R5.DWeb-DI.07 - TP5

Exercice 1

Le but pour ce TP est de créer des objets déformables à l'aide d'un système appelé masse-ressort, comme dans ces exemples :

- Coding Challenge 177: Soft Body Physics
- Soft Body Physics Explained

Commençons en 2D avec la librairie **p5js** déjà vue l'an dernier. Vous avez à disposition le code ci-dessous, avec d'abord le fichier HTML :

```
<html>
<head>
 <meta charset="UTF-8">
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <link rel="stylesheet" href="style.css">
 <script
src="https://cdn.jsdelivr.net/npm/p5@1.7.0/lib/p5.js"></script>
 <script src="Mass.js"></script>
 <script src="q1.js"></script>
</head>
</html>
```

Le fichier **style.css** :

```
* {
   margin: 0;
   padding: 0;
   box-sizing: border-box;
}
```

Et le fichier **Mass.js** :

```
class Mass {
   constructor(x, y) {
       this.position = createVector(x, y);
       this.velocity = createVector(random(-20, 20), random(20));
   }
   updatePosition() {
       this.velocity.y += gravity;
       this.velocity.mult(damping);
       this.velocity.limit(maxVel);
       this.position.x += this.velocity.x*deltaT;
       this.position.y += this.velocity.y*deltaT;
```

```
      if (this.position.x < 0) {
          this.position.x = 0;
          this.velocity.x = 0;
          this.velocity.y *= friction;
        }
        if (this.position.x > width) {
          this.position.x = width;
          this.velocity.x = 0;
          this.velocity.y *= friction;
        }
        if (this.position.y < 0) {
          this.position.y = 0;
          this.velocity.y = 0;
          this.velocity.x *= friction;
        }
        if (this.position.y > height) {
          this.position.y = height;
          this.velocity.y = 0;
          this.velocity.x *= friction;
        }
  }
  display() {
      fill(0);
      circle(this.position.x, this.position.y, 10);
  }
}
```

1. Essayez de comprendre ce qui se passe dans la classe **Mass** puis créez le fichier **q1.js** sur la base du squelette ci-dessous :

```
const deltaT = 0.1;
const gravity = 1;
const damping = 0.99;
const stiffness = 0.99;
const friction = 0.005;
const maxVel = 150;

function setup() {
      ...
}

function draw() {
      ...
```

```
}
```

Le but est d'abord de créer une masse à une position aléatoire et de la faire simplement retomber : https://mediaserver.unilim.fr/videos/08112023-104938/

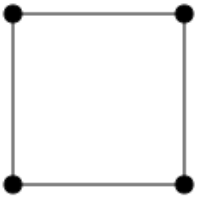
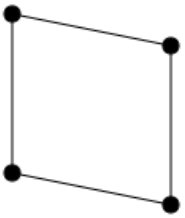
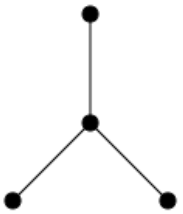
Modifiez le code pour créer 10 masses : https://mediaserver.unilim.fr/videos/08112023-112332/

2. Modifiez la classe **Mass** pour fixer une vélocité nulle dans le constructeur, puis créez une nouvelle classe **Spring** :

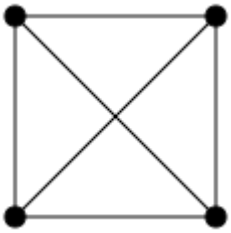
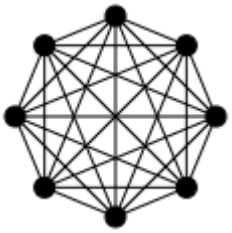
```
class Spring {
   constructor(_m1, _m2) {
       this.m1 = _m1;
       this.m2 = _m2;
       this.restLength = dist(_m1.position.x, _m1.position.y,
           _m2.position.x, _m2.position.y);
   }
   applyConstraint() {
       let d = this.m2.position.copy();
       d.sub(this.m1.position);
       let m = (d.mag() - this.restLength)/this.restLength;
       d.mult(m * 2);
       d.mult(stiffness);
       d.mult(0.5);
       this.m1.velocity.add(d);
       this.m2.velocity.sub(d);
   }
   display() {
       line(this.m1.position.x, this.m1.position.y,
           this.m2.position.x, this.m2.position.y);
   }
}
```

