

Module ReV - Projet

Musée Virtuel

Valentin LOPEZ CARUSO - v1lopezc@enib.fr
Trajano MENA - t1mena@enib.fr

1. Introduction

Nous avons développé un musée virtuel en utilisant Babylon.js, un moteur open source basé sur JavaScript, réputé pour sa capacité à créer des jeux et des applications 3D immersifs sur le web. Le musée virtuel transporte les visiteurs à l'époque connue sous le nom de Belle Époque en France, qui s'étend de 1871 à 1914. Notre musée virtuel présente une collection de peintures réalisées par certains des artistes français les plus estimés de l'époque. Il comprend également des statues de cette période. Le musée est divisé en différentes salles.

En plus des œuvres d'art statiques, le projet comprenait des éléments interactifs dans l'environnement virtuel pour donner vie au musée. Les visiteurs rencontreront des personnages non jouables (NPC) se promenant dans le musée et regardant les tableaux, ajoutant ainsi un élément de réalisme à ce monde. Il y a aussi des signaux dynamiques pour indiquer la direction de la visite. Pour faciliter la navigation, nous avons ajouté des téléporteurs sphériques. Le musée et les salles sont équipés de portes coulissantes.

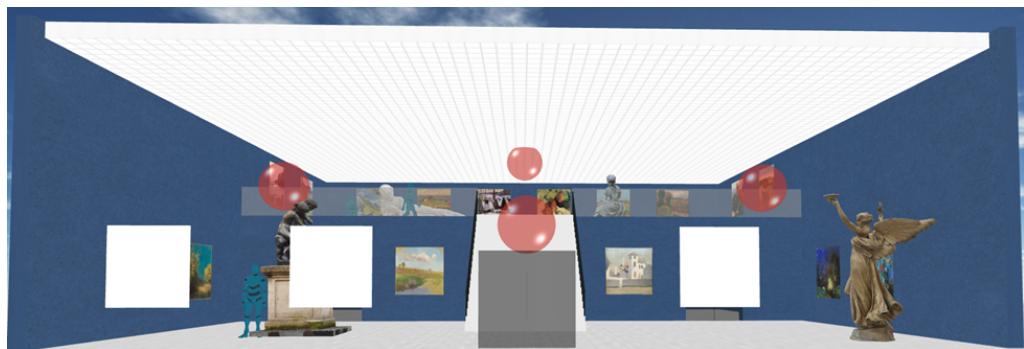


Figure 1 : Vue du musée depuis l'extérieur, sans le mur frontal. Les carrés blancs représentent l'arrière des tableaux qui sont exposés puisque le mur avant est caché dans cette image, et le carré gris semi-transparent est la porte coulissante d'entrée.

Une autre caractéristique de notre musée virtuel est la dynamique des noms et des descriptions des tableaux. En utilisant les capacités de rendu en temps réel de Babylon.js, nous faisons apparaître les noms lorsque le visiteur s'approche du tableau et la description seulement lorsqu'il y reste quelques secondes. Cela permet de ne pas surcharger l'utilisateur d'informations, car lorsqu'il est loin d'un tableau, il ne voit que l'œuvre d'art, sans aucune donnée supplémentaire.

Dans ce rapport, nous examinerons certains aspects techniques du musée virtuel, en explorant le processus de développement, les défis rencontrés et les solutions que nous avons mises en place.

2. Implementation

Le code de notre musée virtuel est structuré en plusieurs fichiers, les principaux étant factory.js et museum.js. Dans museum.js, le monde est initialisé et les fonctionnalités principales sont mises en place. Ce module se charge d'initialiser la scène, de configurer la perspective de la caméra et de verrouiller le pointeur de la souris. Il gère également la mise en place des EventListeners en

JavaScript pour permettre l'interaction et la réactivité dans l'environnement virtuel. La boucle de rendu se trouve également dans ce module, rafraîchissant continuellement l'affichage et gérant les aspects dynamiques de notre monde virtuel.

Une fois que le monde est créé, on initialise la "fabrique" à partir de factory.js. Ce module crée et positionne tous les éléments qui composent le musée virtuel. Il est également responsable de la création des matériaux et de leur attribution aux objets, ainsi que d'autres propriétés.

