la caméra autour du cube:

```
const tick = () => {
    camera.position.x = Math.sin(cursor.x * Math.PI) * 4;
    camera.position.z = Math.cos(cursor.x * Math.PI) * 4;
    camera.position.y = cursor.y * 3;
    camera.lookAt(boxMesh.position);

    renderer.render(scene, camera);

    requestAnimationFrame(tick);
};
```

LES CONTRÔLES EXISTANTS (1/11)

ThreeJS has de nombreux contrôles qui permettent la même chose et même beaucoup plus

LES CONTRÔLES EXISTANTS (2/11)

DRAGCONTROLS

DragControls permet d'effectuer des opérations de drag'n'drop notamment sur des objets.

LES CONTRÔLES EXISTANTS (3/11)

FLYCONTROLS

FlyControls permet d'effectuer une navigation similaire à un vol. On peut tourner sur les 3 axes et avancer ou reculer.

LES CONTRÔLES EXISTANTS (4/11)

FIRSTPERSONCONTROLS

FirstPersonControls est similaire à FlyControls mais avec un axe de fixe. Contrairement à son nom ça ne fonctionne pas pareil que dans les jeux FPS.

LES CONTRÔLES EXISTANTS (5/11)

MAPCONTROLS

MapControls a pour but pour bouger la caméra à travers une carte depuis l'oeil d'un oiseau.

LES CONTRÔLES EXISTANTS (6/11)

POINTERLOCKCONTROLS

PointerLockControls est basé sur l'API de PointerLock et est le choix parfait pour les jeux FPS.

LES CONTRÔLES EXISTANTS (7/11)

TRANSFORMCONTROLS

TransformControls permet d'effectuer des opérations sur le repère d'un objet comme des logiciels de modeling.

LES CONTRÔLES EXISTANTS (8/11)

ORBITCONTROLS

OrbitControls permet de tourner autour d'une cible.

LES CONTRÔLES EXISTANTS (9/11)

ORBITCONTROLS - INSTANCIATION

```
import { OrbitControls } from 'three/examples/jsm/controls/OrbitControls';  
  
// ...  
  
const controls = new OrbitControls(camera, canvas);
```

LES CONTRÔLES EXISTANTS (10/11)

ORBITCONTROLS - CIBLE

Par défaut l'OrbitControls regarde au centre de la scène. On peut changer ce autour de quoi on souhaite tourner via la propriété target

```
const controls = new OrbitControls(camera, canvas);
controls.target = boxMesh.position;
controls.update(); // To force internal updates
```

LES CONTRÔLES EXISTANTS (11/11)

ORBITCONTROLS - AMORTISSEMENT

Par défaut, le contrôle est très brut dans les mouvements. Pour lisser les mouvements, on peut utiliser la propriété enableDamping. Comme cela fonctionne comme une animation, on devra appeler la méthode update à chaque tick.

```
const controls = new OrbitControls(camera, canvas);
controls.enableDamping = true;
```

LES TEXTURES

INTRODUCTION

Une texture est une image qui va recouvrir la surface des objets.

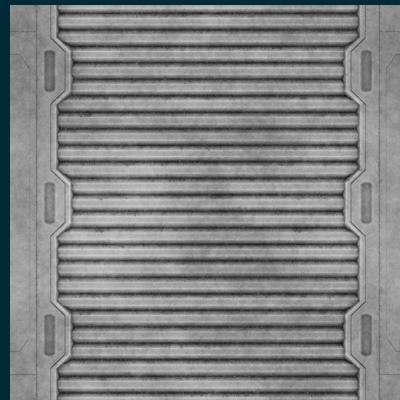
Il en existe plusieurs types et dont chacune d'elles peut servir pour différents effets.

LES TYPES DE TEXTURES (1/9)

Dans le cours, on va utiliser les textures que l'on peut trouver [ici](#). Elles sont à disposition librement et ont été faites par João Paulo.

LES TYPES DE TEXTURES (2/9)

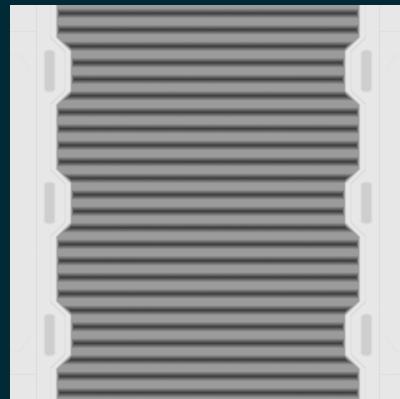
COULEUR OU (ALBEDO)



La texture la plus simple. Elle est simplement appliquée sur la géométrie telle quelle.

LES TYPES DE TEXTURES (3/9)

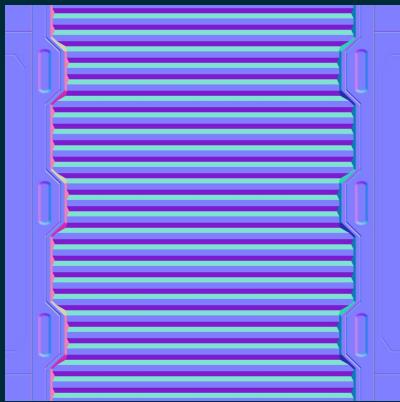
HAUTEUR



La texture est une image en niveau de gris. Elle permet de bouger les vertices pour créer du relief. Bien penser qu'il faut que la géométrie doit être assez subdivisée pour voir ces détails.

LES TYPES DE TEXTURES (4/9)

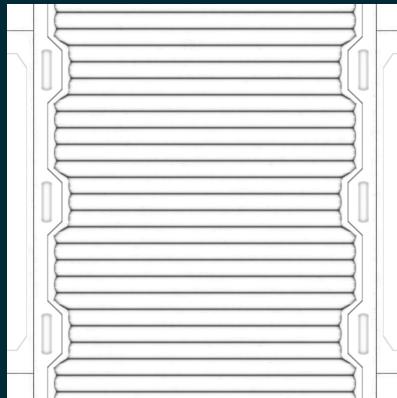
NORMAL



Cela permet d'ajouter des légers détails et notamment de leurer la lumière pour faire croire qu'à un endroit précis finalement la face est orientée différemment. Cela permet de créer des détails sans avoir besoin de subdiviser énormément la géométrie.

LES TYPES DE TEXTURES (5/9)

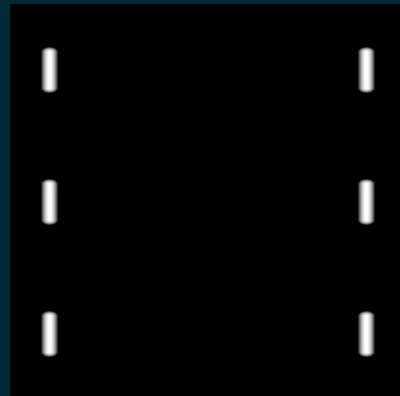
OCCLUSION AMBIENTE



La texture est une image en niveau de gris. Elle permet de tricher sur les ombres dans les recoins de la géométrie.

LES TYPES DE TEXTURES (6/9)

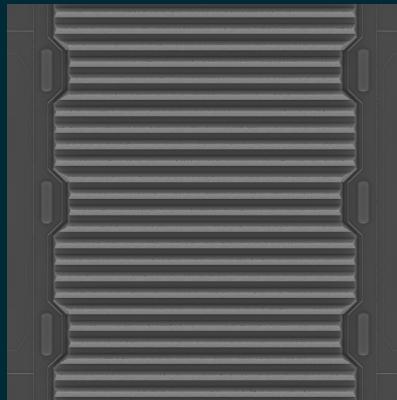
EMISSION



La texture est utilisée pour représenter les zones de l'objet qui émettent de la lumière. Cela décrit la luminosité de l'objet en lui-même.

LES TYPES DE TEXTURES (7/9)

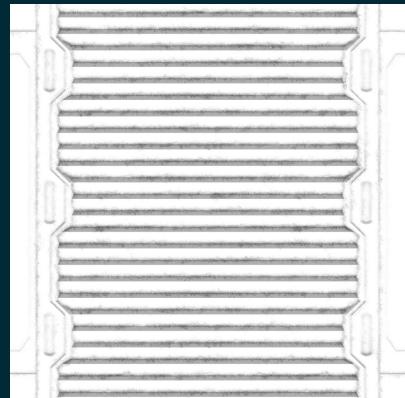
RUGOSITÉ



La texture de rugosité permet de décrire la manière dont la lumière est dispersée ou réfléchie par la surface.

LES TYPES DE TEXTURES (8/9)

MÉTALLIQUE



Permet de donner des propriétés métalliques à la surface qui va une fois de plus influencer sur la manière dont est réfléchie la lumière.

LES TYPES DE TEXTURES (9/9)

ASSEMBLAGE



Une fois tout assemblé, on peut obtenir des résultats assez réalistes grâce au principe de PBR (physical based rendering).

UTILISER DES TEXTURES DANS THREEJS (1/2)

Il faut loader les textures pour pouvoir les utiliser correctement dans ThreeJS

UTILISER DES TEXTURES DANS THREEJS (2/2)

On va utiliser le [TextureLoader](#)

```
const textureLoader = new THREE.TextureLoader();
const albedoTexture = textureLoader.load(
  '/textures/sci-fi_metal_panel/Sci_fi_Metal_Panel_004_basecolor.jpg'
);
albedoTexture.colorSpace = THREE.SRGBColorSpace;

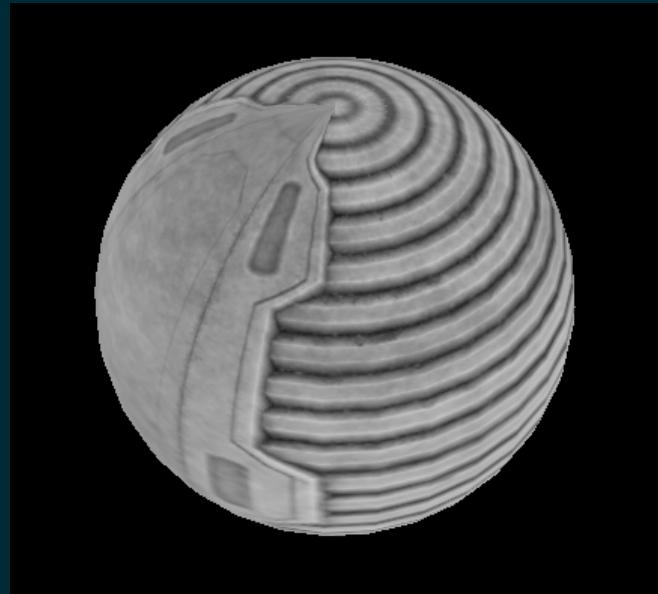
// ...

const boxMaterial = new THREE.MeshBasicMaterial({ map: albedoTexture });
```

UVS (1/6)

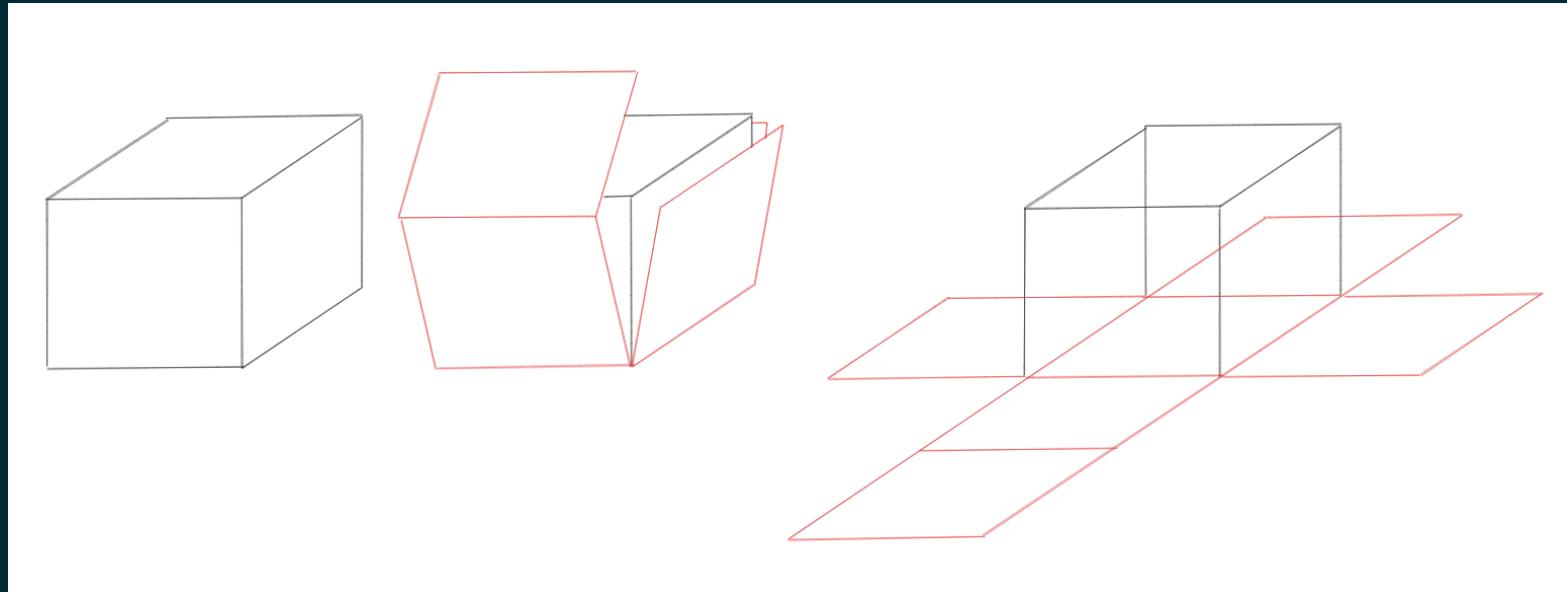
Le résultat qu'on obtient sur un cube semble assez logique, mais que se passe-t-il si on remplace la BoxGeometry par d'autres géométries telles que la SphereGeometry, le ConeGeometry, etc.

UVS (2/6)



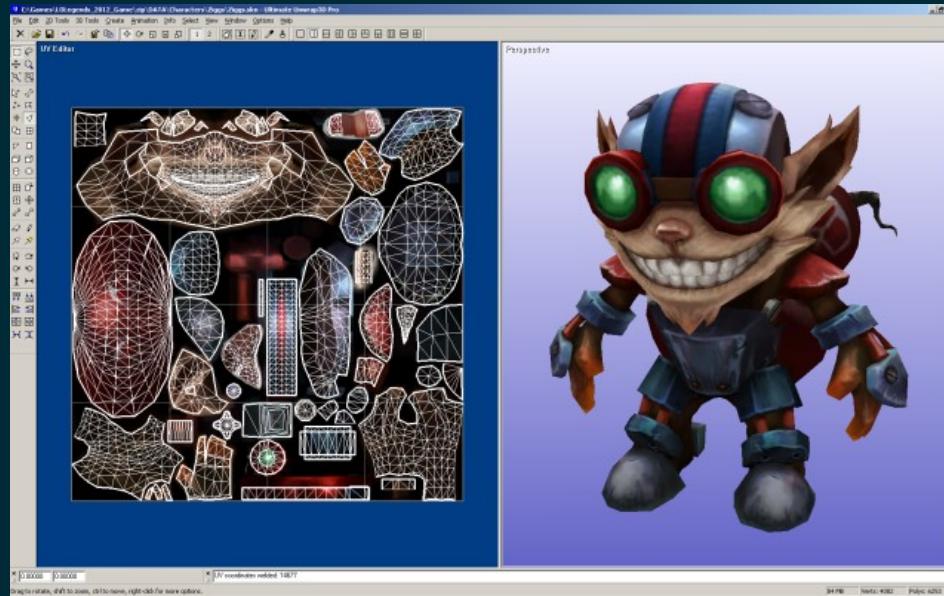
Dans ces cas-ci les textures semblent déformés et/ou étirés par endroit de manière à couvrir la géométrie.

UVS (3/6)



C'est ce qu'on appelle le dépliage des UVs. Il faut imaginer son objet et qu'on déballe et à la fin on obtient une feuille de papier carée sur laquelle il y a la texture.

UVS (4/6)



Chaque vertex de la géométrie a une coordonnée 2D pour aller indiquer quelle partie de la texture est à appliquer.

UVS (5/6)

Pour afficher le buffer uvs du cube par exemple, on peut y accéder comme ça:

```
console.log(boxGeometry.attributes.uv);
```

UVS (6/6)

Pour les primitives de ThreeJS, les uvs sont calculées en interne. Il en est de même la plupart du temps pour les modèles 3D que vous trouverez sur internet.

Si un jour, vous modelez vous même votre objet 3D,
vous devrez vous en charger.

TRANSFORMATION SUR LES TEXTURES (1/9)

RÉPÉTITION

On peut faire répéter une texture sur une surface via la propriété `repeat` que l'on a sur une texture. Cela permet de dire combien de fois on veut répéter la texture sur chaque axe

```
albedoTexture.repeat.x = 2;  
albedoTexture.repeat.y = 4;
```

TRANSFORMATION SUR LES TEXTURES (2/9)

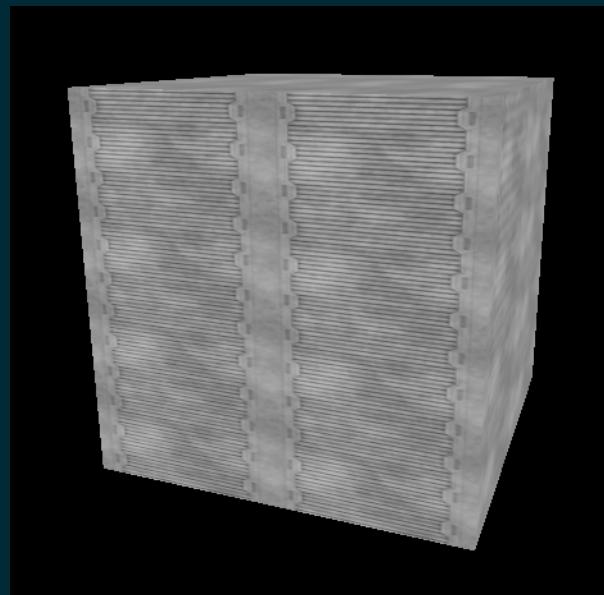
RÉPÉTITION

Par défaut une texture la manière dont est fait une texture ne permet pas le repeat même si la propriété repeat est utilisée. Il faut changer le wrapping:

```
albedoTexture.wrapS = THREE.RepeatWrapping;  
albedoTexture.wrapT = THREE.RepeatWrapping;
```

TRANSFORMATION SUR LES TEXTURES (3/9)

RÉPÉTITION



On obtient ce résultat appliqué sur notre cube.

TRANSFORMATION SUR LES TEXTURES (4/9)

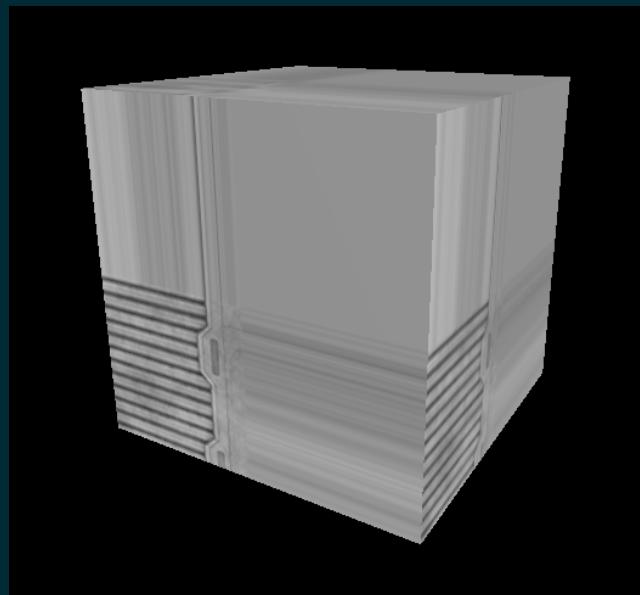
DÉCALAGE

On peut décaler la texture via la propriété offset. Cela revient à décaler les UVs finalement.

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

TRANSFORMATION SUR LES TEXTURES (5/9)

DÉCALAGE



On obtient ce résultat appliqué sur notre cube.

TRANSFORMATION SUR LES TEXTURES (6/9)

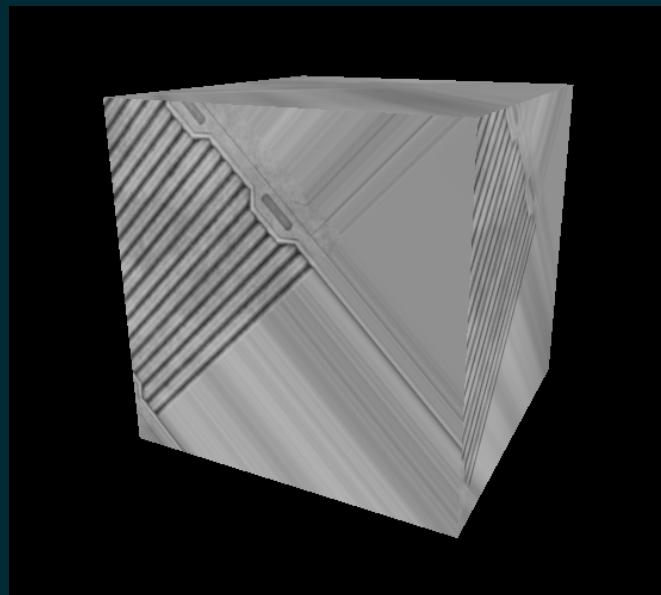
ROTATION

On peut tourner la texture via la propriété rotation d'un angle en radians.

```
albedoTexture.rotation = Math.PI * 0.25;
```

TRANSFORMATION SUR LES TEXTURES (7/9)

ROTATION



On obtient ce résultat appliqué sur notre cube.

TRANSFORMATION SUR LES TEXTURES (8/9)

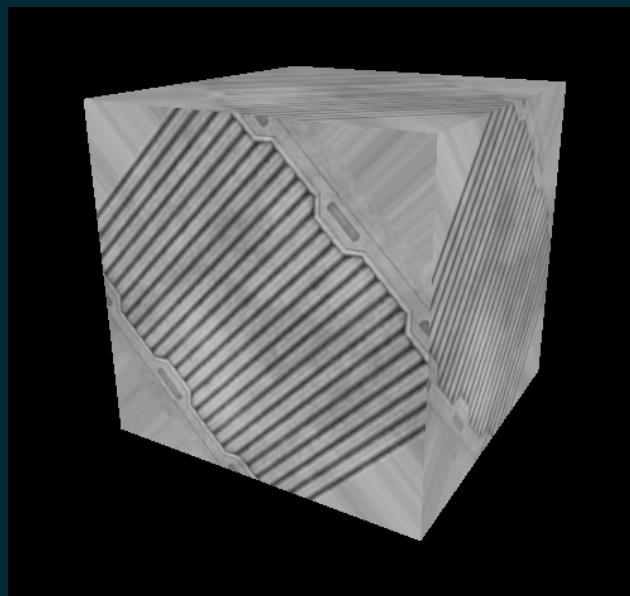
ROTATION

Et si on souhaite changer le centre de rotation de la texture:

```
albedoTexture.center.x = 0.5;  
albedoTexture.center.y = 0.5;
```

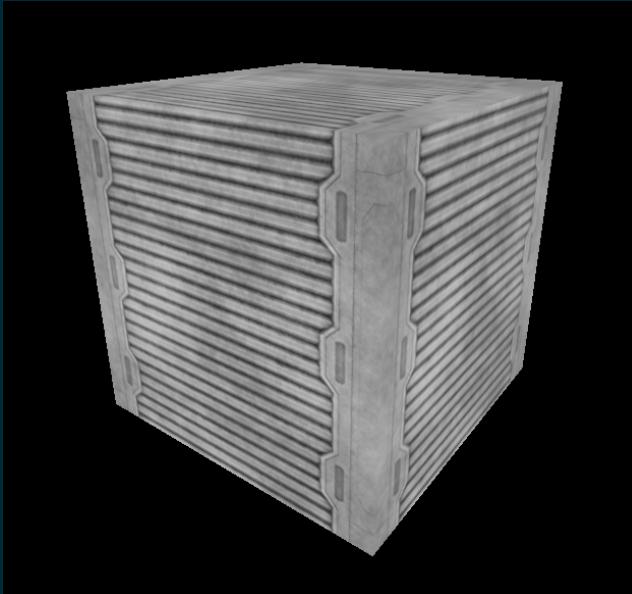
TRANSFORMATION SUR LES TEXTURES (9/9)

ROTATION



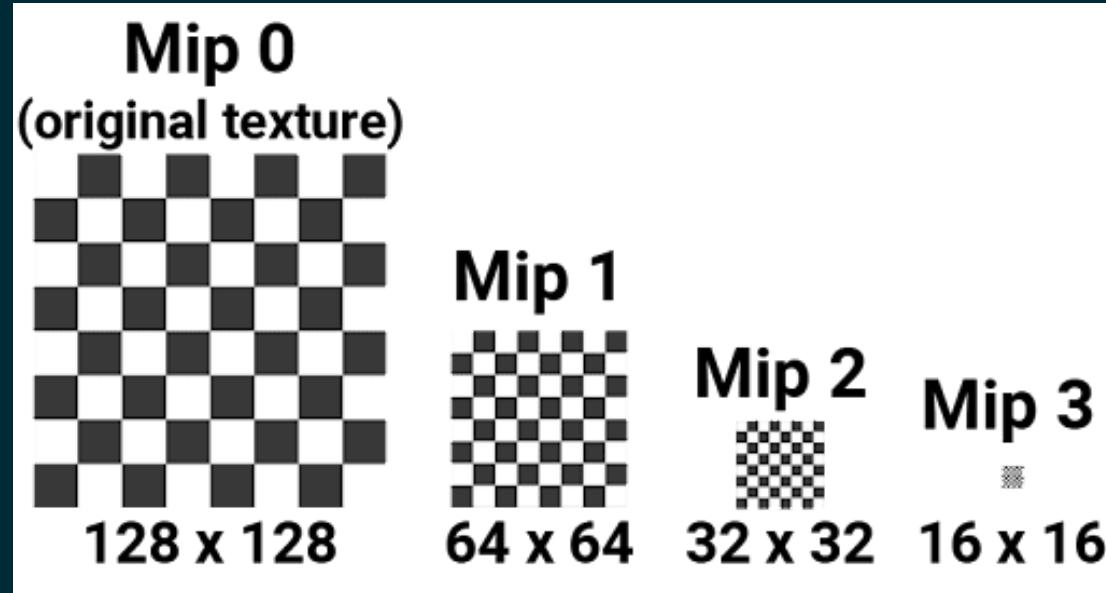
On obtient ce résultat appliqué sur notre cube.

