



# La 3D avec WebGL & three.js

- Ce document est la propriété de **DOTSAFE**. Toute reproduction, même partielle sans l'autorisation expresse de **DOTSAFE** est formellement interdite.

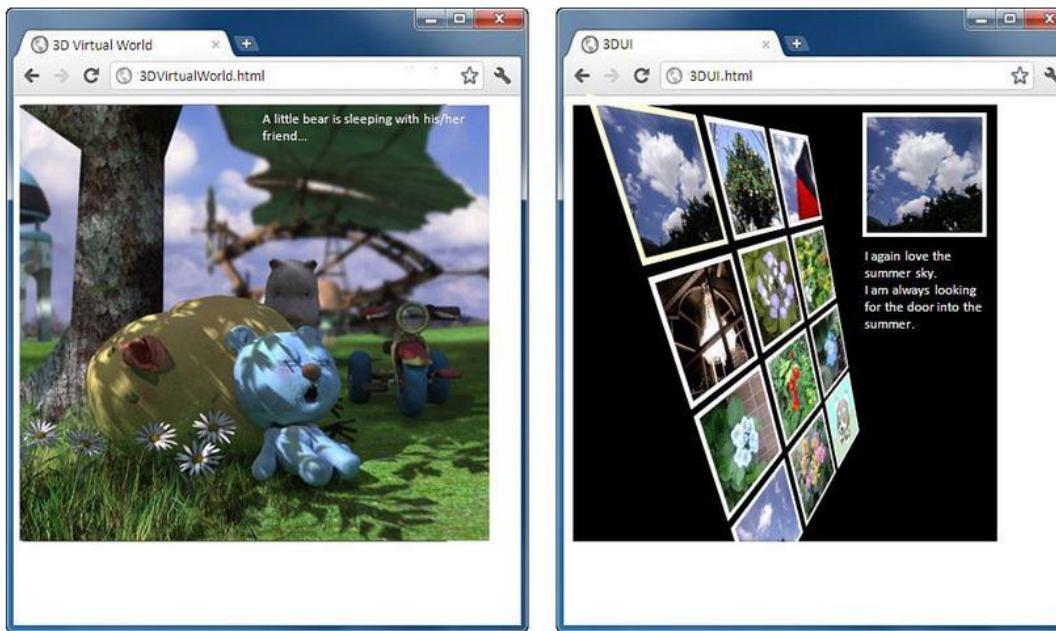
# Introduction à WebGL

- Dessiner, afficher et interagir avec de la 3D directement dans le navigateur
- Plus réservé uniquement aux consoles de jeu
- Accessible au développeur web
- Facile à déployer (pas de pb d'arch, OS, etc...)
- Basé sur des standards du web : HTML5 & Javascript



# Un vieux rêve

- Une technologie du futur pour le Web :
  - Jeu vidéo, data visualization , ecommerce, eLearning, etc..



# Introduction à WebGL

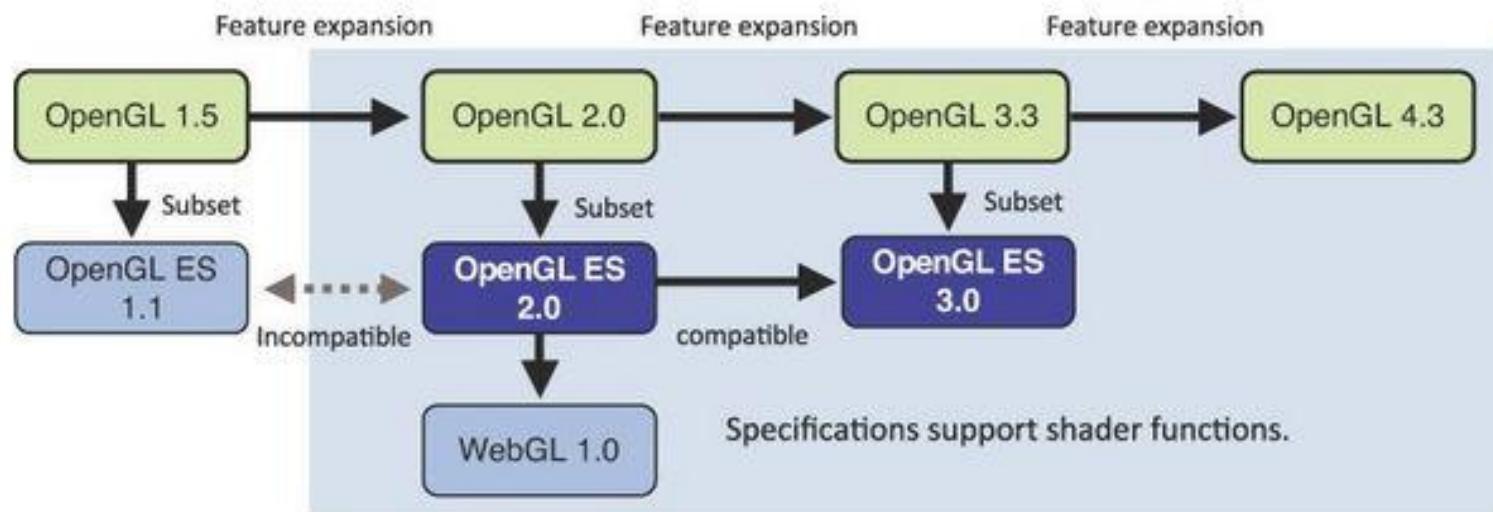
- Compatibilité ordinateur, smartphone & tablette.
- Directement intégré dans les nouveaux navigateurs, pas besoin de plugin !
- Des possibilités infinies grâce aux fonctionnalités déjà présentes dans les navigateurs
- Uniquement besoin d'un éditeur de texte et d'un navigateur Web

# Des superbes exemples

- <https://code.google.com/p/webglsamples/>
- <http://hellorun.helloenjoy.com/> (jeu de rapidité, très beau)
- <https://goote.ch/8dc32a3c778e4a53a14a1f0f42359e3a.project/> (générateur de donjon)
- <http://www.chromeexperiments.com/detail/xibalba/?f=webgl> (un wolfenstein)
- <http://ogreen.special-t.com/en/> (artistique/publicitaire)

# Origine de WebGL

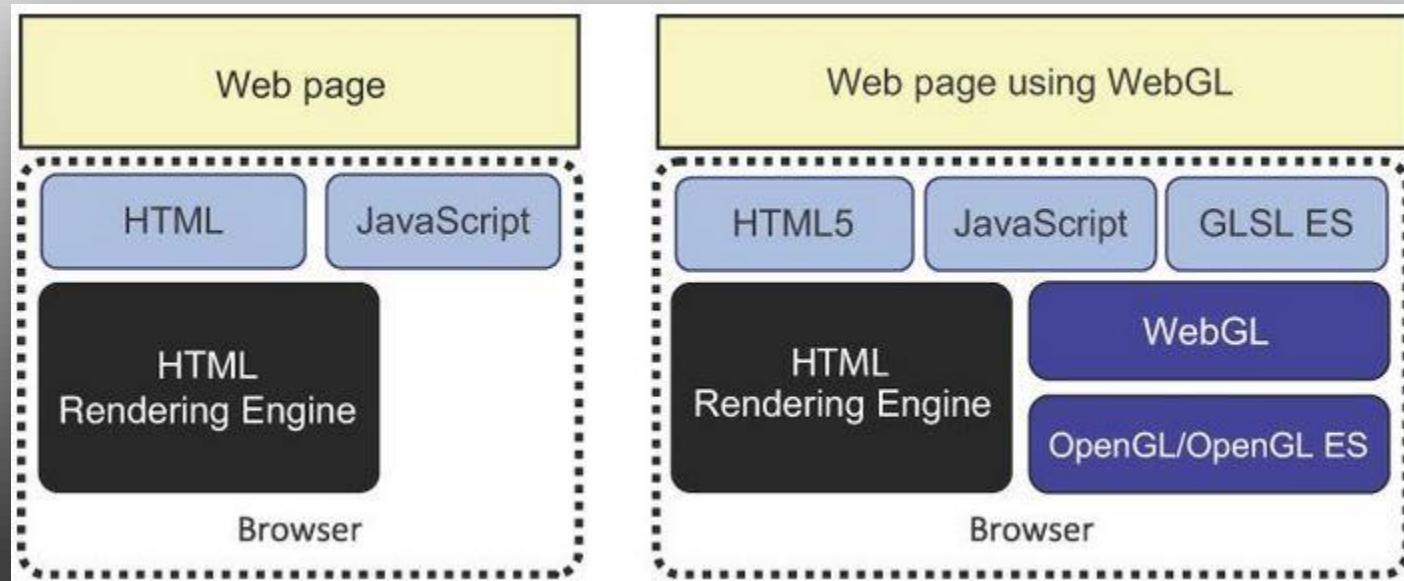
- L'OpenGL du Web
- Standard ouvert défini par le Khronos Group, organisation non lucrative (dont AMD, Apple, ARM, Ericsson, Nokia, IBM, Intel, NVIDIA, Google, Mozilla, Oracle, Samsung et Sony)



# Origine de WebGL

- OpenGL (Open Graphic Library) :
  - une API implémentée au niveau des pilotes graphique
  - Utilisée dans les applications compilée
- OpenGL ES (OpenGL for Embedded System):
  - Version allégée pour le matériel mobile/embarqué

# Pour résumer



# Différences entre WebGL et un moteur de jeux

- WebGL s'occupe uniquement du rendu 3D
- C'est léger et très rapide
- Un moteur de jeux a beaucoup plus de fonctionnalités comme :
  - La détection de collision
  - Les effets de particules
  - La physique
  - Le réseau
- WebGL dispose de fonctions primitives pour afficher des triangles ou des objets plus complexes mais pas de moyen simple et direct d'ajouter une caméra à votre scène

# Comprendre les mathématiques de la 3D

- Développer en 3D demande des bonnes connaissances en mathématique
- Notamment :
  - Les vecteurs
  - Les matrices
  - Les changements de repères
  - La trigonométrie
  - Etc...
- WebGL ne fournit pas de librairie pour gérer tout ça

# Les vecteurs

- Les vecteurs sont utilisés pour :
  - Représenter un point dans l'espace
  - Des directions dans l'espace comme l'orientation de la caméra ou l'orientation des normales d'une surface
- Un point dans l'espace peut être représenté par un vecteur en utilisant les axes x, y, z

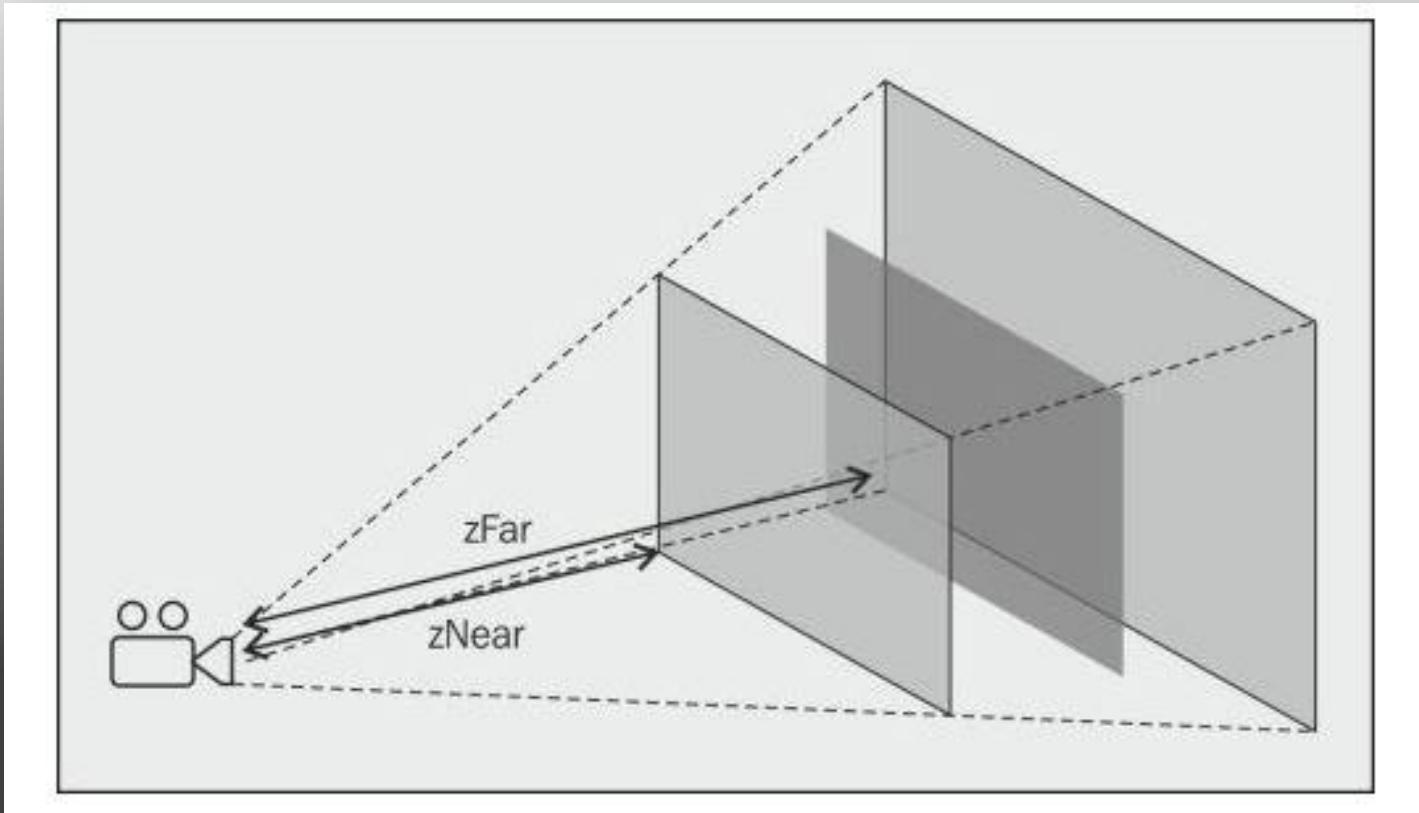
# Les matrices

- Les matrices sont initialement utilisées pour décrire la relation entre deux systèmes de coordonnées spatiales en 3D
- Elles permettent de définir des transformations comme les rotations, les changements d'échelle ou les translations
- On peut les additionner, les soustraire, les multiplier, etc...

# Matrice de projection

- Changer la matrice de projection c'est comme changer de lentille sur une caméra
- Détermination de comment les objets apparaissent dans la scène (près, loin, dans le champ ou non, etc...)
- `mat4.perspective(out, fovy, aspect, near, far)`
  - `fovy` : profondeur du champ
  - `aspect` : ratio de la scène
  - `near` : premier plan
  - `far` : dernier plan

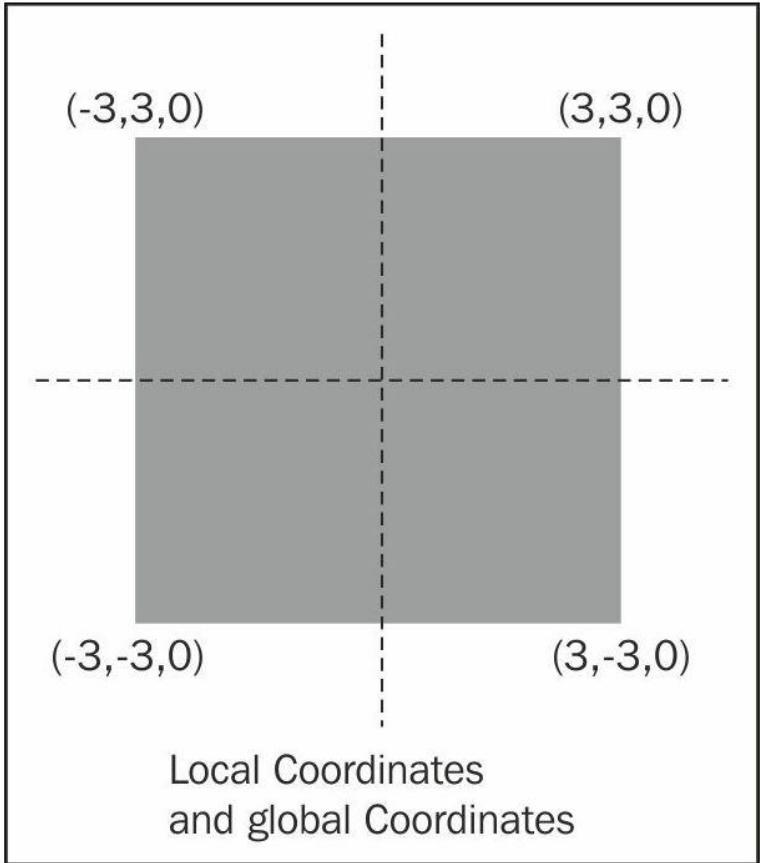
# Matrice de projection



# Les bases de la 3D

- Un modèle en 3D est appelé un **mesh**
- Chaque facette d'un **mesh** est appelé **polygone**
- Un polygone est composé d'au moins 3 angles appelés **vertex**
- Un polygone avec 4 **vertices** forme un **quad**
- Chaque vertex est défini par un vecteur selon 2 à 3 axes

# Les bases de la 3D



```
vertices = [  
    3.0, 3.0, 0.0, //Vertex 0  
    -3.0, 3.0, 0.0, //Vertex 1  
    3.0, -3.0, 0.0, //Vertex 2  
    -3.0, -3.0, 0.0, //Vertex 3  
];
```

# Dessiner des modèles complexes



```
vertices = [  
    ....générées par un outil de  
    modélisation...  
];
```

//Heureusement il n'y a pas à les  
coder à la main !