Cette division du code contribue à maintenir une organisation claire et facilite la gestion du projet.

2.1. Modélisation du musée

Le musée mesure 30m x 30m. Il dispose d'un hall principal à l'entrée, qui fait 15m x 30m. Juste en face de l'entrée, nous avons placé un escalier pour accéder à la mezzanine, également de 15m x 30m. Sous la mezzanine, il y a trois salles, chacune mesurant 15m x 10m. On a utilisé la fonction Factory.prototype.createWall qui nous avait déjà été fournie pour construire les murs. Pour les portes coulissantes, nous avons pratiqué des ouvertures dans les murs à l'aide de la fonction Factory.prototype.hole, également fournie. Nous avons également ajouté des rampes sur les escaliers et la mezzanine pour éviter les chutes. Plus de luminaires ont été ajoutés pour améliorer la visibilité.

La gravité est bien sûr activée afin que la caméra reste au niveau du sol, à hauteur de tête, et les collisions sont également activées pour empêcher de traverser les objets.

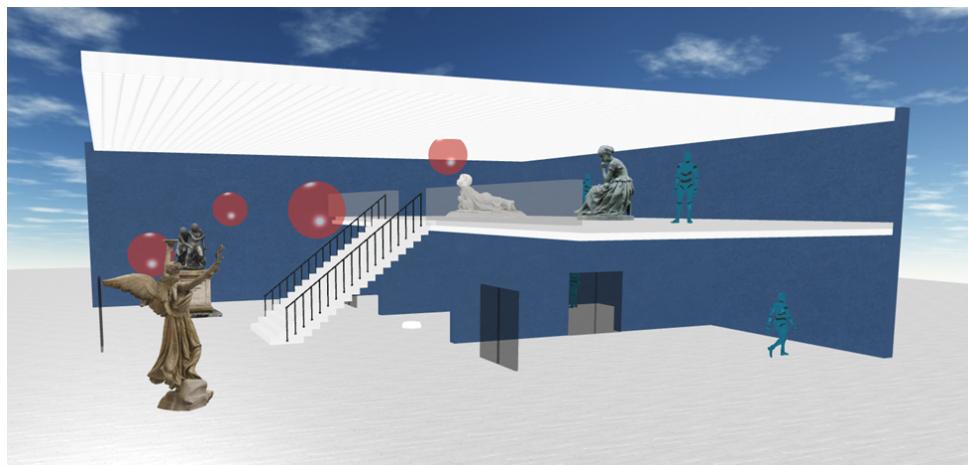


Figure 2 : Vue du musée sans certains murs et sans les tableaux pour apprécier la disposition physique.

L'un des défis que nous avons rencontrés lors de la mise en place de ce modèle initial était lié aux escaliers. Avec les collisions et la gravité activées, il était difficile de faire en sorte que le visiteur ne saute pas de manière étrange en montant les escaliers ou ne reste pas bloqué en essayant de les monter. Nous avons résolu ce défi avec succès en ajustant la largeur et la hauteur de chaque marche. Si elles sont trop grandes, le joueur ne peut pas monter. Si elles sont trop petites, il y a un mouvement de saut irréaliste.

