TP Three.js 4

Licence Informatique 3ème année

Année 2024-2025
durée : 3h

L'objectif de ce quatrième TP est d'expérimenter la manière de sélectionner un objet dans l'application via l'algorithme du *ray casting*, ainsi qu'utiliser cette fonctionnalité pour détecter les collisions.
Dans un premier temps, récupérez l'archive jointe à cet énoncé et extrayez les dossiers et fichiers qui s'y trouvent dans un sous-dossier nommé TP4 qui se trouvera dans le dossier où vous réalisez vos TP d'informatique graphique. Après extraction vous disposez :

— d'un dossier `js` qui contient les scripts nécessaires à ce tp ;
— un fichier `tp4_threejs.html` qui permet l'affichage et l'animation des objets du TP précédent.

Ce TP devra être rendu à la fin de la séance, sous la forme d'une archive nommée `TPIG4-XXX`, où `XXX` sera remplacé par votre nom de famille. Elle devra contenir le dossier TP4 et les dossiers et fichiers qui y seront contenus et sera à envoyer via un site tel que `wetransfer`.

Pour tous les TPs Three.js, vous pourrez consulter la documentation officielle en ligne, à l'adresse :
`https://threejs.org/docs/index.html`

Exercice 1 - Sélection

Dans ce premier exercice, vous allez modifier le code fourni pour permettre la sélection d'un objet via un clic souris.

Gestion d'un clic souris

Dans un premier temps, ajoutez une fonction nommée `onDocumentMouseDown(event)` à votre script principal. Cette fonction devra être associée à un gestionnaire d'événement de type `mousedown`. Vous pourrez faire afficher un message dans la console ou dans le navigateur pour vérifier que l'événement de type *clic souris* est bien pris en compte.

Sélection d'un objet

De manière pratique, la sélection d'un objet se fait en positionnant le pointeur de la souris sur l'objet désiré, puis en effectuant un clic sur l'un des boutons de la souris. Au clic, un événement de type `mousedown` est généré, associé aux coordonnées (x, y) de l'emplacement du pixel auquel s'est produit le clic dans la fenêtre de votre navigateur. Le problème est que disposer uniquement de ces coordonnées ne permet pas de savoir quel est l'objet qui s'y trouve, car le pixel ne contient qu'une information de couleur. Pour pouvoir déterminer l'objet qui se trouve dans ce pixel, nous allons utiliser l'algorithme du tracé de rayons (*ray casting*) qui consiste à lancer un rayon (une demi-droite) depuis la position de la caméra à travers le pixel choisi. L'intersection de cette demi-droite avec tous les objets de la scène va être calculée et chaque intersection va être stockée dans une case d'un tableau, dans l'ordre de distance croissante par rapport à l'origine du rayon, ici la position de la caméra (voir figure 1).

En regardant le contenu du tableau d'intersection il est alors possible de savoir (i) si un objet est présent dans le pixel (le tableau n'est pas vide) et (ii) quel objet est présent dans le pixel (c'est celui qui se trouve dans la case 0 du tableau d'intersections).

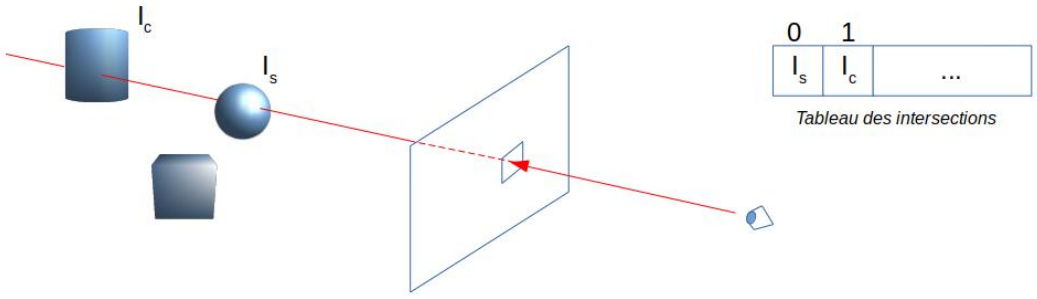


FIGURE 1 – Principe du tracé de rayon : un rayon est lancé à travers un pixel depuis la position de l'oeil ; les intersections avec les objets de la scène sont calculées (ici des intersections sont trouvées avec le cylindre (I_c) et la sphère (I_s)) et elles sont stockées dans un tableau, par ordre croissant de distance (ici I_s est plus proche de l'oeil que I_c).