Essayez de comprendre ce qui se passe ici, puis testez ce code en créant deux masses aléatoires reliées par un ressort : https://mediaserver.unilim.fr/videos/08112023-115856/

3. Vous êtes maintenant équipé pour créer des objets complexes constitués de masses reliées par des ressorts. Construisez les formes ci-dessous, laissez jouer la gravité et identifiez les différents problèmes :

La conclusion que vous pouvez en tirer est que vous avez intérêt, si vous voulez conserver les formes d'origine, à ajouter un maximum de ressorts comme ci-dessous :

4. Ajoutez une interaction pour que les masses soient attirées par le pointeur de souris si un clic est détecté : https://mediaserver.unilim.fr/videos/08112023-151609/

5. Transformez votre code pour gérer plusieurs cercles, les collisions entre eux (en testant si une masse se retrouve à l'intérieur d'un autre cercle), et les collisions avec une boîte posée au sol : https://mediaserver.unilim.fr/videos/09112023-114139/

   Sans écrire de code, expliquez comment vous pourriez traiter des objets déformables plus complexes ?

Exercice 2

Il est temps de revenir à Three.js et d'étendre le principe au cas 3D. Pour cela on va créer des objets constitués d'un maillage 3D, mais avec la possibilité de modifier dynamiquement la position des sommets. La géométrie de type **BufferGeometry** permet de définir un objet avec des sommets ("vertices") et des indices qui décrivent comment joindre ces sommets par des triangles. Pour bien comprendre, dessinez à la main l'objet 3D décrit par ce code :

```
const geometry = new THREE.BufferGeometry();
const vertices = new Float32Array([
  -5, 5, -5,
  5, 5, -5,
  5, 15, -5,
  -5, 15, -5,
  -5, 5, 5,
  5, 5, 5,
  5, 15, 5,
  -5, 15, 5,
```

```
    ]);
    const indices = [
      2, 1, 0, 0, 3, 2,
      0, 4, 7, 7, 3, 0,
      0, 1, 5, 5, 4, 0,
      1, 2, 6, 6, 5, 1,
      2, 3, 7, 7, 6, 2,
      4, 5, 6, 6, 7, 4
    ];
    geometry.setIndex(indices);
    geometry.setAttribute('position', new
THREE.BufferAttribute(vertices, 3));
```

On peut accéder au tableau de sommets grâce à :

```
    geometry.getAttribute('position').array;
```

Si la position des sommets a été modifiée, il suffit pour voir les changements dans la scène d'appliquer les instructions suivantes :

```
    geometry.computeVertexNormals();
    geometry.getAttribute("position").needsUpdate = true;
```

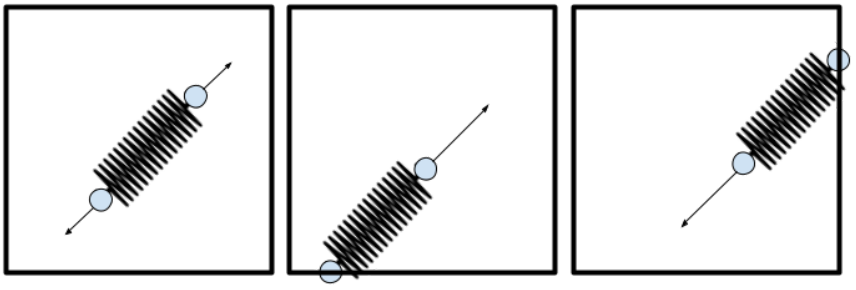
A votre avis, à quoi sert la fonction **computeVertexNormals** ?

Maintenant adaptez votre code 2D pour créer une scène ThreeJS contenant un plan horizontal et un cube déformable qui saute aléatoirement vers le haut si on appuie sur la barre Espace et qui rebondit sur les murs invisibles :
https://mediaserver.unilim.fr/videos/10112023-114155/

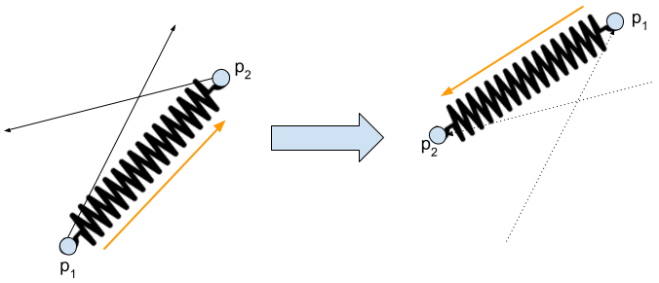
Un problème que vous rencontrez peut-être en passant à la 3D concerne la déformation très importante que les ressorts peuvent subir, notamment à cause de la gravité. Pour améliorer le comportement de votre cube vous pouvez implémenter les deux modifications suivantes.