Pour gérer la disposition des escaliers, nous avons utilisé une boucle qui continue de placer les éléments, car placer chaque marche individuellement ne serait pas pratique. La première rampe et la dernière marche reçoivent un traitement spécial pour être correctement positionnées.

```
for (let i = 0; i < stepNum; i++) {

    if ((i == 0) || (i == stepNum-1)){
        // CREATION AND POSITIONING OF HANDRAIL FOR FIRST STAIR
        const stairHandrailFirstTopL = this.createSphere("stairHandrailFirstTopL",
{material:railMaterial, diameter:handrailSphereDiameter}, scene);
        stairHandrailFirstTopL.position = new BABYLON.Vector3(-stairWidth/2,
(0.5+i)*stepHeight+handrailHeight, -(stairOffset-i*stepThickness/2));
        const stairHandrailFirstTopR = this.createSphere("stairHandrailFirstTopR",
{material:railMaterial, diameter:handrailSphereDiameter}, scene);
        stairHandrailFirstTopR.position = new BABYLON.Vector3(stairWidth/2,
(0.5+i)*stepHeight+handrailHeight, -(stairOffset-i*stepThickness/2));
    }

    // HANDRAIL BARS CREATION AND POSITIONING
    const stairHandrailBaseL = this.createSphere("stairHandrailBaseL",
{material:railMaterial, diameter:handrailSphereDiameter}, scene);
    stairHandrailBaseL.position = new BABYLON.Vector3(-stairWidth/2, (i+0.5)*stepHeight,
-(stairOffset-i*stepThickness/2));
    const stairHandrailBaseR = this.createSphere("stairHandrailBaseR",
{material:railMaterial, diameter:handrailSphereDiameter}, scene);
    stairHandrailBaseR.position = new BABYLON.Vector3(stairWidth/2, (i+0.5)*stepHeight,
-(stairOffset-i*stepThickness/2));

    const stairHandrailColumnL = this.createWall("stairHandrailColumnL",
{material:railMaterial, width:handrailWidth, height:handrailHeight,
depth:handrailDepth}, scene);
    stairHandrailColumnL.position = new BABYLON.Vector3(-stairWidth/2,
(i+0.5)*stepHeight, -(stairOffset-i*stepThickness/2));
    const stairHandrailColumnR = this.createWall("stairHandrailColumnR",
{material:railMaterial, width:handrailWidth, height:handrailHeight,
depth:handrailDepth}, scene);
    stairHandrailColumnR.position = new BABYLON.Vector3(stairWidth/2,
(i+0.5)*stepHeight, -(stairOffset-i*stepThickness/2));

    // STAIR CREATION AND POSITIONING
    const staircase = this.createWall("staircase", {material:stairMaterial,
width:stairWidth, height:stepHeight, depth: stepThickness*2}, scene);
    staircase.position = new BABYLON.Vector3(0, stepHeight*i,
-(stairOffset-i*stepThickness/2));

    if (i == stepNum-1){
        // CREATION AND POSITIONING OF LAST STAIR
        const staircase = this.createWall("staircase", {material:stairMaterial,
width:stairWidth, height:0.24, depth: stepThickness*2}, scene);
        staircase.position = new BABYLON.Vector3(0, stepHeight*i+stepHeight/2,
-(stairOffset-i*stepThickness/2));
    }
}
```

2.2. Matériaux pour structure physique

Afin de rendre le musée réaliste, nous avons ajouté des matériaux à la structure physique, y compris aux murs, au sol, au plafond, aux escaliers, etc. Nous avons utilisé un matériau de mur bleu pour les murs, qui rappelle celui de certains musées, une texture légère pour le sol et une texture de plafond qui simule des lumières. Nous avons obtenu ces matériaux simples à partir de sites qui les proposent gratuitement, notamment : <https://3dtextures.me/>, <https://polyhaven.com/> et <https://ambientcg.com/>.

2.3. Tableaux et Statues

Nous avons créé chaque tableau en utilisant la fonction qui nous a été fournie à cet effet : Factory.prototype.createPoster. Nous avons inclus entre 8 et 10 tableaux provenant de 5 auteurs différents, pour un total de 46 tableaux. Les tableaux de trois des auteurs ont été placés dans chacune des petites salles, un auteur dans le hall et un autre dans la mezzanine. Nous avons créé une fonction qui appelait Factory.prototype.createPoster 46 fois. Le code est répétitif. C'est quelque chose qui pourrait être amélioré, cependant, l'utilisation d'une boucle for n'aurait pas été simple en raison de la disposition irrégulière des tableaux et des murs parentaux. Une autre option serait d'importer les données de chaque tableau à partir d'un fichier texte.

Nous avons également placé 4 statues, 2 dans le hall et 2 dans la mezzanine, en utilisant la fonction intégrée de Babylon.js, BABYLON.SceneLoader.ImportMesh.

```
Factory.prototype.createStatue = function(name, scale, pos, file, scene) {
    let that = this
    BABYLON.SceneLoader.ImportMesh("", "./assets/stat/", file, scene, function(meshes) {

        let groupe = new BABYLON.TransformNode(name);

        for (var i = 0; i < meshes.length; i++) {

            // Set the scaling of each mesh to reduce its size
            meshes[i].scaling = scale;
            meshes[i].parent = groupe;
            //meshes[i].checkCollisions = true;

        }
        groupe.rotation.z = Math.PI
        groupe.position = pos
    });
}
```

Initialement, nous avions activé les collisions, mais cela générait problèmes extrêmes de performance lors des collisions avec les statues en raison de leur complexité, donc les collisions ont été désactivées.