FILTRES ET MIPMAP (1/11)



Si on regarde de plus près la face du dessus, on observe qu'elle est un peu floue et pas aussi nette que les autres faces. Cela est dû au type de filtre ainsi qu'au mipmapping.

FILTRES ET MIPMAP (2/11)



Le mipmappping est une technique qui consiste à créer des textures de moitié de résolution jusqu'à la version la plus petite de 1x1. Toutes ses textures sont envoyées sur le GPU et il choisira la plus adapté en fonction du rendu. ThreeJS s'en charge pour nous par défaut.

FILTRES ET MIPMAP (3/11)

FILTRE DE MINIFICATION

Ce filtre est utilisé quand les pixels de la texture sont plus petits que les pixels du rendus.

Il existe 6 valeurs possibles:

- THREE.NearestFilter
- THREE.LinearFilter
- THREE.NearestMipmapNearestFilter
- THREE.NearestMipmapLinearFilter
- THREE.LinearMipmapNearestFilter
- THREE.LinearMipmapLinearFilter (valeur par défaut)

FILTRES ET MIPMAP (4/11)

FILTRE DE MINIFICATION - NEARESTFILTER

C'est le filtre le plus simple. Il renvoie la valeur de l'élément de texture le plus proche (en distance manhattan) des coordonées de texture spécifiées.

FILTRES ET MIPMAP (5/11)

FILTRE DE MINIFICATION - LINEARFILTER

Il renvoie la moyenne pondérée des quatres éléments de texture les plus proches des coordonnées de textures spécifiées.

FILTRES ET MIPMAP (6/11)

FILTRE DE MINIFICATION - NEARESTMIPMAPNEARESTFILTER

Il choisit le niveau de mipmap qui correspond le mieux à la taille du pixel à texturer et applique NearestFilter pour produire la valeur de texture finale.

FILTRES ET MIPMAP (7/11)

FILTRE DE MINIFICATION - NEARESTMIPMAPLINEARFILTER

Il choisit les deux niveau de mipmap qui correspondent le mieux à la taille du pixel à texturer et applique NearestFilter pour produire la valeur de texture à partir de chaque niveau de mipmap. La valeur de texture finale est une moyenne pondérée de ces deux valeurs.

FILTRES ET MIPMAP (8/11)

FILTRE DE MINIFICATION - LINEARMIPMAPNEARESTFILTER

Il choisit le niveau de mipmap qui correspond le mieux à la taille du pixel à texturer et applique LinearFilter pour produire la valeur de texture finale.

FILTRES ET MIPMAP (9/11)

FILTRE DE MINIFICATION - LINEARMIPMAPLINEARFILTER

Il choisit les deux niveau de mipmap qui correspondent le mieux à la taille du pixel à texturer et applique LinearFilter pour produire la valeur de texture à partir de chaque niveau de mipmap. La valeur de texture finale est une moyenne pondérée de ces deux valeurs.

FILTRES ET MIPMAP (10/11)

FILTRE DE GROSSISSEMENT

Ce filtre fonctionne comme le précédent mais quand les pixels de la texture sont plus gros que les pixels du rendus.

Il existe 2 valeurs possibles (avec les mêmes principes):

- THREE.NearestFilter
- THREE.LinearFilter (valeur par défaut)

FILTRES ET MIPMAP (11/11)

Le mipmapping n'est utile que pour le minFilter. Par conséquent, si vous utilisez THREE.NearestFilter, vous pouvez les désactiver:

```
albedoTexture.generateMipmaps = false;  
albedoTexture.minFilter = THREE.NearestFilter;  
albedoTexture.magFilter = THREE.NearestFilter;
```

POIDS ET OPTIMISATION (1/2)

POIDS

On est sur le web et donc les utilisateurs vont devoir télécharger toutes les textures que vous utilisez. On peut utiliser plusieurs types d'images tels que .jpg ou .png. Essayez de réduire au maximum le poids en gardant une qualité qui convient.

POIDS ET OPTIMISATION (2/2)

TAILLE

Par principe, il est recommandé de toujours utiliser des textures qui sont en puissance de 2 étant que les niveaux de mipmap divise par 2 jusqu'à atteindre une texture de 1x1.

Exemple:

- 2048x2048
- 1024x1024
- 512x512

LES MATÉRIAUX

ThreeJS a pleins de matériaux différents et on va les découvrir un à un pour voir leurs spécificités.

MESH BASIC MATERIAL (1/6)

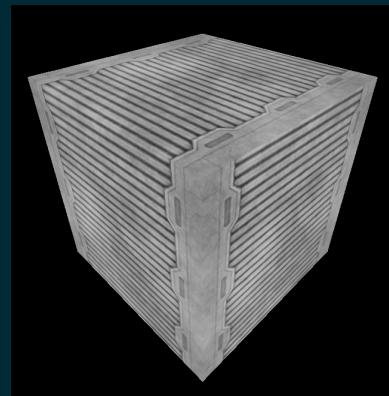
C'est le matériau le plus simple qui existe. Il existe pleins de propriétés que l'on n'a pas encore eu l'occasion d'exploiter

MESHBASICMATERIAL (2/6)

MAP

Cela permet d'appliquer une texture à la surface de la géométrie.

```
material.map = albedoTexture;
```

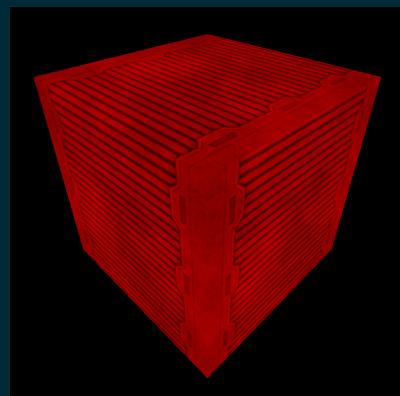


MESHBASICMATERIAL (3/6)

COLOR

Cela permet d'appliquer une couleur uniforme à la surface de la géométrie. Si une texture est appliquée, cela teintera la texture.

```
material.color = new THREE.Color(0xff0000);
```

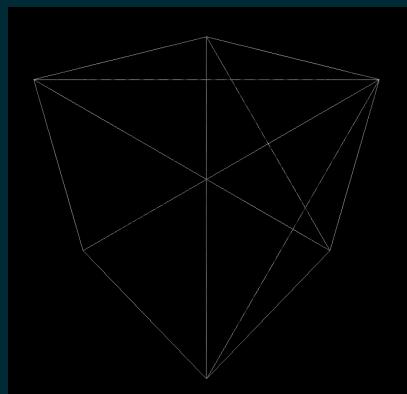


MESHBASICMATERIAL (4/6)

WIREFRAME

Cela permet de voir les triangles de la géométrie avec des lignes de 1px (peu importe la distance avec la caméra)

```
material.wireframe = true;
```

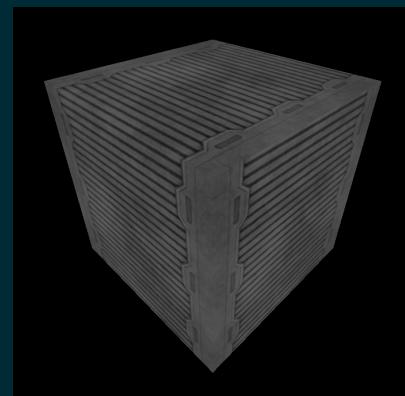


MESHBASICMATERIAL (5/6)

TRANSPARENCE

On peut dire que notre objet est transparent ainsi que sa valeur

```
material.transparent = true;  
material.opacity = 0.5;
```

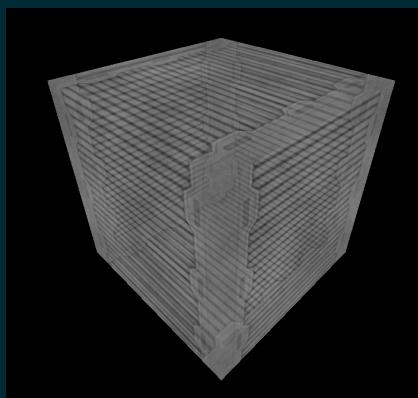


MESHBASICMATERIAL (6/6)

CÔTÉ

On peut choisir quel côté des faces on souhaite rendre visible. Par défaut ce sont les faces avant THREE.FrontSide, mais on peut choisir celles de derrières THREE.BackSide ou même les deux THREE.DoubleSide.

```
material.side = THREE.DoubleSide
```



MESHNORMALMATERIAL (1/3)

Ce matériau fait correspondre les normales aux couleurs RVB.

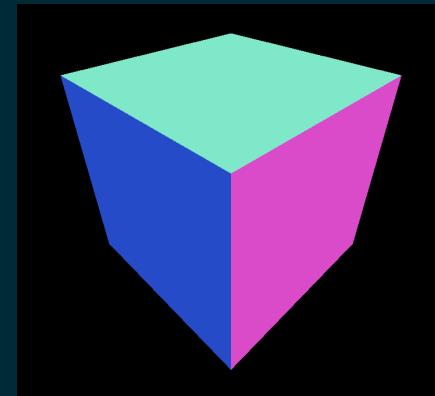
Et si on observe bien les couleurs sont similaires à ce que l'on a vu précédemment dans la texture de normale.



MESHNORMALMATERIAL (2/3)

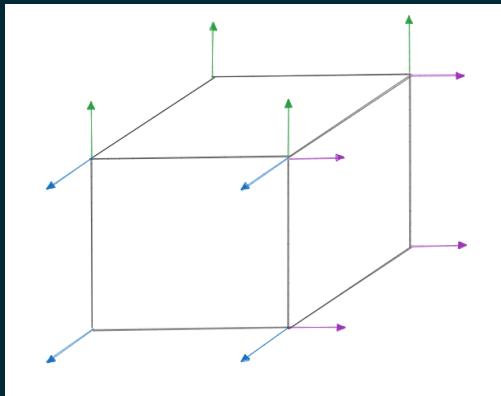
On obtient ce résultat:

```
material = new THREE.MeshNormalMaterial();
```



MESHNORMALMATERIAL (3/3)

Les normales sont des informations pour chaque vertex qui contienne la direction perpendiculaire vers l'extérieur de la face.



Les normales sont utilisées notamment pour calculer comment éclairer les faces ou comment l'environnement devrait se refléter ou se réfracter.

MESHMATCAPMATERIAL (1/2)

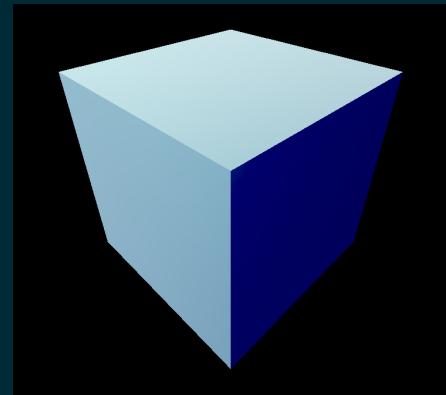
Ce matériau est défini par ce qu'on appelle une texture MapCap (ou Lit Sphere) qui encode la couleur et l'ombrage du matériau. Néanmoins ce matériau ne réagit plus à la lumière vu que le comportement est embarqué dans la texture.



MESHMATCAPMATERIAL (2/2)

On obtient ce résultat:

```
material = new THREE.MeshMatcapMaterial();
material.matcap = matcap;
```



On peut trouver pleins de matcap [ici](#)

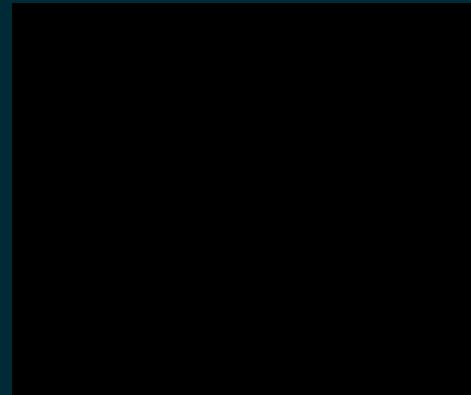
MESHLAMBERTMATERIAL (1/5)

Ce matériau utilise le modèle Lambertien non physique pour calculer la réflectance. Il peut bien simuler certaines surfaces (comme le bois non traité ou la pierre), mais ne peut pas simuler les surfaces brillantes avec des reflets spéculaires (comme le bois verni).

MESHLAMBERTMATERIAL (2/5)

On obtient ce résultat:

```
material = new THREE.MeshLambertMaterial();
```



MESHLAMBERTMATERIAL (3/5)

Mais où a bien pu passer l'objet ?



MESHLAMBERTMATERIAL (4/5)

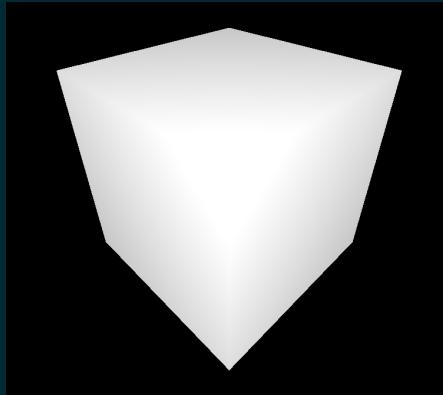
Ce matériau et tous les suivants nécessitent des lumières pour fonctionner.

```
const ambientLight = new THREE.AmbientLight(0xffffff, 1);
scene.add(ambientLight);

const pointLight = new THREE.PointLight(0xffffff, 30);
pointLight.position.x = 2;
pointLight.position.y = 2;
pointLight.position.z = 2;
scene.add(pointLight);
```

MESHLAMBERTMATERIAL (5/5)

Une fois les lumières ajoutée:



On revoit notre objet et on commence à percevoir un peu de réalisme. Ce n'est pas le matériau le plus réaliste mais cela reste le plus performant.

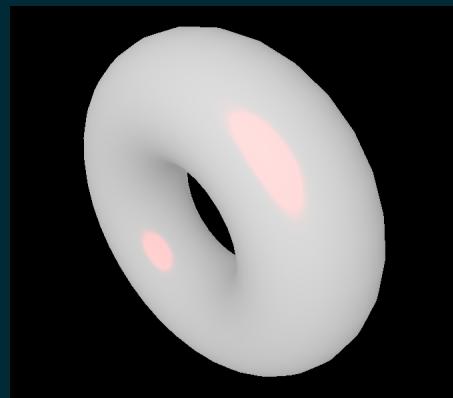
MESHPHONGMATERIAL (1/2)

Ce matériau utilise le modèle Blinn-Phong non physique pour calculer la réflectance. Contrairement au MeshLambertMaterial, il permet de simuler des surfaces brillantes avec des reflets spéculaires (comme le bois verni).

MESHPHONGMATERIAL (2/2)

On obtient ce résultat:

```
material = new THREE.MeshPhongMaterial();
material.shininess = 100;
material.specular = new THREE.Color(0xff0000);
```



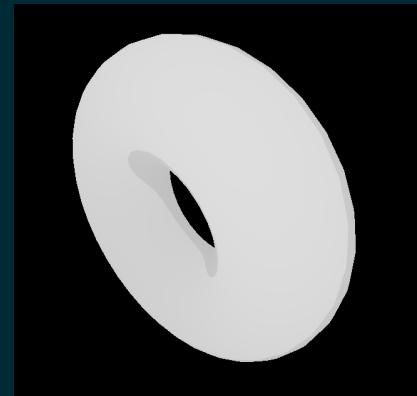
MESHTOONMATERIAL (1/3)

Ce matériau utilise le principe du toon shading.

MESHTOONMATERIAL (2/3)

On obtient ce résultat:

```
material = new THREE.MeshToonMaterial();
```



MESHTOONMATERIAL (3/3)

On observe qu'il y a deux colorations (une pour l'ombre et une pour la surface éclairée). On peut décider d'avoir plus d'étapes de coloration en donnant une gradientMap.

```
gradientTexture.minFilter = THREE.NearestFilter;  
gradientTexture.magFilter = THREE.NearestFilter;  
material.gradientMap = gradientTexture;
```

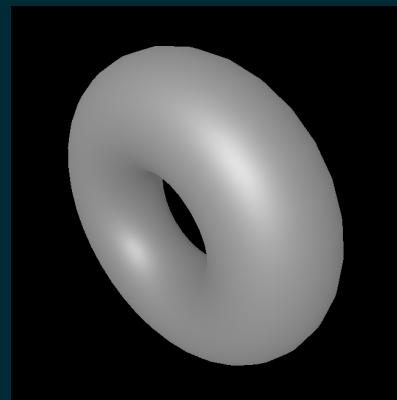
MESHSTANDARDMATERIAL (1/10)

Ce matériau se repose sur le concept de rendu basé sur la physique (PBR) par le biais de la rugosité et de la réflexion métallique de la surface. On parle de standard c'est devenu le standard dans pleins d'autres logiciels, moteurs de rendu, librairies. L'idée est d'obtenir un rendu réaliste avec des paramètres plus fins.

MESHSTANDARDMATERIAL (2/10)

On obtient ce résultat:

```
material = new THREE.MeshStandardMaterial();  
material.roughness = 0.6;  
material.metalness = 0.6;
```



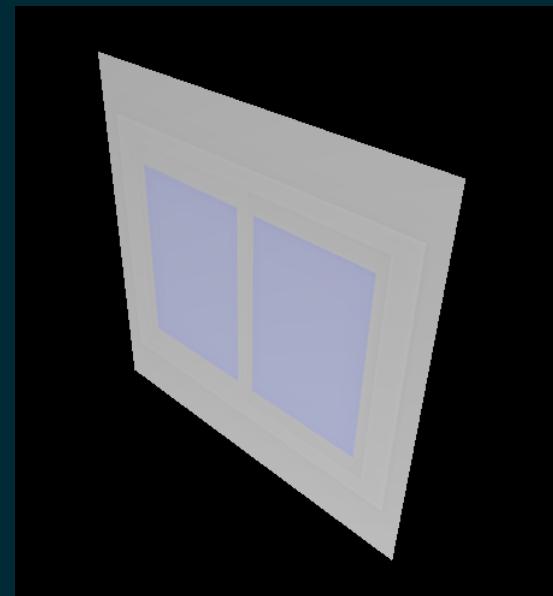
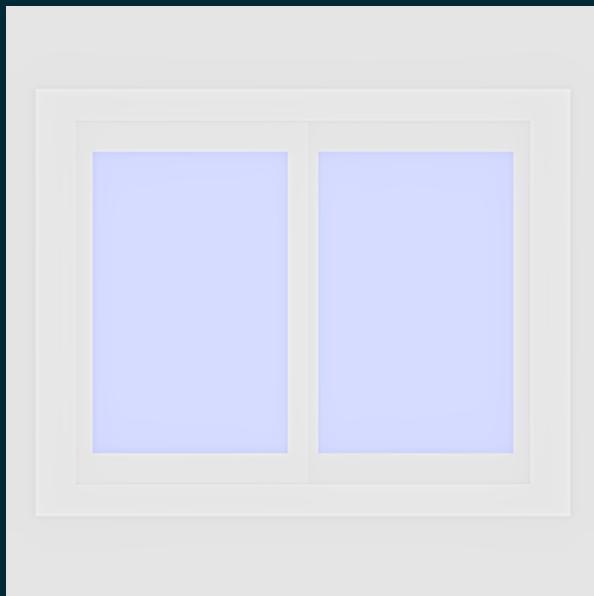
MESHSTANDARDMATERIAL (3/10)

Dans le chapitre des textures, on en a vu plusieurs types, on va donc voir concrètement ce que cela provoque:

MESHSTANDARDMATERIAL (4/10)

MAP

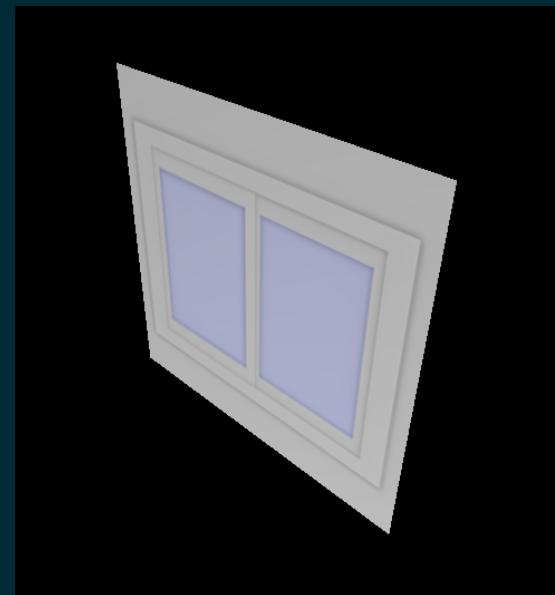
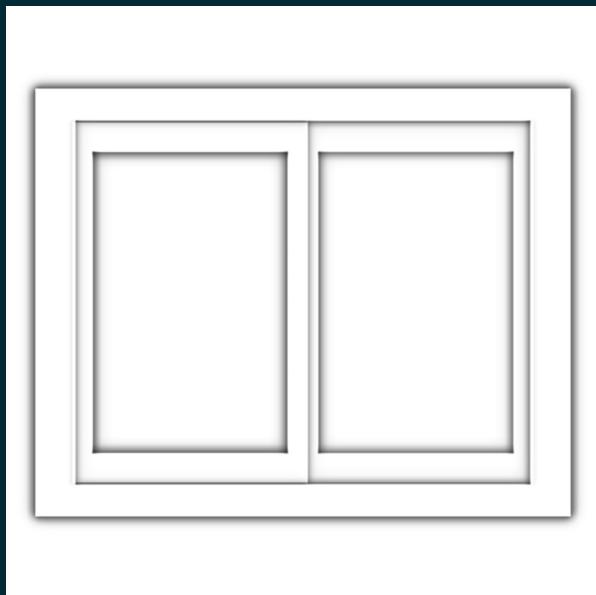
```
material.map = albedoTexture;
```



MESHSTANDARDMATERIAL (5/10)