Cet algorithme est implémenté dans `Three.js` au sein de la classe `RayCaster`. Pour pouvoir l'utiliser il faut alors créer une nouvelle instance de `RayCaster` :

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

puis préciser (i) les coordonnées du pixel traversé et la position de la caméra par l'intermédiaire de la méthode `setFromCamera` :

```
const coord = new THREE.Vector2();
        ...
raycaster.setFromCamera( coord, camera );
```

Ces coordonnées seront de type `Vector2` (composantes (x, y)) et devront être comprises dans l'intervalle $[-1, 1]$, l'origine $(0, 0)$ se trouvant au centre de la fenêtre d'affichage. Cependant, les coordonnées fournies par le gestionnaire d'événements sont des coordonnées « pixel » comprises entre 0 et $window.innerWidth$ pour x et 0 et $window.innerHeight$ pour y. Pour effectuer la conversion, il suffit d'utiliser le code ci-dessous :

```
const coord = new THREE.Vector2();
coord.x = ( event.clientX / window.innerWidth ) * 2 - 1;
coord.y = - ( event.clientY / window.innerHeight ) * 2 + 1;
```

ou (`event.clientX`, `event.clientY`) représentent les coordonnées du pixel dans lequel l'utilisateur a cliqué.

Lorsque le `raycaster` a été correctement initialisé, on lui demande de calculer les intersections avec l'ensemble des objets contenus dans la scène (`scene.children`) par l'intermédiaire de sa méthode `intersectObjects`. Celle-ci prend comme premier paramètre la liste des objets à tester et comme second paramètre une variable booléenne, qui précise si on doit tester l'intersection avec les objets contenus dans un groupe (valeur `true`) de manière récursive, ou pas (valeur `false`) :

```
var intersects = raycaster.intersectObjects(scene.children, true);
```

La méthode retourne un tableau qui contient les intersections triées par ordre de distance croissante. Il peut être vide si aucune intersection n'a été trouvée. Le code classique d'utilisation de l'objet le plus proche intersecté prend alors la forme suivante :

```
if (intersects.length > 0) {// le tableau n'est pas vide
  // exploiter le premier objet intersecté qui se trouve
  // dans intersects[0].object
}
```

Application 1 Complétez votre fonction `onDocumentMouseDown` de telle sorte que lorsque l'utilisateur clique sur l'un de vos objets, la transparence de son matériau change : au premier clic, l'objet devient transparent (voir figure 2) et au second il redevient opaque (et ainsi de suite). Pour ce faire on précise les points suivants :

- l'accès au matériel d'un objet s'effectue par l'intermédiaire de son attribut `material` ;
- `material` possède un attribut booléen `transparent` qui permet de spécifier si le matériau est transparent (valeur `true`) ou non (valeur `false`) ;
- le niveau de transparence, compris entre 0 (totalement transparent) et 1 (totalement opaque) peut être réglé via l'attribut `opacity` du matériau.

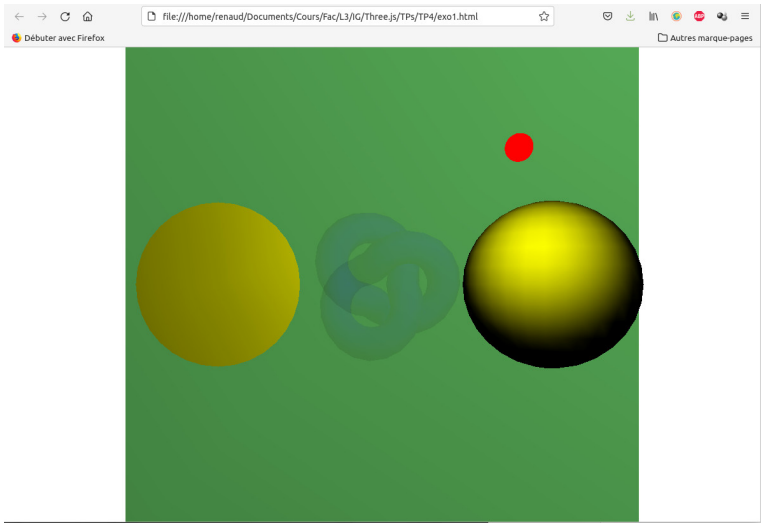


FIGURE 2 – Illustration de la sélection d'un objet : l'objet central a été sélectionné et rendu transparent (opacité à 0.1 dans cet exemple).

