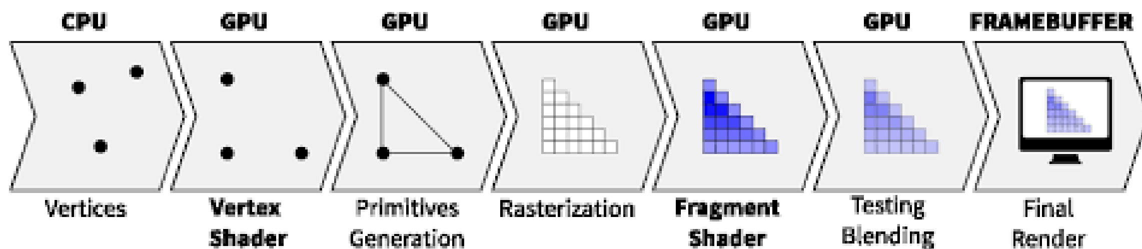
R5.DWeb-DI.07 - TP5

Un **shader** est un bout de code écrit dans le langage GLSL (proche du langage C, un peu moins du Javascript), qui permet de paralléliser des calculs sur GPU. En général on considère deux types de shaders dans une appli ThreeJS :

- le **vertex shader** calcule la projection des sommets des objets 3D sur l'écran
- le **fragment shader** s'occupe de calculer la couleur des pixels visibles à l'écran, obtenus à partir de la projection des triangles des objets 3D



La première utilisation en ThreeJS consiste à définir un plan orienté face à la caméra, qui occupe tout ou une partie de l'écran. Le **fragment shader** permet de calculer la couleur de chaque pixel du plan et l'ordinateur va appliquer ce traitement en parallèle pour obtenir la couleur de tous les pixels presque instantanément, alors qu'en JS classique il faudrait le faire pixel par pixel.

Exercice 1 - Exemple minimal

- On veut obtenir un simple disque blanc sur fond noir
- Le **vertex shader** (shaders/vertex.glsl.js) est minimal : il se contente de passer les positions au shader suivant

```
export const vertexShader = `
varying vec2 vUv;
void main() {
   vUv = uv;
   gl_Position = projectionMatrix * modelViewMatrix * vec4(position,
1.0);
}
`;
```

Essayez de répondre aux questions suivantes :

1. Qu'est-ce que “uv” ?
2. A quoi sert “varying” ?
3. Qu'est-ce que “position” ?
4. A quoi sert la ligne “gl_Position = …” ?

- Le **fragment shader** (shaders/fragment.glsl.js) peut accéder à la valeur vUv pour chaque pixel visible. Ici on va calculer simplement la distance au centre (0.5, 0.5) et on colorie en blanc si on est dans le cercle de rayon 0.2, sinon en noir :

```
export const fragmentShader = `
```

```
varying vec2 vUv;
void main() {
   float dist = distance(vUv, vec2(0.5));
   if (dist < 0.2) {
       gl_FragColor = vec4(1.0);
   } else {
       gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0);
   }
}
`;
```

Essayez de répondre aux questions suivantes :

1. Qu’est-ce que “vUv” et quelles sont ses valeurs minimale et maximale ?
2. Que fait la fonction distance(vUv, vec2(0.5)) ?
3. Que se passe-t-il si on remplace 0.2 par une autre valeur ?
4. Que signifie la ligne gl_FragColor = vec4(1.0) ?
5. Que signifie la ligne gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0); ?