AOMAP

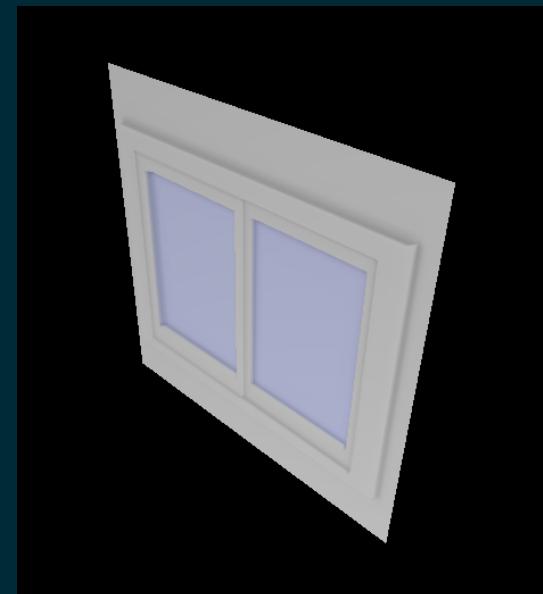
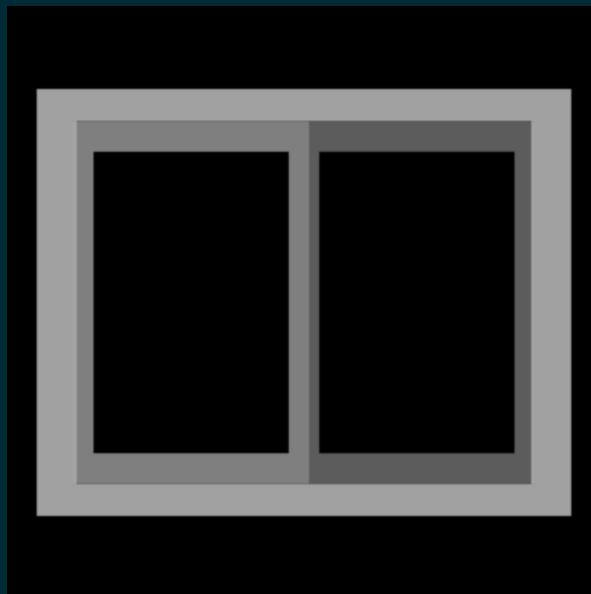
```
material.aoMap = aoTexture;  
material.aoIntensity = 1;
```



MESHSTANDARDMATERIAL (6/10)

DISPLACEMENTMAP

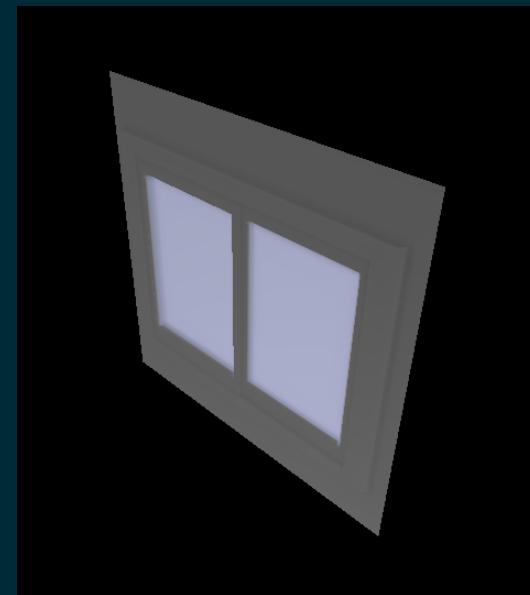
```
material.displacementMap = displacementTexture;  
material.displacementScale = 0.05;
```



MESHSTANDARDMATERIAL (7/10)

METALNESSMAP

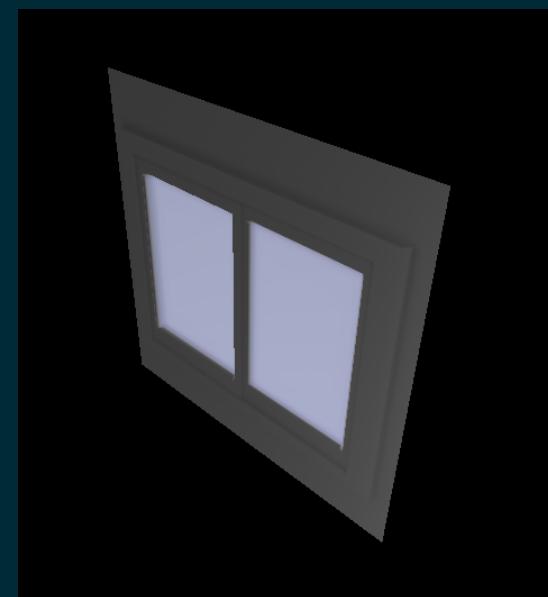
```
material.metalnessMap = metalnessTexture;  
material.metalness = 1;
```



MESHSTANDARDMATERIAL (8/10)

ROUGHNESSMAP

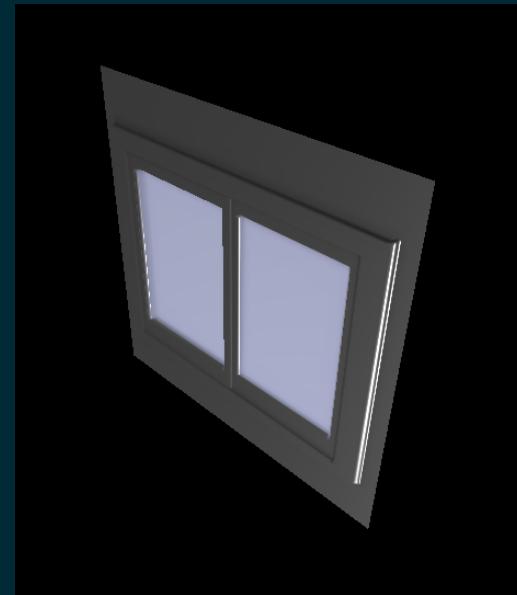
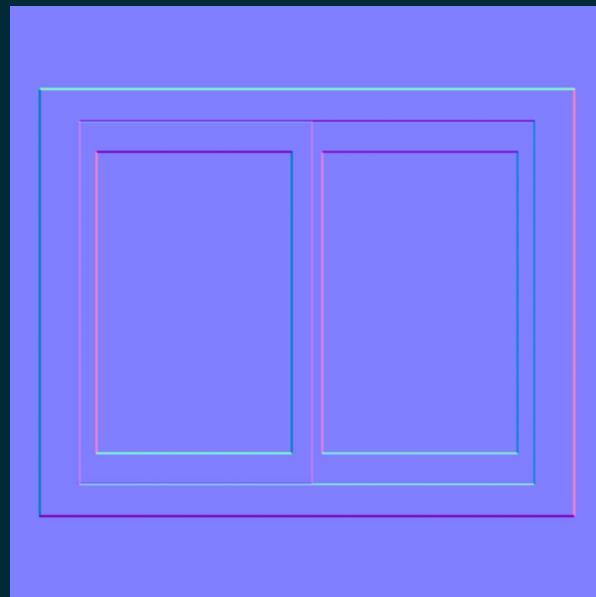
```
material.roughnessMap = roughnessTexture;  
material.roughness = 1;
```



MESHSTANDARDMATERIAL (9/10)

NORMALMAP

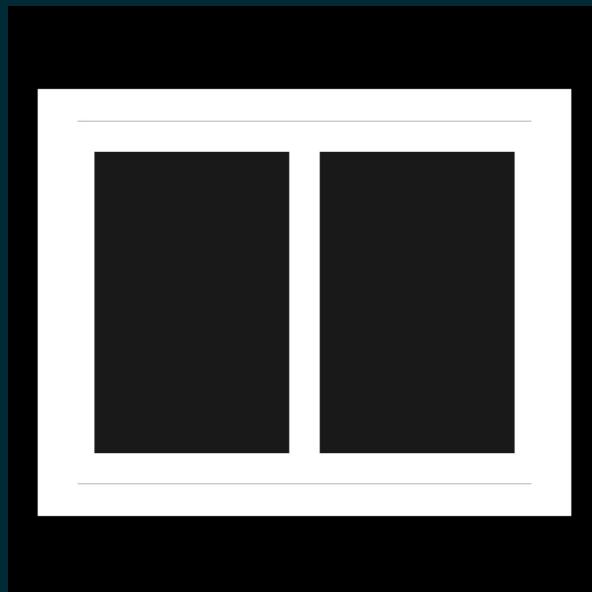
```
material.normalMap = normalTexture;
```



MESHSTANDARDMATERIAL (10/10)

ALPHAMAP

```
material.alphaMap = alphaTexture;  
material.transparent = true;
```



ENVIRONNEMENT (1/2)

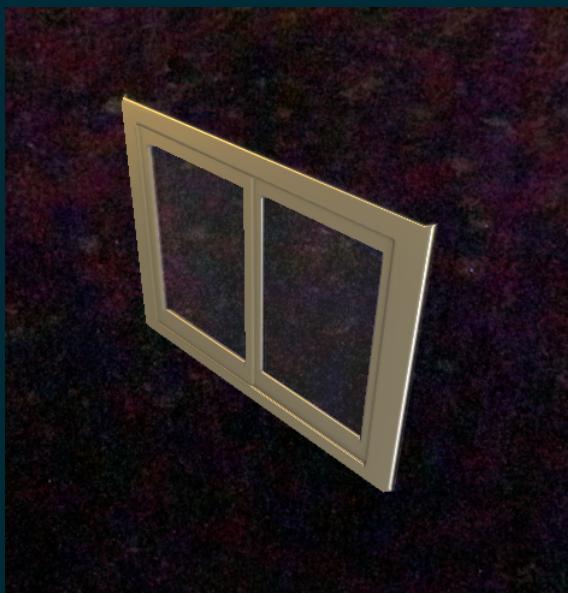
Pour donner plus de réflexion, de réfraction mais aussi de lumière aux objets en plus des lumières, on peut ajouter une texture d'environnement à la scène.



ENVIRONNEMENT (2/2)

```
const envLoader = new EXRLoader();
envLoader.load('textures/satara_night_4k.exr', (envTexture) => {
    envTexture.mapping = THREE.EquirectangularReflectionMapping;

    scene.environment = envTexture;
    scene.background = envTexture;
});
```



MESHPHYSICALMATERIAL (1/5)

Ce matériau est une extension du
MeshStandardMaterial avec des nouvelles propriétés:

MESHPHYSICALMATERIAL (2/5)

VERNIS

Certains matériaux comme les peintures automobiles, les fibres de carbone, les surfaces humides, nécessitent une couche transparent et réfléchissante au-dessus d'une autre couche qui peut être irrégulière ou rugueuse. La propriété clearcoat permet d'obtenir un tel effet.

```
material.clearcoat = 1;  
material.clearcoatRoughness = 0;
```

Exemple ici

MESHPHYSICALMATERIAL (3/5)

BRILLANCE

Peut être utilisé pour représenter les tissus et les matières textiles.

```
material.sheen = 1  
material.sheenRoughness = 0.25  
material.sheenColor.set(1, 1, 1)
```

Exemple ici

MESHPHYSICALMATERIAL (4/5)

IRIDESCENCE

Pour représenter des artefacts de couleur comme des bulles de savon ou les effets de reflet sur les CDs.

```
material.iridescence = 1  
material.iridescenceIOR = 1  
material.iridescenceThicknessRange = [100, 800]
```

Exemple ici

MESHPHYSICALMATERIAL (5/5)

TRANSMISSION

Pour simuler la lumière qui traverse l'objet. Et cela permet donc de déformer à travers l'objet.

```
material.transmission = 1  
material.ior = 1.5  
material.thickness = 0.5
```

Exemple ici

Les deux matériaux peuvent être très coûteux en terme de performance. Ne vous attendez pas à une expérience fluide sur tous les appareils si vous commencez à entre mettre sur tous les objets.

Il reste les PointsMaterial, ShaderMaterial et RawShaderMaterial qui auront leurs chapitres dédiés.

LES LUMIÈRES

AMBIENTLIGHT (1/2)

Cette lumière illumine globalement tous les objets de la scène de manière équitable. On le verra par la suite pour les ombres, mais cette lumière ne provoque pas d'ombre vu qu'elle n'a pas de direction

AMBIENTLIGHT (2/2)

```
const ambientLight = new THREE.AmbientLight(0xffffff, 1.5)  
scene.add(ambientLight)
```



DIRECTIONALLIGHT (1/3)

Cette lumière est émise dans une direction spécifique.

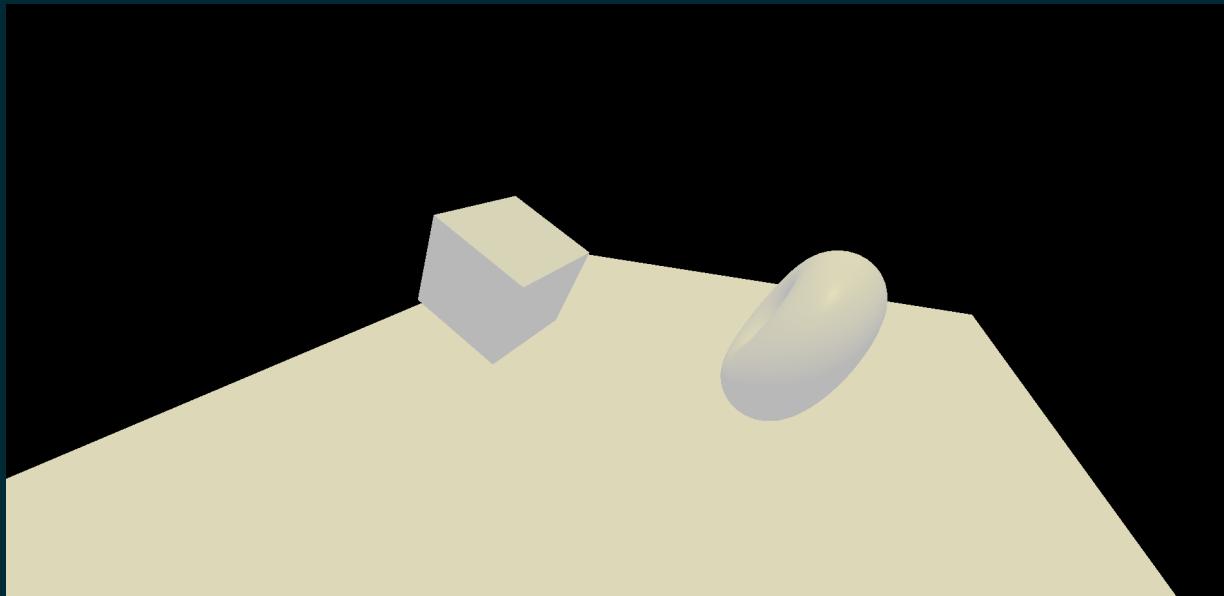
Cette lumière se comportera comme si était infinie
éloignée et que les rayons produits étaient tous
parallèles.

A quoi peut-on la comparer ?

Le soleil

DIRECTIONALLIGHT (2/3)

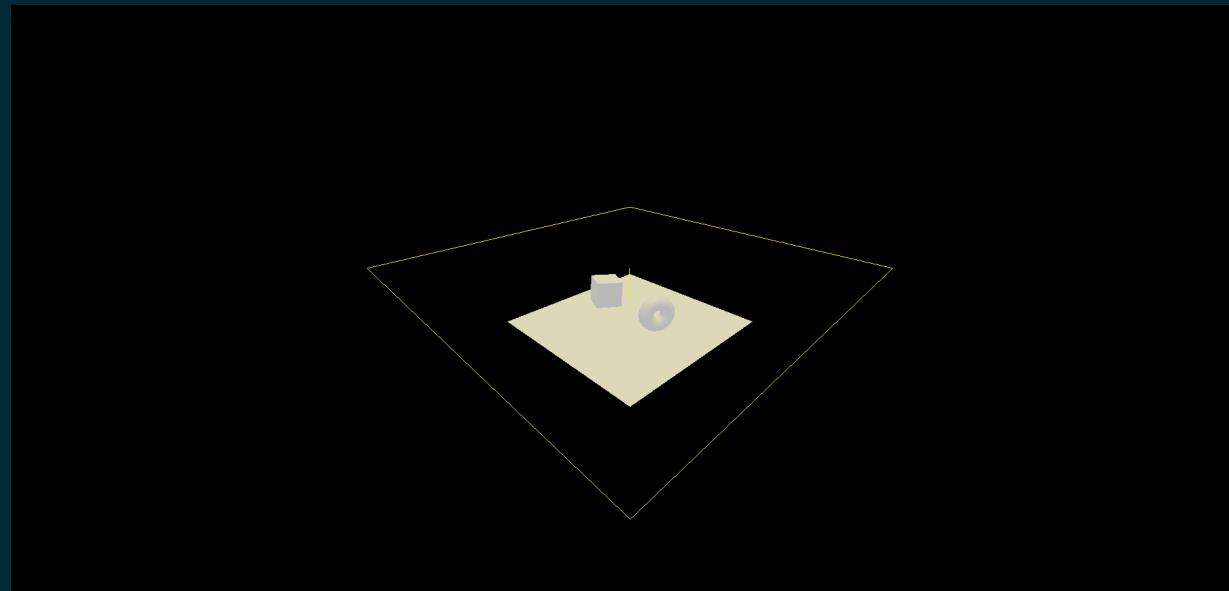
```
const directionalLight = new THREE.DirectionalLight(0xffff00, 0.75)  
scene.add(directionalLight)
```



DIRECTIONALLIGHT (3/3)

Pour aider la visualisation, on dispose d'un helper

```
const helper = new THREE.DirectionalLightHelper(directionalLight, 5);  
scene.add(helper);
```

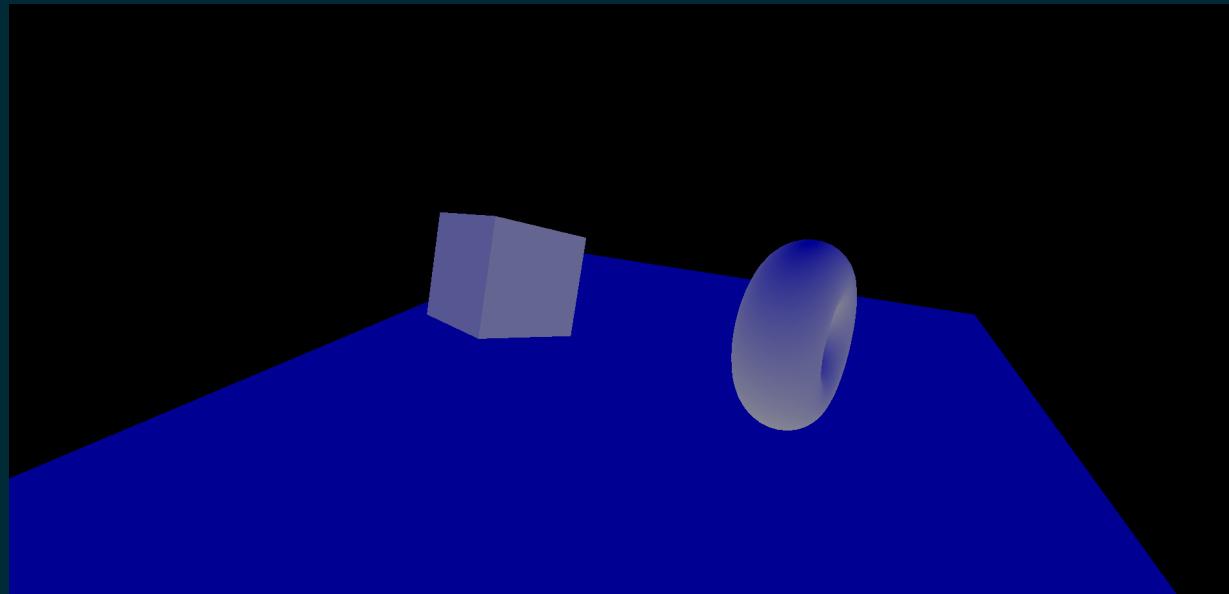


HEMISPHERELIGHT (1/3)

Cette lumière est une source de lumière positionnée juste au-dessus de la scène, avec une couleur qui s'estompe de la couleur du ciel à celle du sol.

HEMISPERELIGHT (2/3)

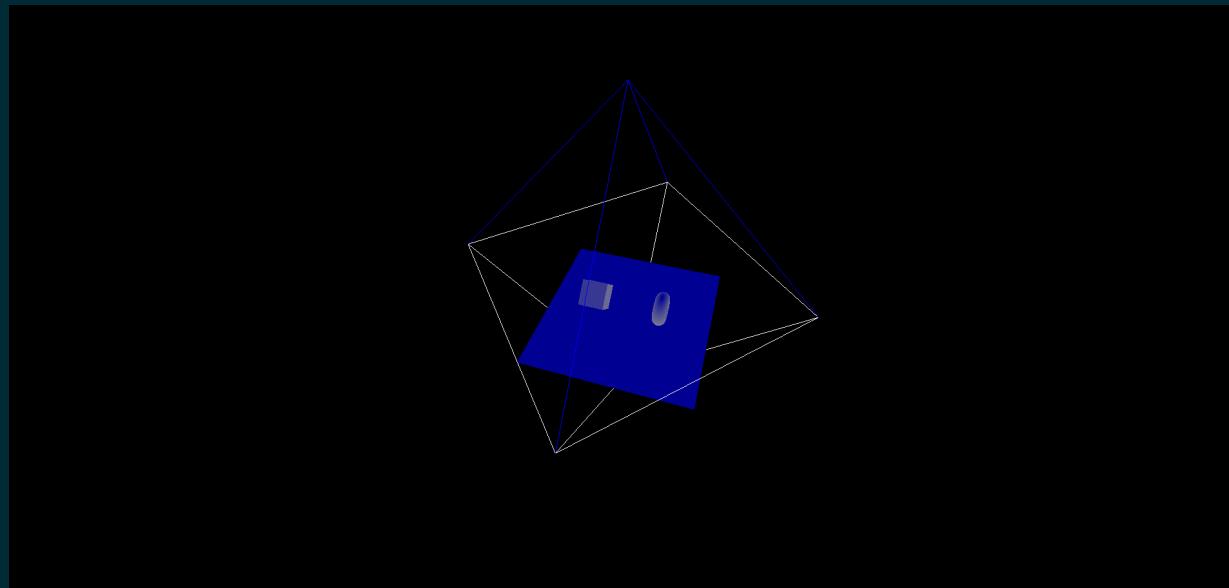
```
const hemisphereLight = new THREE.HemisphereLight(0x0000ff, 0xffffffff, 0.9)  
scene.add(hemisphereLight)
```



HEMISPHERELIGHT (3/3)

Pour aider la visualisation, on dispose d'un helper

```
const helper = new THREE.HemisphereLightHelper(directionalLight, 5);  
scene.add(helper);
```



POINTLIGHT (1/3)

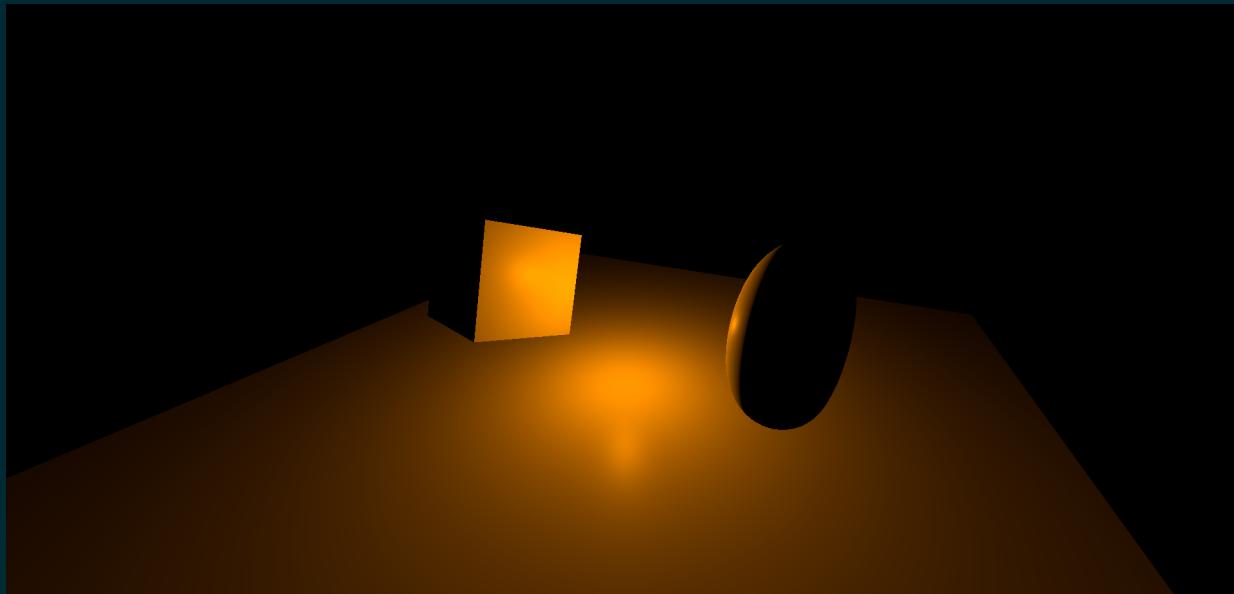
Cette lumière est une source de lumière qui est émise à partir d'un seul point dans toutes les directions.