# Utiliser des indices pour sauver la mémoire

- Pour chaque vertex défini, une valeur numérique est associée, par exemple
  - Vertex 0 : 0
  - Vertex 1 : 1
  - Etc...
- Il s'agit des indices

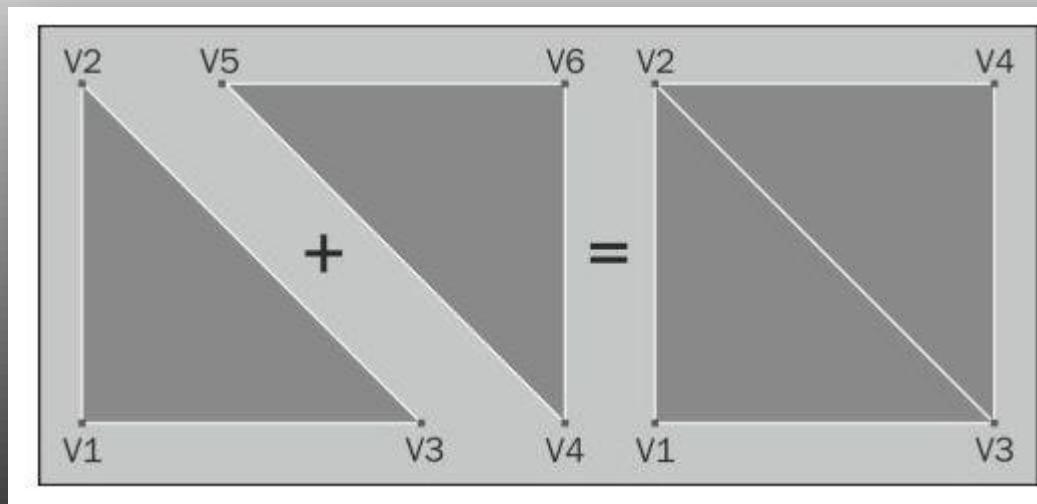
```
vertices = [  
    3.0, 3.0, 0.0, //Vertex 0  
    -3.0, 3.0, 0.0, //Vertex 1  
    3.0, -3.0, 0.0, //Vertex 2  
    -3.0, -3.0, 0.0, //Vertex 3
```

```
];
```

```
Indices = [0, 2, 3, 0, 3, 1];
```

- En utilisant les indices, on sauve la mémoire car plusieurs vertices sont partagées entre plusieurs triangles (surtout si on imagine une sphère)

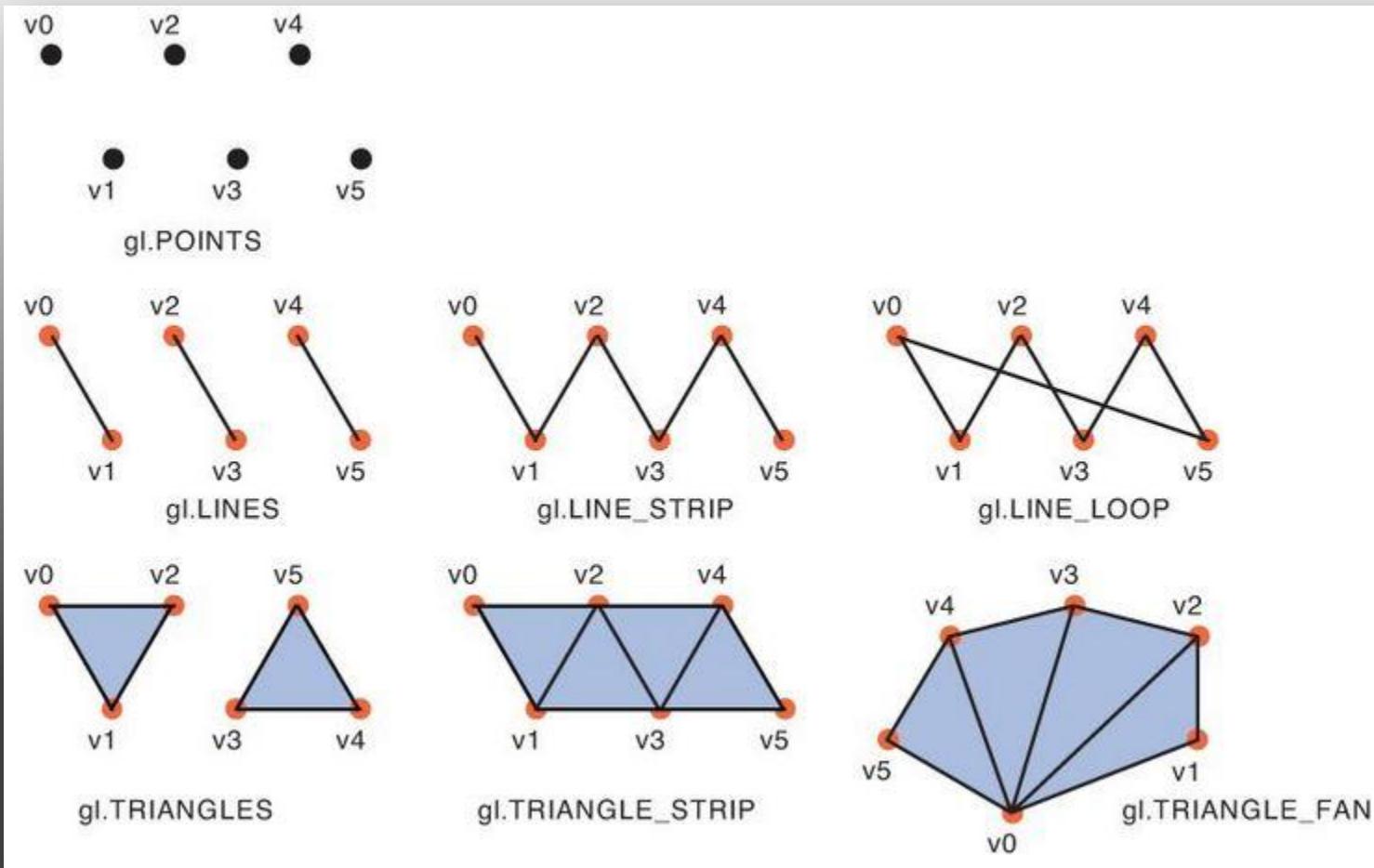
# Utiliser des indices pour sauver la mémoire



# VBO

- Les vertices sont envoyées à un buffer nommé VERTEX BUFFER OBJECT
- Il peut être de deux type :
  - GL.ARRAY\_BUFFER
  - GL.ELEMENT\_ARRAY\_BUFFER (utilisation des indices)
- Le VBO est ensuite envoyé à la carte graphique à travers le programme de shaders

# Plusieurs méthodes de dessin



# Shaders

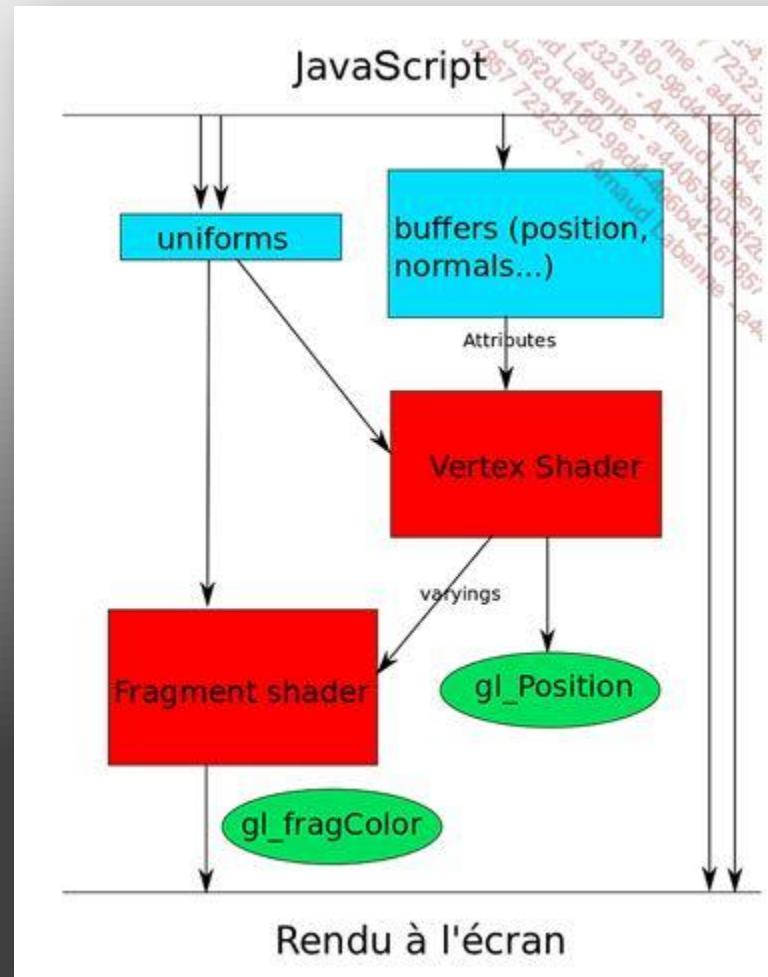
- Comme toute application 3D, une application WebGL est composée d'une partie de code exécutée sur CPU & une autre partie exécutée sur le GPU
- Le code graphique est regroupé en unités d'exécution appelés shaders
- Rédigé dans un langage spécial, le GLSL (OpenGL Shading Language) – très proche du C !

# Shaders

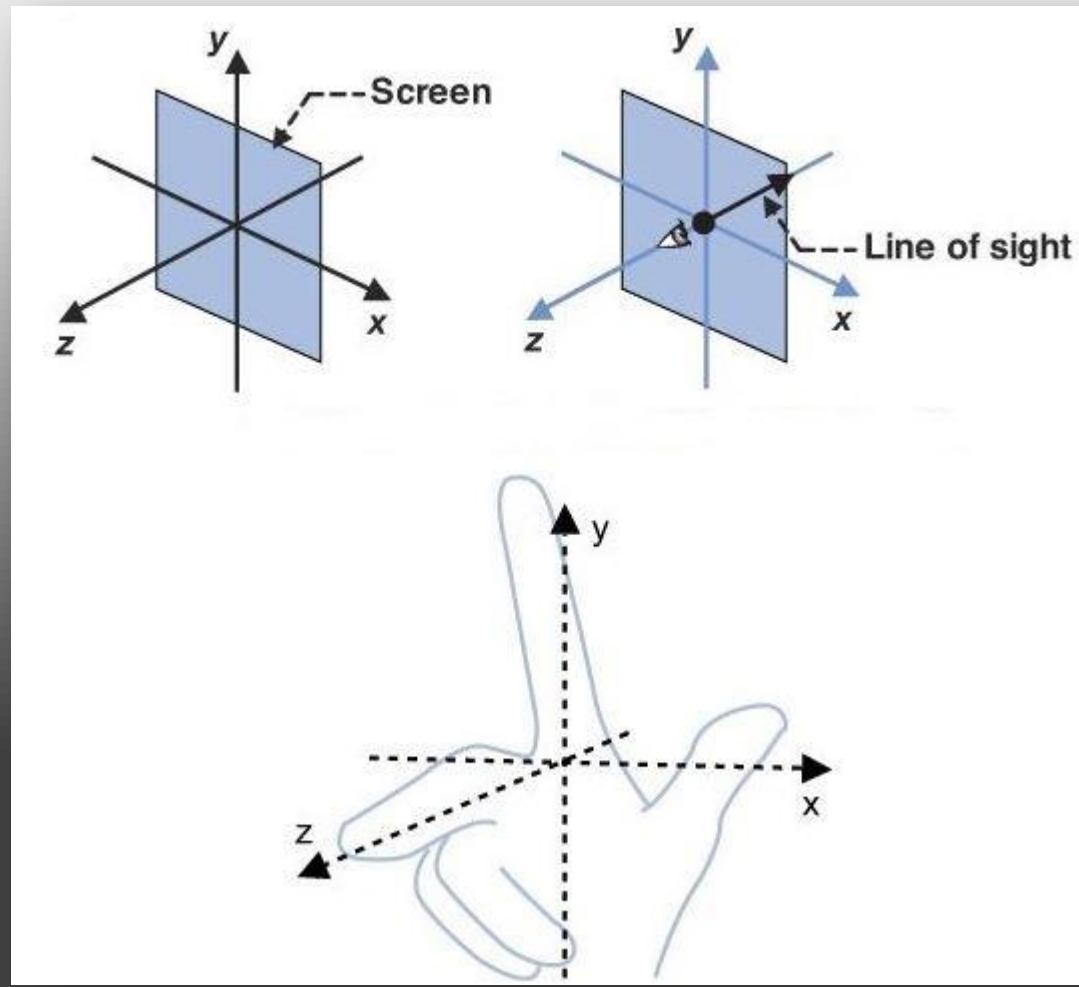
- Regroupement des shaders est appelé « programme de shaders »
- Il est compilé !
- Pour un objet à afficher, ce dernier reçoit les différents paramètres de l'objet (position des points, textures, faces, etc...) et calcule la couleur de chaque pixel du rendu de l'objet à l'écran
- Ces calculs sont effectués en parallèle !
- Leur distribution dépend du matériel et n'est pas à la charge du développeur

# Shaders

- Les shaders reçoivent du JavaScript des variables spéciales qu'on appelle attributes et uniforms.
- Les attributes sont des variables prédéfinies correspondant à des VBO de type `GL.ARRAY_BUFFER`.
- Les uniforms sont définies depuis le code hôte et correspondent à des paramètres personnalisés et constants au cours du rendu d'un objet. Cela peut être la position de l'objet, d'une lumière, l'intensité du brouillard.
- Cela ne vous rappelle rien ?



# Le système de coordonnées de WebGL



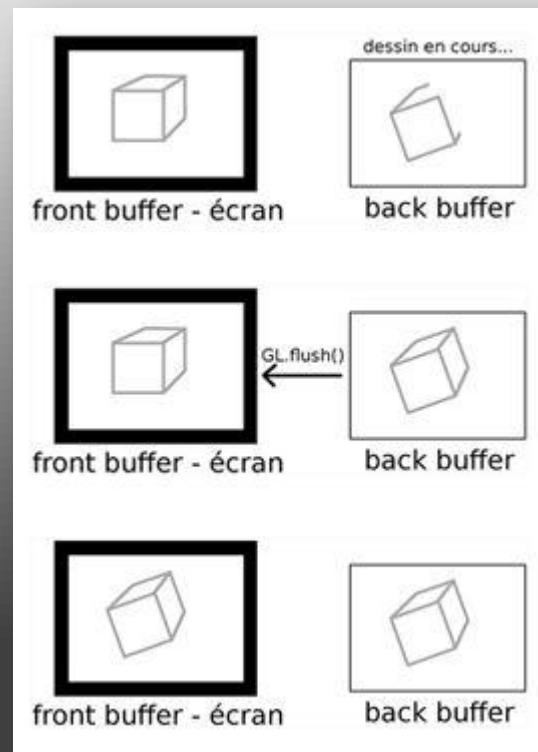
# Gestion du temps

- Dessiner une scène c'est bien, mais pour créer une sensation de mouvement, il faut recalculer la scène plusieurs fois par seconde
- Nombre de rendus générés par seconde == FPS == Frames Per Second
- Recommandé d'avoir au moins 20 FPS
- Conseillé 40 FPS
- Fluidité maximale 60 FPS

# Front & Back Buffer

- Un buffer d'affichage ou framebuffer est un tampon dans lequel écrivent les shaders.
- Pour effectuer un rendu correct, WebGL possède deux buffers : front et back buffer
- Le dessin de la scène s'effectue dans le back buffer. Une fois la scène dessinée, on appelle la fonction `GL.flush()`. Les buffers sont swappés
- Cela permet d'afficher que des rendus terminés

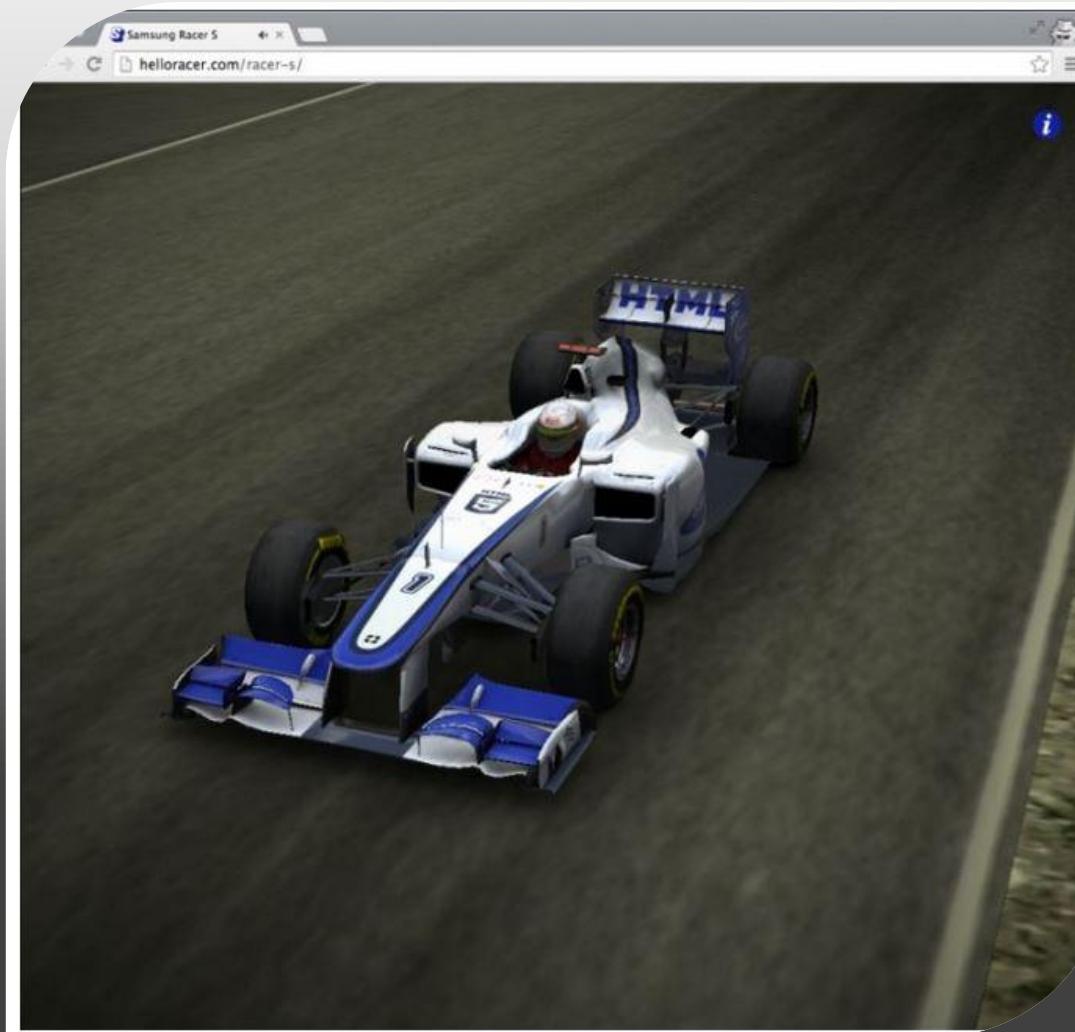
# Front & Back buffer



# Three.js

- Three.js est une librairie OpenSource qui s'appuie sur WebGL pour dessiner des scènes 3D dans le navigateur
- Créeé en 2010
- <http://threejs.org/>
- Facilite grandement le processus de création
- Conserve le support multidevice