- Code principal (main.js) :

```
import * as THREE from 'three';
import { vertexShader } from './shaders/vertex.glsl.js';
import { fragmentShader } from './shaders/fragment.glsl.js';

const scene = new THREE.Scene();
const camera = new THREE.OrthographicCamera(-1, 1, 1, -1, 0, 1);
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

const geometry = new THREE.PlaneGeometry(2, 2);
const material = new THREE.ShaderMaterial({
   vertexShader,
   fragmentShader
});
const mesh = new THREE.Mesh(geometry, material);
scene.add(mesh);

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

Essayez de répondre aux questions suivantes :

1. Représentez graphiquement la scène 3D définie dans ce code
2. Qu’est-ce qu’une caméra orthographique et pourquoi l’utiliser ici ?

3. Par défaut PlaneGeometry représente un plan avec seulement deux triangles (https://threejs.org/docs/#api/en/geometries/PlaneGeometry), est-ce que ça changerait quelque chose d'augmenter la résolution widthSegments ou heightSegments ?

Il vous reste à créer le **index.html** (cf cours de l'an dernier) et vous devriez voir un cercle blanc sur fond noir. Essayez ensuite de modifier le code pour :
1. dessiner un carré au lieu d'un cercle
2. afficher un dégradé de couleur du centre vers l'extérieur du cercle

Exercice 2 - Fonctions de distance

Il existe énormément de tutoriels sur les shaders, et pour continuer l'exercice précédent il existe aussi énormément de moyens de calculer des distances permettant d'obtenir d'autres formes qu'un cercle ou un carré. Essayez d'abord de reproduire quelques exemples issus de Inigo Quilez, "2D distance functions"

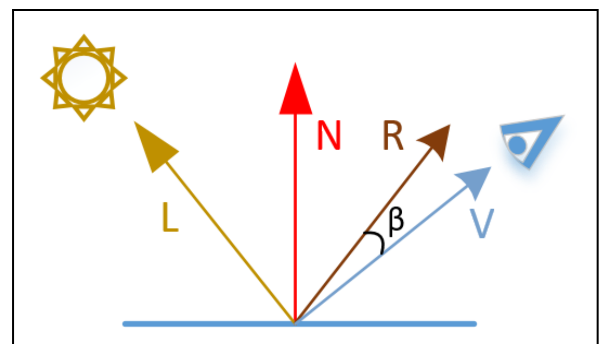
Exercice 3 - Uniforms

Les **uniforms** sont un moyen de communiquer au fragment shader des données venant de l'application ThreeJS, comme la position de la souris, une texture ou un compteur de temps que le shader peut utiliser pour modifier ses calculs. Essayez de reproduire ces 3 exemples : https://mediaserver.unilim.fr/videos/19092025-110409/

Exercice 4 - Passage en 3D

Il est temps de passer à la 3D en modifiant d'abord le code ThreeJS: utilisez une sphère à la place du plan, ajoutez un OrbitControls, une lumière ambiante, une lumière fixe et une lumière tournante, et ajoutez des **uniforms** pour que ces lumières soient utilisables dans le fragment shader.

Le premier objectif est de reproduire avec vos shaders une simple "illumination", c'est-à-dire simuler l'interaction de la lumière avec une surface, sur la base du modèle ci-contre. Expliquez quels éléments sur ce schéma influent sur la façon dont un observateur "voit" un point sur une surface ?



Tout cela nécessite de mettre à jour le vertex shader:

```
varying vec3 vNormal;
varying vec3 vViewPosition;

void main() {
   vec4 modelViewPosition = modelViewMatrix * vec4(position, 1.0);
   gl_Position = projectionMatrix * modelViewPosition;
```

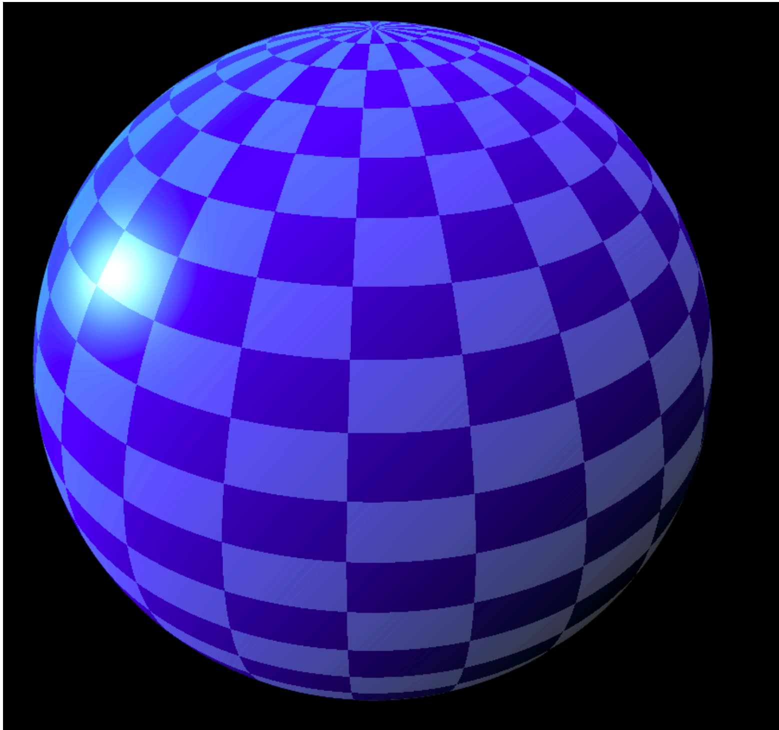
```
   vViewPosition = -modelViewPosition.xyz;
   vNormal = normalize(normalMatrix * normal);
}
```
Essayez de déterminer quelles sont les modifications importantes ici ?

Et pour le fragment shader:
```
varying vec3 vNormal;
varying vec3 vViewPosition;
uniform vec3 u_ambient;
uniform vec3 u_lightPosition;
uniform vec3 u_fixedLightPosition;