A quoi peut-on la comparer ?

Une ampoule

POINTLIGHT (2/3)

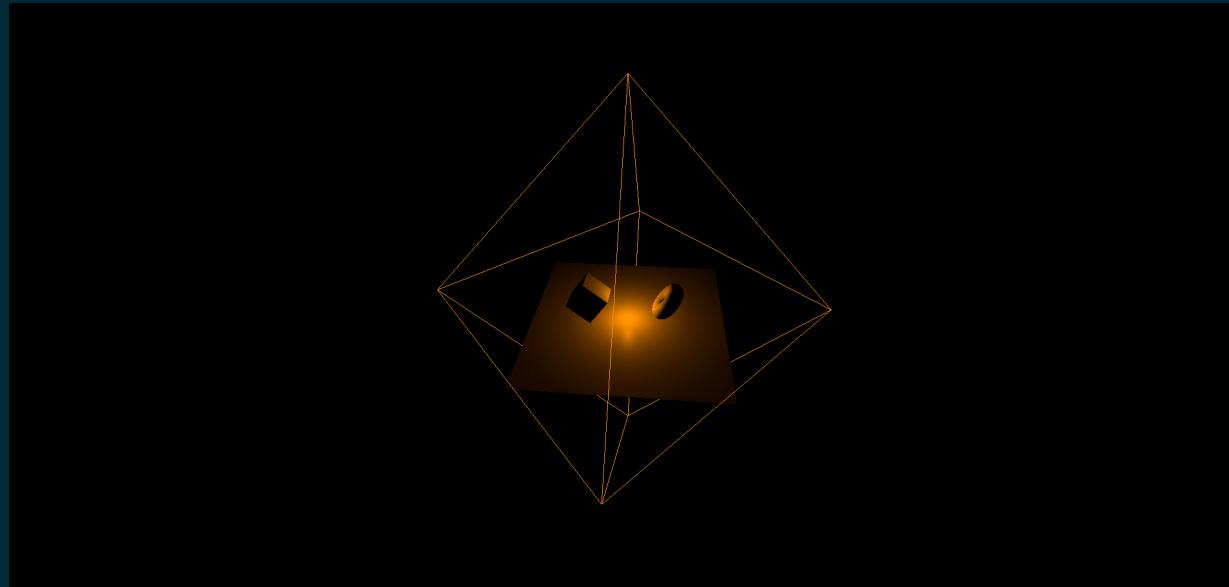
```
const pointLight = new THREE.PointLight(0xff9000, 1.5)  
scene.add(pointLight)
```



POINTLIGHT (3/3)

Pour aider la visualisation, on dispose d'un helper

```
const helper = new THREE.PointLightHelper(pointLight, 5);  
scene.add(helper);
```



RECTAREALIGHT (1/3)

Cette lumière émet uniformément sur la face d'un plan rectangulaire.

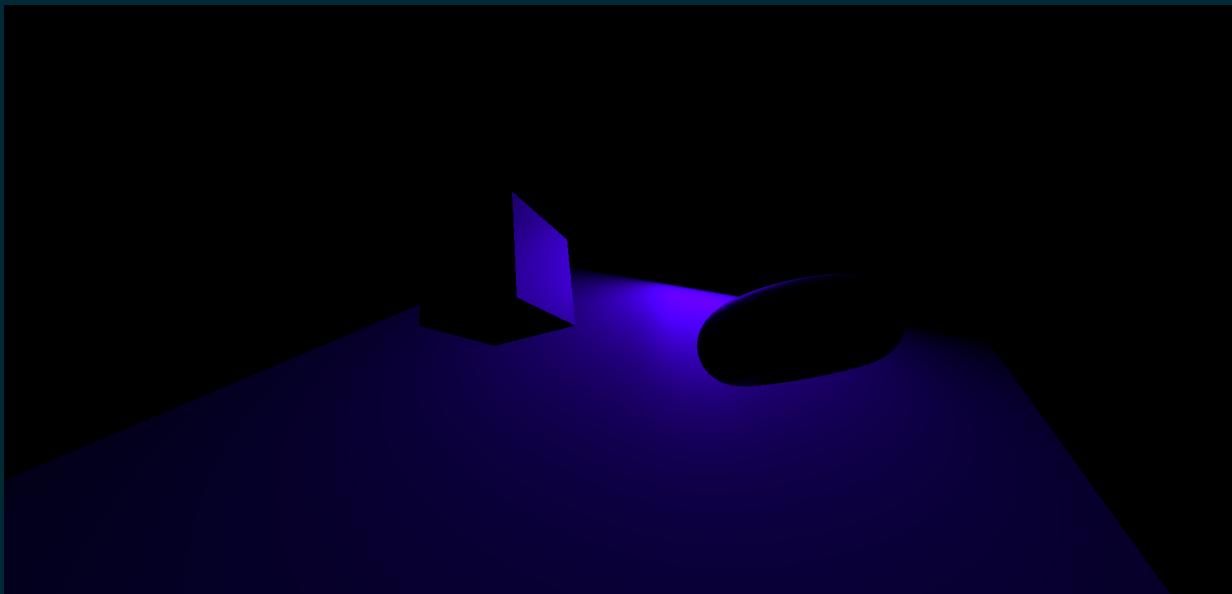
A quoi peut-on la comparer ?

Une fenêtre lumineuse ou des bandes lumineuses

RECTAREALIGHT (2/3)

```
RectAreaLightUniformsLib.init();

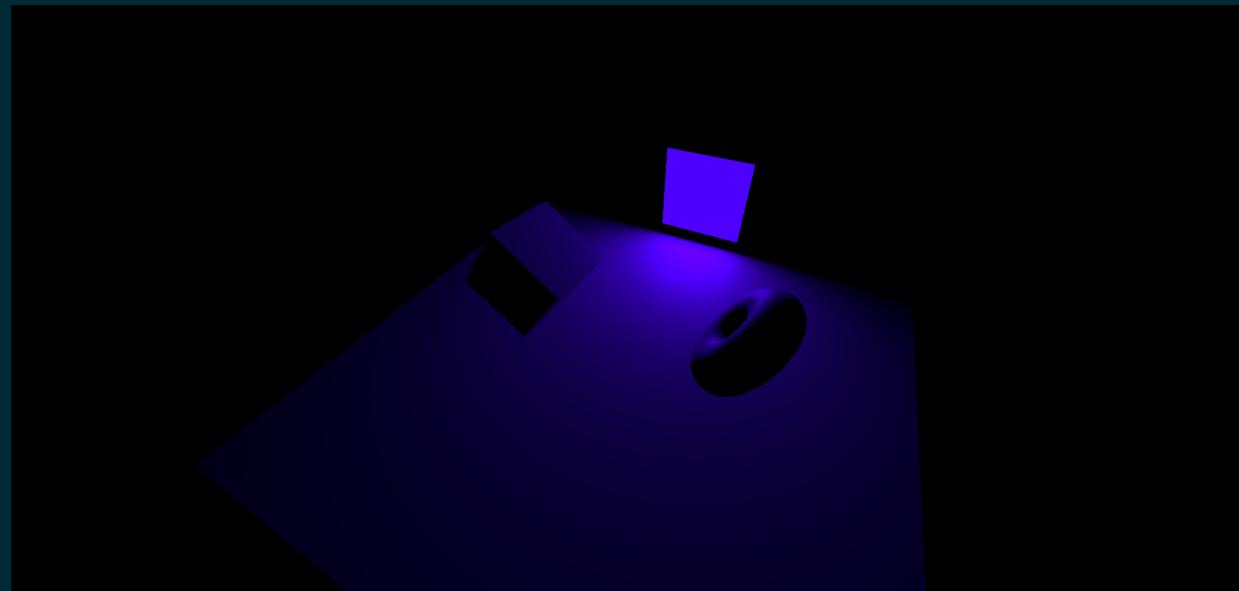
const rectAreaLight = new THREE.RectAreaLight(0x4e00ff, 6, 1, 1)
scene.add(rectAreaLight)
rectAreaLight.position.z = -2;
rectAreaLight.lookAt(new THREE.Vector3())
```



RECTAREALIGHT (3/3)

Pour aider la visualisation, on dispose d'un helper

```
const helper = new RectAreaLightHelper(rectAreaLight)  
scene.add(helper)
```



SPOTLIGHT (1/3)

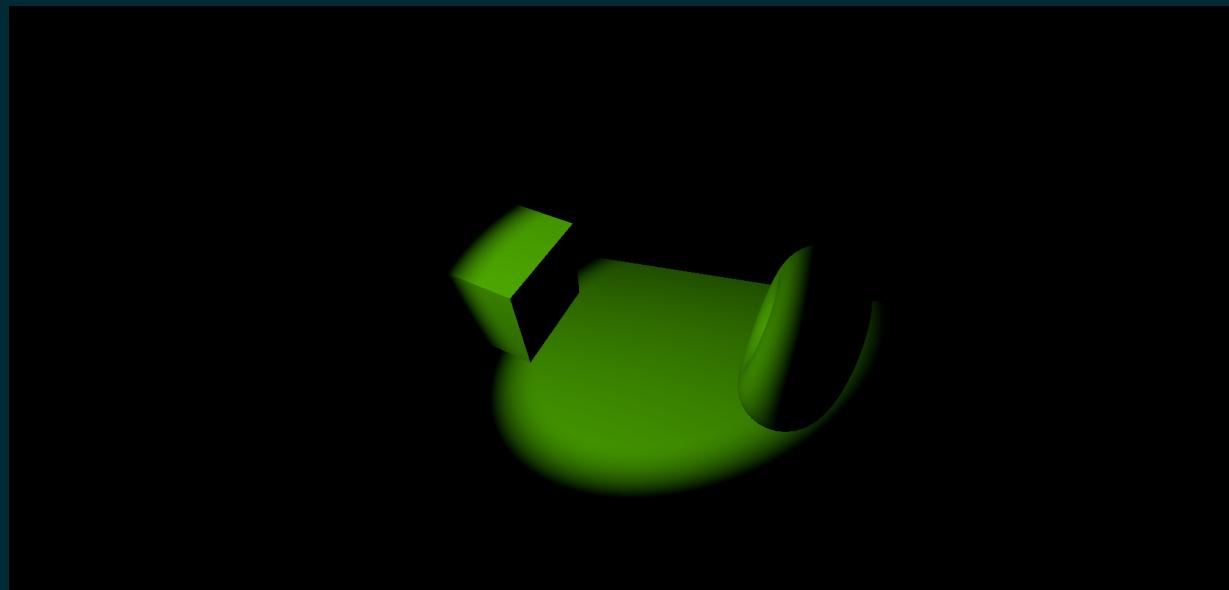
Cette lumière émet à partir d'un seul point dans une seule direction, le long d'un cône dont la taille augmente au fur et à mesure que l'on s'éloigne de la lumière.

A quoi peut-on la comparer ?

Un projecteur

SPOTLIGHT (2/3)

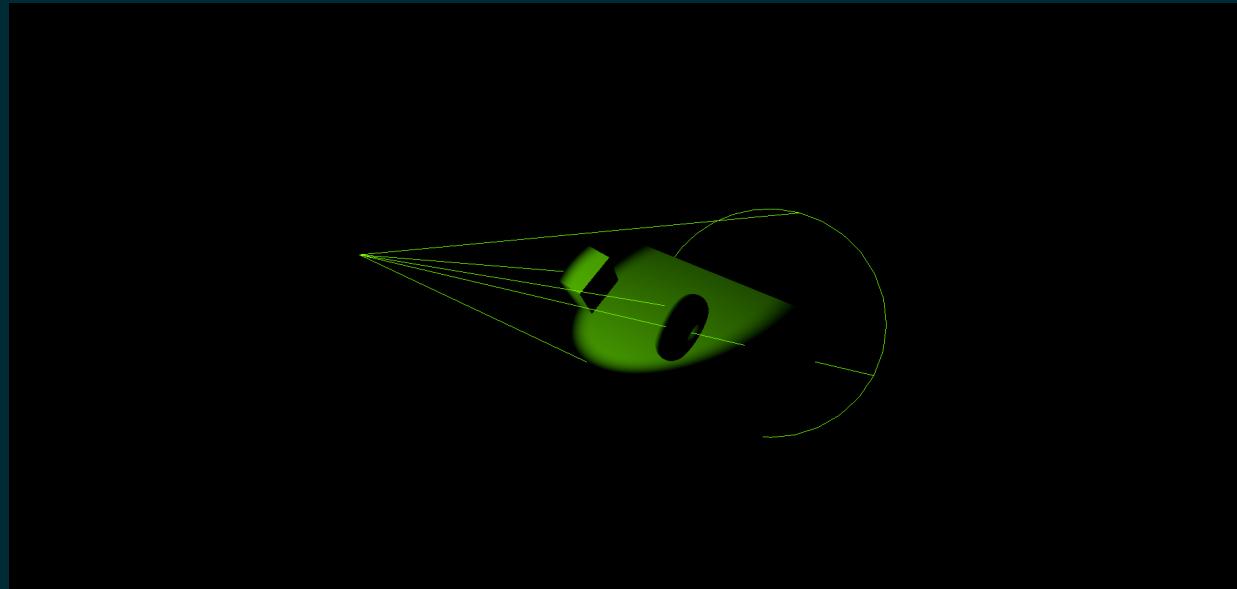
```
const spotLight = new THREE.SpotLight(  
  0x78ff00, 4.5,  
  10, Math.PI * 0.1,  
  0.25, 1  
)  
spotLight.position.set(0, 2, 3)  
scene.add(spotLight)
```



SPOTLIGHT (3/3)

Pour aider la visualisation, on dispose d'un helper

```
const helper = new THREE.SpotLightHelper(spotLight)  
scene.add(helper)
```



LES OMBRES (1/12)

Maintenant que l'on sait ajouter des lumières, on va pouvoir commencer à avoir des ombres.

On a 2 types d'ombres:

- les ombres dites **core**: ce sont les ombres sur les objets eux-mêmes. Les parties naturellement sombres sur les objets
- les ombres dites **portées**: ce sont les ombres que les objets créent sur les autres

LES OMBRES (2/12)

COMMENT CELA FONCTIONNE ?

Quand une frame est rendue, ThreeJS va d'abord faire un rendu pour chacune des lumières qui sont supposées émettre des ombres.

LES OMBRES (3/12)

COMMENT CELA FONCTIONNE ?

L'idée c'est de rendre du point de vue de chaque lumière, comme s'il y avait une caméra à chaque emplacement de lumière. Pendant ce rendu, les objets voient leurs matériaux changer pour un MeshDepthMaterial

LES OMBRES (4/12)

COMMENT CELA FONCTIONNE ?

Le résultat de ces rendus intermédiaires est stocké dans ce qu'on appelle des shadow maps et elles sont ensuite utilisées sur les matériaux des objets qui sont supposés recevoir les ombres

Voilà un excellent exemple ici

LES OMBRES (5/12)

De même que pour les lumières, gardez à l'esprit que les ombres sont coûteuses en terme de performance, donc utilisez vraiment à petite dose.

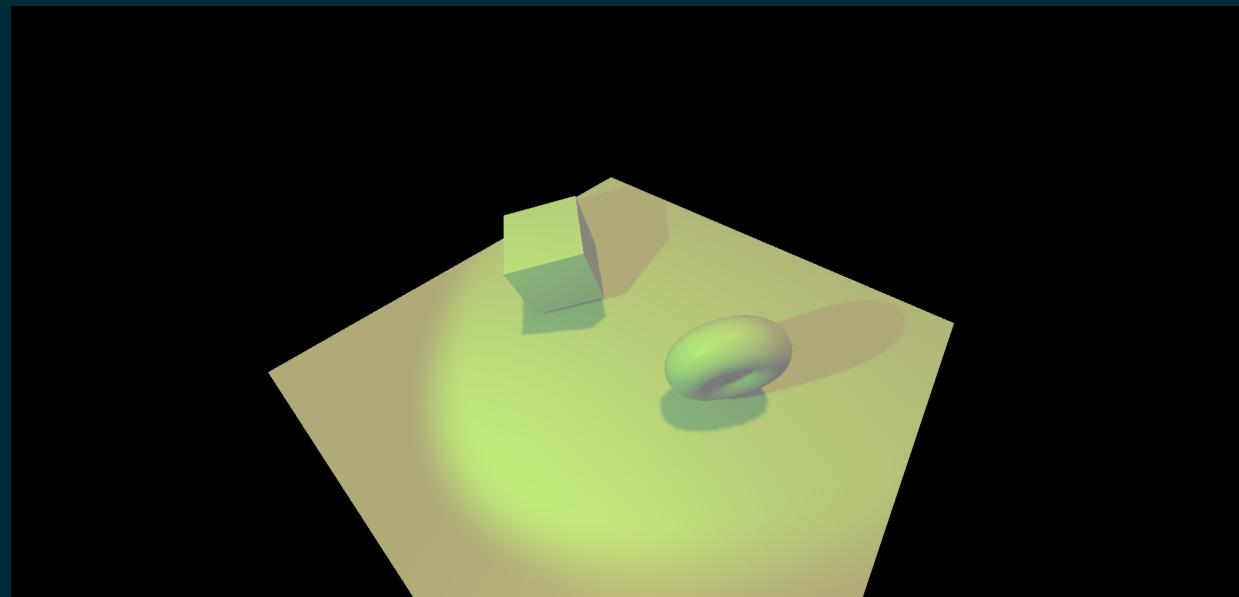
LES OMBRES (6/12)

Pour activer les ombres:

```
// Active le système d'ombre pour le renderer  
renderer.shadowMap.enabled = true;  
  
// Sur chaque objet que l'on veut qu'il émette une ombre  
object.castShadow = true;  
  
// Sur chaque objet que l'on veut qu'il reçoive les ombres des autres  
object.receiveShadow = true;  
  
// Sur chaque lumière dont on veut émettre des ombres  
light.castShadow = true;
```

LES OMBRES (7/12)

On obtient ce résultat:



LES OMBRES (8/12)

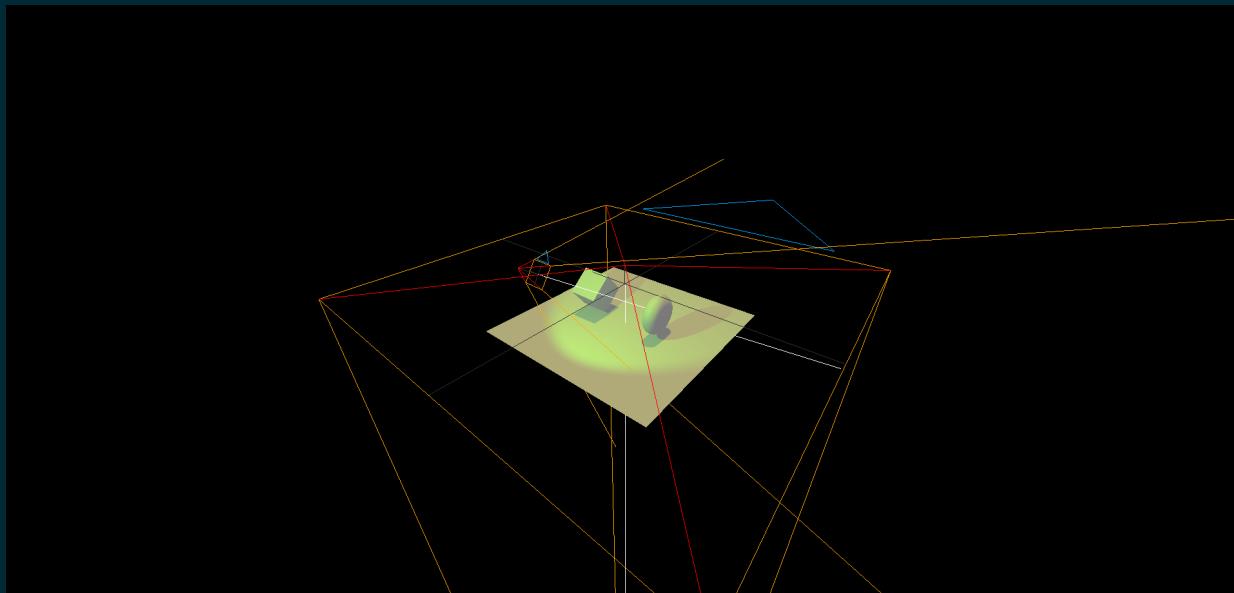
On observe que la qualité des ombres n'est pas terrible.
Voyons comment améliorer ça:

```
/*
 * On peut améliorer la taille de la texture utilisée pour générer
 * la shadow map. Par défaut elle est de 512x512.
 * On se rappelle, on garde une taille en puissance de 2 pour le mipmapping
 */
directionalLight.shadow.mapSize.width = 1024
directionalLight.shadow.mapSize.height = 1024
```

LES OMBRES (9/12)

On peut debug la caméra utilisée par une lumière:

```
const helper = new THREE.CameraHelper(directionalLight.shadow.camera);  
scene.add(helper)
```



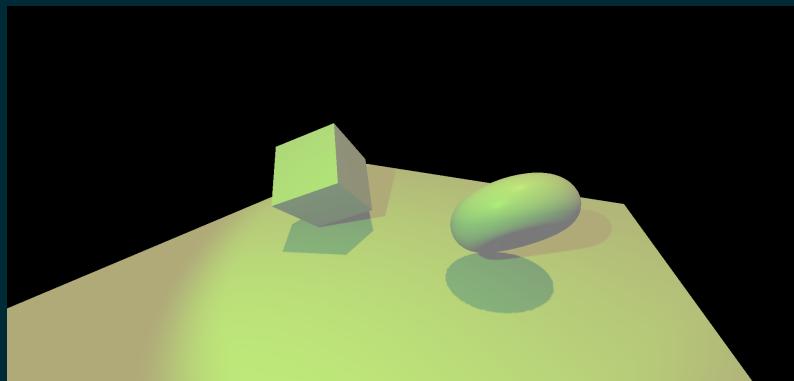
LES OMBRES (10/12)

Ces caméras possèdent les mêmes propriétés que l'on a vu lors du chapitre des caméras.

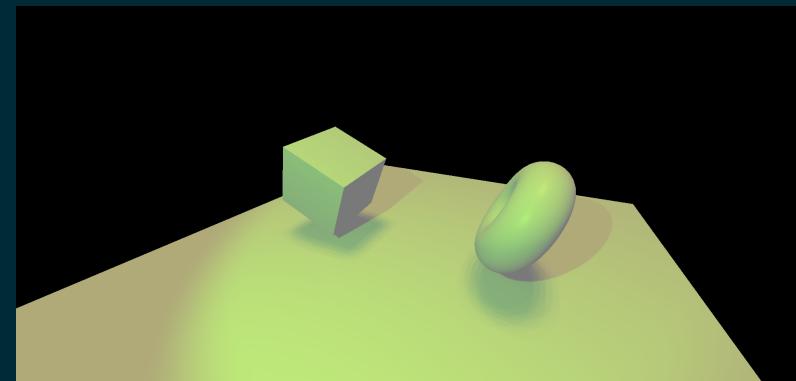
Changer ces valeurs ne changera pas fondamentalement la qualité mais pourra limiter l'apparition de bugs.