Application 2 A ce stade, tous les objets de la scène sont sélectionnables. On souhaite ne pouvoir sélectionner que les trois objets principaux. Pour ce faire, il suffit de les copier dans un tableau annexe et de passer ce tableau à la méthode `intersectObjects` à la place du tableau contenant tous les objets contenus dans la scène. Effectuez les modifications requises.

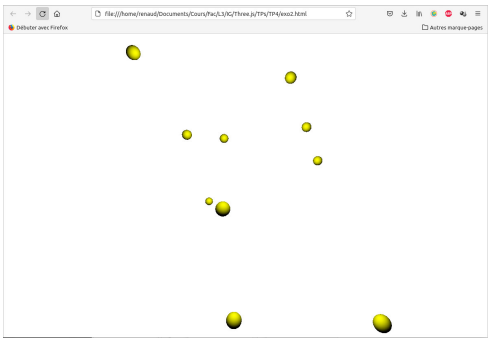
Exercice 2 - Petit jeu de « tir »

Le but de cet exercice est de développer une petite application de jeu, dans laquelle des balles vont rebondir de manière désordonnée sur des murs et dont l'objectif, pour le joueur, est de cliquer sur chacune des balles pour les faire disparaître le plus vite possible. Recopiez dans un premier temps le fichier `exo1.html` dans un fichier `exo2.html`, afin de pouvoir repartir d'un grand nombre de choses déjà existantes.

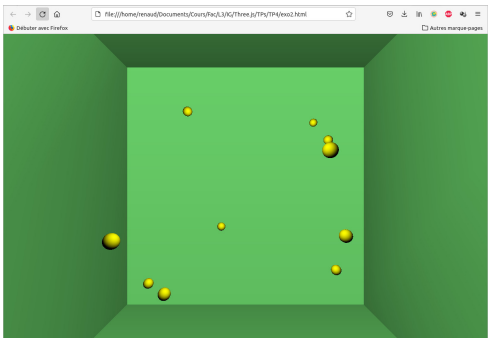
Première étape

Remplacez dans un premier temps la création des 4 objets précédents par un tableau de 10 sphères de diamètre 0.1 unité. La position de ces sphères sera initialisée aléatoirement entre -2.0 et 2.0 pour chacune de ses 3 coordonnées (x, y, z). Vous pourrez utiliser pour ce faire la fonction `Math.random()`, qui retourne un nombre réel compris entre 0.0 et 1.0. Vous supprimerez également le **code** de la fonction `update()` qui sera mis à jour dans l'une des prochaines questions.

Une fois créées, ces 10 sphères seront ajoutées à la scène et vous visualiserez le résultats obtenu (voir figure 3a).



(a) Rendu après la première étape.



(b) Rendu après la seconde étape.

FIGURE 3 – Evolution des rendus à obtenir pour les deux premières étapes.

Seconde étape

Ajoutez au fichier `js/mesObjets.js` une fonction permettant de créer les 5 murs d'une boite de côté 4.2 unités. La face située côté caméra ($Z > 0$) ne sera pas ajoutée, afin que la caméra puisse voir l'intérieur de la boite. Modifiez ensuite votre script principal pour ajouter ces 5 faces (voir figure 3b).

Troisième étape

Complétez la fonction `update()` de votre script principal, de telle sorte qu'à chaque *frame*, la position de chaque sphère soit modifiée d'une valeur $(\Delta x, \Delta y, \Delta z)$. Vous pourrez prendre aléatoirement l'une des deux valeurs 0.01 ou -0.01 pour initialiser chacune de ces quantités, sachant que chaque sphère devra avoir des incréments différents.

Pour simuler le rebond sur les murs, vous vérifierez à chaque mise à jour que la valeur de la coordonnée correspondante n'est pas supérieure à 2 ou inférieure à -2. Si c'est le cas, l'incrément devra changer de signe (cf le mouvement de la sphère dans le premier exercice).