# Three.js



# Prérequis

- 100% standalone, pas de dépendance
- Besoin d'un navigateur qui supporte WebGL

IE : à partir de la version 11

Firefox : à partir de la version 4

Chrome : à partir de la version 10

Safari : à partir de la version 5.1

Opera : à partir de la version 12

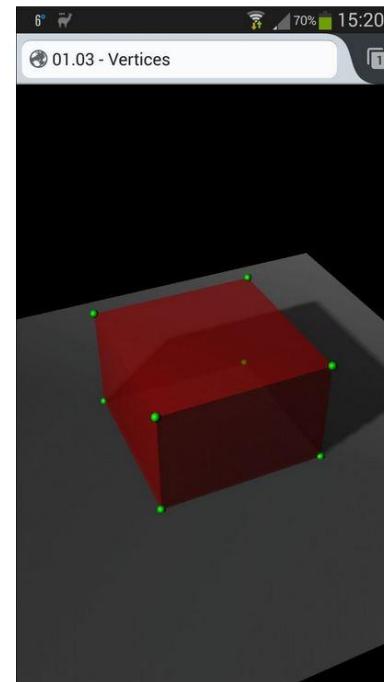
Mobile Firefox for Android

Google Chrome for Android

Opera Mobile

Blackberry Browser

Safari sur iOS



# Installer un serveur Web

- Pour éviter les problèmes de sécurité navigateur (chargement ajax), il vaut mieux installer un serveur web
- On pourra aussi prévoir une partie client/serveur à nos applications three.js
- 1000 solutions :
  - EasyPHP
  - Mongoose portable
  - Node.js
  - Etc...

# HelloWorld three.js

```
<!DOCTYPE html>
<html>
<head>
    <title>01.01 - Basic scene</title>
    <script src="../libs/three.js"></script>
    <style>
        body {
            /* set margin to 0 and overflow to hidden, to go fullscreen */
            margin: 0;
            overflow: hidden;
        }
    </style>
</head>
```

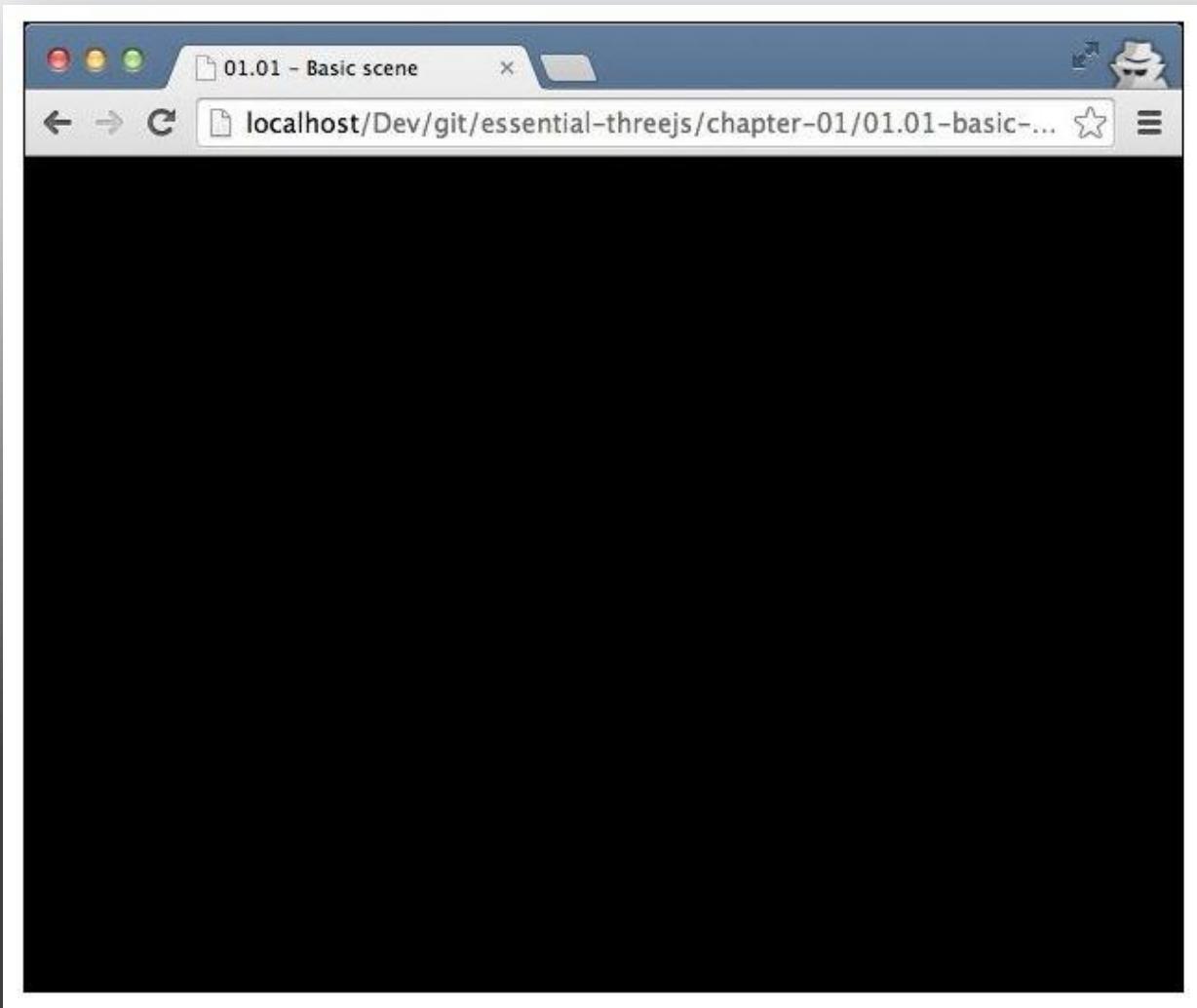
# HelloWorld three.js

```
/**  
 * Initializes the scene, camera and objects. Called when the window is  
 * loaded by using window.onload (see below)  
 */  
function init() {  
  
    // create a scene, that will hold all our elements such as objects, cameras and lights.  
    scene = new THREE.Scene();  
  
    // create a camera, which defines where we're looking at.  
    camera = new THREE.PerspectiveCamera(45, window.innerWidth / window.innerHeight, 0.1, 1000);  
  
    // create a render, sets the background color and the size  
    renderer = new THREE.WebGLRenderer();  
    renderer.setClearColor(0x000000, 1.0);  
    renderer.setSize(window.innerWidth, window.innerHeight);  
    renderer.shadowMapEnabled = true;  
  
    // position and point the camera to the center of the scene  
    camera.position.x = 15;  
    camera.position.y = 16;  
    camera.position.z = 13;  
    camera.lookAt(scene.position);  
  
    // add the output of the renderer to the html element  
    document.body.appendChild(renderer.domElement);  
  
    // call the render function, after the first render, interval is determined  
    // by requestAnimationFrame  
    render();  
}
```

# HelloWorld three.js

```
/**  
 * Called when the scene needs to be rendered. Delegates to requestAnimationFrame  
 * for future renders  
 */  
function render() {  
    // render using requestAnimationFrame  
    requestAnimationFrame(render);  
    renderer.render(scene, camera);  
}  
  
/**  
 * Function handles the resize event. This make sure the camera and the renderer  
 * are updated at the correct moment.  
 */  
function handleResize() {  
    camera.aspect = window.innerWidth / window.innerHeight;  
    camera.updateProjectionMatrix();  
    renderer.setSize(window.innerWidth, window.innerHeight);  
}  
  
// calls the init function when the window is done loading.  
window.onload = init;  
// calls the handleResize function when the window is resized  
window.addEventListener('resize', handleResize, false);
```

# Whoua



# Ajoutons un sol

```
// create the ground plane
var planeGeometry = new THREE.PlaneGeometry(20, 20);
var planeMaterial = new THREE.MeshLambertMaterial({color: 0xcccccc});
var plane = new THREE.Mesh(planeGeometry, planeMaterial);
plane.receiveShadow = true;

// rotate and position the plane
plane.rotation.x = -0.5 * Math.PI;
plane.position.x = 0;
plane.position.y = -2;
plane.position.z = 0;

// add the plane to the scene
scene.add(plane);
```

# Ajoutons un cube

```
// create a cube
var cubeGeometry = new THREE.BoxGeometry(6, 4, 6);
var cubeMaterial = new THREE.MeshLambertMaterial({color: 'red'});
var cube = new THREE.Mesh(cubeGeometry, cubeMaterial);

cube.castShadow = true;

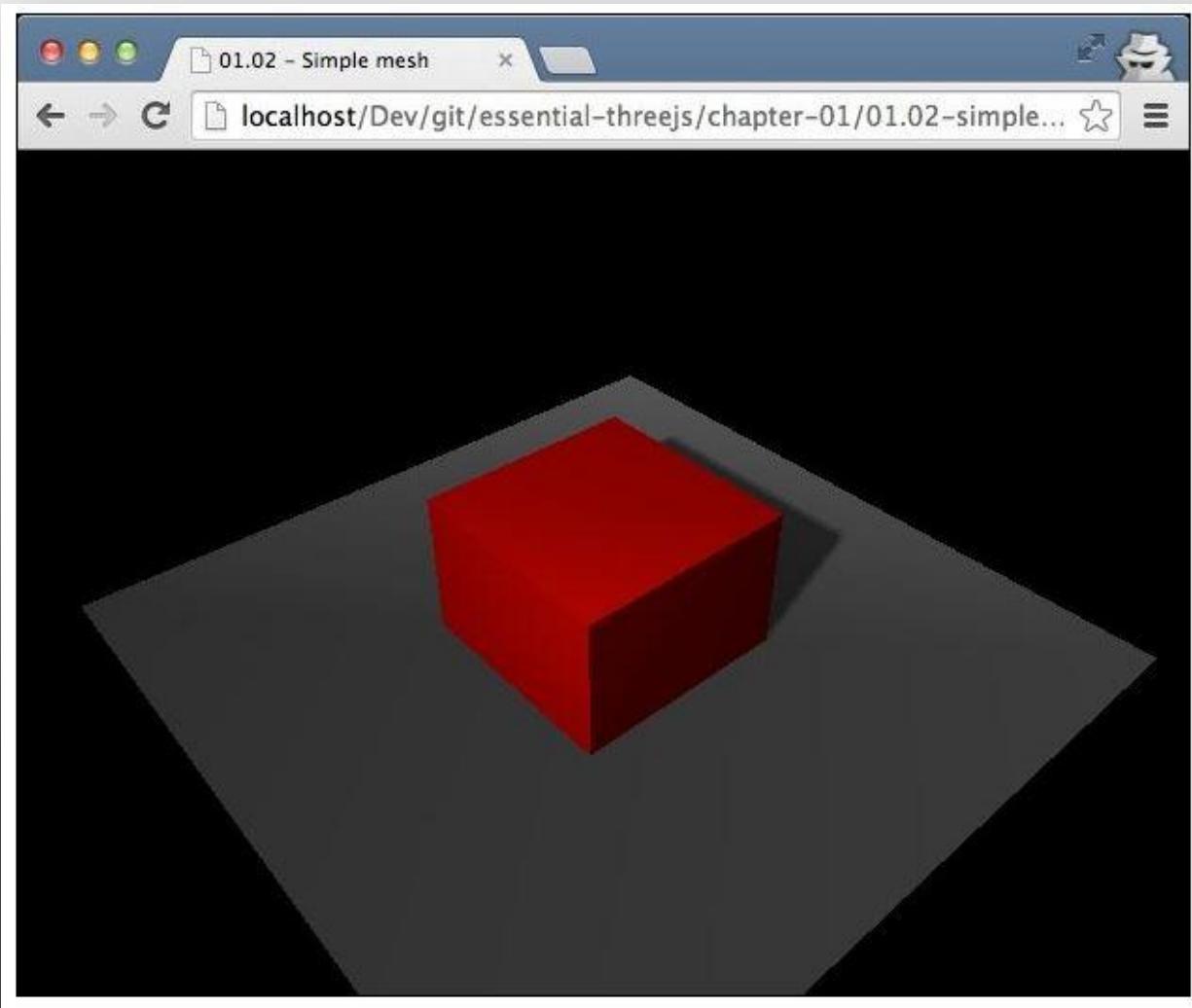
// add the cube to the scene
scene.add(cube);
```

# Ajoutons une lumière

```
// add spotlight for the shadows
var spotLight = new THREE.SpotLight(0xffffffff);
spotLight.position.set(10, 20, 20);
spotLight.shadowCameraNear = 20;
spotLight.shadowCameraFar = 50;
spotLight.castShadow = true;

scene.add(spotLight);
```

# Notre HelloWorld



# Mouvement de caméra

```
/**  
 * Called when the scene needs to be rendered. Delegates to  
requestAnimationFrame  
 * for future renders  
 */  
function render() {  
    var rotSpeed = 0.01;  
    camera.position.x = camera.position.x * Math.cos(rotSpeed)  
+ camera.position.z * Math.sin(rotSpeed);  
    camera.position.z = camera.position.z * Math.cos(rotSpeed)  
- camera.position.x * Math.sin(rotSpeed);  
    camera.lookAt(scene.position);  
  
    // render using requestAnimationFrame  
    requestAnimationFrame(render);  
    renderer.render(scene, camera);  
}
```

# Exercice

- Modifier les différents paramètres :
  - Caméra
  - Sol
  - Cube
  - Couleurs
  - Lumière
  - Etc...
- Les autres formes sur [threejs.org](http://threejs.org)

WORK IN  
PROGRESS

# Améliorons l'affichage

- On peut ajouter la superbe librairie dat.GUI  
<https://code.google.com/p/dat-gui/>
- Permet d'ajouter des éléments d'interfaces très simplement
- On peut aussi ajouter stats.js  
<http://github.com/mrdoob/stats.js>
- On peut ajouter un menu qui permettra de configurer l'affichage et de présenter des infos supplémentaires (HUD)

# Améliorons l'affichage

```
scene.add(spotLight);

// setup the control object for the control gui
control = new function() {
    this.rotationSpeed = 0.005;
    this.opacity = 0.6;
    this.color = cubeMaterial.color.getHex();
};

// add extras
addControlGui(control);
addStatsObject();

// add the output of the renderer to the html element
document.body.appendChild(renderer.domElement);
```

# Améliorons l'affichage

```
function addControlGui(controlObject) {  
    var gui = new dat.GUI();  
    gui.add(controlObject, 'rotationSpeed', -0.01, 0.01);  
    gui.add(controlObject, 'opacity', 0.1, 1);  
    gui.addColor(controlObject, 'color');  
}  
  
function addStatsObject() {  
    stats = new Stats();  
    stats.setMode(0);  
  
    stats.domElement.style.position = 'absolute';  
    stats.domElement.style.left = '0px';  
    stats.domElement.style.top = '0px';  
  
    document.body.appendChild( stats.domElement );  
}
```