LES OMBRES (11/12)

La propriété `radius` sur la shadow d'une lumière permet d'augmenter ou diminuer le flou autour de l'ombre.



```
directionalLight.shadow.radius = 0.5;
```



```
directionalLight.shadow.radius = 10;
```

LES OMBRES (12/12)

Il existe plusieurs algorithmes pour le rendu des shadows:

- THREE.BasicShadowMap: très performant mais qualité moyenne
- THREE.PCFShadowMap: moins performant mais arrêtes lissées (valeur par défaut)
- THREE.PCFSOFTShadowMap: moins performant mais arrêtes encore plus lissées
- THREE.VSMShadowMap: moins performant, plus de contraintes et résultats inattendus

BAKING (1/5)

LUMIÈRES

On a vu que lumières pouvaient faire baisser les performances. Il existe une autre technique qui permet d'avoir des rendus réalistes sans pour autant avoir les lumières en temps réel. C'est ce qu'on appelle le baking.

BAKING (2/5)

LUMIÈRES

L'idée est de rendre le résultat des couleurs (avec la lumière) directement dans la texture. Cela peut être fait dans des logiciels 3D. Revers de la médaille, on ne pourra plus bouger les lumières en temps réel par exemple.

Donc cela dépendra réellement du résultat voulu.

BAKING (3/5)

LUMIÈRES

Voilà un exemple de ce que cela peut donner:



BAKING (4/5)

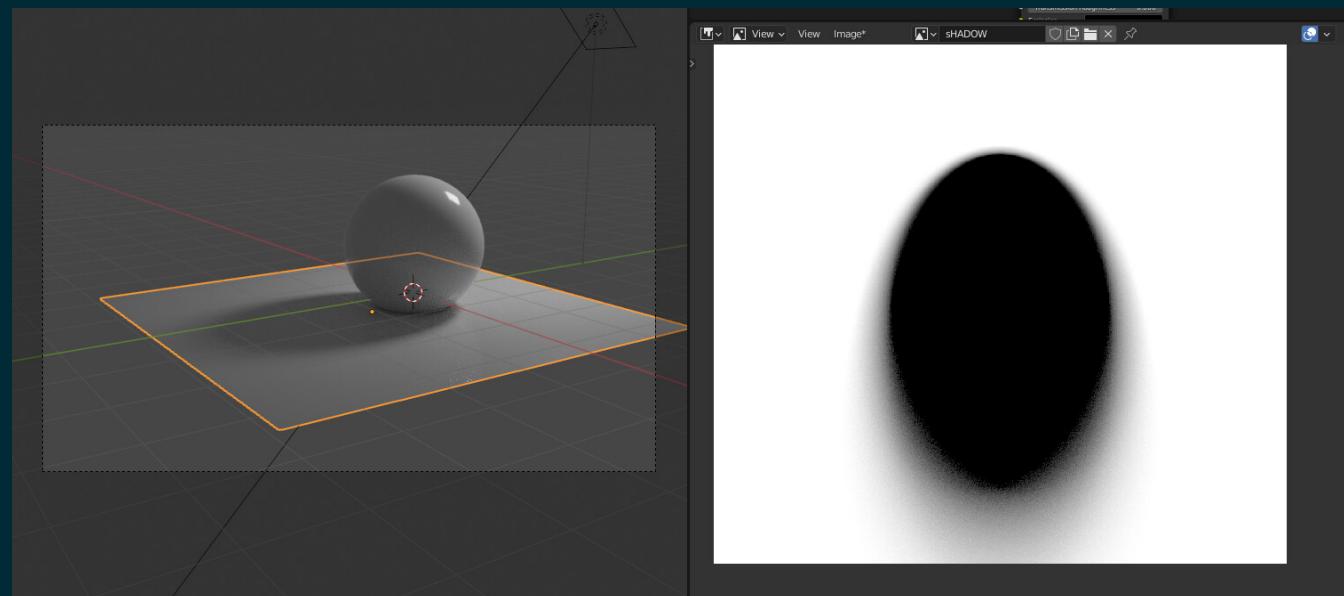
OMBRES

Pour les ombres, c'est exactement pareil, on peut créer des textures qui encapsulent les ombres. Mais là encore une fois, ce n'est là que pour créer de l'illusion, et donc si les objets ou les lumières bougent, les ombrent ne suivront pas.

BAKING (5/5)

OMBRES

Voilà un exemple de ce que cela peut donner:



TEXTE 3D

INTRODUCTION (1/3)

On a vu pour ajouter des types de contenus simples fournis par ThreeJS. On va maintenant voir comment ajouter du texte dans nos scènes 3D.

INTRODUCTION (2/3)

ThreeJS supporte les textes 3D à travers la classe
[TextGeometry](#)

Pour que cela puisse fonctionner, nous allons avoir
besoin de fichiers de polices d'écriture dans un format
JSON particulier appelé typeface

INTRODUCTION (3/3)

Si vous possédez une font, vous pourrez la convertir via ce lien. Ou alors ThreeJS a déjà quelques fonts de base déjà prêtées à l'emploi dans le dossier
three/examples/fonts

CHARGER UNE FONT (1/2)

Première manière de faire:

```
import { FontLoader } from 'three/examples/jsm/loaders/FontLoader';
import myFontFace from 'three/examples/fonts/gentilis_regular.typeface.json';

// Permet de parse directement depuis le fichier JSON de la typeface
const fontLoader = new FontLoader();
const myFont = fontLoader.parse(myFontFace);
console.log(myFont);
```

CHARGER UNE FONT (2/2)

Deuxième manière de faire:

```
import { FontLoader } from 'three/examples/jsm/loaders/FontLoader';

const fontLoader = new FontLoader();
fontLoader.load('./assets/fonts/gentilis_regular.typeface.json', (myFont) => {
  console.log(myFont);
});
```

LA GÉOMÉTRIE (1/2)

```
import { TextGeometry } from 'three/examples/jsm/geometries/TextGeometry';

fontLoader.load('./assets/fonts/gentilis_regular.typeface.json', (font) => {
  const textGeometry = new TextGeometry(
    'Hello world !',
    {
      font,
      size: 0.5,
      height: 0.2,
      curveSegments: 12,
      bevelEnabled: true,
      bevelThickness: 0.03,
      bevelSize: 0.02,
      bevelOffset: 0,
      bevelSegments: 5
    }
  );
  const text = new THREE.Mesh(textGeometry, new THREE.MeshBasicMaterial());
  scene.add(text);
});
```

LA GÉOMÉTRIE (2/2)

On obtient ce résultat:

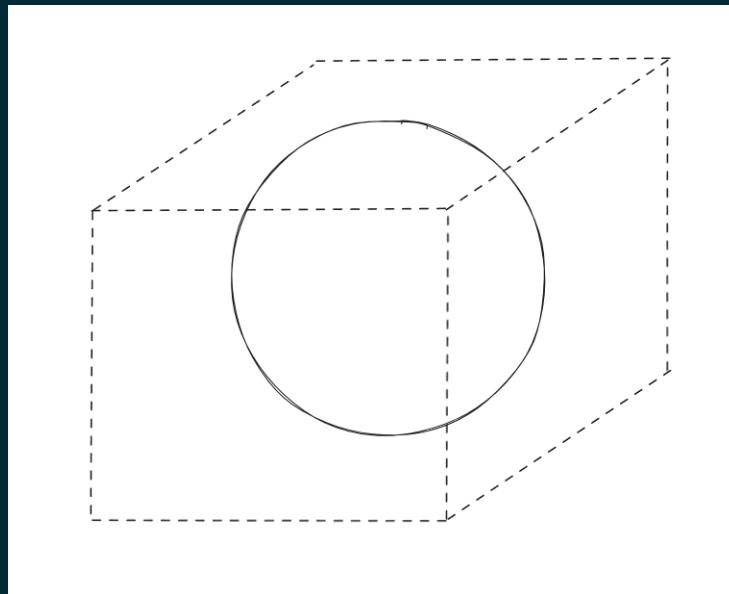


CENTRAGE DU TEXTE (1/6)

On observe que le texte n'est pas centré et on va essayer d'y remédier de 2 manières

CENTRAGE DU TEXTE (2/6)

On va parler de bounding box, c'est-à-dire la boîte qui englobe notre objet.



CENTRAGE DU TEXTE (3/6)

Pour pouvoir centrer le texte, on a besoin de connaître la taille de la bounding box:

```
textGeometry.computeBoundingBox();  
console.log(textGeometry.boundingBox);
```

CENTRAGE DU TEXTE (4/6)

Maintenant on peut déplacer la géométrie elle-même

```
const bbox = textGeometry.boundingBox;  
textGeometry.translate(  
  -(bbox.max.x - bbox.min.x) * 0.5,  
  -(bbox.max.y - bbox.min.y) * 0.5,  
  -(bbox.max.z - bbox.min.z) * 0.5,  
);
```

CENTRAGE DU TEXTE (5/6)

Il existe une méthode qui fait ça pour nous

```
textGeometry.center();
```

CENTRAGE DU TEXTE (6/6)

On obtient ce résultat:



OBJETS 3D

INTRODUCTION

On a créé des primitives simples, des textes en 3D mais quand on a besoin de modèles 3D plus complexes, on passe par des logiciels prévus pour ça.

FORMATS (1/12)

Il existe pleins de formats de modèles 3D dont chacun répondait à des problèmes précis:

- Poids
- Compression
- Compatibilité
- Droits
- ...

FORMATS (2/12)

Vous pouvez retrouver une liste complète de formats
[ici](#)

Certains formats sont propriétaires et dédiés à certains logiciels. D'autres sont connus pour être légers. Encore d'autres ont tous types de données mais plus lourds. Enfin certains sont open source, écrits en binaires ou en ASCII, etc.

FORMATS (3/12)

Voilà une petite liste des formats les plus populaires:

- OBJ (.obj)
- FBX (.fbx)
- PLY (.ply)
- COLLADA (.dae)
- GLTF (.gltf)

FORMATS (4/12)

GLTF (1/9)

- GL Transmission Format
- Créé par Khronos Group (OpenGL, WebGL, Vulkan, etc.)
- Le format le plus populaire depuis quelques années
- Supporte pleins de types de données: géométries, caméras, lumières, animations, squelettes, morphing

FORMATS (5/12)

GLTF (2/9)

- Supporte plusieurs formats comme JSON, binaires et même textures embarquées
- Supporté dans pleins de logiciels

Ce n'est pas la réponse à tout. En fonction des besoins, d'autres formats peuvent être plus adaptés

FORMATS (6/12)

GLTF (3/9)

A l'intérieur même du glTF, il existe plusieurs formats

FORMATS (7/12)

GLTF (4/9) - DEFAULT

C'est un fichier JSON qui contient différentes informations telles que les caméras, les lumières, les scènes, les matériaux, les transformations des objets mais pas les géométries ou les textures elle-mêmes.

Les informations manquantes sont stockées dans d'autres fichiers et le fichier default va load tout de manière automatique

FORMATS (8/12)

GLTF (5/9) - BINARY

Ce fichier contient tout ce dont on a parlé dans le format default. Ce fichier est en binaire et donc on ne peut pas l'ouvrir facilement dans notre éditeur. Ce fichier est souvent plus léger et plus simple à load par le fait qu'il n'y en a qu'un.

FORMATS (9/12)

GLTF (6/9) - DRACO

Ce format est le même que le default mais le buffer de données utilise l'algorithme **Draco**. On constate que ce fichier est plus léger comparé au format binary.

FORMATS (10/12)

GLTF (7/9) - EMBEDDED

Ce format est le même que le binary car il n'y a qu'un seul fichier mais c'est un JSON au lieu d'être un fichier binaire et donc on peut l'ouvrir dans notre éditeur

FORMATS (11/12)

GLTF (8/9) - LEQUEL ?

Cela va dépendre dans la manière sont gérés les assets.

Si vous voulez interagir avec les lumières ou les coordonnées de texture, le default est mieux. De plus, du fait de charger plusieurs fichiers va améliorer la rapidité de chargement.

Si vous voulez qu'un seul fichier par modèle, le binary semblera plus adapté.

FORMATS (12/12)

GLTF (9/9) - LEQUEL ?

Peu importe le choix, il faudra choisir si vous voulez utiliser la compression draco ou non.

PREMIER MODÈLE 3D (1/12)

Vous pouvez créer vos propres modèles 3D à partir de logiciels comme Blender, Maya, 3DSMax, etc.

Dans le cours, nous allons utiliser ceux que mette à disposition l'équipe en charge du GLTF que vous pouvez retrouver [ici](#)

PREMIER MODÈLE 3D (2/12)

Première chose dont on va avoir besoin, c'est le
[GLTFLoader](#)

```
import { GLTFLoader } from 'three/examples/jsm/loaders/GLTFLoader';  
  
const gltfLoader = new GLTFLoader();
```

PREMIER MODÈLE 3D (3/12)

Ensuite on peut load notre fichier .gltf

```
gltfLoader.load(  
    './assets/models/duck/glTF/Duck.gltf',  
    (gltf) => {  
        console.log(gltf);  
    },  
    (progress) => {  
        console.log(progress);  
    },  
    console.error,  
);
```

PREMIER MODÈLE 3D (4/12)

La structure du gltf peut être un peu complexe et on va devoir identifier ce qui nous intéresse:

```
▼ Object 1
  ► animations: []
  ► asset: {generator: 'COLLADA2GLTF', version: '2.0'}
  ► cameras: [PerspectiveCamera]
  ► parser: GLTFParser {json: {...}, extensions: {...}, plugins: {...}, options: {...}, cache: {...}, ...}
  ► scene: Group {isObject3D: true, uuid: '8683b745-7562-40d6-aff7-7024a06b3a22', name: '', type: 'Group', parent: null, ...}
  ► scenes: [Group]
  ► userData: {}
  ► [[Prototype]]: Object
```

PREMIER MODÈLE 3D (5/12)

Dans ce cas-ci, on va venir récupérer le bon élément
dans la hiérarchie

```
▼ Object [1]
  ► animations: []
  ► asset: {generator: 'COLLADA2GLTF', version: '2.0'}
  ► cameras: [PerspectiveCamera]
  ► parser: GLTFParser {json: {...}, extensions: {...}, plugins: {...}, options: {...}, cache: {...}, ...}
  ▼ scene: Group
    ► animations: []
    castShadow: false
    ▼ children: Array(1)
      ▼ 0: _Object3D
        ► animations: []
        castShadow: false
        ► children: (2) [Mesh, PerspectiveCamera]
        frustumCulled: true
        isObject3D: true
        ► layers: Layers {mask: 1}
        ► matrix: _Matrix4 {elements: Array(16)}
        matrixAutoUpdate: true
        ► matrixWorld: _Matrix4 {elements: Array(16)}
        matrixWorldAutoUpdate: true
        matrixWorldNeedsUpdate: true
        name: ""
        ► parent: Group {isObject3D: true, uuid: '93ff59e3-e682-40fe-a2f3-548c8ebfd7b5', name: '', type: 'Group', parent: null, ...}
```

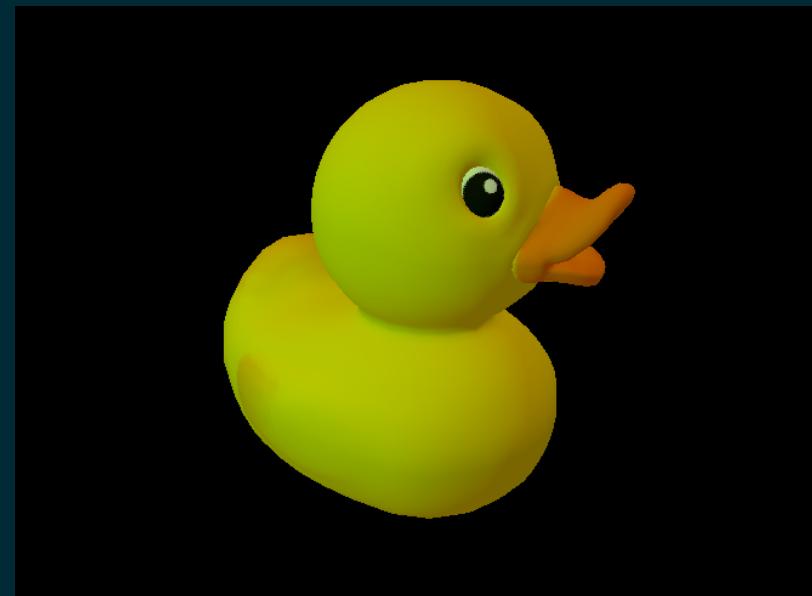
PREMIER MODÈLE 3D (6/12)

Dans notre cas, c'est le premier enfant de la scène:

```
gltfLoader.load(  
    './assets/models/duck/glTF/Duck.gltf',  
    (gltf) => {  
        scene.add(gltf.scene.children[0]);  
    },  
    (progress) => {  
        console.log(progress);  
    },  
    console.error,  
);
```

PREMIER MODÈLE 3D (7/12)

On obtient ce résultat:



PREMIER MODÈLE 3D (8/12)

Comme vu précédemment, on peut essayer différents formats:

```
gltfLoader.load(  
  './assets/models/duck/glTF/Duck.gltf', // default  
  
gltfLoader.load(  
  './assets/models/duck/glTF-Binary/Duck.glb', // binary  
  
gltfLoader.load(  
  './assets/models/duck/glTF-Embedded/Duck.gltf', // embedded
```

Draco ne fonctionnera pas pour le moment, on le verra un peu plus loin

PREMIER MODÈLE 3D (9/12)

Dans notre cas on ajoutait seulement le premier enfant de la scène du gltf. Mais il arrivera qu'il peut y avoir plusieurs objets enfants voire même plusieurs scènes.

PREMIER MODÈLE 3D (10/12)

Voilà un exemple avec un autre modèle:

```
▼ {scene: Group, scenes: Array(1), animations: Array(0), cameras: Array(0), asset: {...}, ...} ⓘ
  ► animations: []
  ► asset: {copyright: 'Original assets by Moeen Sayed and Mujtaba Sayed for SideFX.', generator: 'Khronos glTF Blender I/O v3.3.17', version: '2.0'}
  ► cameras: []
  ► parser: GLTFParser {json: {...}, extensions: {...}, plugins: {...}, options: {...}, cache: {...}, ...}
  ▼ scene: Group
    ► animations: []
    ► castShadow: false
    ▼ children: Array(33)
      ► 0: Mesh {isObject3D: true, uuid: 'eleae177-6fe1-42c7-b555-2e0a301efcd5', name: 'King_B', type: 'Mesh', parent: Group, ...}
      ► 1: Mesh {isObject3D: true, uuid: '211db393-69ee-4272-8c44-7e8720722be5', name: 'King_W', type: 'Mesh', parent: Group, ...}
      ► 2: Mesh {isObject3D: true, uuid: 'ce0b10d1-46ce-4374-90c4-cb8dc36bfd18', name: 'Queen_B', type: 'Mesh', parent: Group, ...}
      ► 3: Mesh {isObject3D: true, uuid: 'ba4c219a-712f-4530-90c3-cf27166d09c', name: 'Queen_W', type: 'Mesh', parent: Group, ...}
      ► 4: Mesh {isObject3D: true, uuid: '6a4b3e10-c8bd-4aea-92e5-e633bc801287', name: 'Chessboard', type: 'Mesh', parent: Group, ...}
      ► 5: Mesh {isObject3D: true, uuid: '7d13df24-971e-4a17-8c2a-fa15cd3e6364', name: 'Pawn_Body_W1', type: 'Mesh', parent: Group, ...}
      ► 6: Mesh {isObject3D: true, uuid: '5123d942-60ad-4000-a796-a73a5728d23', name: 'Pawn_Body_W2', type: 'Mesh', parent: Group, ...}
      ► 7: Mesh {isObject3D: true, uuid: '09c5a297-e4f2-46f0-b79b-a6d36533600', name: 'Pawn_Body_W3', type: 'Mesh', parent: Group, ...}
      ► 8: Mesh {isObject3D: true, uuid: '628796a2-df85-43b5-b278-754e510f2153', name: 'Pawn_Body_W4', type: 'Mesh', parent: Group, ...}
      ► 9: Mesh {isObject3D: true, uuid: '7eb49ba-9c8b-403b-a991-9d74744eda75', name: 'Pawn_Body_W5', type: 'Mesh', parent: Group, ...}
      ► 10: Mesh {isObject3D: true, uuid: 'cb5b68aa-039d-479e-8274-8915b61c76d0', name: 'Pawn_Body_W6', type: 'Mesh', parent: Group, ...}
      ► 11: Mesh {isObject3D: true, uuid: '68d5ae68-f3f4-4056-97fb-f8d22148ae95', name: 'Pawn_Body_W7', type: 'Mesh', parent: Group, ...}
      ► 12: Mesh {isObject3D: true, uuid: '4290d7d1-71b3-4689-b9c0-e644f3755f34', name: 'Pawn_Body_W8', type: 'Mesh', parent: Group, ...}
      ► 13: Mesh {isObject3D: true, uuid: '281f0b36-6fff-4961-b265-d0ae4701236c', name: 'Pawn_Body_W9', type: 'Mesh', parent: Group, ...}
      ► 14: Mesh {isObject3D: true, uuid: '09efca72-e42e-414f-a8cc-2a0915e4f34c', name: 'Pawn_Body_W10', type: 'Mesh', parent: Group, ...}
      ► 15: Mesh {isObject3D: true, uuid: '836928b3-c8d0-447a-88a8-698d6f8a46c0', name: 'Pawn_Body_W11', type: 'Mesh', parent: Group, ...}
      ► 16: Mesh {isObject3D: true, uuid: 'def9709a-f886-4b03-b899-21652513df5', name: 'Pawn_Body_W12', type: 'Mesh', parent: Group, ...}
      ► 17: Mesh {isObject3D: true, uuid: 'e50d9cf3-4959-47a4-bf0b-58d77c5cf14d', name: 'Pawn_Body_W13', type: 'Mesh', parent: Group, ...}
      ► 18: Mesh {isObject3D: true, uuid: '6d1e19c4-46eb-4a87-8052-fd035b4ada9', name: 'Pawn_Body_W14', type: 'Mesh', parent: Group, ...}
      ► 19: Mesh {isObject3D: true, uuid: '4a0e6870-0e02-4e53-b228-5c5aa639c2d0', name: 'Pawn_Body_W15', type: 'Mesh', parent: Group, ...}
      ► 20: Mesh {isObject3D: true, uuid: 'c69f8c1a-248c-4dd3-ab4a-46ff9ce337e2', name: 'Pawn_Body_W16', type: 'Mesh', parent: Group, ...}
      ► 21: Mesh {isObject3D: true, uuid: '80aa3fee-0696-4251-a8fa-0acb495e040', name: 'Castle_B1', type: 'Mesh', parent: Group, ...}
      ► 22: Mesh {isObject3D: true, uuid: 'a3562d87-346c-4f5e-add2-d4ee6fd404b', name: 'Castle_B2', type: 'Mesh', parent: Group, ...}
      ► 23: Mesh {isObject3D: true, uuid: '78c920c3-9886-49e4-a051-53f7f808866f', name: 'Castle_W1', type: 'Mesh', parent: Group, ...}
      ► 24: Mesh {isObject3D: true, uuid: 'ce44ea46-0a24-43aa-9933-2800ecfa9bcd', name: 'Castle_W2', type: 'Mesh', parent: Group, ...}
      ► 25: Mesh {isObject3D: true, uuid: 'c513a26e-e677-4664-a6b6-5239b42ce507', name: 'Knight_B1', type: 'Mesh', parent: Group, ...}
      ► 26: Mesh {isObject3D: true, uuid: '04e22138-3ea9-470d-945f-c6bf79937b0e', name: 'Knight_B2', type: 'Mesh', parent: Group, ...}
      ► 27: Mesh {isObject3D: true, uuid: '3b6acd9d-8d88-4a46-8102-1abbc924c899', name: 'Knight_W1', type: 'Mesh', parent: Group, ...}
      ► 28: Mesh {isObject3D: true, uuid: 'bdae0769-14af-4eda-ace8-a7c56a2f1faa', name: 'Knight_W2', type: 'Mesh', parent: Group, ...}
      ► 29: Mesh {isObject3D: true, uuid: '62715e06-fa1b-4fdc-bb8a-ebc46e068f2f', name: 'Bishop_B1', type: 'Mesh', parent: Group, ...}
      ► 30: Mesh {isObject3D: true, uuid: 'd2366ebc-3094-4f8d-aa53-58847bcb22b4', name: 'Bishop_B2', type: 'Mesh', parent: Group, ...}
      ► 31: Mesh {isObject3D: true, uuid: '9fb340d0-4a0f-48fb-ae07-bfc0f6b000', name: 'Bishop_W1', type: 'Mesh', parent: Group, ...}
      ► 32: Mesh {isObject3D: true, uuid: '456be8f3-3912-49ec-a625-ac8c09fdc7e5', name: 'Bishop_W2', type: 'Mesh', parent: Group, ...}
      length: 33
    ▶ [[Prototype]]: Array(0)
    frustumCulled: true
    isGroup: true
```

PREMIER MODÈLE 3D (11/12)

```
gltfLoader.load(  
    './assets/models/ABeautifulGame/glTF/ABeautifulGame.gltf',  
    (gltf) => {  
        while (gltf.scene.children.length) {  
            scene.add(gltf.scene.children[0]);  
        }  
    },  
);
```

PREMIER MODÈLE 3D (12/12)

On obtient ce résultat:



DRACO (1/6)

Pour utiliser des fichiers qui ont été compressés avec Draco, on va avoir besoin d'utiliser le [DracoLoader](#)

```
import { DRACOLoader } from 'three/examples/jsm/loaders/DRACOLoader';  
  
const dracoLoader = new DRACOLoader();
```

DRACO (2/6)

On a parlé de compression, cela veut dire qu'on va avoir besoin d'un décodeur pour décompresser notre modèle 3D. Ce décodeur est disponible en JavaScript natif ou en [Web Assembly](#) et peut tourner dans un autre thread (un Worker). Ces deux fonctionnalités réunies permettent d'améliorer grandement les performances mais cela implique des parties de code complètement séparé.