La première amélioration consiste à introduire un booléen appelé **fixed** dans la classe **Mass**, qui aura la valeur **true** seulement si la masse entre en collision avec le sol ou les murs (à rajouter dans la fonction **updatePosition**). Ensuite, dans la fonction **applyConstraint** de la classe **Spring**, on peut traiter 3 cas différents qui sont surtout importants quand le ressort est en compression :

- si aucune des deux masses n'est fixe, dans ce cas on applique comme précédemment le calcul pour repousser les deux masses à parts égales (cf ci-dessous à gauche)
- si l'une des deux masses est fixe, alors le déplacement total est appliqué seulement à l'autre (cf ci-dessous au centre et à droite)



La seconde amélioration consiste à empêcher les deux masses d'un ressort de changer subitement d'orientation, ce qui peut arriver dans certains cas comme sur l'image ci-dessous :



Pour traiter ce cas vous pouvez rajouter dans la boucle d'animation l'appel à la fonction suivante une fois que toutes les vélocités ont été calculées (essayez de comprendre ce qu'elle fait exactement) :

```
   avoidExchange() {
        let d = this.p2.position.clone();
        d.sub(this.p1.position);
        let nextP1 = this.p1.position.clone();
        nextP1.add(this.p1.velocity);
        let nextP2 = this.p2.position.clone();
        nextP2.add(this.p2.velocity);
        let dNext = nextP2.clone();
        dNext.sub(nextP1);
        // If the orientation from p1 to p2 is not the same from nextP1
to nextP2, then reduce velocities
        if (dNext.dot(d) < 0) {
            this.p1.velocity.multiplyScalar(0.5);
            this.p2.velocity.multiplyScalar(0.5);
        }
   }
```

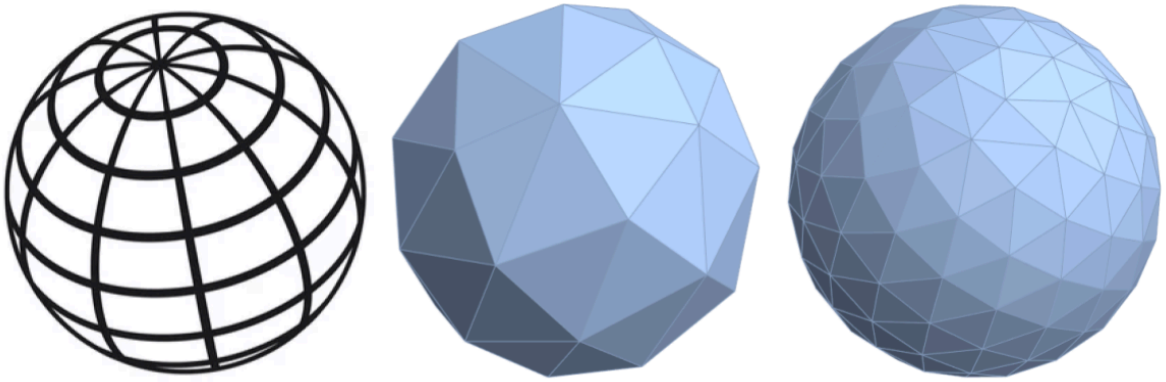
Exercice 3

Le but maintenant est de remplacer le cube par une sphère déformable, puis d'ajouter d'autres sphères déformables et de traiter les collisions entre ces sphères : https://mediaserver.unilim.fr/videos/25072024-124006/

Ici la touche "f" permet de lancer des sphères, et la touche "Espace" de les faire sauter aléatoirement vers le haut.

La première question à résoudre est la génération de points bien répartis sur une sphère. On a l'habitude des représentations de type "latitude / longitude" comme ci-dessous à gauche, mais cela impliquerait trop de ressorts connectés avec les deux pôles. On préfère dans ce cas utiliser un algorithme qui garantit une bonne répartition, comme ci-dessous au

centre et à droite, même pour un faible nombre de points (35 pour les sphères présentes dans la vidéo) : https://openprocessing.org/sketch/392066/



Une difficulté est ensuite de définir les triangles, mais il y a une solution simple (même si cela induit parfois des petites erreurs d'affichage comme on peut le voir sur la vidéo) : créer des triangles entre tous les triplets possibles de sommets.