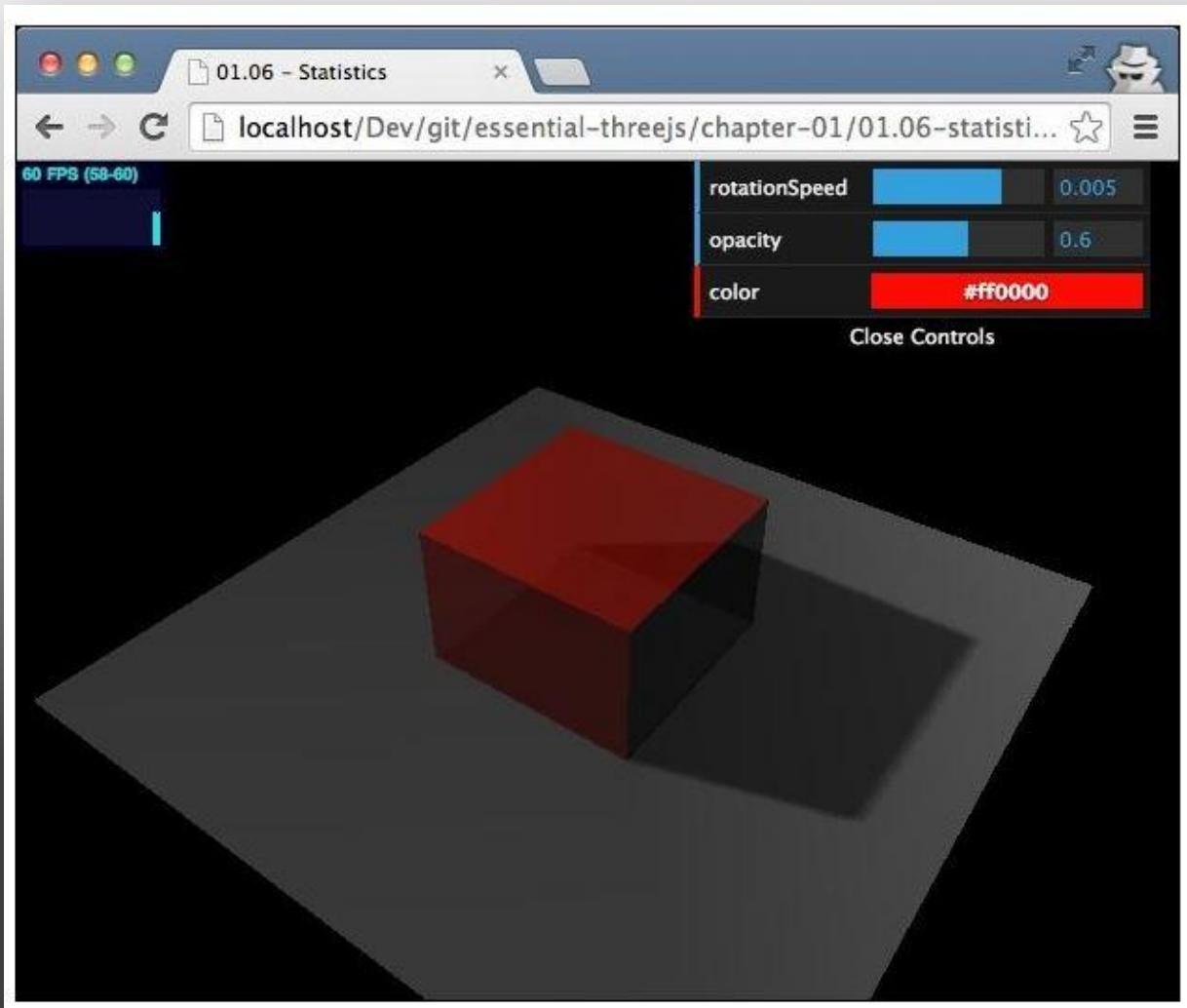
# Améliorons l'affichage

```
function render() {
    // update the camera
    var rotSpeed = control.rotationSpeed;
    camera.position.x = camera.position.x * Math.cos(rotSpeed) +
camera.position.z * Math.sin(rotSpeed);
    camera.position.z = camera.position.z * Math.cos(rotSpeed) -
camera.position.x * Math.sin(rotSpeed);
    camera.lookAt(scene.position);

    // change opacity
    scene.getObjectByName('cube').material.opacity = control.opacity;

    // change color
    scene.getObjectByName('cube').material.color = new THREE.Color(
control.color);
}
```

# Tada !



# Globe controls

- Il est possible de faire « orbiter » la caméra autour du centre de la scène grâce à la classe THREE.OrbitControls

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

camera.position.z = 0,
camera.lookAt(scene.position);

// add controls
cameraControl = new THREE.OrbitControls(camera);

// setup the control object for the control gui
control = new function () {
    this.rotationSpeed = 0.005.
```

# Globe controls

```
/**  
 * Called when the scene needs to be rendered. Delegates to requestAnimationFrame  
 * for future renders  
 */  
function render() {  
  
    // update stats  
    stats.update();  
  
    // update the camera  
    cameraControl.update();  
  
    // and render the scene  
    renderer.render(scene, camera);  
  
    // render using requestAnimationFrame  
    requestAnimationFrame(render);  
}
```

# Globe controls

Contrôle	Action
Clic gauche et bouger la souris	Rotation de la caméra autour de la scène
Molette	Zoom
Clic droit et bouger la souris	Se déplacer dans la scène

# Exercice

- Appliquer la classe THREE.OrbitControls à l'exemple précédent (celui qui ne tourne pas)

WORK IN  
PROGRESS

# Textures

- Avec Three.js, on peut appliquer simplement des textures

```
var matexture =  
    THREE.ImageUtils.loadTexture("../assets/image.jpeg");  
  
var material = new THREE.MeshBasicMaterial({map :  
    matexture});
```

- Le chargement est asynchrone
- Supporte la transparence

# Exercice

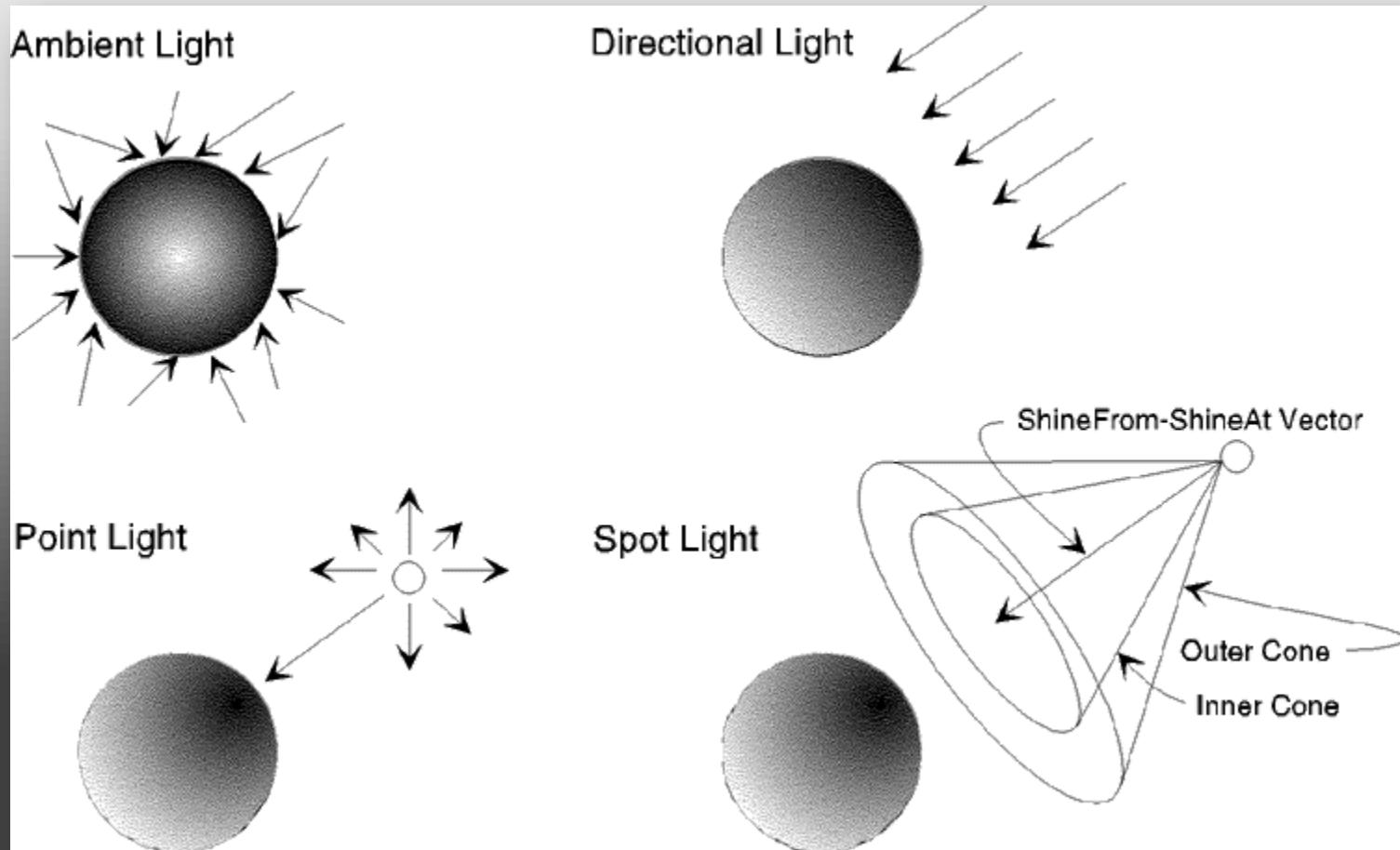
- Appliquer une texture au sol du HelloWorld

WORK IN  
PROGRESS

# Lumières ambiantes et directionnelles

Nom	Description
AmbientLight	Une simple lumière dont la couleur est ajoutée à la couleur d'un material
PointLight	Un point dans l'espace dont émane une lumière dans toutes les directions
SpotLight	Une lumière avec un effet cône, comme une lampe torche
DirectionalLight	Une lumière très lointaine : les rayons lumineux sont parallèles. Par exemple le soleil

# Lumières ambiantes et directionnelles



# Lumières ambiantes et directionnelles

- Attention aux materials utilisés
- THREE.MeshBasicMaterial ne réagit pas à la lumière contrairement à THREE.MeshPhongMaterial

```
var directionalLight = new THREE.DirectionalLight(0xffffff, 1);
directionalLight.position = new THREE.Vector3(100,10,-50);
directionalLight.name='directional';
scene.add(directionalLight);
```

# Lumières ambiantes et directionnelles

```
var ambientLight = new  
THREE.AmbientLight(0x111111);  
ambientLight.name='ambient';  
scene.add(ambientLight);
```

# Exercice

- Créer une sphère en rotation sur elle-même
- Intégrer un orbital control
- Ajouter des lumières
- Appliquer les textures de Terre et de nuage (sur une sphère légèrement plus grande)
- Ajouter la lune avec sa texture en rotation autour de la Terre à l'échelle

WORK IN  
PROGRESS

# Peaufinons l'exercice

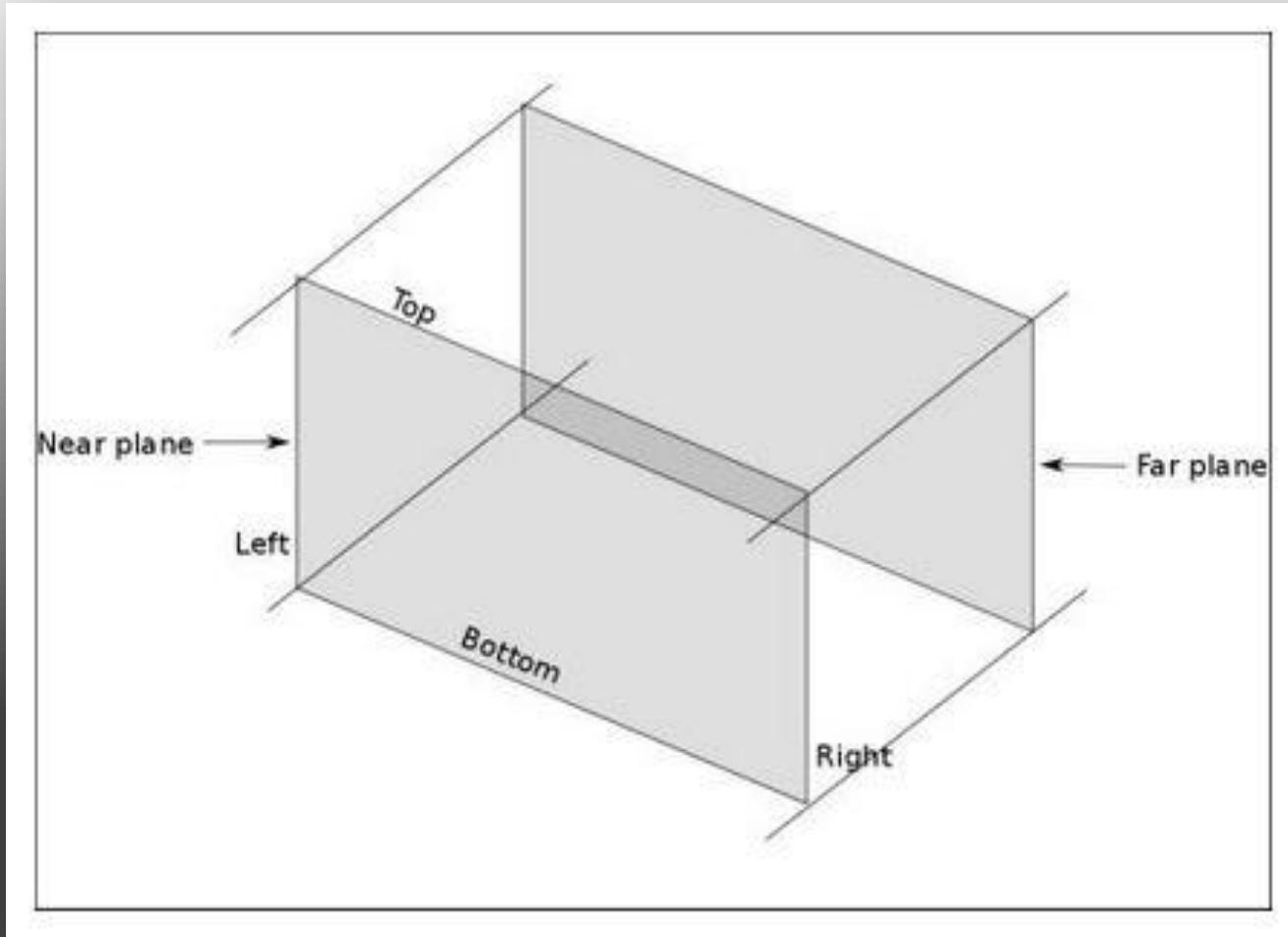
- Il serait intéressant d'ajouter un fond d'étoile
- 2 solutions :
  - Créer un plan vertical très très gros derrière
  - Utiliser une caméra orthogonale
- Utilisons la deuxième solution bien sûr

# Peaufinons l'exercice

- Avec une caméra orthogonale, tous les éléments de la scène ont toujours la même taille sans prendre en compte leur distance
- Un peu comme dans Baldur's Gate ou Civilization
- Pour déclarer une caméra orthogonale :

```
cameraBG = new THREE.OrthographicCamera(  
    -window.innerWidth,  
    window.innerWidth,  
    window.innerHeight,  
    -window.innerHeight,  
    -10000,  
    10000);  
cameraBG.position.z = 50;
```

# Caméra orthogonale



# Caméra orthogonale

```
// add background using a camera
cameraBG = new THREE.OrthographicCamera(-window.innerWidth,
window.innerWidth, window.innerHeight, -window.innerHeight, -10000
, 10000);
cameraBG.position.z = 50;
sceneBG = new THREE.Scene();

var materialColor = new THREE.MeshBasicMaterial({ map:
THREE.ImageUtils.loadTexture(
"../assets/textures/planets/starry_background.jpg"), depthTest:
false });
var bgPlane = new THREE.Mesh(new THREE.PlaneGeometry(1, 1),
materialColor);
bgPlane.position.z = -100;
bgPlane.scale.set(window.innerWidth * 2, window.innerHeight * 2, 1
);
sceneBG.add(bgPlane);
```

# EffectComposer

- Pour combiner les deux scènes, nous devons utiliser EffectComposer

```
<script src="../libs/EffectComposer.js"></script>
<script src="../libs/RenderPass.js"></script>
<script src="../libs/CopyShader.js"></script>
<script src="../libs/ShaderPass.js"></script>
<script src="../libs/MaskPass.js"></script>
```

# EffectComposer

```
// setup the composer steps
// first render the background
var bgPass = new THREE.RenderPass(sceneBG, cameraBG);
// next render the scene (rotating earth), without clearing the current output
var renderPass = new THREE.RenderPass(scene, camera);
renderPass.clear = false;
// finally copy the result to the screen
var effectCopy = new THREE.ShaderPass(THREE.CopyShader);
effectCopy.renderToScreen = true;