void main() {
   vec3 normal = normalize(vNormal);
   vec3 viewDir = normalize(vViewPosition);
   vec3 lightDir = normalize(u_lightPosition - vViewPosition);
   vec3 baseColor = vec3(0.1, 0.3, 0.9);
   float diffuse = max(dot(normal, lightDir), 0.7);
   vec3 halfDir = normalize(lightDir + viewDir);
   float spec = pow(max(dot(normal, halfDir), 0.0), 64.0);
   vec3 ambient = baseColor * u_ambient;
   vec3 color = ambient + baseColor * diffuse + vec3(1.0, 1.0, 1.0) *
spec * 0.7;
   vec3 fixedLightDir = normalize(u_fixedLightPosition -
vViewPosition);
   float fixedDiffuse = dot(normal, fixedLightDir);
   color += baseColor * fixedDiffuse * 0.93; // contribution plus douce

   gl_FragColor = vec4(color, 1.0);
}
```
Essayez de déterminer à quoi correspondent ces calculs par rapport au schéma précédent ?

Avec ce code vous devriez voir une simple sphère illuminée par vos lumières, ce que ThreeJS sait très bien déjà faire sans shaders. L'intérêt est ensuite de pouvoir modifier les calculs dans le fragment shader pour obtenir par exemple le résultat ci-dessous (vous aurez besoin de modifier aussi le vertex shader pour transmettre la position du point d'origine) :



Exercice 5 - Du bruit

Une autre chose difficile à faire en ThreeJS sans shaders est d'appliquer des fonctions de bruit : https://fr.wikipedia.org/wiki/Bruit_de_Perlin

Le code ci-dessous est un peu différent mais donne le même type de résultat :

```
float hash(float n) {
   return fract(sin(n) * 43758.5453123);
}
float noise(vec3 p) {
   vec3 i = floor(p);
   vec3 f = fract(p);
   f = f * f * f * (f * (f * 6.0 - 15.0) + 10.0);
   float n = dot(i, vec3(1.0, 57.0, 113.0));
   float res = mix(mix(mix( hash(n + 0.0), hash(n + 1.0), f.x),
                       mix( hash(n + 57.0), hash(n + 58.0), f.x), f.y),
                   mix(mix( hash(n + 113.0), hash(n + 114.0), f.x),
                       mix( hash(n + 170.0), hash(n + 171.0), f.x),
f.y), f.z);
   return res;
}
float fbm(vec3 p) {
   float value = 0.0;
   float amplitude = 0.5;
   float frequency = 1.0;
```

```
    for(int i = 0; i < 4; i++) {
        value += amplitude * noise(p * frequency);
        frequency *= 2.0;
        amplitude *= 0.5;
    }
    return value;
}
```

Il peut être utilisé pour modifier le fragment shader, voir la première partie de la vidéo : https://mediaserver.unilim.fr/videos/19092025-194711/

Exercice 6 - Des piques

On peut aussi modifier le vertex shader pour appliquer aux sommets du maillage un déplacement positif ou négatif le long du vecteur normal, comme dans la seconde partie de la vidéo.

Exercice 7 - TSL

La dernière façon d'utiliser les shaders est le langage TSL : [Three.js Shading Language](#)
Ici les shaders peuvent s'écrire directement en JS, mais avec des fonctions spécifiques qui permettent beaucoup de choses avec (un peu) moins de complexité qu'en GLSL.

On peut par exemple faire bouger des objets :