DRACO (3/6)

ThreeJS a déjà prévu ces fichiers prêts à l'emploi.

Il va falloir récupérer le dossier situé dans
node_modules/three/examples/jsm/libs/draco et le
copier dans votre dossier de votre projet

```
// Spécifie où se trouve les fichiers pour décoder le DRACO
dracoLoader.setDecoderPath('./assets/libs/draco/');

// On donne le DRACOLoader au GLTFLoader
gltfLoader.setDRACOLoader(dracoLoader);
```

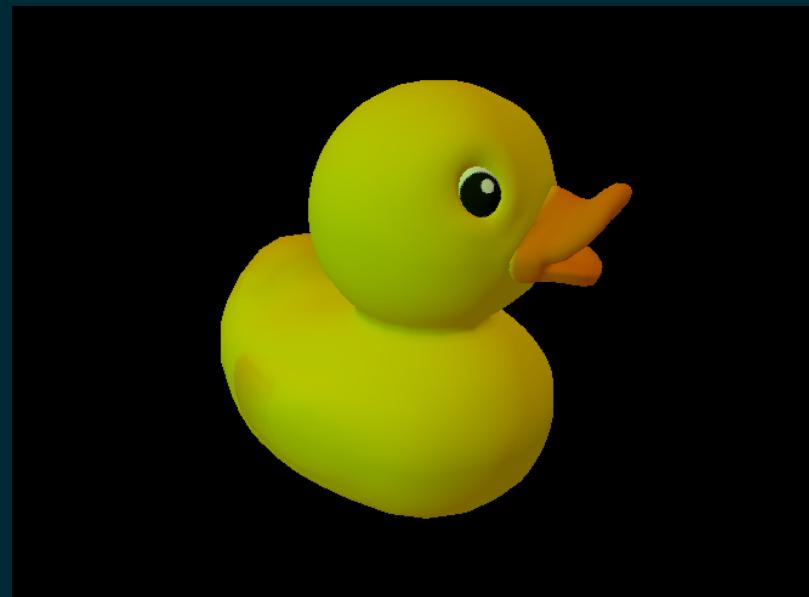
DRACO (4/6)

Une fois que cela est prêt, on peut remplacer le load par
le fichier compressé:

```
gltfLoader.load(  
    './assets/models/duck/glTF-Draco/Duck.gltf' ,
```

DRACO (5/6)

On obtient ce résultat:



DRACO (6/6)

On pourrait penser que DRACO est toujours la bonne solution pour charger des modèles 3D, mais cela n'est pas forcément vrai. Encore une fois, cela dépendra de la quantité et du poids des modèles 3D à charger. C'est pertinent avec des quantités plus importantes mais moins avec très peu de modèles.

De plus, il faut garder en tête que cela peut créer des mini freeze au début, le temps de tout setup.

ANIMATIONS (1/9)

On a parlé d'animations qui étaient supportées par le format glTF. Et bonne nouvelle ThreeJS supporte aussi cela.

On va avoir besoin d'un modèle animé comme [celui-ci](#)

ANIMATIONS (2/9)

```
gltfLoader.load(  
    './assets/models/BrainStem/glTF-Draco/BrainStem.gltf',  
    (gltf) => {  
        scene.add(gltf.scene);  
    },  
);
```

ANIMATIONS (3/9)

On obtient ce résultat:



ANIMATIONS (3/9)

Si on affiche une nouvelle fois le gltf récupéré, on observe qu'il y a une propriété `animations` qui contient un ou plusieurs `AnimationClip`

```
▼ Object ⓘ
  ► animations: [AnimationClip]
  ► asset: {generator: 'COLLADA2GLTF', version: '2.0'}
  ► cameras: []
  ► parser: GLTFParser {json: {...}, extensions: {...}, plugin: ...}
  ► scene: Group {isObject3D: true, uuid: 'f4d9552e-7398-4...'}
  ► scenes: [Group]
  ► userData: {}
  ► [[Prototype]]: Object
```

ANIMATIONS (4/9)

Pour jouer les animations, il va nous falloir un player et c'est le rôle du **AnimationMixer** que l'on crée pour chaque modèle dont on souhaite jouer les animations.

```
let mixer = null;
gltfLoader.load(
  './assets/models/BrainStem/glTF-Draco/BrainStem.gltf',
  (gltf) => {
    // ...

    mixer = new THREE.AnimationMixer(gltf.scene);
  },
);
```

ANIMATIONS (5/9)

Une fois le mixer de prêt, on va pourvoir ajouter les différents clips et récupérer les actions qui en découlent

```
gltfLoader.load(  
  './assets/models/BrainStem/glTF-Draco/BrainStem.gltf',  
  (gltf) => {  
    // ...  
  
    // Notre modèle n'a qu'une seule animation  
    const action = mixer.clipAction(gltf.animations[0]);  
  },  
);
```

ANIMATIONS (6/9)

On récupère une `AnimationAction` que l'on va pouvoir jouer quand on veut

```
gltfLoader.load(  
  './assets/models/BrainStem/glTF-Draco/BrainStem.gltf',  
  (gltf) => {  
    // ...  
  
    action.play();  
  },  
);
```

ANIMATIONS (7/9)

Malheureusement, malgré tout ça, on a toujours notre
objet qui est très statique



ANIMATIONS (8/9)

De la même manière que pour les autres updates, il va falloir dire à notre mixer de se mettre à jour en temps réel. Quoi de mieux qu'utiliser notre fonction tick dédié pour ça

```
const tick = () => {
    const deltaTime = clock.getDelta();

    // On met à jour le mixer
    if (mixer !== null) {
        mixer.update(deltaTime);
    }

    // ...
}
```

ANIMATIONS (9/9)

On obtient ce résultat:



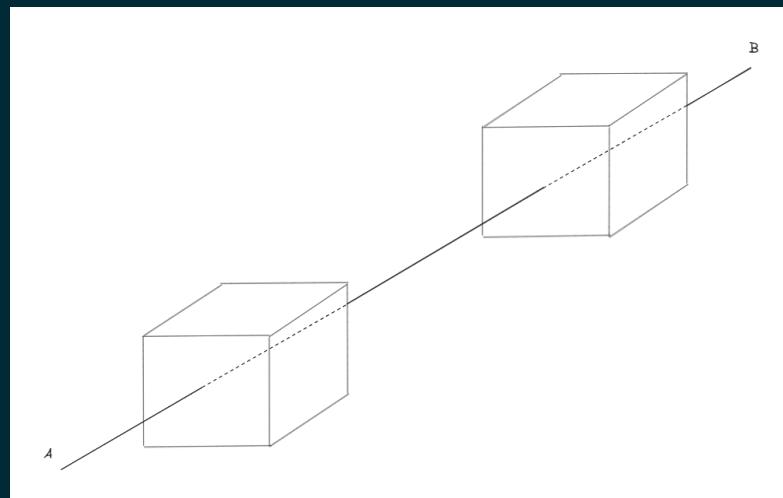
INTERACTIONS

INTRODUCTION

On va maintenant voir comment on peut interagir avec les objets de notre scène notamment avec la souris

LANCER DE RAYONS (1/9)

La technique du lancer de rayons permet de, comme son nom l'indique, de lancer un rayon dans une direction et de détecter les objets que ce rayon traverse.



Cette technique peut être utile pour détecter des collisions, tester les objets sous la souris, etc.

LANCER DE RAYONS (2/9)

ThreeJS met à disposition ce qu'on appelle un **Raycaster** pour nous aider à faire nos lancers de rayon

```
const raycaster = new THREE.Raycaster();
```

LANCER DE RAYONS (3/9)

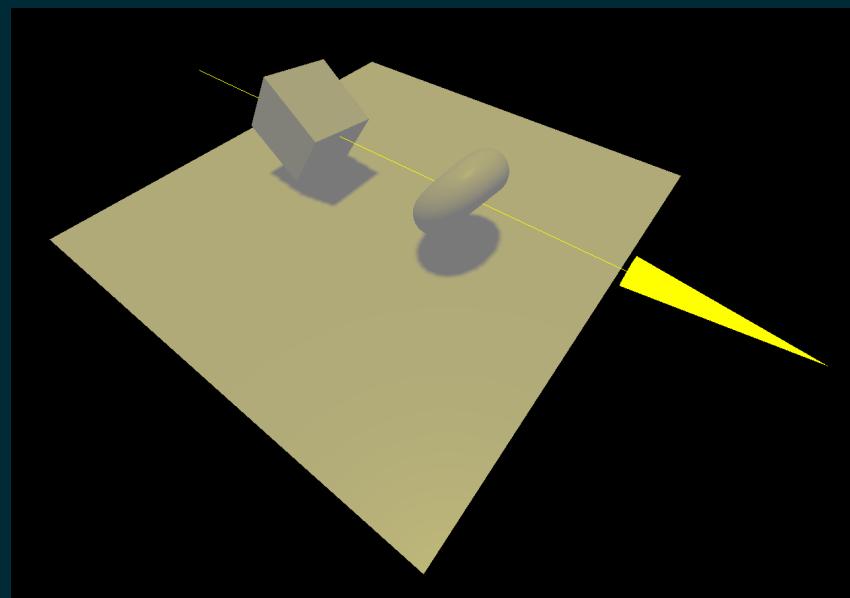
Un rayon a une origine (de là où il part) et une direction.
Ces 2 valeurs seront des `Vector3`. La seule chose à savoir est qu'il faut que la direction soit normalisée (`.normalize()`).

```
const rayOrigin = new THREE.Vector3(-3, 0, 0);
const rayDirection = new THREE.Vector3(7, 0, 0);
const rayDirectionNormalized = rayDirection.clone().normalize();

const arrowHelper = new THREE.ArrowHelper(
    rayDirectionNormalized,
    rayOrigin,
    rayDirection.length(),
    0xffff00
);
scene.add(arrowHelper);
```

LANCER DE RAYONS (4/9)

On obtient ce résultat:



LANCER DE RAYONS (5/9)

On peut setup notre raycaster et lui demander de lancer un rayon et de nous retourner tout ce qu'il a traversé:

```
raycaster.set(rayOrigin, rayDirectionNormalized);
const intersected = raycaster.intersectObjects([cube, torus]);
```

LANCER DE RAYONS (6/9)

Le résultat est un tableau d'intersections avec les propriétés suivantes:

- **distance**: la distance entre l'origine du rayon et le point de collision
- **face**: la face traversée
- **faceIndex**: l'index de cette face
- **object**: l'objet traversé
- **point**: un Vector3 de la position de la collision
- **uv**: la coordonnée UV dans cette géométrie

LANCER DE RAYONS (7/9)

Toutes ses données peuvent être utiles pour détecter des collisions entre le joueur et le décor, notamment avec la distance, changer les aspects des matériaux des objets touchés ou encore faire apparaître des choses aux impacts grâce au point.

LANCER DE RAYONS (8/9)

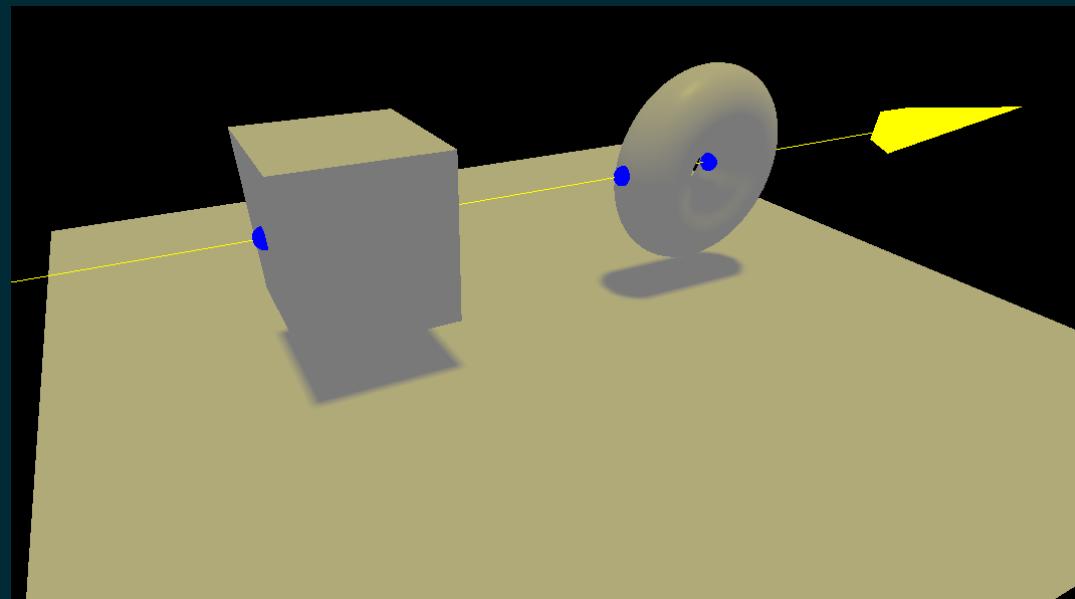
On peut faire le lancer de rayons en temps réel

```
const intersectObjects = () => {
    raycaster.set(rayOrigin, rayDirectionNormalized);
    const objects = raycaster.intersectObjects([cube, torus]);

    // Ce sont des petites sphères pour symboliser les collisions
    for (let i = 0; i < debugsCount; i += 1) {
        if (i >= objects.length) {
            debugPoints[i].visible = false;
        } else {
            debugPoints[i].visible = true;
            debugPoints[i].position.fromArray(objects[i].point.toArray());
        }
    }
}
```

LANCER DE RAYONS (9/9)

On obtient ce résultat:



LA SOURIS (1/5)

On peut utiliser le système de raycast avec la souris pour détecter les objets qui sont sous la souris.

LA SOURIS (2/5)

On va réutiliser ce que l'on avait fait pour le contrôle de caméra pour avoir la position de la souris

```
const mouse = new THREE.Vector2();
canvas.addEventListener('mousemove', (evt) => {
  mouse.x = evt.clientX / sizes.width * 2 - 1;
  mouse.y = -(evt.clientY / sizes.height) * 2 + 1;
});
```

LA SOURIS (3/5)

On va changer de méthode pour set notre raycaster

```
raycaster.setFromCamera(mouse, camera);
```

LA SOURIS (3/5)

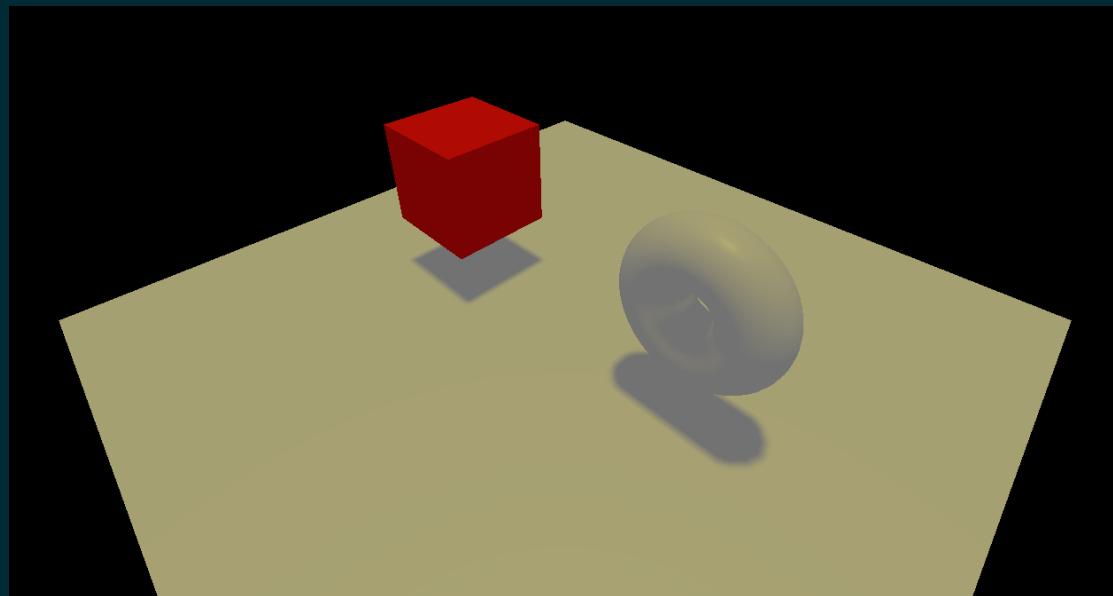
```
const objectsToIntersectWith = [cube, torus];
const intersectObjects = () => {
  const intersected = raycaster.intersectObjects(objectsToIntersectWith);
  for (const obj of objectsToIntersectWith) {
    obj.material.color.set(
      // On regarde si l'objet est traversé par le rayon
      intersected.find((intersect) => intersect.object === obj)
        ? '#ff0f0f'
        : '#f0f0f0'
    );
  }
}

const intersectObjectsFromCamera = (mouse, camera) => {
  raycaster.setFromCamera(mouse, camera);
  intersectObjects();
}

const tick = () => {
  intersectObjectsFromCamera(mouse, camera);
  // ...
}
```

LA SOURIS (4/5)

On obtient ce résultat:



LA SOURIS (5/5)

Pour des usages plus avancés notamment mouseenter, mouseleave et mouseclick, il va falloir faire les comportements soi-même. A savoir qu'il faudrait maintenir une liste des objets courants et regarder si c'est la première fois, alors c'est équivalent à un mouseenter. Si l'objet disparait de la liste alors c'est un mouseleave

DEBUG & PERFORMANCE

DEBUG UI (1/12)

Un aspect essentiel de projets créatifs comme ceux en WebGL est la capacité à modifier n'importe quelle valeur en temps réel sans devoir à chaque fois aller dans le code. C'est pratique à la fois pour les développeurs mais aussi pour les autres acteurs du projet (designers, clients, etc.) qui veulent pouvoir changer n'importe quelle valeur pour avoir une expérience qui leur convient parfaitement.

DEBUG UI (2/12)

Pour faire ça on va avoir besoin d'une petite librairie qui permet d'afficher des contrôles dans l'interface. Il en existe pleins avec notamment:

- dat.gui
- lil-gui
- etc.

DEBUG UI (3/12)

On va garder `lil-gui` car la librairie est populaire,
maintenue et simple à utiliser.

DEBUG UI (3/12)

On va commencer par installer la librairie:

```
npm install lil-gui
```

Puis l'importer dans notre code

```
import GUI from 'lil-gui';
```

DEBUG UI (4/12)

On peut maintenant instancier notre panneau de debug d'UI

```
const gui = new GUI();
```

DEBUG UI (5/12)

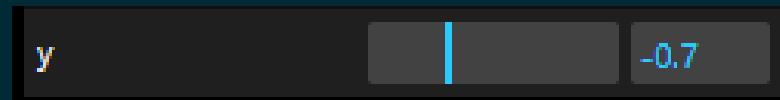
Cette librairie permet de contrôler plusieurs types de valeurs:

- Nombre
- Couleur
- Texte
- Case à cocher
- Sélecteur
- Bouton

DEBUG UI (6/12)

NOMBRE

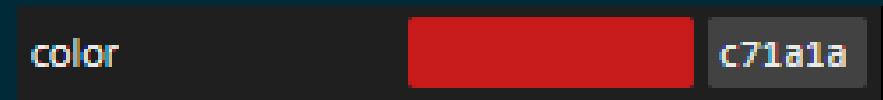
```
gui.add(cube.position, 'y', -2, 2, 0.1);
```



DEBUG UI (7/12)

COULEUR

```
gui.addColor(cube.material, 'color');
```



DEBUG UI (8/12)

CASE À COCHER

```
gui.add(cube.material, 'wireframe');
```



DEBUG UI (9/12)

SÉLECTEUR

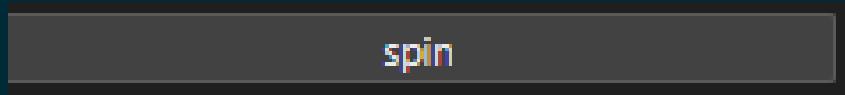
```
gui.add(  
  cube.material,  
  'side',  
  { Front: THREE.FrontSide, Back: THREE.BackSide, Double: THREE.DoubleSide }  
);
```



DEBUG UI (10/12)

BOUTON

```
const myObject = {
  spin: () => {
    anime({ targets: cube.rotation, y: cube.rotation.y + Math.PI * 2 });
  },
};
gui.add(myObject, 'spin');
```



DEBUG UI (11/12)

DOSSIER

```
const folder = gui.addFolder('Custom Folder');
folder.add(cube.position, 'x', -2, 2, 0.01);
```



DEBUG UI (12/12)

Il existe pleins d'autres configurations possibles et pleins de méthodes. Vous pouvez retrouver la documentation complète [ici](#).

PERFORMANCE (1/28)

Comme dit auparavant dans le cours, on cherche à atteindre 60 fps minimum pour avoir une expérience fluide et qualitative.

Les limitations de performance peuvent venir de soit du CPU soit du GPU.

Il est donc très important de toujours avoir en tête la performance sur différents appareils voire même mobile, si le site y est aussi destiné.

PERFORMANCE (2/28)

CONTRÔLE DU FPS (1/2)

On va utiliser une petite librairie qui va nous permettre de suivre à minima les FPS de notre expérience

```
npm install stats.js
```

PERFORMANCE (3/28)

Contrôle du FPS (2/2)

```
import Stats from 'stats.js';

// On ajoute dans le dom
const stats = new Stats();
stats.showPanel(0);
document.body.appendChild(stats.dom);

// On ajoute le begin et end tout autour des traitements fait à chaque tick
const tick = () => {
    stats.begin();

    // ...

    stats.end();
};
```

PERFORMANCE (4/28)

CONTRÔLE DES APPELS DE DESSIN (1/2)

Les appels de dessin sont les actions déclenchées sur le GPU. Il y a de nombreux appels de dessin à chaque frame.

Ce qu'il faut retenir c'est que moins il y a d'appels de dessin, mieux c'est pour les performances.

PERFORMANCE (5/28)

CONTRÔLE DES APPELS DE DESSIN (2/2)

On va utiliser une extension (sur Chrome): [Spector.js](#) pour nous aider à contrôler ce qu'il se passe.

PERFORMANCE (6/28)

AFFICHER LES INFOS DE RENDU

```
console.log(renderer.info);
```

PERFORMANCE (7/28)

CODE PERFORMANT

Cela va sembler logique, mais là tout gain de performance peut s'avérer très important. Donc il faudra toujours garder à l'esprit la performance lorsqu'on écrira du code.