// add these passes to the composer
composer = new THREE.EffectComposer(renderer);
composer.addPass(bgPass);
composer.addPass(renderPass);
composer.addPass(effectCopy);
```

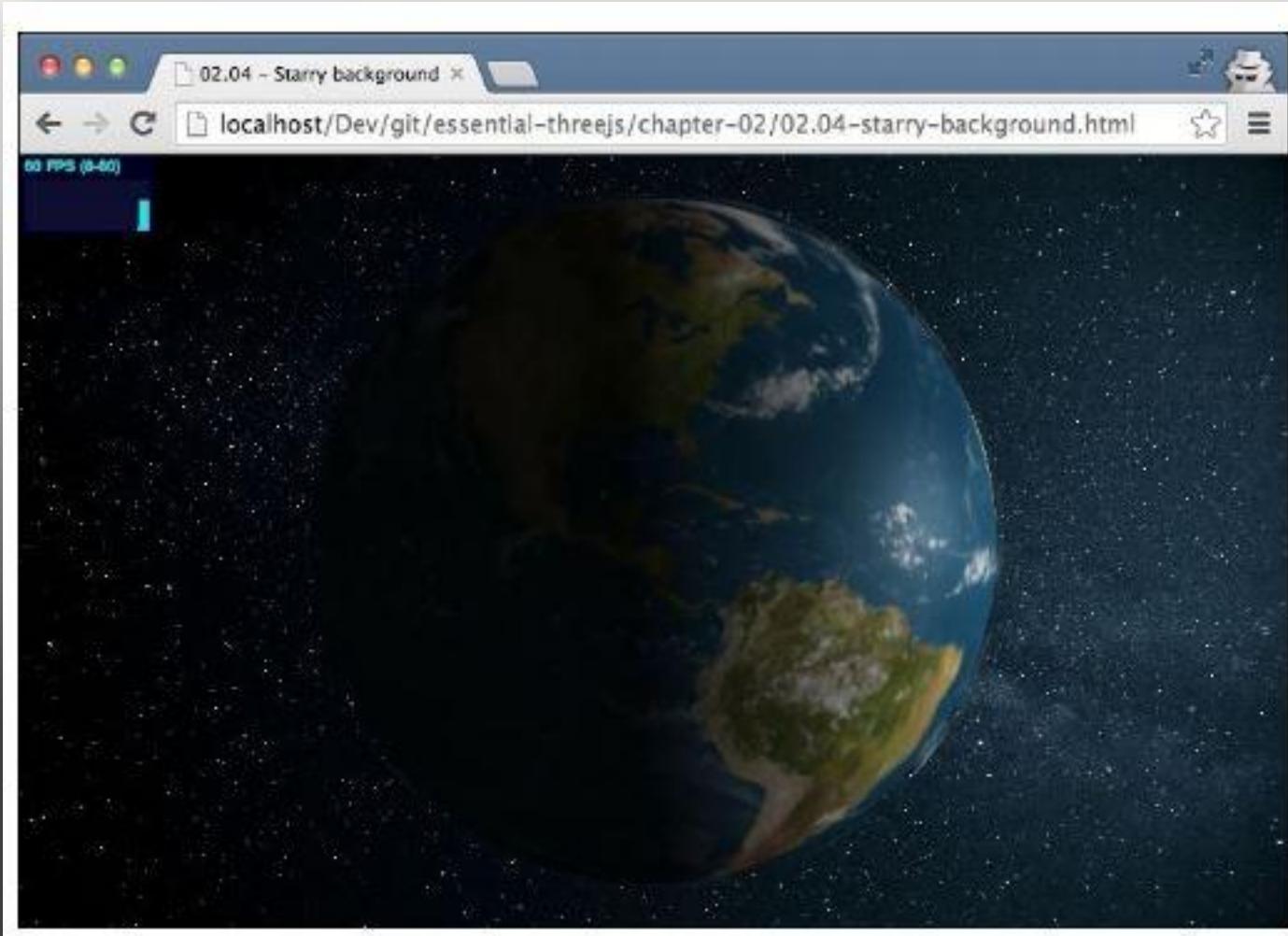
# EffectComposer

- On utilise d'abord deux objets THREE.RenderPass
- Ils permettent de rendre une scène avec une caméra sans afficher le résultat à l'écran
- Le comportement par défaut de THREE.RenderPass est de vider l'affichage en précédent avant de faire le rendu. Il est passé à false pour que l'on puisse voir le rendu du fond étoilé

# Touche finale

```
// and render the scene, renderer shouldn't autoclear,  
// we let the composer steps do that themselves  
// rendering is now done through the composer, which  
// executes the render steps  
renderer.autoClear = false;  
composer.render();
```

# La Terre

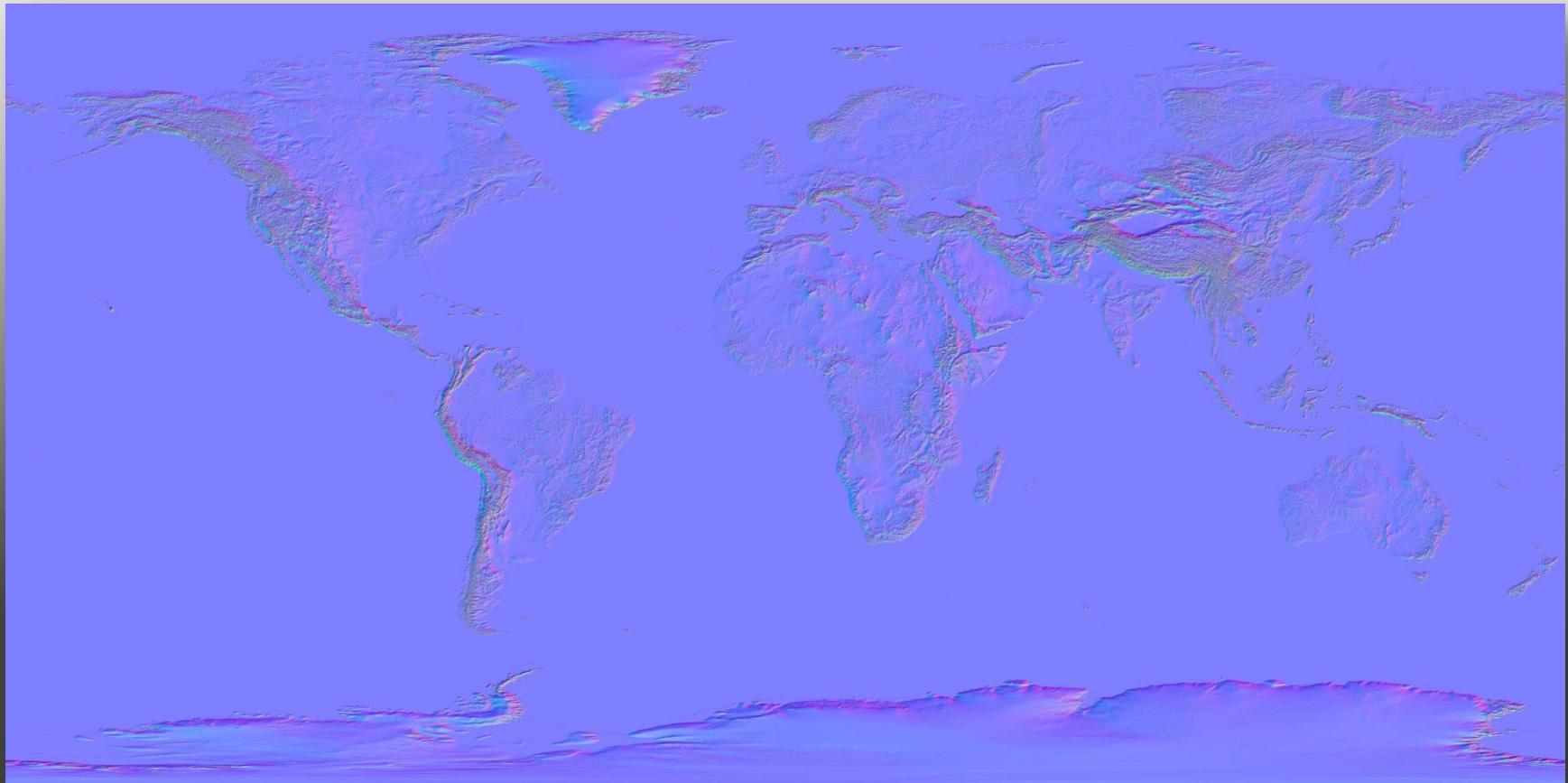


# Textures avancées

- Pour améliorer le rendu de notre Terre, nous allons utiliser une « normal map » pour simuler la hauteur du relief
- Une « normal map » ne stocke pas l'intensité de chaque pixel mais plutôt son « orientation »
- Chaque pixel de la « normal map » est un vecteur qui définit la direction
- Il suffit de modifier la propriété `normalMap` du material

```
var normalMap = THREE.ImageUtils.loadTexture(  
  ".../assets/textures/planets/earth_normalmap_flat4k.jpg");  
earthMaterial.normalMap = normalMap;
```

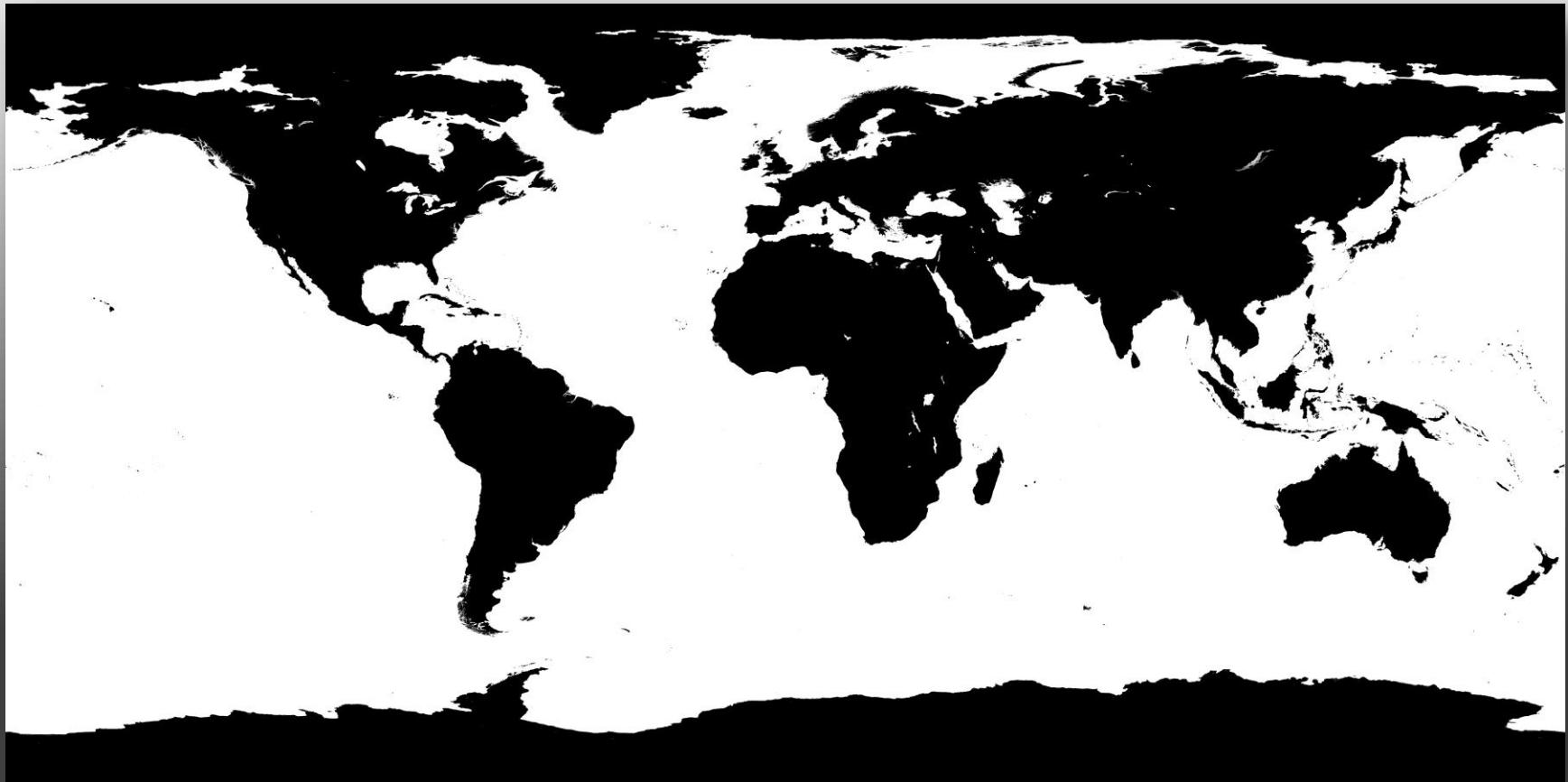
# Textures avancées



# Textures avancées

- Pour améliorer encore le rendu de notre Terre, nous pouvons utiliser une « specular map »
- Les océans reflètent bien la lumière tandis que les continents non
- La couleur blanche désigne les zones qui reflètent complètement la lumière

# Textures avancées



# Ajouter des informations 2D

- Notre globe peut nous servir pour une multitude de visualisation de données
- A condition que l'on puisse mapper une carte dynamique sur un mesh
- WebGL nous donne la possibilité d'utiliser un <canvas> comme une texture pour notre globe

# Ajouter des informations 2D

- L'important est de localiser une bonne API ou CSV : population mondiale, localisation des aéroports, foyers d'Ebola, etc...
- Les coordonnées GPS peuvent être traduites en 2D sur « une carte du monde rectangulaire »
- En HTML5, on peut dessiner des ronds sur un canvas

# Ajouter des informations 2D



# Exercice

- Utiliser la source de données JSON sur le nucléaire civil pour afficher des barres en 3D (une barre nombre de centrale, une barre puissance totale)
- Utiliser les coordonnées GPS pour placer les barres sur le globe

WORK IN  
PROGRESS

# Exercice

- Créer une scène composée de 5 cubes verts de taille 10 et d'un cube rouge de taille 6

WORK IN  
PROGRESS

# THREE.Raycaster

- Grâce à l'objet THREE.Raycaster, on peut lancer un rayon depuis un point spécifique dans une direction et obtenir la liste des objets qui « intersectent » le rayon
- C'est très souvent utilisé pour sélectionner des objets
- Mais aussi pour la détection de collision

# THREE.Raycaster

```
var projector = new THREE.Projector();
function onDocumentMouseDown( event ) {

    event.preventDefault();

    var vector = new THREE.Vector3( ( event.clientX / window.innerWidth
        ) * 2 - 1, - ( event.clientY / window.innerHeight ) * 2 + 1, 0.5 );
    projector.unprojectVector(vector,camera);

    var raycaster = new THREE.Raycaster(camera.position,vector.sub(
        camera.position).normalize() );
    var intersects = raycaster.intersectObjects( collidableMeshList );

    if ( intersects.length > 0 ) {
        intersects[ 0 ].object.material.transparent=true;
        intersects[ 0 ].object.material.color=new THREE.Color(0x0000ff);
    }
}

window.onmousedown=onDocumentMouseDown;
```

# THREE.Raycaster

- On calcule les coordonnées NDC (comprises entre -1 et 1) à partir du clic sur le canvas. Le but est d'obtenir des coordonnées indépendantes de la taille et forme du canvas

Device Coordinate  $\longrightarrow$  Normalized Device Coordinate  $\longrightarrow$  World Coordinate

- THREE.Projector va convertir les coordonnées de l'écran (vector) en coordonnées de la scène 3D grâce à la méthode « unprojectVector »
- THREE.Raycaster va lancer un rayon depuis la caméra dans la direction cliquée
- En utilisant la méthode raycaster.intersectObjects on détermine si le rayon coupe un objet
- collidableMeshList est un tableau qui contient les meshs de la scène
- Le premier élément du tableau intersects est le plus proche

# Détection des collisions

```
function detectCollision(objet, directionVector) {  
  
    for (var vertexIndex = 0; vertexIndex < objet.geometry.vertices.length;  
        vertexIndex++)  
    {  
        var localVertex = objet.geometry.vertices[vertexIndex].clone();  
        var globalVertex = localVertex.applyMatrix4( objet.matrix );  
        .....  
        var ray = new THREE.Raycaster( globalVertex, directionVector.clone() ).  
            normalize() ;  
        var collisionResults = ray.intersectObjects( collidableMeshList );  
        if ( collisionResults.length > 0 && collisionResults[0].distance <  
            directionVector.length() )  
        {  
            return true;  
        }  
    }  
  
    return false;  
}
```

# Contrôle du clavier

```
function checkKey(e) {  
    e = e || window.event;  
  
    var factor = 0.3;  
  
    if (e.keyCode == '37') {  
        // left  
        move(new THREE.Vector3(0, 0, factor));  
    }  
    if (e.keyCode == '38') {  
        // up  
        move(new THREE.Vector3(-factor, 0, 0));  
    }  
    if (e.keyCode == '39') {  
        // right  
        move(new THREE.Vector3(0, 0, -factor));  
    }  
    else if (e.keyCode == '40') {  
        // down  
        move(new THREE.Vector3(factor, 0, 0));  
    }  
  
    e.preventDefault();  
}  
}
```

# Contrôle du clavier

```
function move(direction) {
    if (!selectedObject) {
        return;
    }

    if (!detectCollision(selectedObject,direction)) {
        selectedObject.position.x += direction.x;
        selectedObject.position.y += direction.y;
        selectedObject.position.z += direction.z;
    }
}
```

# Exercice

- Dans l'exercice précédent,
- Sélectionner un cube pour le bouger
- Implémenter la gestion des collisions
- Implémenter le contrôle clavier

WORK IN  
PROGRESS

# Créer un terrain depuis scratch

- On ne peut pas toujours utiliser des formes géométriques pour tout dessiner
- Notamment si on veut dessiner un terrain  
`create3DTerrain(width, depth, spacingX, spacingY, height)`
  - Créer les vertices
  - Créer les faces
  - Créer le Mesh et l'ajouter à la scène

# Créer un terrain depuis Math.random()

- **Création des vertices**

```
var geometry = new THREE.Geometry();
```

```
var vertex = new THREE.Vector3(x, y, z);
```

```
geometry.vertices.push(vertex);
```

# Créer un terrain depuis Math.random()

- Création des faces

```
//a, b, c les indices du tableau de vertices
```

```
var face = new THREE.Face3(a, b, c);
```

```
face.color = new THREE.Color(0x262626);
```

```
geometry.faces.push(face);
```

# Créer un terrain depuis Math.random()

- ## • Création du Mesh

```
var mat = new THREE.MeshPhongMaterial();
var groundMesh = new THREE.Mesh(geometry,
    mat);
scene.add(groundMesh);
```

# Créer un terrain depuis Math.random()

```
// first create all the individual vertices
var geometry = new THREE.Geometry();
for (var z = 0 ; z < depth ; z++) {
    for (var x = 0 ; x < width ; x++) {
        var vertex = new THREE.Vector3(x*spacingX,
            Math.random()*height,z*spacingZ);
        geometry.vertices.push(vertex);
    }
}
```

# Créer un terrain depuis Math.random()