```
import * as THREE from 'three';
import { Fn, instancedArray, deltaTime, hash, instanceIndex, vec3,
billboarding, If } from 'three/tsl';

// Essayer avec 1000, 2000, 5000, 10000...
const maxParticleCount = 50;
const globalSpeed = 0.001;
const radius = 0.5;
let camera, scene, renderer;
let computeParticles;
let particles;

init();

function init() {
    const { innerWidth, innerHeight } = window;
    camera = new THREE.PerspectiveCamera(60, innerWidth / innerHeight,
0.1, 100);
    camera.position.set(0, 0, 30);
```

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

    const positionBuffer = instancedArray(maxParticleCount, 'vec2');
    const velocityBuffer = instancedArray(maxParticleCount, 'vec2');

    const computeInit = Fn(() => {
        const pos = positionBuffer.element(instanceIndex);
        pos.x = hash(instanceIndex).mul(20).sub(10);
        pos.y = hash(instanceIndex.add(12345)).mul(20).sub(10);
        const vel = velocityBuffer.element(instanceIndex);
        vel.x = hash(instanceIndex.add(67890)).mul(2 *
globalSpeed).sub(globalSpeed);
        vel.y = hash(instanceIndex.add(54321)).mul(2 *
globalSpeed).sub(globalSpeed);
    })().compute(maxParticleCount);

    const computeUpdate = Fn(() => {
        const pos = positionBuffer.element(instanceIndex);
        const vel = velocityBuffer.element(instanceIndex);

        const randomX =
hash(instanceIndex.add(deltaTime.mul(12345))).mul(2 *
globalSpeed).sub(globalSpeed);
        const randomY =
hash(instanceIndex.add(deltaTime.mul(234567))).mul(2 *
globalSpeed).sub(globalSpeed);

        vel.x = vel.x.add(randomX);
        vel.y = vel.y.add(randomY);
        const len = (vel.x.mul(vel.x).add(vel.y.mul(vel.y)));
        If(len.greaterThan(10*globalSpeed), () => {
            vel.x = vel.x.mul(10*globalSpeed).div(len);
            vel.y = vel.y.mul(10*globalSpeed).div(len);
        });
        pos.addAssign(vel);

        If(pos.x.greaterThan(10), () => {
            pos.x = 10;
            vel.x = vel.x.mul(-1);
        });
        If(pos.x.lessThan(-10), () => {
            pos.x = -10;
            vel.x = vel.x.mul(-1);
```

```
        });
        If(pos.y.greaterThan(10), () => {
            pos.y = 10;
            vel.y = vel.y.mul(-1);
        });
        If(pos.y.lessThan(-10), () => {
            pos.y = -10;
            vel.y = vel.y.mul(-1);
        });
    });

    computeParticles = computeUpdate().compute(maxParticleCount);

    const material = new THREE.MeshBasicNodeMaterial();
    material.color = new THREE.Color(0x000000); // Noir
    material.vertexNode = billboarding({
        position: vec3(positionBuffer.toAttribute(), 0)
    });
    material.opacity = 0.6;
    material.transparent = true;
    material.depthWrite = false;

    particles = new THREE.Mesh(new THREE.PlaneGeometry(radius, radius),
material);
    particles.count = maxParticleCount;
    scene.add(particles);

    renderer = new THREE.WebGPURenderer({ antialias: true });
    renderer.setPixelRatio(window.devicePixelRatio);
    renderer.setSize(innerWidth, innerHeight);
    renderer.setAnimationLoop(animate);
    document.body.appendChild(renderer.domElement);

    renderer.computeAsync(computeInit);
    window.addEventListener('resize', onWindowResize);
}

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

```
function animate() {
   renderer.compute(computeParticles);
   renderer.render(scene, camera);
}
```

Le but est ici de comprendre ce que font les deux “shaders” et comment ils sont déclenchés dans le code ThreeJS.

Et ensuite de produire une animation 3D montrant l’augmentation du nombre de satellites autour de la Terre, un peu comme sur cette vidéo :
https://www.linkedin.com/posts/recreasciences_le-nombre-de-satellites-lanc%C3%A9s-depuis-1957-ugcPost-7372168916090978304-NOAv/