Il ne vous reste plus qu'à finaliser votre jeu, en modifiant la fonction de traitement des intersections : plutôt que de modifier la transparence de l'objet trouvé dans un pixel lors d'un clic souris, il faut désormais le supprimer de la scène, afin qu'il n'apparaisse plus et ne soit plus pris en compte dans les calculs d'intersection. Vous utiliserez pour ce faire la méthode `remove()` de la classe `scene`, qui doit prendre en paramètre **l'objet** trouvé.

Exercice 3 - Détection de collision

Dans l'exercice précédent, on a simulé la collision d'une sphère avec l'une des parois du cube en détectant le dépassement de position par rapport aux dimensions de ce cube. Cela implique de connaître celles-ci, ce qui peut être limitatif. Dans cet exercice, vous allez utiliser une autre méthode, basée sur le tracé de rayons, pour pouvoir calculer la présence ou non de collisions avec les parois du cube.

Principe Chaque sphère se déplace d'un incrément selon les 3 axes Ox, Oy et Oz, tandis que les parois du cube se trouvent dans les plans Oxy, Oxz et Oyz. Une collision ne peut donc survenir que si, pour une direction de déplacement donnée (Ox, Oy ou Oz), la sphère se trouve à une distance inférieure ou égale à la valeur de son rayon du plan perpendiculaire considéré (voir figure 4 qui illustre le cas pour la direction Δx).

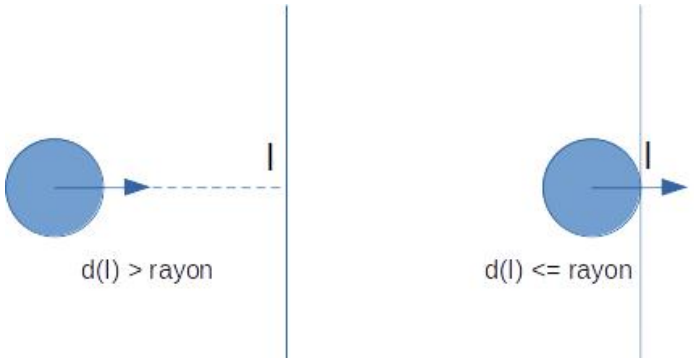


FIGURE 4 – Schéma du calcul d'intersection selon la direction Ox pour $\Delta x > 0$: le rayon est lancé depuis le centre de la sphère dans la direction $(1, 0, 0)$ (vecteur bleu). Dans le premier cas (à gauche), une intersection I est trouvée avec le plan vertical à une distance supérieure à la valeur du rayon de la sphère. Il n'y a donc pas de collision entre la sphère et le plan. Dans le second cas (à droite), une intersection I est également trouvée, mais cette fois à une distance inférieure ou égale au rayon de la sphère. Il y a donc collision et la sphère doit rebondir dans l'autre sens selon l'axe Ox.

L'idée est alors d'utiliser un objet de type `RayCaster` pour lancer un rayon depuis le centre de la sphère, dirigé selon chacune des 3 directions principales du repère. Ce rayon détermine les intersections rencontrées (si elles existent) et stocke la distance à laquelle celles-ci se sont produites. Il suffit ensuite de vérifier que la distance de l'intersection la plus proche est inférieure ou égale au rayon de la sphère pour déterminer qu'une collision a eu lieu. Si c'est le cas, on change le signe de la valeur de l'incrément, de manière à faire repartir la sphère dans l'autre sens.

Application Après avoir recopié votre fichier `exo2.html` dans un fichier nommé `exo3.html`, modifiez sa fonction `update` de manière à effectuer les tests de collisions avec un objet de type `RayCaster` et les mises à jour des directions de déplacement en fonction des rebonds. On précise les points suivants :

- le cube dans lequel se déplacent les sphères n'a pas de face avant (face en z positif). Il ne sera donc pas possible de trouver une intersection avec cette face et les sphères pourront donc sortir du cube. Il faudra donc, pour cette face, utiliser le même système que celui utilisé à la question précédente ;
- dans la question précédente, vous testiez les intersections avec tous les objets de la scène dans la méthode `intersectObjects` du rayon créé. Ici seules les faces doivent être testées ; il faudra alors créer un tableau d'objets dans lequel les murs seront recopiés et passer ce tableau à cette méthode, plutôt que l'ensemble de la scène [1].

1. On précise que la manière de tester l'intersection avec les murs, telle que proposée ici, n'est pas valide pour tester l'intersection des sphères entre-elles ...