```
// next we need to define the faces. Which are triangles
// we create a rectangle between four vertices, and we do
// that as two triangles.
for (var z = 0 ; z < depth-1 ; z++) {
    for (var x = 0 ; x < width-1 ; x++) {
        // we need to point to the position in the array
        // a - - b
        // |   x   |
        // c - - d
        var a = x + z*width;
        var b = (x+1) + (z * width);
        var c = x + ((z+1) * width);
        var d = (x+1) + ((z+1) * width);

        var face1 = new THREE.Face3(b, a, c );
        var face2 = new THREE.Face3(c ,d, b );

        face1.color = new THREE.Color(scale(getHighPoint(geometry, face1)).hex());
        face2.color = new THREE.Color(scale(getHighPoint(geometry, face2)).hex())
        geometry.faces.push(face1);
        geometry.faces.push(face2);
    }
}
```

# Créer un terrain depuis Math.random()

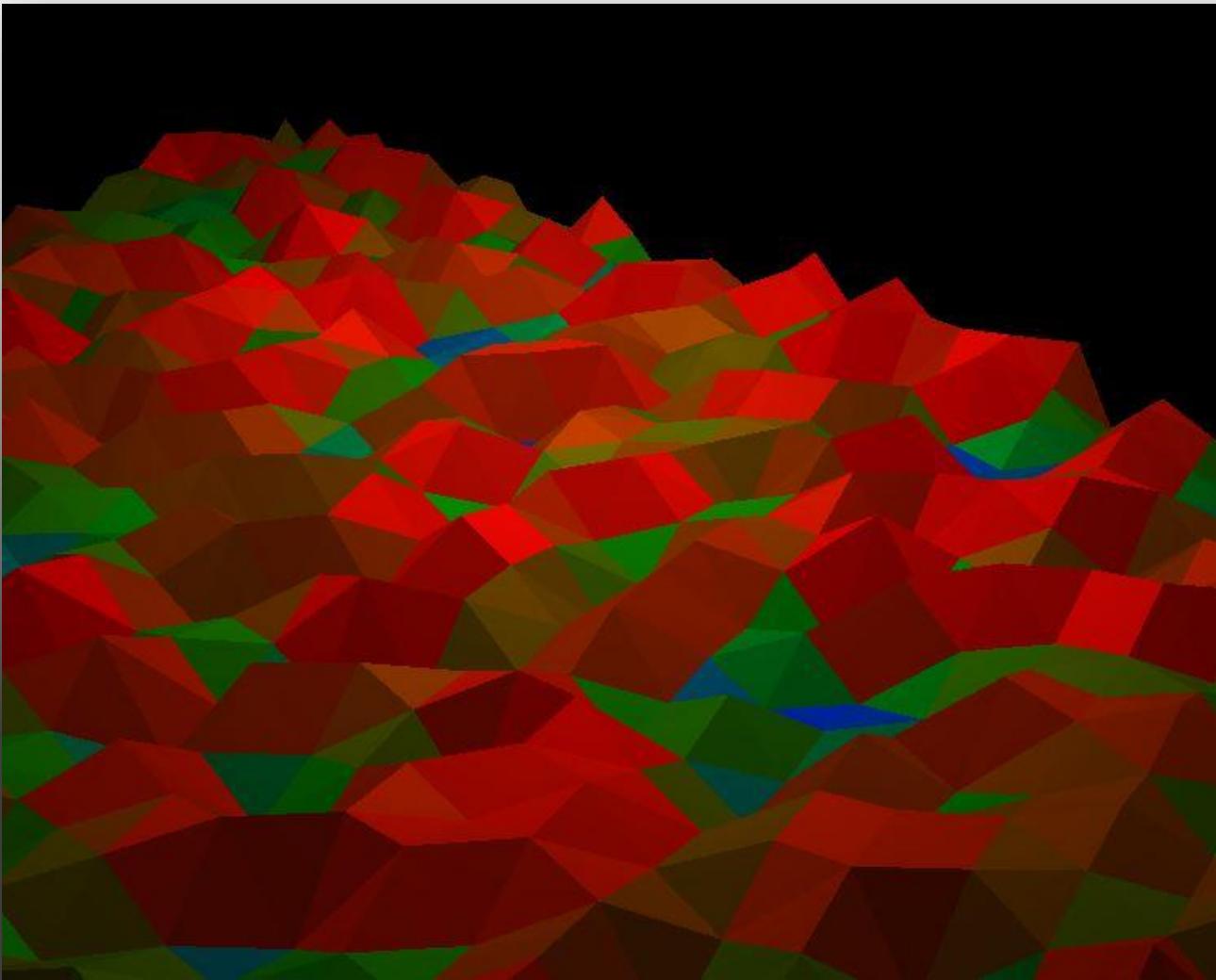
```
// compute the normals
geometry.computeVertexNormals(true);
geometry.computeFaceNormals();

// setup the material
var mat = new THREE.MeshPhongMaterial();
mat.vertexColors = THREE.FaceColors;
mat.shading = THREE.NoShading;

// create the mesh
var groundMesh = new THREE.Mesh(geometry,mat);
groundMesh.translateX(-width/1.5);
groundMesh.translateZ(-depth/4);
groundMesh.name = 'terrain';

scene.add(groundMesh);
```

# Un résultat pas si naturel

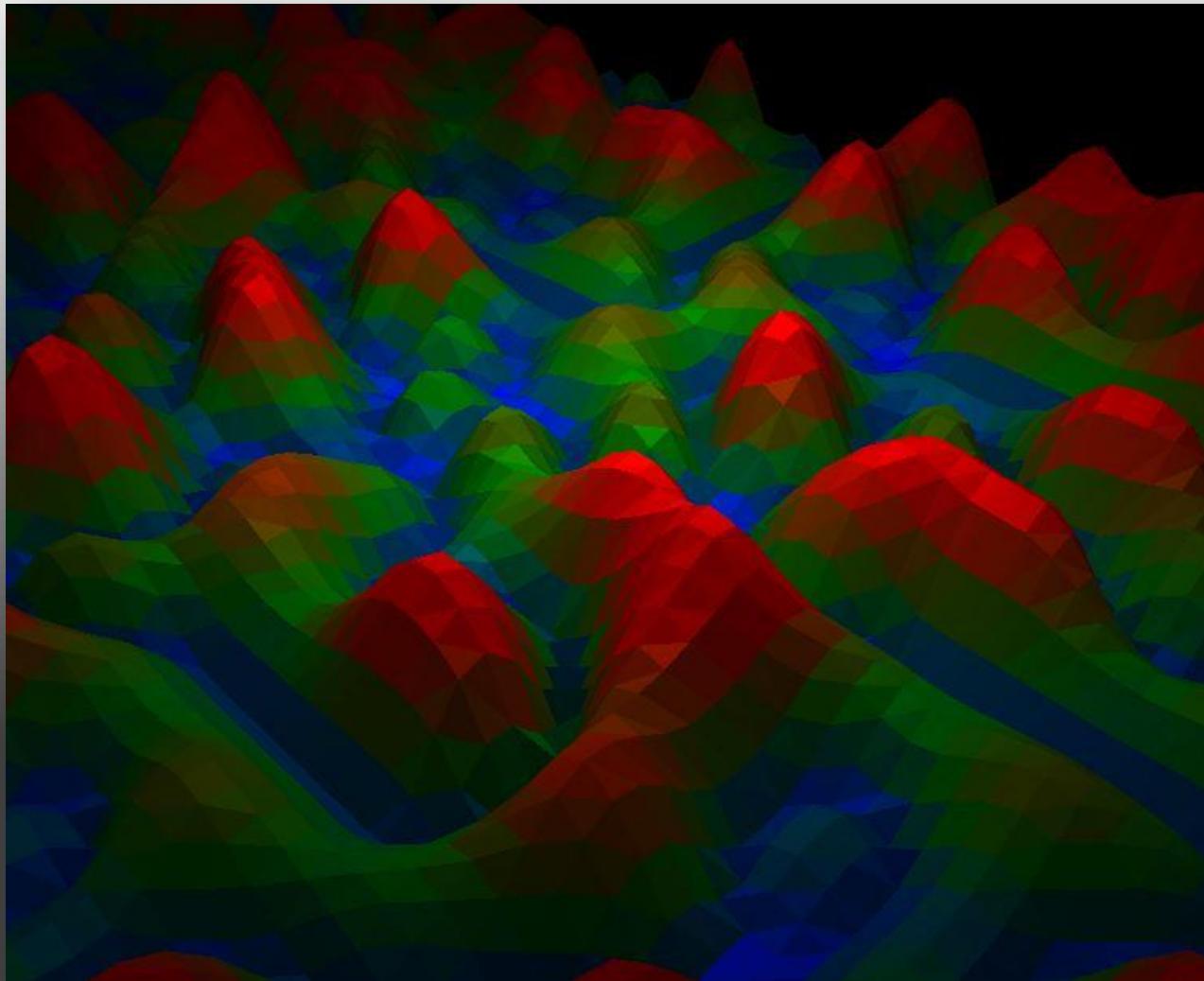


# Utilisation de Perlin

- Utilisation de la librairie Perlin
- Générateur de bruit
- Permet d'obtenir un effet beaucoup plus naturel qu'un aléa total

```
var yValue = Math.abs(noise.perlin2(x/8, z/8) * height*2);
```

# Un résultat beaucoup mieux



# Créer un terrain depuis une géométrie Plane

```
function create3DTerrain(height) {  
  
    var plane = new THREE.PlaneGeometry(120, 100, 100, 100);  
  
    for (var i = 0; i < plane.vertices.length ; i++) {  
        var vertex = plane.vertices[i];  
        var value = Math.abs(noise.perlin2(vertex.x / 10, vertex.y / 10, 0));  
        vertex.z = value * 10;  
    }  
  
    //Math.random() * height;  
    // compute the normals  
    plane.computeVertexNormals(true);  
    plane.computeFaceNormals();  
  
    // setup the material  
    var mat = new THREE.MeshPhongMaterial();  
    mat.map = THREE.ImageUtils.loadTexture("../assets/textures/wood_1-1024x1024.png");  
}
```

# Exercice

- Générer un terrain de taille 80 \* 80 grâce à Perlin
- Plus la face est basse plus la couleur est verte
- Plus la face est haute plus la couleur est rouge
- Utiliser THREE.FlyControls

WORK IN  
PROGRESS

# Fusionner des géométries

```
//A capsule mesh
var cyl = new THREE.CylinderGeometry(2, 2, 6);
var top = new THREE.SphereGeometry(2);
var bot = new THREE.SphereGeometry(2);

var topMesh = new THREE.Mesh(top);
var botMesh = new THREE.Mesh(bot);
topMesh.position.y = 3
botMesh.position.y = -3

THREE.GeometryUtils.merge(cyl, topMesh);
THREE.GeometryUtils.merge(cyl, botMesh);
```

# Gérer la physique avec Physijs



# Gérer la physique avec Physijs

- Construit sur ammo.js
- Les calculs de simulation sont effectués sur un thread différent grâce aux « web workers » pour conserver la performance
- Supporte de multiples formes
- Gère la friction et la restitution
- Intègre la détection de collision
- Système de contrainte

# Installer Physijs

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

<script type="text/javascript">
Physijs.scripts.worker = '../libs/physijs_worker.js';
Physijs.scripts.ammo = '../libs/ammo.js';

var scene = new Physijs.Scene;
scene.setGravity(new THREE.Vector3(0, -50, 0));

</script>
```

# Gérer la physique avec Physijs

```
// Materials
ground_material = Physijs.createMaterial(
    new THREE.MeshPhongMaterial({ map: THREE.ImageUtils.loadTexture(
        '../assets/textures/general/floor-wood.jpg') }),
    .9, // high friction
    .6 // low restitution
);

// Ground
ground = new Physijs.BoxMesh(
    new THREE.CubeGeometry(60, 1, 130),
    ground_material,
    0 // mass
);
ground.receiveShadow = true;
```

# Exercice

- Utiliser le terrain généré toute à l'heure
- Faire une rotation du terrain pour l'incliner
- Lâcher des sphères sur le terrain pour constater le fonctionnement de Physijs

WORK IN  
PROGRESS

# Charger un model



# Charger un model

```
var loader = new THREE.JSONLoader();
loader.load('../assets/models/ogre/ogre.js', function (geometry, mat) {
    geometry.computeMorphNormals();

    var mat = new THREE.MeshLambertMaterial(
        {
            map: THREE.ImageUtils.loadTexture(
                '../assets/models/ogre/skins/skin.jpg'),
            morphTargets: true, morphNormals: true
        });
    mesh = new THREE.MorphAnimMesh(geometry, mat);

    mesh.parseAnimations();

    mesh.playAnimation('stand', 10);

    scene.add(mesh);
});
```

# Charger un model

```
/**  
 * Called when the scene needs to be rendered.  
 * for future renders  
 */  
function render() {  
    // render using requestAnimationFrame  
    var delta = clock.getDelta();  
  
    if (mesh) {  
        mesh.updateAnimation(delta * 1000);  
    }  
  
    requestAnimationFrame(render);  
    renderer.render(scene, camera);  
  
    scene.simulate(undefined, 1);  
}
```

# Exercice

- Charger le model « ogre » animé dans notre scène
  - Utiliser Physijs pour gérer les collisions avec le modèle
  - Ajouter les contrôles pour bouger le modèle en jouant les animations
  - Positionner la caméra de manière à voir le modèle
- WORK IN PROGRESS*

<http://oos.moxiecode.com/blog/index.php/experiments/javascript-webgl/>

# Exercice

- On peut contrôler Ogro grâce au clic
- Ajouter des ennemis
- Ajouter des murs
- Les ennemis peuvent voir ou non Ogro et se déplacer vers lui pour l'attaquer
- Ogro peut frapper pour détruire les ennemis
- Générer automatiquement la scène
- les ennemis

WORK IN  
PROGRESS

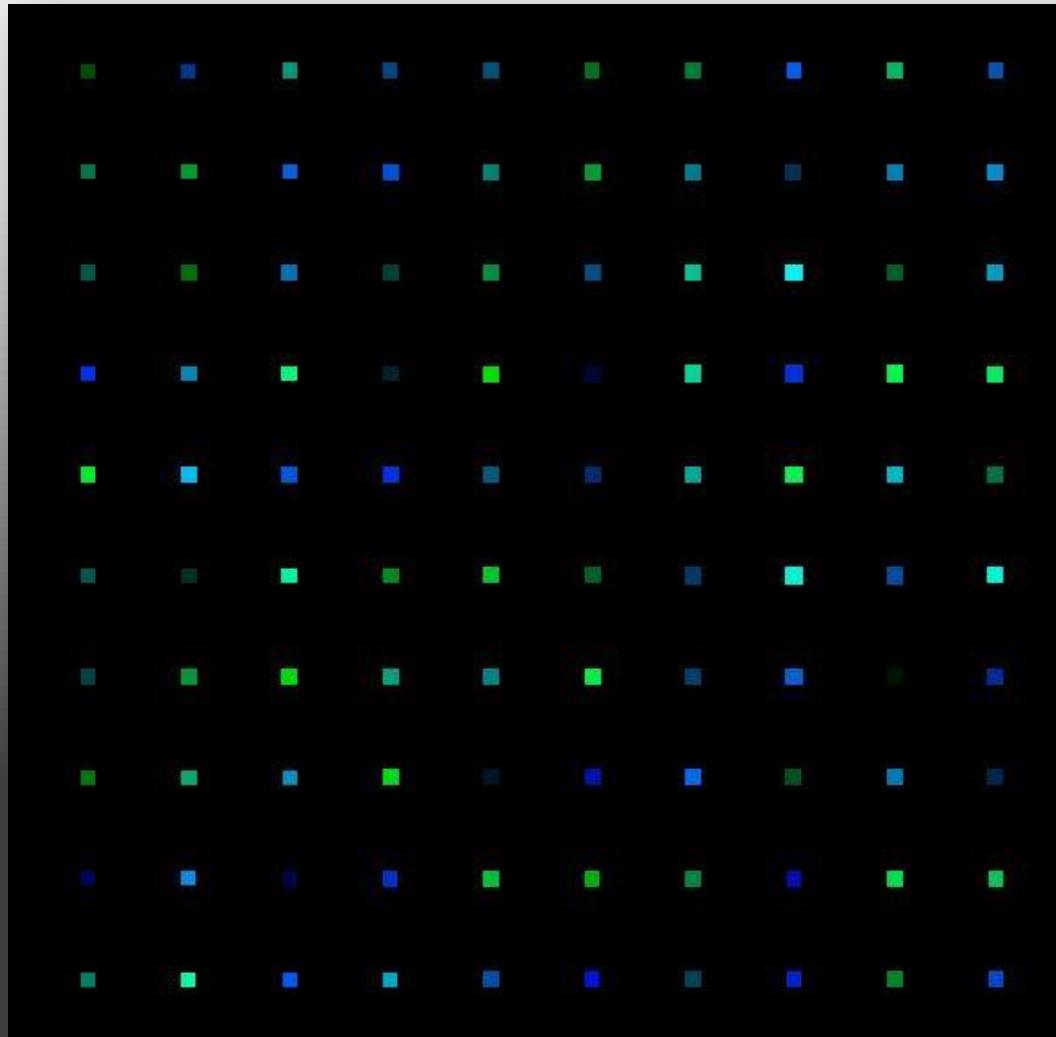
# Système de particules

- Avec les particules, on peut très facilement simuler des petits objets comme la pluie ou la neige
- On peut également créer de toute pièce une géométrie à partir des particules et contrôler ces particules séparément

# Créer des particules

```
function createParticles() {  
  
    var geom = new THREE.Geometry();  
    var material = new THREE.ParticleBasicMaterial({size: 4, vertexColors: true, color: 0xffffffff});  
  
    for (var x = -5; x < 5; x++) {  
        for (var y = -5; y < 5; y++) {  
            var particle = new THREE.Vector3(x * 10, y * 10, 0);  
            geom.vertices.push(particle);  
            geom.colors.push(new THREE.Color(Math.random() * 0x00ffff));  
        }  
    }  
  
    var system = new THREE.ParticleSystem(geom, material);  
    scene.add(system);  
}
```