PERFORMANCE (8/28)

GESTION MÉMOIRE

Il faut aussi avoir en tête la mémoire et donc dès qu'une ressource n'est plus utilisée, il faudra nettoyer la mémoire afin d'éviter des fuites mémoire. ThreeJS a une page dédiée pour expliquer comment nettoyer proprement les éléments en fonction des besoin

```
scene.remove(cube);
cube.geometry.dispose();
cube.material.dispose();
```

PERFORMANCE (8/28)

LUMIÈRES (1/2)

Dès que c'est possible, évitez d'utiliser les lumières de ThreeJS ou uniquement les moins coûteuses. Les calculs sont coûteux. Privilégiez d'autres matériaux qui ne nécessitent pas de lumières (des matcap, des textures déjà bakes)

PERFORMANCE (9/28)

LUMIÈRES (2/2)

Si vous utilisez des lumières, évitez absolument de les ajouter ou de les supprimer en temps réel. Cela provoquera une recompilation des matériaux qui les utilisent et donc potentiellement un freeze.

PERFORMANCE (10/28)

OMBRES (1/4)

De la même manière que pour les lumières, dès que c'est possible, évitez d'utiliser les ombres pour maximiser les performances.

PERFORMANCE (11/28)

OMBRES (2/4)

Si vous utilisez les ombres, optimisez les textures des ombres.

- Réduisez les zones des caméras des ombres
- Réduisez au maximum la résolution des textures des ombres

PERFORMANCE (12/28)

OMBRES (3/4)

On a vu qu'on pouvait utiliser les propriétés `castShadow` et `receiveShadow`. Utilisez les sur le minimum d'objets possibles. Ne l'activez pas partout et essayez de trouver des alternatives.

PERFORMANCE (13/28)

OMBRES (4/4)

Les textures des ombres sont automatiquement mises à jour à chaque rendu. Si vous en avez la possibilité, désactivez ça et faites les mises à jour vous-même quand c'est nécessaire.

```
renderer.shadowMap.autoUpdate = false;  
renderer.shadowMap.needsUpdate = true;
```

PERFORMANCE (14/28)

TEXTURES (1/3)

Les textures prennent beaucoup de mémoire sur le GPU et encore pire avec les mipmaps quand elles sont activées.

Essayez de toujours réduire vos textures aux plus petites résolutions possibles.

PERFORMANCE (15/28)

TEXTURES (2/3)

Faites toujours des textures en puissance de 2. C'est très important pour la génération des mipmaps.

PERFORMANCE (16/28)

TEXTURES (3/3)

Utilisez le bon format pour les textures .jpg ou .png pour utiliser les bonnes compressions ou même ne pas avoir le canal alpha quand ce n'est pas utile.

Vous pouvez utiliser [tinify](#)

PERFORMANCE (17/28)

GÉOMÉTRIES (1/3)

Evitez de mettre à jour en temps réel les vertices sur le CPU. Vous pouvez le faire une fois à la création mais après évitez de le faire à chaque tick.

Si vous avez besoin de mettre à jour les vertices, passez par un vertex shader.

PERFORMANCE (18/28)

GÉOMÉTRIES (2/3)

Si vous avez plusieurs mesh qui utilisent la même géométrie. Mutualisez la géométrie plutôt que d'en créer autant qu'il y a de meshes.

PERFORMANCE (19/28)

GÉOMÉTRIES (3/3)

Si les géométries n'ont pas vocation à bouger, vous pouvez même les fusionner grâce à [BufferGeometryUtils](#)

```
const mergedGeometries = BufferGeometryUtils.mergeBufferGeometries([  
    geometry1,  
    geometry2  
]);
```

PERFORMANCE (20/28)

MATÉRIAUX (1/2)

Si vous avez plusieurs meshes qui utilisent le même type de matériaux, alors mutualisez-les et créez-les qu'une seule fois

```
const myMaterial = new THREE.MeshLambertMaterial();
const myGeometry = new THREE.BoxGeometry();

for (let i = 0; i < 100; i += 1) {
  scene.add(
    new THREE.Mesh(myGeometry, myMaterial)
  );
}
```

PERFORMANCE (21/28)

MATÉRIAUX (2/2)

Comme vu précédemment, certains matériaux nécessitent plus de ressources que d'autres.

Privilégiez des matériaux plus légers comme `MeshBasicMaterial`, `MeshLambertMaterial` ou encore `MeshPhongMaterial`.

Et évitez les `MeshStandardMaterial` ou `MeshPhysicalMaterial`

PERFORMANCE (22/28)

MODÈLES 3D (1/2)

Utilisez des modèles low-poly. Moins vous avez de polygones, mieux sera l'expérience en terme de fluidité.

Vous pouvez palier ce manque de détails via les normal maps qui sont très efficaces en termes de performance.

PERFORMANCE (23/28)

MODÈLES 3D (2/2)

Si vous avez des modèles avec beaucoup de détails avec des géométries complexes, utilisez la compression DRACO. Cela réduira considérablement le poids de vos modèles 3D.

Les inconvénients seront à la décompression de vos modèles au chargement de votre expérience qui pourront potentiellement créer des freeze.

PERFORMANCE (24/28)

CAMÉRAS (1/2)

Essayez d'adapter au mieux votre champs de vision pour ne rendre que ce qui est strictement nécessaire.

PERFORMANCE (25/28)

CAMÉRAS (2/2)

De la même manière que le champs de vision, essayez de réduire le near et le far de vos caméras. Avec des décors complexes et loins, pas besoin de rendre tout cela et donc trouvez les meilleures valeurs qui vous conviennent pour optimiser les rendus.

PERFORMANCE (26/28)

MOTEUR DE RENDU (1/2)

Comme on a vu, certains appareils peuvent avoir des gros ratio de pixels. Essayez de limiter vos ratio de pixel pour ne pas trop dégrader les performances

```
myRenderer.setPixelRatio(Math.min(window.devicePixelRatio), 2);
```

PERFORMANCE (27/28)

MOTEUR DE RENDU (2/2)

L'antialias par défaut est performant. Mais comme on est toujours à la recherche de la moindre optimisation, utilisez-le que si vous commencez à percevoir de l'aliasing.

```
const myRenderer = new THREE.WebGLRenderer({ antialias: true });
```

PERFORMANCE (28/28)

Vous pouvez trouver d'autres conseils pour améliorer vos expriences WebGL: [ici](#)

PARTICULES

INTRODUCTION

Les particules ont pleins d'usage comme réaliser des étoiles, de la fumée, de la pluie, du feu, etc.

Vous pouvez en avoir des milliers avec de bonnes performances.

Chaque particule est composée d'un plan qui regarde toujours dans la direction de la caméra.

INTRODUCTION

Créer des particules, cela revient à créer un Mesh avec:

- Une géométrie: `BufferGeometry`
- Un matériau: `PointsMaterial`
- Une instance de l'objet: `Points`

PARTICULES À PARTIR D'UNE GÉOMÉTRIE EXISTANTE (1/4)

Chaque vertex d'une géométrie va devenir une particule

```
const particlesGeometry = new THREE.TorusGeometry(1, 0.2, 32, 64);
```

PARTICULES À PARTIR D'UNE GÉOMÉTRIE EXISTANTE (2/4)

On va créer notre PointsMaterial

```
const particlesMaterial = new THREE.PointsMaterial({  
    // Taille d'une particule  
    size: 0.03,  
    // Pour garder une notion de profondeur avec les particules  
    // Et d'avoir des plus grosses plus elles sont proches de la caméra  
    // Et inversement plus petites plus elles en sont loin  
    sizeAttenuation: true,  
});
```

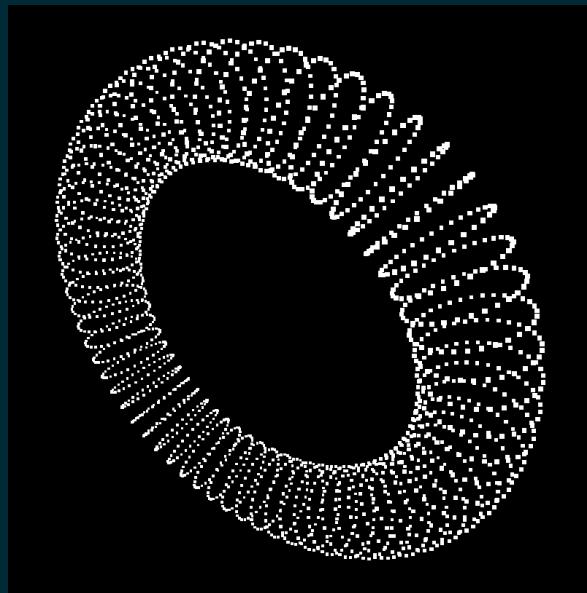
PARTICULES À PARTIR D'UNE GÉOMÉTRIE EXISTANTE (3/4)

On va instancier nos particules avec un [Points](#)

```
const particles = new THREE.Points(  
    particlesGeometry,  
    particlesMaterial  
);  
scene.add(particles);
```

PARTICULES À PARTIR D'UNE GÉOMÉTRIE EXISTANTE (4/4)

On obtient ce résultat avec chaque vertex qui est une particule:



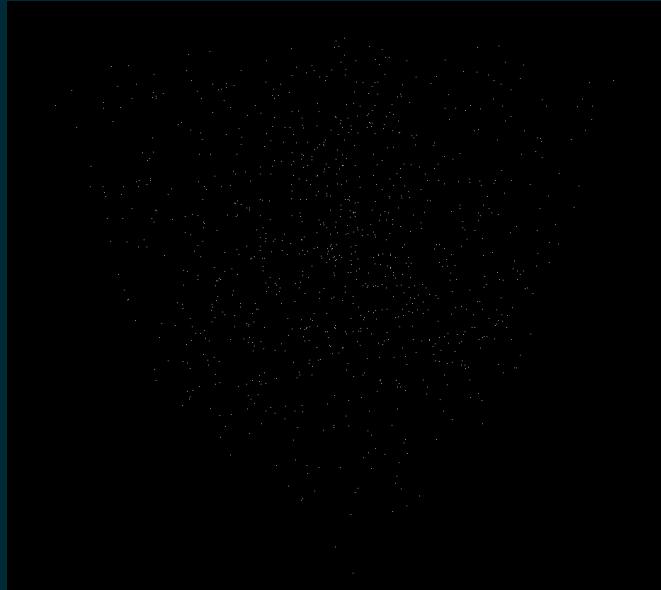
PARTICULES À PARTIR D'UNE GÉOMÉTRIE PERSONNALISÉE (1/2)

On va créer des particules avec une géométrie personnalisée:

```
const particlesGeometry = new THREE.BufferGeometry();
const count = 1000;
const positions = new Float32Array(count * 3);
for (let i = 0; i < count * 3; i += 1) {
  positions[i] = (Math.random() - 0.5) * 10;
}
particlesGeometry.setAttribute(
  'position',
  new THREE.BufferAttribute(positions, 3)
);
```

PARTICULES À PARTIR D'UNE GÉOMÉTRIE PERSONNALISÉE (2/2)

On obtient ce résultat:



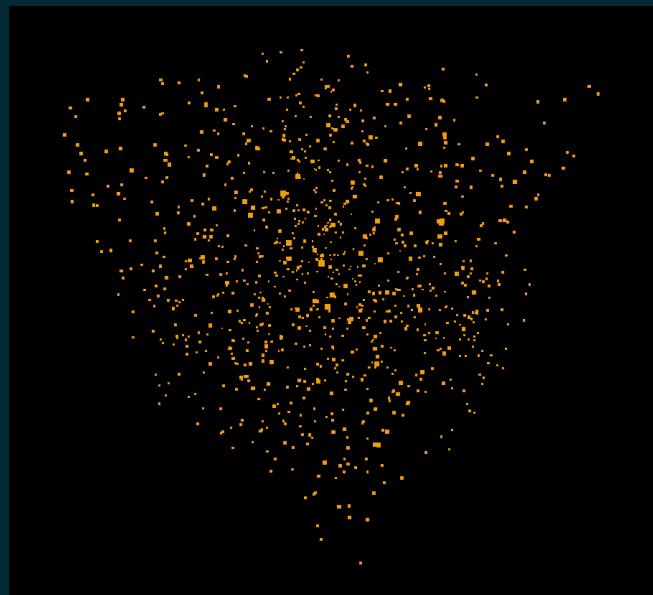
PERSONNALISER LES PARTICULES (1/10)

On peut changer la couleur de toutes les particules via
le material

```
particlesMaterial.color = new THREE.Color('#ff88cc');
```

PERSONNALISER LES PARTICULES (2/10)

On obtient ce résultat:



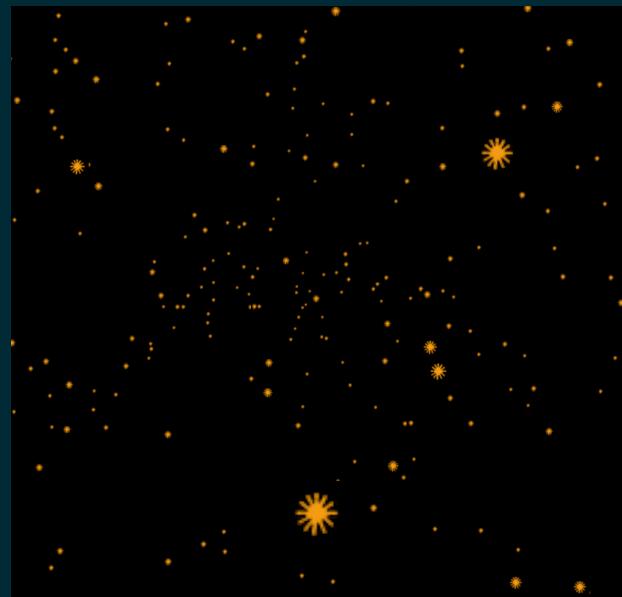
PERSONNALISER LES PARTICULES (3/10)

On peut appliquer une texture sur les particules

```
const texture = textureLoader.load('../assets/textures/snowflake1.png');  
particlesMaterial.map = texture;
```

PERSONNALISER LES PARTICULES (4/10)

On obtient ce résultat:



PERSONNALISER LES PARTICULES (5/10)

On peut trouver un pack de textures de particules: [ici](#)

PERSONNALISER LES PARTICULES (6/10)

On observe que les particules se cachent les unes des autres. On va devoir activer la transparence sur nos particules

```
const texture = textureLoader.load('./../assets/textures/snowflake1.png');
particlesMaterial.alphaMap = texture;
particlesMaterial.transparent = true;
```

PERSONNALISER LES PARTICULES (7/10)

Malgré ça, on observe que certaines particules cachent encore d'autres. Cela est dû au fait que les particules sont dessinées dans l'ordre de leur création et WebGL ne sait pas réellement quelle particule est devant une autre.

PERSONNALISER LES PARTICULES (8/10)

On peut fixer le problème en changeant l'alphaTest du matériau, qui correspond à la valeur à laquelle le WebGL rendra ou non le pixel

```
particlesMaterial.alphaTest = 0.001;
```

PERSONNALISER LES PARTICULES (9/10)

On peut fixer le problème en désactivant le depth testing (qui correspond à tester ce qui a déjà été dessiné pour rendre un nouveau pixel ou non)

```
particlesMaterial.depthTest = false
```

Note: Cela peut créer des bugs avec les autres objets de la scène

PERSONNALISER LES PARTICULES (10/10)

On peut fixer le problème en désactivant le depth writing (qui correspond à ne pas écrire et donc les autres objets n'en tiendront pas compte)

```
particlesMaterial.depthWrite = false
```

Note: Cela peut créer des bugs avec les autres objets de la scène

BLENDING

Le WebGL dessine les pixels des particules par dessus les uns des autres. Avec la propriété blending on peut dire au WebGL comment mélanger les couleurs des pixels

```
particlesMaterial.depthWrite = false  
particlesMaterial.blending = THREE.AdditiveBlending
```

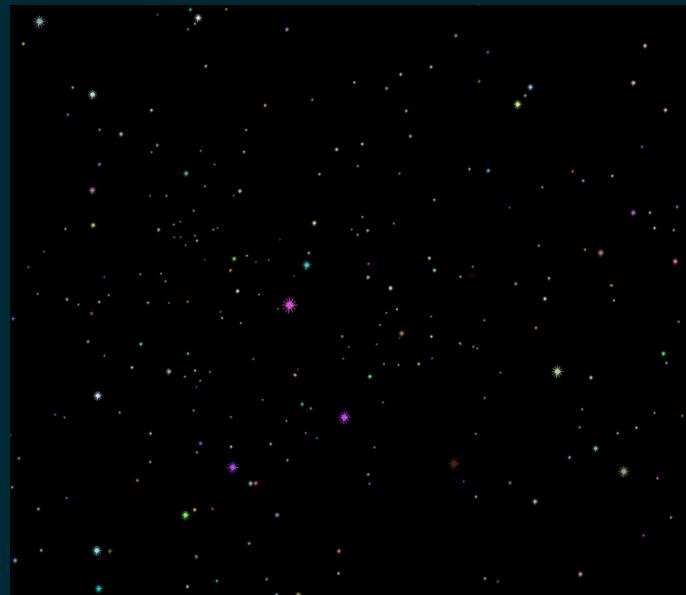
COULEUR DISTINCTE DES PARTICULES (1/2)

On a vu pour appliquer une même couleur sur toutes les particules. On peut faire en sorte d'appliquer des couleurs par particule

```
const colors = new Float32Array(count * 3);
for (let i = 0; i < count * 3; i += 1) {
  colors[i] = Math.random();
}
particlesGeometry.setAttribute('color', new THREE.BufferAttribute(colors, 3));
particlesMaterial.vertexColors = true;
```

COULEUR DISTINCTE DES PARTICULES (2/2)

On obtient ce résultat:



ANIMER LES PARTICULES

```
const positionsArray = particlesGeometry.attributes.position.array;
for (let i = 0; i < particlesCount; i += 1) {
  const indexForVertex = i * 3;
  // On bouge sur y
  positionsArray[indexForVertex + 1] = Math.sin(
    elapsedTime + positionsArray[indexForVertex]
  );
}
// Important pour dire qu'il faut upload le buffer de nouveau sur le GPU
particlesGeometry.attributes.position.needsUpdate = true;
```

POST-TRAITEMENT

INTRODUCTION (1/2)

Le post traitement consiste à ajouter des effets sur l'image finale. Cette technique est souvent utilisée pour les films mais on peut réaliser le même genre d'effets en WebGL.

Cela peut permettre d'améliorer la qualité d'image ou même de créer des effets.

INTRODUCTION (2/2)

Voici une liste non exhaustive de type d'effets que l'on peut réaliser:

- Bloom
- Depth of field
- Motion blur
- Glitch
- Outlines
- Antialiasing
- etc.

COMMENT CELA FONCTIONNE (1/4)

Jusqu'à présent on a toujours rendu dans un canvas,
mais cette fois on va rendre dans une texture qui sera
utilisée plus tard dans le cycle du rendu

COMMENT CELA FONCTIONNE (2/4)

On utilise cette texture sur un plan qui regarde la caméra et qui couvre l'entièreté de la vue.

Et ensuite on utilise un fragment shader qui va appliquer les effets de post-traitements sur la texture.

Dans ThreeJS, on parle de passage pour parler d'effets

COMMENT CELA FONCTIONNE (3/4)

On peut avoir plusieurs passages pendant le post-traitement. Et à cause de cela, le système de post-traitement va avoir besoin de 2 textures.

La raison est qu'on ne peut pas rendre dans une texture en même temps qu'on l'a lit pour rendre l'effet. Donc on va dessiner toujours dans l'une pendant qu'on lit dans l'autre et vice versa à la passe d'après.

C'est ce qu'on appelle le ping pong buffering.

COMMENT CELA FONCTIONNE (4/4)

Le dernier passage ne fera pas de rendu dans une texture mais bel et bien dans le canvas comme avant pour voir le résultat final à l'écran.

EFFECTCOMPOSER (1/3)

La technique semble assez complexe de devoir le faire à la main, mais ThreeJS a rajouté un système qui gère tout le mécanisme pour nous: **EffectComposer**

EFFECTCOMPOSER (2/3)

Pour mettre en place le minimum:

```
import {
  EffectComposer
} from 'three/examples/jsm/postprocessing/EffectComposer';

const composer = new EffectComposer(renderer);
composer.setSize(sizes.width, sizes.height);
composer.setPixelRatio(Math.min(window.devicePixelRatio, 2));

const renderPass = new RenderPass(scene, camera);
composer.addPass(renderPass);

const tick = () => {
  // ...

  // renderer.render(scene, camera)
  composer.render();

  // ...
}
```

EFFECTCOMPOSER (3/3)

Avec ça, on devrait voir aucun changement visuel à proprement parlé, mais ça y est le système de post traitement est en place et on va pouvoir tester différents effets à travers l'EffectComposer. Vous trouverez d'ailleurs dans la [documentation](#) une liste des effets disponibles dans ThreeJS

On va en tester certains pour voir comment cela fonctionne dans la pratique

DOT EFFECT (1/2)

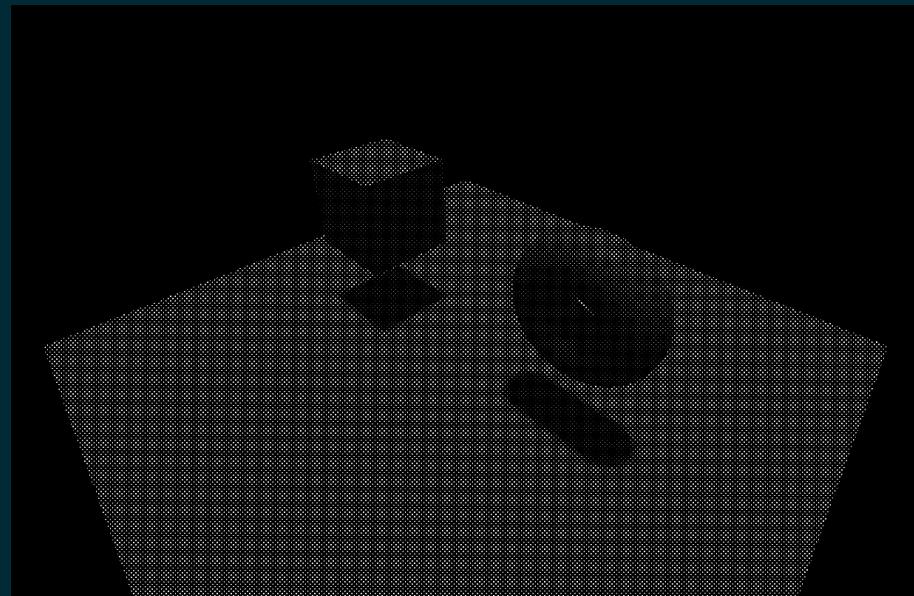
On va rajouter un post-traitement qui fait le rendu avec un effet point

```
import {  
  DotScreenPass  
} from 'three/examples/jsm/postprocessing/DotScreenPass';  
  
// Après la render pass  
  
const dotScreenPass = new DotScreenPass();  
composer.addPass(dotScreenPass);
```

Note: On peut désactiver chacun des passages avec la propriété `enabled` sur chaque effet

DOT EFFECT (2/2)

On obtient ce résultat:



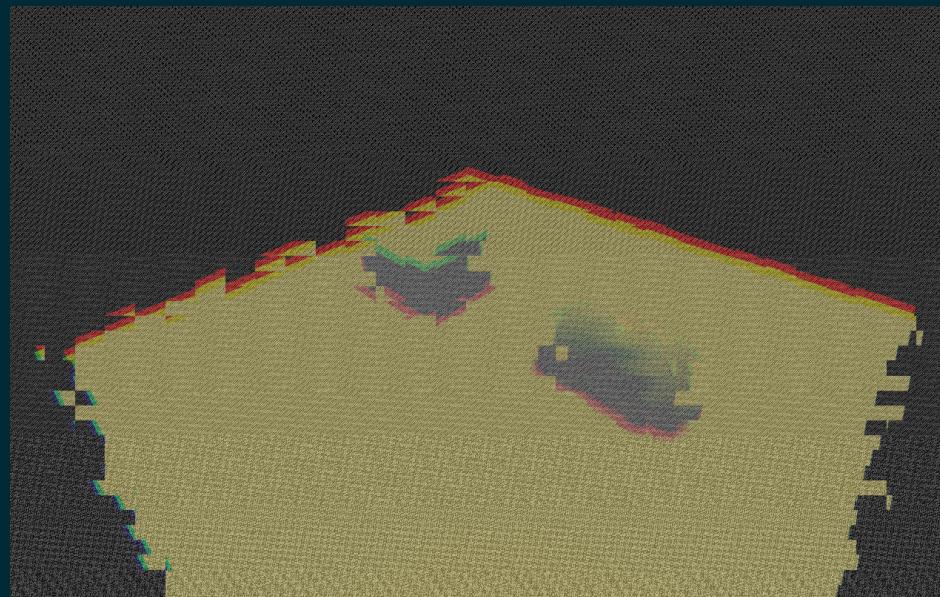
GLITCH EFFECT (1/2)

