TP Noté : La Boîte à Particules (Broad & Narrow Phases)

Objectif du TP

L'objectif est de créer une simulation de « gaz » haute performance. Vous devez gérer 500+ sphères rebondissant dans un cube 3D. Le défi est double : assurer une physique réaliste (Narrow Phase) et maintenir 60 FPS via un partitionnement spatial (Broad Phase).

1. Barème de Notation (Total /30)

Critère d'évaluation	Points
Narrow Phase : Calcul d'impulsion 3D, restitution et correction d'overlap.	10 pts
Broad Phase : Implémentation d'une Grille Uniforme (Spatial Hashing).	10 pts
Environnement : Gestion des rebonds sur les 6 faces du cube conteneur.	5 pts
Optimisation & Code : Gestion de la mémoire (pas de `new` en boucle) et GUI.	5 pts

2. La Narrow Phase (Physique)

Calcul de l'impulsion

Détecter une collision entre 2 sphères est très simple :

$$\|P_A - P_B\| < 2R$$

Lorsqu'une collision est détectée entre deux sphères A et B, vous pouvez calculer l'impulsion j (en utilisant la formule de restitution) et appliquer l'impulsion à chaque sphère.

3. La Broad Phase (Optimisation)

La Grille Uniforme

Le monde est divisé en cases de taille $2 \times$ Diamètre.

- Enregistrement : À chaque frame, chaque boule est indexée dans une cellule : $X_{cell} = \left\lfloor \frac{P_x}{S} \right\rfloor$.
- Voisinage : Pour une boule donnée, vous ne devez tester que les boules présentes dans sa cellule et les 26 cellules adjacentes (en 3D).
- Performance : Vous devez inclure un bouton dans la GUI pour activer/désactiver la Broad Phase et constater la différence de FPS.

4. Contraintes Techniques

> ⚠ **Optimisation de la mémoire**
>
> Dans la fonction `updatePhysics`, il est strictement interdit de créer de nouveaux objets Three.js (`new THREE.Vector3()`).
> - Utilisez des variables globales ou des variables `static` pour vos calculs temporaires.
> - La création d'objets en boucle déclenche le **Garbage Collector** et provoque des saccades (stuttering).

Structure attendue du Spatial Hashing.

```
import * as THREE from 'three';
import { OrbitControls } from 'jsm/controls/OrbitControls.js';
import { GUI } from 'https://cdn.jsdelivr.net/npm/lil-gui@0.19/+esm';

// --- CONFIGURATION ---
const CUBE_SIZE = 10; // Le cube ira de -5 à 5
const BALL_RADIUS = 0.2;

let scene, camera, renderer, balls = [];
let grid;

const params = {
    nbBalls: 200,
    restitution: 0.8,
    useBroadPhase: true, // Pour comparer !
    gravity: 0,
    reset: () => initSimulation()
};

// --- CLASSE À COMPLÉTER (BROAD PHASE) ---
class SpatialGrid {
    constructor(size, cellSize) {
        this.cellSize = cellSize;
        this.grid = {}; // Dictionnaire de cellules
    }

    // Convertir une position 3D en clé de dictionnaire (ex: "1,2,-1")
    getKey(pos) {
        const x = Math.floor(pos.x / this.cellSize);
        const y = Math.floor(pos.y / this.cellSize);
        const z = Math.floor(pos.z / this.cellSize);
        return `${x},${y},${z}`;
    }

    update(balls) {
        this.grid = {};
        // 1. [À COMPLÉTER] : Parcourir les boules et les ranger dans les bonnes cases
    }

    getNeighbors(ball) {
        // 2. [À COMPLÉTER] : Récupérer les boules dans la case actuelle + 26 cases voisines
        return balls; // Par défaut renvoie tout (Brute force)
```

```
    }
}

// --- INITIALISATION ---
function init() {
    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera(75, window.innerWidth/window.innerHeight, 0.1, 1000);
    camera.position.set(15, 15, 15);

    renderer = new THREE.WebGLRenderer({ antialias: true });
    renderer.setSize(window.innerWidth, window.innerHeight);
    document.body.appendChild(renderer.domElement);

    // Wireframe du cube conteneur
    const geo = new THREE.BoxGeometry(CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
    const wireframe = new THREE.LineSegments(
        new THREE.EdgesGeometry(geo),
        new THREE.LineBasicMaterial({ color: 0xffffff })
    );
    scene.add(wireframe);

    grid = new SpatialGrid(CUBE_SIZE, BALL_RADIUS * 4);

    initSimulation();
    setupGUI();
    new OrbitControls(camera, renderer.domElement);
    animate();
}

function initSimulation() {
    // Nettoyage
    balls.forEach(b => scene.remove(b));
    balls = [];

    // Création des boules aléatoires
    for(let i=0; i < params.nbBalls; i++) {
        const mesh = new THREE.Mesh(
            new THREE.SphereGeometry(BALL_RADIUS),
            new THREE.MeshStandardMaterial({ color: Math.random() * 0xffffff })
        );
        mesh.position.set(
            (Math.random()-0.5) * (CUBE_SIZE - 1),
            (Math.random()-0.5) * (CUBE_SIZE - 1),
            (Math.random()-0.5) * (CUBE_SIZE - 1)
        );
        mesh.userData.velocity = new THREE.Vector3(
            (Math.random()-0.5) * 5,
            (Math.random()-0.5) * 5,
            (Math.random()-0.5) * 5
        );
        mesh.userData.mass = 1;
        balls.push(mesh);
        scene.add(mesh);
    }
}
```

```
function updatePhysics(dt) {
    // 1. Intégration
    balls.forEach(b => {
        b.position.addScaledVector(b.userData.velocity, dt);

        // 2. Murs (Narrow Phase contre le cube)
        // [À COMPLÉTER] : Rebond sur les 6 faces avec correction de position
    });

    // 3. Collisions entre boules
    if (params.useBroadPhase) {
        grid.update(balls);
        balls.forEach(ballA => {
            const potentialTargets = grid.getNeighbors(ballA);
            potentialTargets.forEach(ballB => {
                if (ballA !== ballB) resolveCollision(ballA, ballB);
            });
        });
    } else {
        // Brute force O(N^2)
        for(let i=0; i<balls.length; i++) {
            for(let j=i+1; j<balls.length; j++) {
                resolveCollision(balls[i], balls[j]);
            }
        }
    }
}

function resolveCollision(ballA, ballB) {
    const dist = ballA.position.distanceTo(ballB.position);
    if (dist > BALL_RADIUS * 2) return;

    // [À COMPLÉTER] :
    // 1. Calcul de la normale
    // 2. Calcul de l'impulsion (J) en 3D
    // 3. Mise à jour des vitesses
    // 4. Correction de l'interpénétration (très important pour éviter que les boules collent)
}

// ... loop et setupGUI ...
```

5. Bonus : Friction et Gravité

Ajoutez un curseur de gravité et un coefficient de friction lors des collisions avec les murs pour obtenir un comportement de « tas de sable » au fond de la boîte (+2 pts bonus).

Date de rendu : Fin de semaine (2026-01-12)

Livrable : Archive .zip (Code source + Démo fonctionnelle)