# Créer des particules



# THREE.ParticleBasicMaterial

```
var material = new THREE.ParticleBasicMaterial({  
    size: size,  
    transparent: transparent,  
    opacity: opacity,  
    vertexColors: vertexColors,  
  
    sizeAttenuation: sizeAttenuation,  
    color: color});
```

# THREE.ParticleBasicMaterial

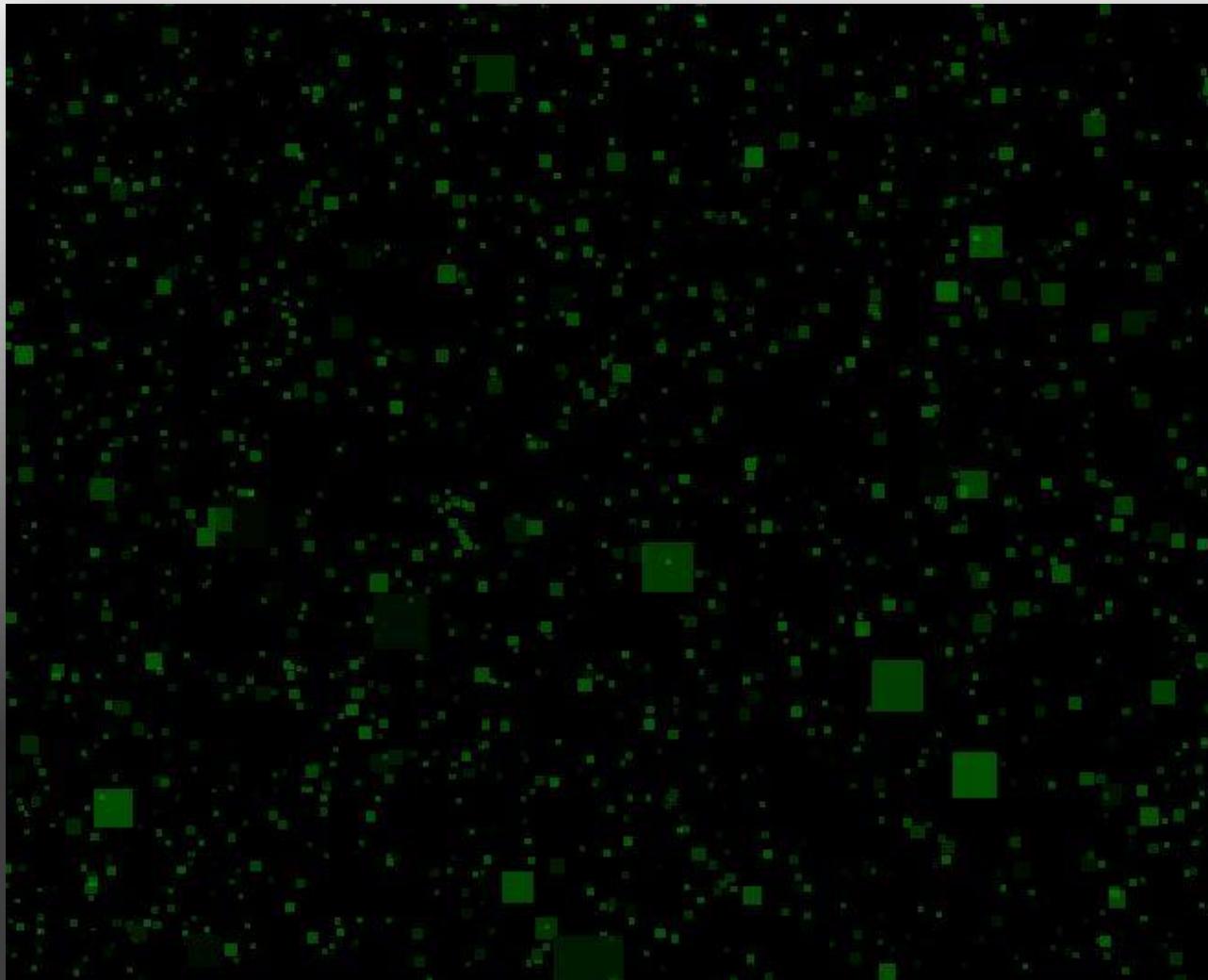
map	Permet d'appliquer une texture aux particules
size	Taille de la particule
sizeAnnotation	Si défini à <b>false</b> , toutes les particules ont la même taille sans rapport avec la distance à la caméra. Par défaut à <b>true</b> .
vertexColors	Normalement toutes les particules ont la même couleur. En définissant la propriété à <b>true</b> , les couleurs du tableau <b>colors</b> de la géométrie seront utilisées.
opacity	Facteur d'opacité
transparent	Active l'opacité
blending	Le blend à utiliser <code>THREE.NormalBlending = 0; THREE.AdditiveBlending = 1;</code> <code>THREE.SubtractiveBlending = 2; THREE.MultiplyBlending = 3;</code> <code>THREE.AdditiveAlphaBlending = 4;</code>
fog	Détermine si le fog affecte les particules. Par défaut à <b>true</b> .

# Exercice

- Créer un système de 15000 particules en faisant varier la couleur pour chacune
- Appliquer une rotation au système sur x et z

WORK IN  
PROGRESS

# Créer des particules

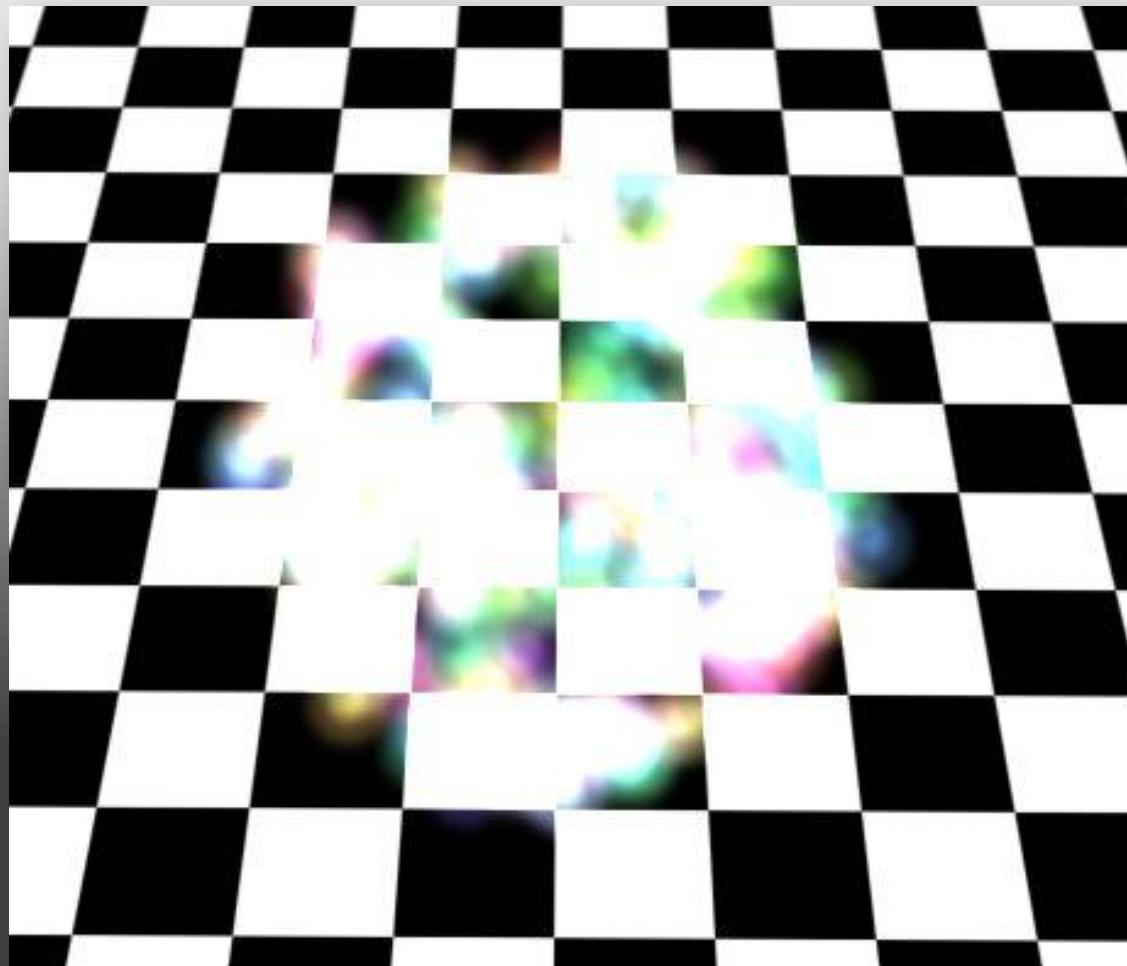


# Exercice

- Appliquer une texture sur la particule
- Activer le blending THREE.AdditiveBlending

WORK IN  
PROGRESS

# Créer des particules



# Exercice

- Appliquer une texture de goutte d'eau
- Créer une scène de pluie
- Créer de la fumée
- Activer le fog

WORK IN  
PROGRESS

# Post processing

- Three.js nous permet de générer le rendu 3D d'une scène
- Il est possible d'ajouter un ou plusieurs effets de post processing à ce rendu
- Pour créer :
  - Un effet N&B
  - Un effet de film TV
  - Un effet de flou
  - ...

# Configurer le post processing

- Créer un objet EffectComposer
- Lui ajouter les différentes phases d'effet
- Modifier notre code pour utiliser EffectComposer pour rendre la scène
- Modifier la fonction render pour afficher le rendu de EffectComposer

# FilmPass

```
<script src="../libs/postprocessing/EffectComposer.js"></script>
<script src="../libs/postprocessing/ShaderPass.js"></script>
<script src="../libs/shaders/CopyShader.js"></script>
<script src="../libs/postprocessing/MaskPass.js"></script>
<script src="../libs/postprocessing/FilmPass.js"></script>
<script src="../libs/shaders/FilmShader.js"></script>
<script src="../libs/postprocessing/RenderPass.js"></script>
```

# FilmPass

```
var renderPass = new THREE.RenderPass(scene, camera);
var effectFilm = new THREE.FilmPass(0.8, 0.325, 256, false);
effectFilm.renderToScreen = true;

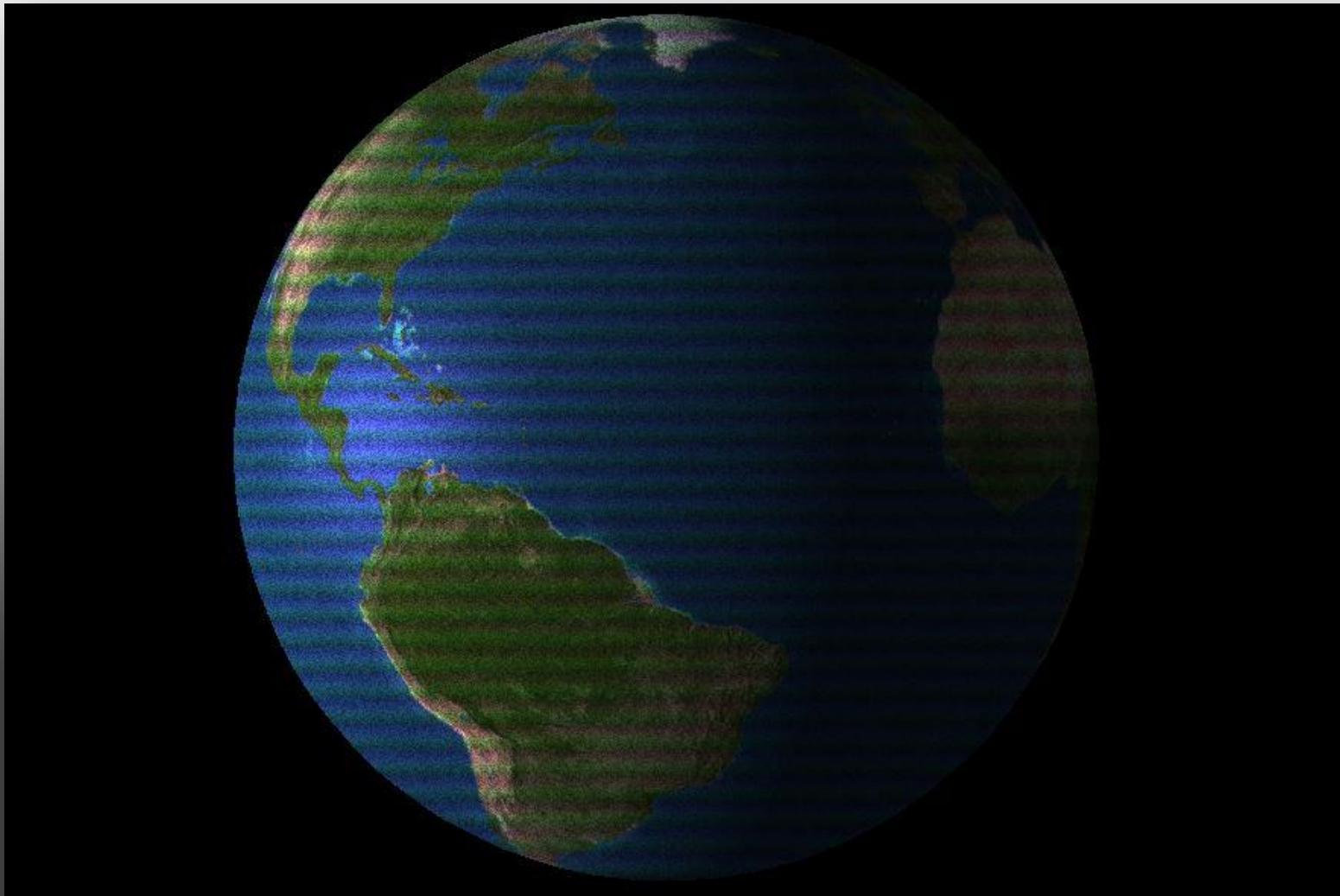
var composer = new THREE.EffectComposer(webGLRenderer);
composer.addPass(renderPass);
composer.addPass(effectFilm);

}

function render() {
    var delta = clock.getDelta();

    requestAnimationFrame(render);
    webGLRenderer.render(scene, camera);
    composer.render(delta);
//
```

# FilmPass



# Différents post processing

BloomPass	Les zones éclairées « coulent » sur les zones sombres. Effet d'éblouissement.
DotScreenPass	Ajoute un calque de points noirs
FilmPass	Simule un affichage TV
MaskPass	Applique un masque sur le rendu, les passes suivantes sont appliquées sur la zone masquée
RenderPass	Rend une scène en utilisant une caméra
SavePass	Fait une copie du rendu courant
ShaderPass	Applique un shader particulier
TexturePass	Stocke le rendu comme une texture que l'on peut utiliser pour EffectComposer

# Multi post processing

```
var renderPass = new THREE.RenderPass(scene, camera);
var effectCopy = new THREE.ShaderPass(THREE.CopyShader);
effectCopy.renderToScreen = true;

var bloomPass = new THREE.BloomPass(3, 25, 5.0, 256);
var effectFilm = new THREE.FilmPass(0.8, 0.325, 256, false);
effectFilm.renderToScreen = true;

var dotScreenPass = new THREE.DotScreenPass();

// basic renderer that renders the scene, and uses the
// effectCopy shader to output the image to the defined
// rendertarget.
var composer = new THREE.EffectComposer(webGLRenderer);
composer.addPass(renderPass);
composer.addPass(effectCopy);

// we use a texture pass to pass the rendered output to
// a texture, so we can reuse it
var renderScene = new THREE.TexturePass(composer.renderTarget2);
```