On va rajouter un post-traitement qui rajoute des
glitches sur l'image

```
import {  
  GlitchPass  
} from 'three/examples/jsm/postprocessing/GlitchPass';  
  
// Après la render pass  
  
const glitchPass = new GlitchPass();  
composer.addPass(glitchPass);
```

GLITCH EFFECT (2/2)

On obtient ce résultat:



RGBSHIFT EFFECT (1/2)

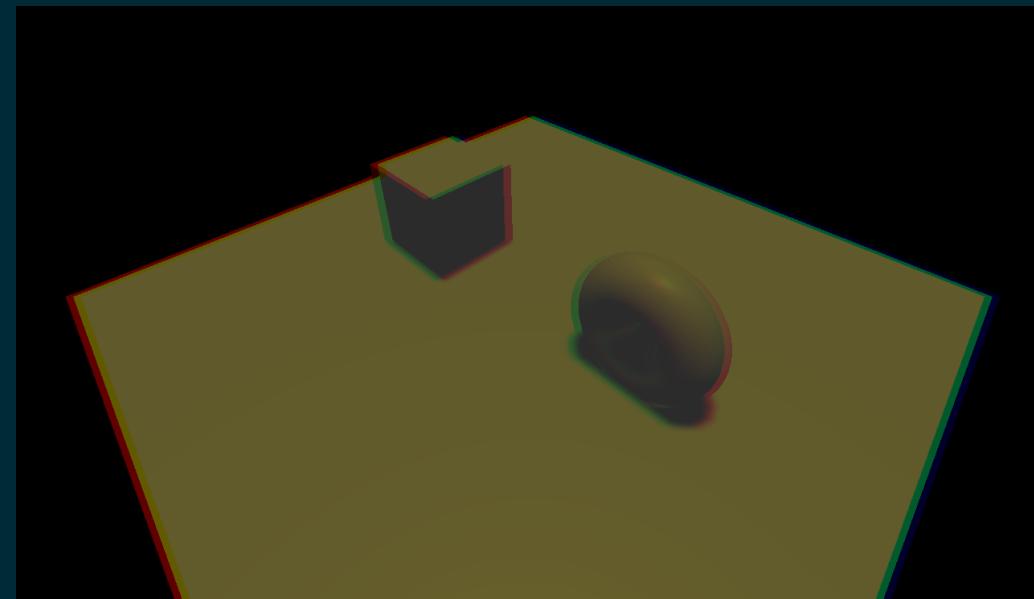
Si on veut faire un passage de RGBShift, il n'existe pas de classe à proprement parlé mais un shader, donc on va devoir utiliser l'effet qui permet de donner un shader directement

```
import {
  ShaderPass
} from 'three/examples/jsm/postprocessing/ShaderPass';
import {
  RGBShiftShader
} from 'three/examples/jsm/shaders/RGBShiftShader';

const rgbShiftPass = new ShaderPass(RGBShiftShader);
composer.addPass(rgbShiftPass);
```

RGBSHIFT EFFECT (2/2)

On obtient ce résultat:



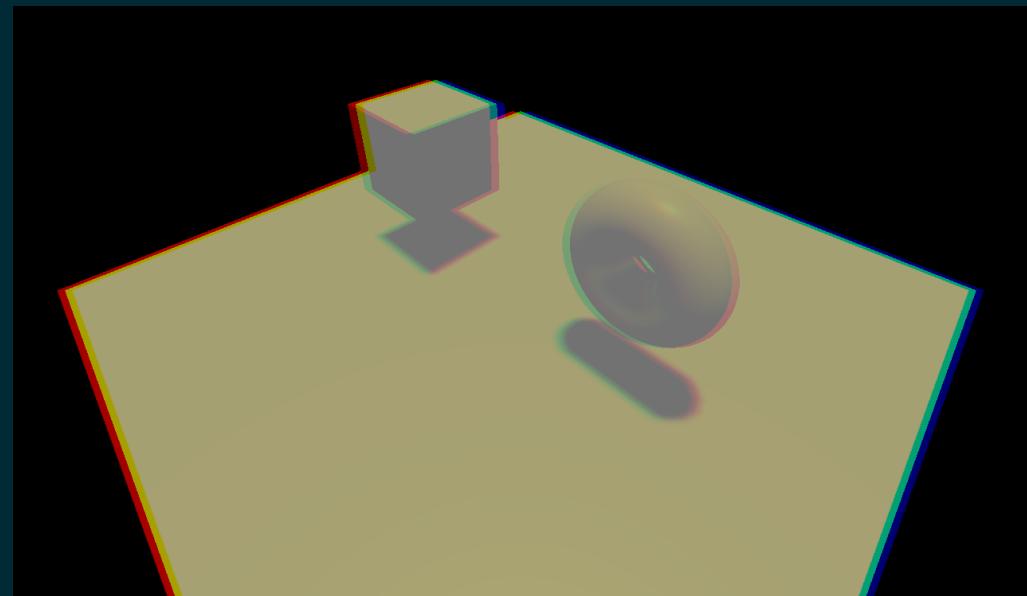
CORRIGER LES COULEURS (1/2)

On observe que les couleurs ont changé et sont plus sombres. Cela est dû au fait que les effets de passage ne supportent pas les espaces de couleur de la même manière. On va devoir corriger ça avec un passage de correction gamma

```
import {  
    GammaCorrectionShader  
} from 'three/examples/jsm/shaders/RGBShiftShader';  
  
const gammaCorrectionPass = new ShaderPass(GammaCorrectionShader);  
composer.addPass(gammaCorrectionPass);
```

CORRIGER LES COULEURS (2/2)

On obtient ce résultat:

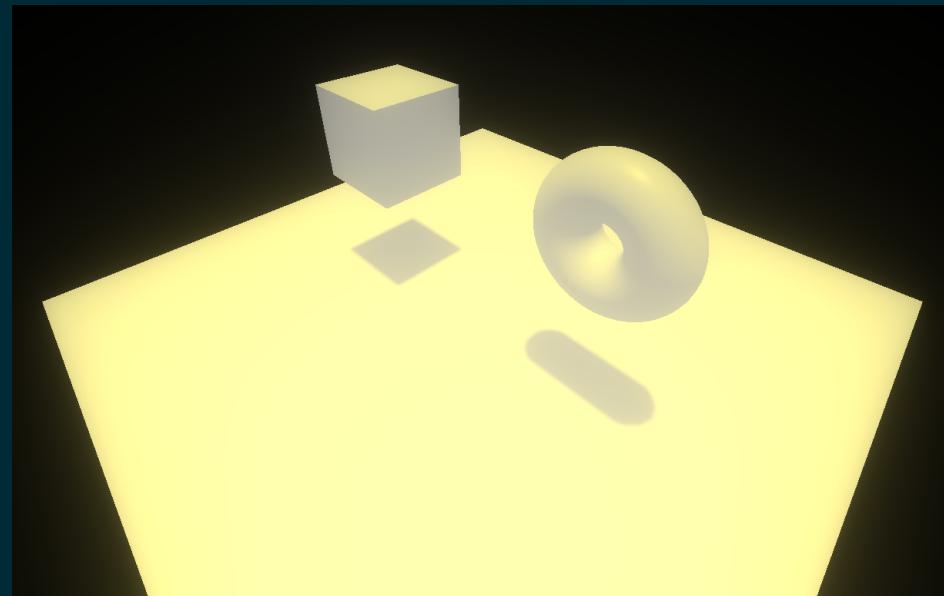


BLOOM EFFECT (1/2)

```
import {  
    UnrealBloomPass  
} from 'three/examples/jsm/postprocessing/UnrealBloomPass';  
  
const unrealBloomPass = new UnrealBloomPass();  
composer.addPass(unrealBloomPass);  
unrealBloomPass.strength = 0.5;  
unrealBloomPass.radius = 1;
```

BLOOM EFFECT (2/2)

On obtient ce résultat:



EXTRA

Comme on a vu, on peut faire des `ShaderPass` pour appliquer n'importe quel fragment shader en tant que pass. On en reparlera dans le chapitre des shaders

PHYSIQUES

INTRODUCTION (1/2)

La physique permet d'ajouter du réalisme et un effet wahou aux expériences WebGL. On pourrait le faire via des mathématiques et des solutions avec le Raycaster que l'on a vu précédemment, mais si on souhaite aller plus loin avec de la friction, des rebonds, des contraintes, etc. il existe des librairies qui nous permettent de faire ça.

INTRODUCTION (2/2)

L'idée est de reproduire un monde physique théorique (rien de visuel concrètement). A chaque fois que l'on créera un objet dans la scène sur lequel on veut de la physique, on créera son équivalent dans le monde physique.

Puis à chaque frame, avant de faire le rendu, on demandera à la physique de se mettre à jour et on mettra à jour nos objets réels à partir des données calculées par la physique.

LIBRAIRIES (1/3)

Il va falloir choisir la librairie qui nous convienne le mieux, et en 2D ou en 3D. Toujours se demander dans quel repère a-t-on besoin de la physique pour trancher la question sur une librairie 2D ou 3D.

LIBRAIRIES (2/3)

LIBRAIRIES 3D

- [ammo.js](#)
- [cannon.js](#)
- [oimo.js](#)
- [Rapier](#)

LIBRAIRIES (3/3)

LAQUELLE CHOISIR ?

Les **examples** de ThreeJS utilisent ammo.js. Dans le cours, on va plutôt partir sur cannon.js qui est plus simple dans nos cas d'utilisation.

UTILISATION DE CANNON (1/5)

```
npm i cannon
```

puis on l'importe

```
import CANNON from 'cannon';
```

UTILISATION DE CANNON (2/5)

On va commencer par créer un monde avec une gravité équivalente à celle que l'on connaît

```
const world = new CANNON.World();
world.gravity.set(0, -9.81, 0);
```

UTILISATION DE CANNON (3/5)

On va créer les équivalents de nos objets dans le monde physique

```
const boxShape = new CANNON.Box(0.375);
const boxBody = new CANNON.Body({
  mass: 1,
  position: new CANNON.Vec3(0, 2, 0),
  shape: boxShape
});
world.addBody(boxBody);
```

UTILISATION DE CANNON (4/5)

Il faut mettre à jour le monde à chaque frame et reporter la mise à jour sur nos objets réels

```
const tick = () => {
    // ...

    const deltaTime = clock.getDelta();
    // On veut que le monde tourne en 60 fps
    // le delta time va permettre d'update le monde du bon laps de temps
    // le 3 c'est le nombre d'itérations pour la physique
    world.step(1 / 60, deltaTime, 3);
    cube.position.copy(boxBody.position);

    // ...
}
```

UTILISATION DE CANNON (5/5)

Vous devriez voir votre cube tomber mais traverser le sol. Ajoutons ce qu'il faut pour que le cube s'arrête sur le plan

```
const planeShape = new CANNON.Plane();
const planeBody = new CANNON.Body({
    // permet d'avoir un objet static qui ne subit pas la gravité
    mass: 0,
    shape: planeShape
});
planeBody.quaternion.setFromAxisAngle(
    new CANNON.Vec3(-1, 0, 0),
    Math.PI * 0.5
);
world.addBody(planeBody);
```

PROPRIÉTÉ PHYSIQUE DU MATERIAU (1/4)

Le cube rebondit que très très légèrement sur notre plan. Cela est dû aux propriétés physiques du matériau appliqué au body. On peut créer toutes sortes de matériaux

PROPRIÉTÉ PHYSIQUE DU MATERIAU (2/4)

Pour créer un nouveau **matériaux**

```
const concreteMaterial = new CANNON.Material('concrete');
const plasticMaterial = new CANNON.Material('plastic');
```

PROPRIÉTÉ PHYSIQUE DU MATÉRIAU (3/4)

Maintenant on peut créer un `ContactMaterial` et donner les propriétés physiques de comment les 2 objets avec les matériaux appliqués doivent se comporter lorsqu'ils rentrent en collision

```
const concretePlasticMaterial = new CANNON.ContactMaterial(  
    concreteMaterial,  
    plasticMaterial,  
    {  
        friction: 0.1,  
        restitution: 0.7,  
    }  
);  
world.addContactMaterial(concretePlasticMaterial);
```

PROPRIÉTÉ PHYSIQUE DU MATERIAU (4/4)

On affecte nos matériaux à nos body

```
boxBody.material = plasticMaterial;  
planeBody.material = concreteMaterial;
```

On note maintenant une différence de comportement
du cube quand il rentre en contact avec le sol.

APPLIQUER DES FORCES (1/6)

Il peut arriver que l'on souhaite appliquer des forces à nos objets. Il existe plusieurs manières de faire.

APPLIQUER DES FORCES (2/6)

`applyForces` pour appliquer une force à un `Body` à partir d'un point dans l'espace global.

APPLIQUER DES FORCES (3/6)

`applyImpulse` est comme `applyForce` mais au lieu d'ajouter la force, cela s'appliquera sur la vitesse

APPLIQUER DES FORCES (4/6)

`applyLocalForce` est comme `applyForce` mais le point
est exprimé en local

APPLIQUER DES FORCES (5/6)

`applyLocalImpulse` est comme `applyImpulse` mais le point est exprimé en local

APPLIQUER DES FORCES (6/6)

On se servira du applyForce car en interne au final ça agira sur la vitesse.

```
boxBody.applyLocalForce(new CANNON.Vec3(150, 0, 0), new CANNON.Vec3(0, 0, 0));
```

Vous devriez remarquer le cube qui rebondit vers la droite sur l'axe des x

BROADPHASE (1/4)

Quand vous testez les collisions, une approche naïve est que chaque body va tester ses collisions avec tous les autres body. Simple comme technique mais qui peut s'avérer très peu performante lorsque l'on commence à avoir beaucoup d'objets.

BROADPHASE (2/4)

C'est à ce moment là que la notion de broadphase arrive, qui va d'abord trier les objets avant d'effectuer les tests de collision.

BROADPHASE (3/4)

Il existe 3 algorithmes de broadphase dans Cannon.js:

- **NaiveBroadphase**: Teste tous les bodies avec tous les autres
- **GridBroadphase**: Fait un quadrillage du monde et tester les bodies qui sont les mêmes cases
- **SAPBroadphase**: Teste les bodies sur un axe arbitraire pendant plusieurs itérations

BROADPHASE (4/4)

Par défaut, c'est la naïve qui est utilisée mais vous pouvez tester de changer pour la SAPBroadphase.

```
world.broadphase = new CANNON.SAPBroadphase(world);
```

Note: Utiliser cette broadphase peut provoquer des bugs mais c'est très rare, sauf si par exemple les bodies bougent extrêmement vite

ENDORMISSEMENT (1/3)

Même avec une amélioration de l'algorithme de la broadphase, il peut s'avérer de vouloir tester avec objets qui ne bougent plus du tout. Pour éviter cela, on peut se servir du système d'endormissement.

ENDORMISSEMENT (2/3)

Quand un objet devient extrêmement lent de sorte de ne plus pouvoir percevoir son mouvement, on peut dire au body de s'endormir et il sera ignoré pour les tests tant qu'il ne subit pas une force assez suffisante pour le réveiller (force ou collision)

ENDORMISSEMENT (3/3)

Pour autoriser l'endormissement d'un body, on peut faire:

```
body.allowSleep = true;
```

ÉVÉNEMENTS (1/2)

On peut écouter plusieurs événement sur les bodies:

- colide: une collision vient de se produire
- sleep: l'objet vient de s'endormir
- wakeup: l'objet vient de se réveiller

ÉVÉNEMENTS (2/2)

Par exemple pour écouter une éventuelle collision:

```
body.addEventListener('collide', (collision) => console.log(collision));
```

REACT THREE FIBER

INTRODUCTION (1/3)

Jusqu'à maintenant on a appris à faire du ThreeJS en
pure vanilla

Mais des fois on a besoin d'intégrer nos expériences
dans des environnements particuliers voire même à
travers d'autres frameworks

INTRODUCTION (2/3)

Et quand on parle de framework, on va parler d'un des plus populaires: React

INTRODUCTION (3/3)

On pourrait voir cela comme une contrainte dans la manière de concevoir l'expérience, mais cela rend le processus de création d'expériences ThreeJS encore plus simple

REACT THREE FIBER (1/2)

ThreeJS s'intègre bien avec React grâce à [React Three Fiber \(R3F\)](#)

REACT THREE FIBER (2/2)

Vous pouvez trouver une liste d'exemple sur ce lien

PREMIÈRE APPLICATION R3F (1/10)

R3F est juste un renderer sous React
On écrit en JSX et c'est convertit en ThreeJS

PREMIÈRE APPLICATION R3F (2/10)

On va d'abord créer une application React

```
npm create vite@latest
```

Et donner un nom à votre projet, choisissez React puis
JavaScript

PREMIÈRE APPLICATION R3F (3/10)

On va ensuite installer les dépendances minimum pour faire du R3F

```
npm i three @react-three/fiber
```

PREMIÈRE APPLICATION R3F (4/10)

Première chose, on va avoir besoin d'un canvas pour faire le rendu

```
import { Canvas } from '@react-three/fiber';

root.render(
  <>
    <Canvas></Canvas>
  </>
)
```

Après ça, si vous inspectez le DOM, vous devriez y trouver un canvas

PREMIÈRE APPLICATION R3F (5/10)

Par défaut, le canvas fait 100% la taille de son parent,
donc à vous de styliser correctement pour avoir l'effet
voulu

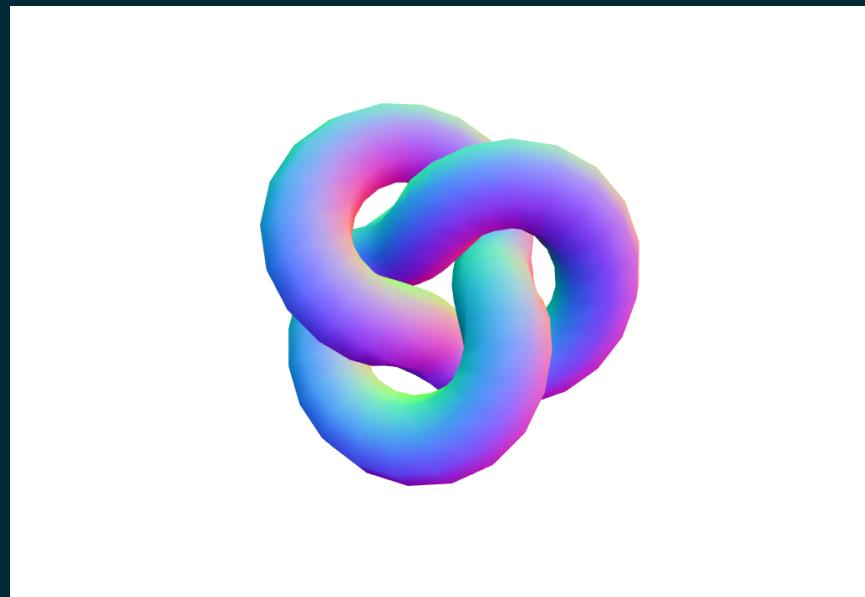
PREMIÈRE APPLICATION R3F (6/10)

On va maintenant y rajouter un mesh

```
root.render(  
  <>  
    <Canvas>  
      <mesh>  
        <torusKnotGeometry />  
        <meshNormalMaterial />  
      </mesh>  
    </Canvas>  
  </>  
)
```

PREMIÈRE APPLICATION R3F (7/10)

On obtient ce résultat:



PREMIÈRE APPLICATION R3F (8/10)

Vous avez sûrement observé qu'on a créé notre premier mesh avec des tags mais sans faire aucun import de ces derniers.

C'est ce qu'on appelle du code *déclaratif*
R3F créera automatiquement les bonnes instances et
les ajoutera à la scène

PREMIÈRE APPLICATION R3F (9/10)

Quand on a créé now tags, on a laissé les valeurs par défaut mais vous pouvez mettre des arguments dans le même ordre que sa classe équivalent dans ThreeJS via l'attribute args

```
root.render(  
  <>  
    <Canvas>  
      <mesh>  
        <boxGeometry args={[1, 1, 1]} />  
        <meshBasicMaterial color="red" />  
      </mesh>  
    </Canvas>  
  </>  
)
```

PREMIÈRE APPLICATION R3F (10/10)

De la même manière, on a accès à tous les attributes qu'on aurait accès dans les classes équivalentes

```
root.render(  
  <>  
  <Canvas>  
    <mesh positions={[1, 0, 0]} scale={1.5} rotation-y={Math.PI * 0.25}>  
      <boxGeometry args={[1, 1, 1]} />  
      <meshBasicMaterial color="red" />  
    </mesh>  
  </Canvas>  
</>  
)
```

ANIMATIONS (1/2)

On va voir maintenant de comment ajouter des animations mais il faut d'abord récupérer le mesh

```
import { useRef } from 'react';

// Pour obtenir le mesh instancié
const cube = useRef()

return (
  <mesh ref={cube}>
    ...
  </mesh>
)
```

ANIMATIONS (2/2)

L'équivalent de notre tick que l'on faisait habituellement, c'est d'utiliser le hook `useFrame` fourni par R3F.

```
import { useFrame } from '@react-three/fiber';

useFrame((state, delta) => (cube.current.rotation.y += delta; ));

return (
  <mesh ref={cube}>
    ...
  </mesh>
)
```

Note: le `useFrame` ne peut fonctionner lorsqu'il est dans un composant qui est à l'intérieur d'un `Canvas`

LUMIÈRES

On va voir comment ajouter des lumières dans notre expérience

```
return (
  <Canvas>
    <ambientLight intensity={1.5} />
    <directionalLight position={ [1, 2, 3] }/>
    <mesh>
      <boxGeometry />
      <meshStandardMaterial color="orange" />
    </mesh>
  </Canvas>
)
```

DREI (1/10)

Un des avantages de React est de pouvoir faire des composants (aussi appelés helpers) et de les réutiliser à l'infini.

Certains peuvent être ajoutés en tant que composants, d'autres en tant que hooks. Vous pouvez trouver une

liste avec tous les composants / helpers: [ici](#)

Découvrez-les et testez-les pour voir l'étendu du possible

DREI (2/10)

On va commencer par installer les dépendances qu'il nous manque

```
npm @react-three/drei
```

DREI (3/10)

On va pouvoir par exemple instancier un OrbitControls

```
import { OrbitControls } from '@react-three/drei';

...

return (
  <Canvas>
    <OrbitControls />
  </Canvas>
)
```

DREI (4/10)

On peut facilement rajouter des manipulateurs pour les objets

```
import { TransformControls } from '@react-three/drei';

...

return (
  <Canvas>
    <TransformControls>
      <mesh ... />
    </TransformControls>
  </Canvas>
)
```

DREI (5/10)

On observe 2 problèmes:

- Il y a un clash entre l'OrbitControls et le TransformControls
- Le TransformControls est ajouté au milieu de la scène et non au même endroit que l'objet qu'il est censé contrôlé

DREI (6/10)

Pour corriger le premier, il existe un attribute sur
OrbitControls:

```
return (
  <Canvas>
    <OrbitControls makeDefault />
    ...
  </Canvas>
)
```

DREI (7/10)

Pour corriger le deuxième, on va donner le mesh en question en tant que ref

```
const cube = useRef();  
  
return (  
  <Canvas>  
    <mesh ref={cube} .../>  
    <TransformControls object={cube} />  
  </Canvas>  
)
```

DREI (8/10)

Pour ajouter des éléments HTML qui vont suivre vos objets 3D

```
import { Html } from '@react-three/drei'

return (
  <mesh ref={cube}>
    <Html>This is a mesh</Html>
  </mesh>
)
```

DREI (9/10)

Pour ajouter du texte en 3D dans la scène

```
import { Text } from '@react-three/drei'

return (
  <Text>This is a text</Text>
)
```

DREI (10/10)

Pour charger des modèles 3D

```
import { useGLTF } from '@react-three/drei'

const model = useGLTF('./models.glb');

return (
  <primitive object={model.scene} />
)

useGLTF.preload('./models.glb');
```