Les difficultés restantes sont encore les problèmes de stabilité et de résistance à la gravité, qui risquent d'aplatir la sphère au sol. Il faut donc encore rajouter des améliorations, d'abord ajouter une masse correspondant au centre de la sphère, qui l'aidera à conserver son volume.

Ensuite, il faut modifier la fonction **applyConstraint** pour "rigidifier" les ressorts :

```
        let d = this.p2.position.clone();
        d.sub(this.p1.position);
        let dMult = (d.length() - this.restLength) / this.restLength;
        dMult = dMult * dMult * dMult;
        d.multiplyScalar(dMult);
        d.multiplyScalar(STIFFNESS);
        ...
```

Et enfin, il faut ajouter aussi des tests de distance entre le centre de gravité de la sphère (à vous de trouver comment le calculer) et le sol, pour la repousser vers le haut le cas échéant :

```
        if (this.center.y < this.size) {
            this.jump(new THREE.Vector3(0, this.size - this.center.y,
0));
        }
```

La fonction **jump** va simplement appliquer la même force à toutes les masses de la sphère, et vous pouvez ajouter le même type de code pour les murs ou les collisions avec les autres sphères.

Exercice 4

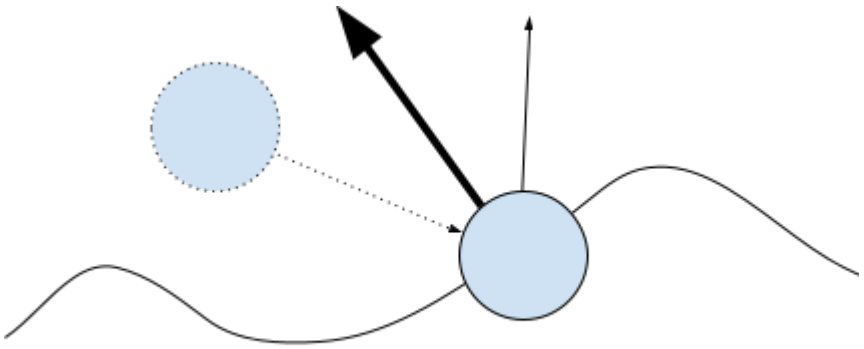
Dernier exercice, créer un terrain un peu plus complexe et gérer les interactions avec les balles, en ajoutant la possibilité de les faire avancer (toutes en même temps) dans les directions X, -X, Z et -Z : https://mediaserver.unilim.fr/videos/25072024-160035/

La première difficulté est la définition du terrain, qui peut se faire à l'aide d'un tableau contenant des des coordonnées planaires associées à un rayon et une hauteur. On peut calculer à partir de ce tableau deux fonctions **hmap** et **grad** qui donnent respectivement la hauteur et le vecteur normal à la surface :

```
export const hmap = (x, y, hills) => {
    let dTotal = 0;
    for (let j = 0; j < hills.length; j++) {
        let dx = x - hills[j].x;
        let dy = y - hills[j].y;
        let d = Math.sqrt(dx*dx + dy*dy)/hills[j].radius;
        if (d < 1) {
            dTotal += 0.5*hills[j].height*(1 - d*d) * (1 - d*d);
        }
    }
    return dTotal;
}
export const grad = (x, y, hills) => {
    let dx = 0.5;
    let dy = 0.5;
    let d = hmap(x - dx, y, hills) - hmap(x + dx, y, hills);
    let d2 = hmap(x, y - dy, hills) - hmap(x, y + dy, hills);
    let vec = new THREE.Vector3(d, 1, -d2);
    vec.normalize();
    return vec;
}
```

Pour le terrain il suffit alors de créer un plan de type **PlaneGeometry** (qui peut être manipulé comme **BufferGeometry**) et de déplacer la hauteur de chaque sommet en fonction de ses coordonnées.

La fonction peut aussi servir à déterminer si une masse s'est enfoncée dans la surface en fonction de ses coordonnées planaires. Dans ce cas il faut replacer la masse sur la surface, et la faire "rebondir" comme sur le dessin ci-dessous où le vecteur normal apparaît en gras (utilisez la fonction **reflect** de la classe **Vector3**) :



Le même principe peut aussi s’appliquer pour faire “jumper” une balle en direction du vecteur normal si elle s’approche trop du sol.