# Multi post processing

```
// we use a texture pass to pass the rendered output to
// a texture, so we can reuse it
var renderScene = new THREE.TexturePass(composer.renderTarget2);

// lower left corner
var composer1 = new THREE.EffectComposer(webGLRenderer);
composer1.addPass(renderScene);
composer1.addPass(dotScreenPass);
composer1.addPass(effectCopy);

// lower right corner
var composer2 = new THREE.EffectComposer(webGLRenderer);
composer2.addPass(renderScene);
composer2.addPass(effectCopy);

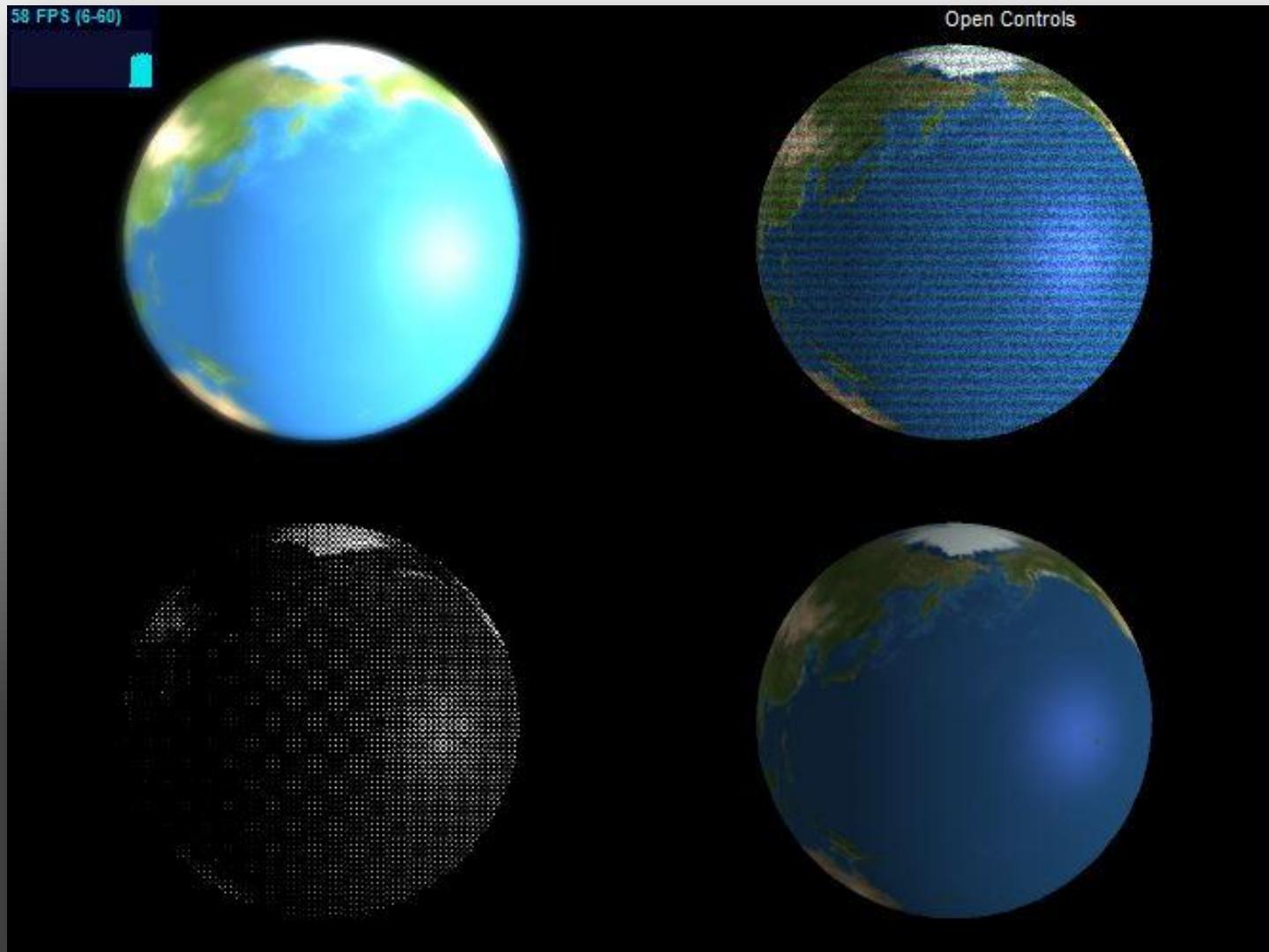
// upper left corner
var composer3 = new THREE.EffectComposer(webGLRenderer);
composer3.addPass(renderScene);
composer3.addPass(bloomPass);
composer3.addPass(effectCopy);

// upper right corner
var composer4 = new THREE.EffectComposer(webGLRenderer);
composer4.addPass(renderScene);
composer4.addPass(effectFilm);
```

# Multi post processing

```
webGLRenderer.autoClear = false;  
webGLRenderer.clear();  
  
webGLRenderer.setViewport(0, 0, 2 * halfWidth, 2 * halfHeight);  
composer.render(delta);  
  
webGLRenderer.setViewport(0, 0, halfWidth, halfHeight);  
composer1.render(delta);  
  
webGLRenderer.setViewport(halfWidth, 0, halfWidth, halfHeight);  
composer2.render(delta);  
  
webGLRenderer.setViewport(0, halfHeight, halfWidth, halfHeight);  
composer3.render(delta);  
  
webGLRenderer.setViewport(halfWidth, halfHeight, halfWidth, halfHeight);  
composer4.render(delta);
```

# Post processing

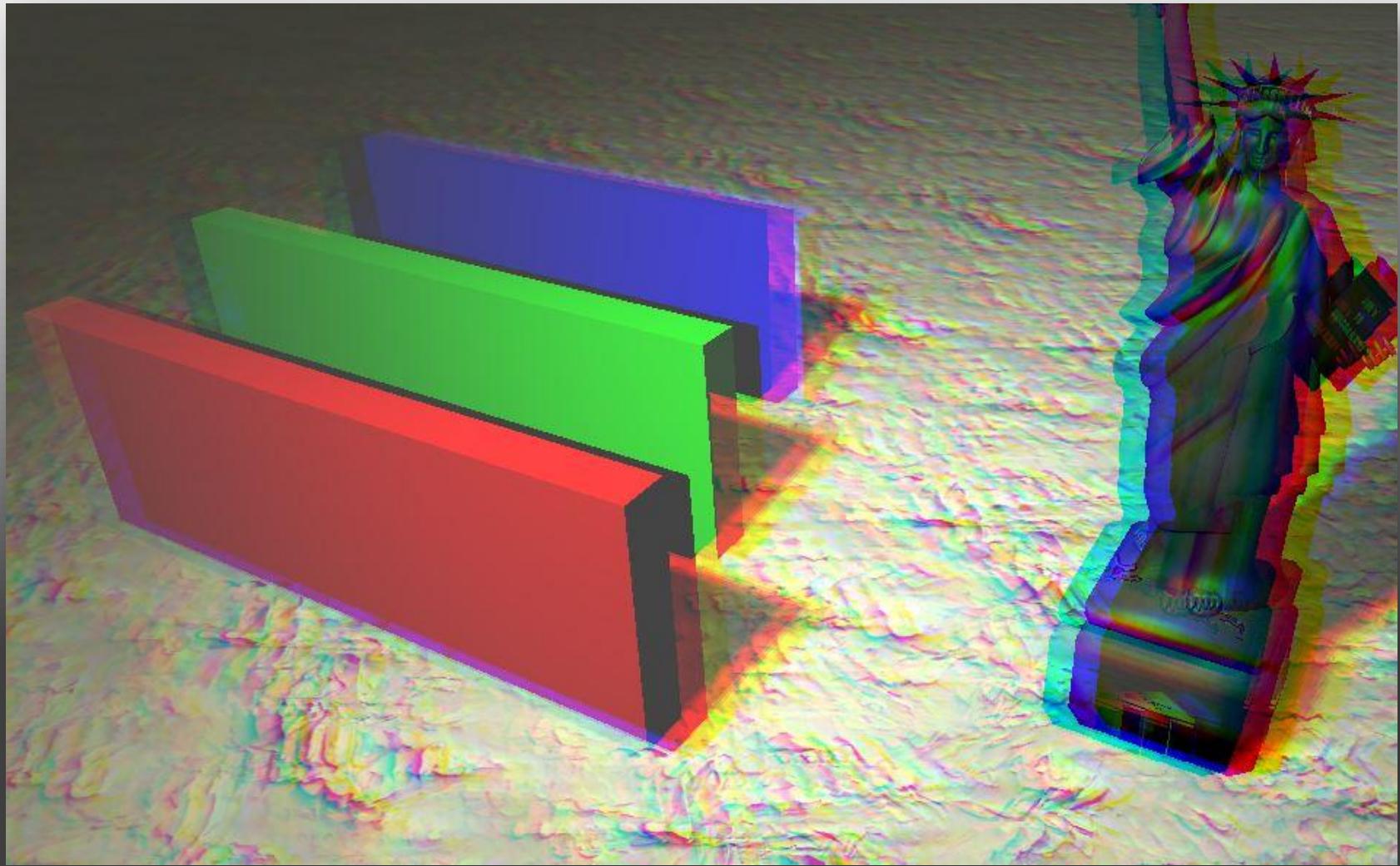


La 3D avec WebGL &  
three.js

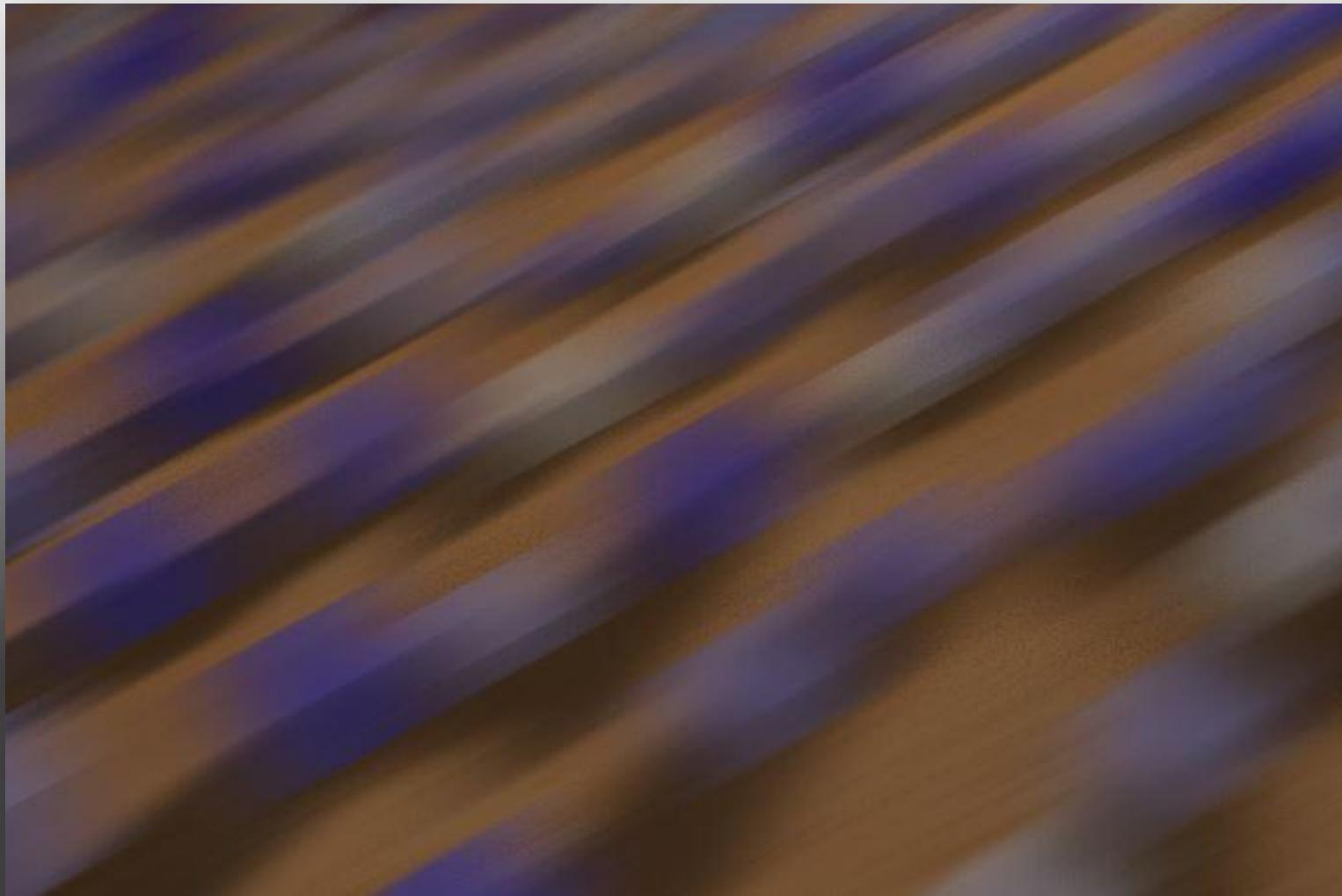
# ShaderPass

Effets simples	MirrorShader, HueSaturationShader, VignetteShader, ColorCorrectionShader, RGBShiftShader, BrightnessContrastShader, ColorifyShader, SepiaShader
Effets de flou	HorizontalBlurShader, VerticalBlurShader, HorizontalTiltShiftShader, VerticalTiltShiftShader, TriangleBlurShader
Effets avancés	BleachBypassShader, EdgeShader (détection des angles), FXAAShader (anti alias), FocusShader,

# RGB Shift



# TriangleBlurShader



# EdgeShader



# Exercice

- Appliquer le post traitement FilmPass

WORK IN  
PROGRESS

# CustomShader

- De nombreux effets sont disponibles en ligne
- Mais on peut en construire de toute pièce en créant des shaders : vaste sujet !
- On a besoin d'implémenter 2 composants :
  - Le vertexShader (défini la position de chaque vertice)
  - Le fragmentShader (défini la couleur de chaque pixel)
- Pour un shader de post processing, nous avons juste besoin d'un fragmentShader (on utilisera le vertexShader par défaut)

# CustomShader

```
THREE.CustomGrayScaleShader = {
    uniforms: {
        "tDiffuse": { type: "t", value: null },
        "rPower": { type: "f", value: 0.2126 },
        "gPower": { type: "f", value: 0.7152 },
        "bPower": { type: "f", value: 0.0722 }
    },

    vertexShader: [
        "varying vec2 vUv;",
        "void main() {",
        "vUv = uv;",
        "gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );",
        "}"
    ].join("\n"),
}
```

# CustomShader

```
fragmentShader: [
    // pass in our custom uniforms
    "uniform float rPower;",
    "uniform float gPower;",
    "uniform float bPower;",

    // pass in the image/texture we'll be modifying
    "uniform sampler2D tDiffuse;",

    // used to determine the correct texel we're working on
    "varying vec2 vUv;",

    // executed, in parallel, for each pixel
    "void main() {",
        // get the pixel from the texture we're working with (called a texel)
        "vec4 texel = texture2D( tDiffuse, vUv );",
        // calculate the new color
        "float gray = texel.r*rPower + texel.g*gPower + texel.b*bPower;",

        // return this new color
        "gl_FragColor = vec4( vec3(gray), texel.w );",
    }
]

].join("\n")
```

# CustomShader

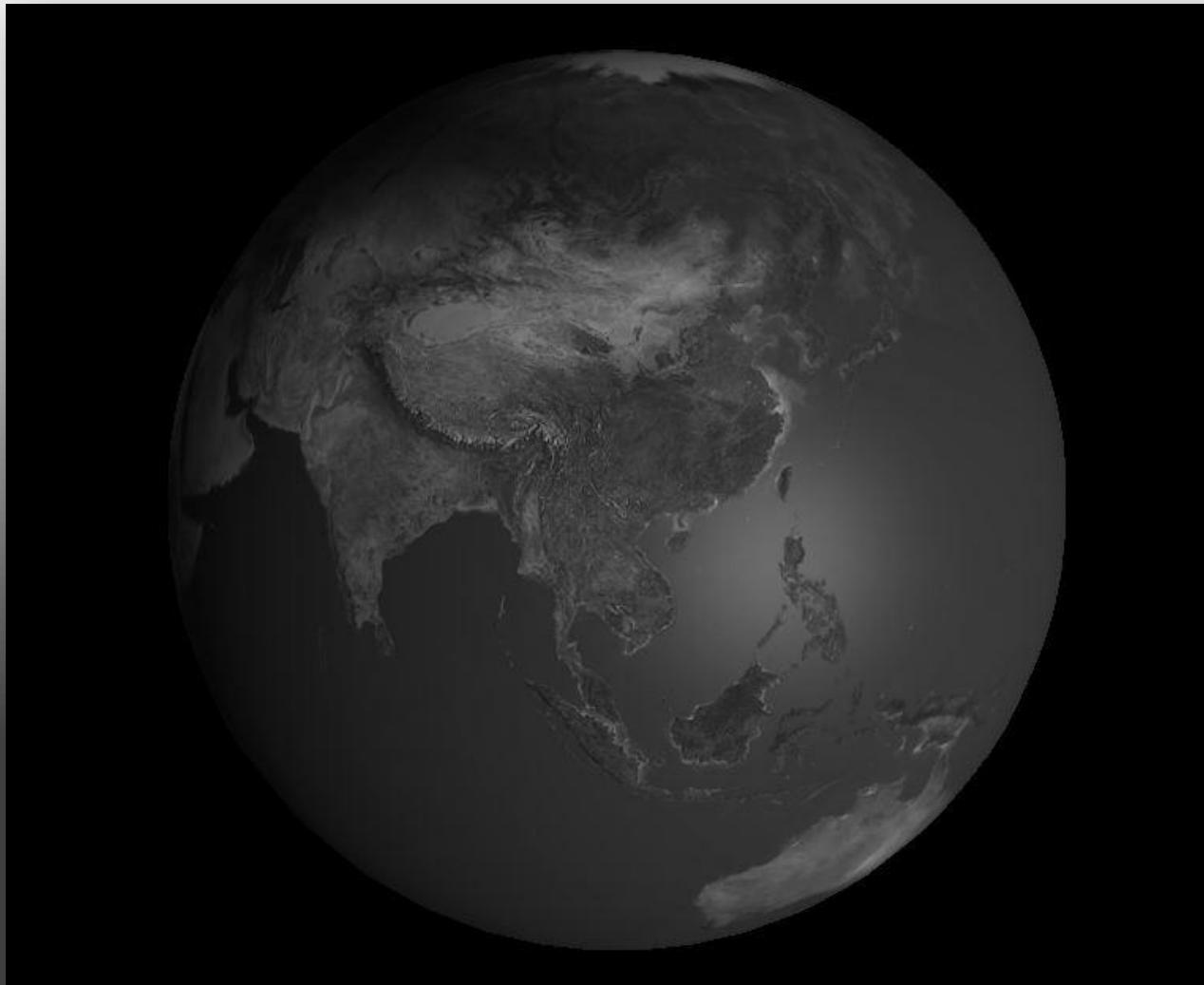
```
var renderPass = new THREE.RenderPass(scene, camera);
var effectCopy = new THREE.ShaderPass(THREE.CopyShader);
effectCopy.renderToScreen = true;

var shaderPass = new THREE.ShaderPass(THREE.CustomGrayScaleShader);

shaderPass.uniforms.rPower.value = 0.2126;
shaderPass.uniforms.gPower.value = 0.7152;
shaderPass.uniforms.bPower.value = 0.0722;

var composer = new THREE.EffectComposer(webGLRenderer);
composer.addPass(renderPass);
composer.addPass(shaderPass);
composer.addPass(effectCopy);
```

# CustomShader



# Exercice

- Créer un CustomShader qui permet de coloriser une scène selon une couleur passée

WORK IN  
PROGRESS

# Fin

Merci pour votre attention !  
**Des questions ?**



[contact@dotsafe.fr](mailto:contact@dotsafe.fr)  
05 47 74 86 88