Nous avons obtenu les meshes des statues sur sketchfab.com.

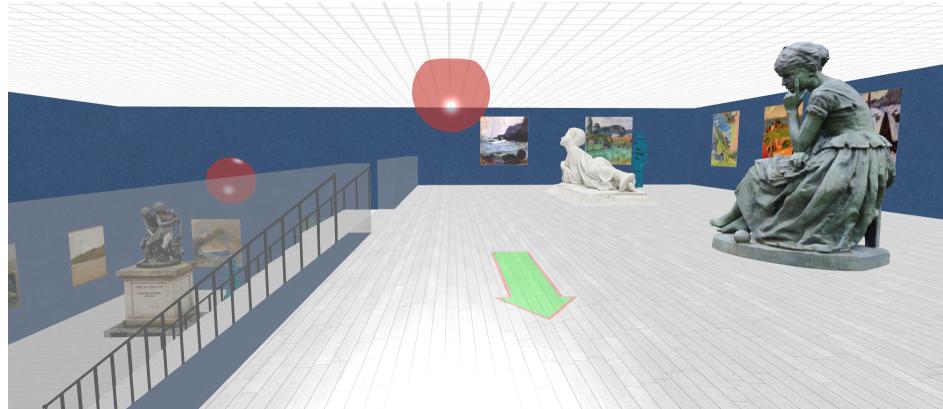


Figure 3 : Vue du musée depuis la mezzanine où l'on peut voir 3 statues et quelques tableaux.

2.4. Portes Coulissantes

Pour l'implémentation des portes coulissantes deux méthodes ont été mises en place. Un pour détecter la proximité du visiteur (caméra) et un autre pour détecter la proximité de chaque NPC dans le musée.

D'abord pour activer l'animation des portes et les ouvrir il faut changer une drapeau associé à chaque paire des portes et donc déclencher l'animation. Si le drapeau est 'faux' alors les portes se ferment par contre si le drapeau est 'vrai' alors les portes s'ouvrent. Le drapeau est défini dans le component 0 d'un tableau dans "colliderCollisions" et le comportement expliqué est traitée dans ce partie du code ci-dessous du module 'museum.js'

```
if((that.colliderCollisions.get(actor.name)[0])) {
    that.targets.get(actor.name).forEach((component) =>
component[0].forceGenerator[0].arriveOn(component[2]));

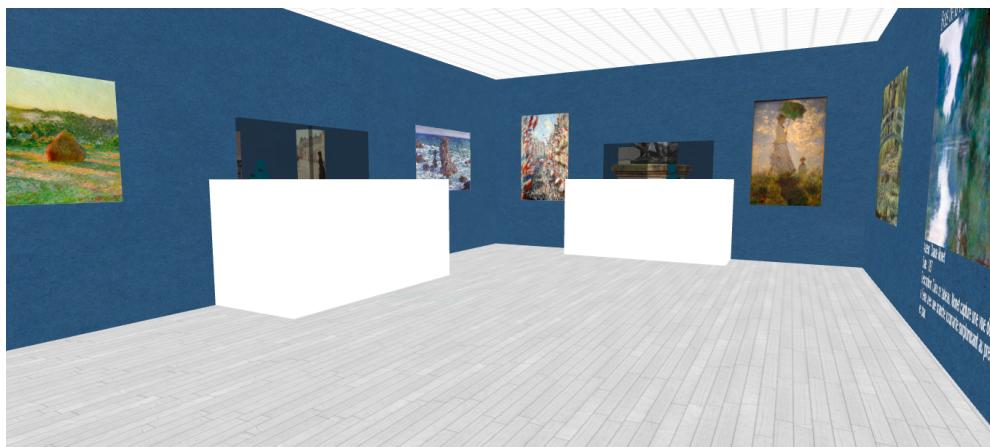
    if(that.clock - that.colliderCollisions.get(actor.name)[1] > 5) {
        that.colliderCollisions.set(actor.name, [false,
that.clock]);
    }
}
else{
    that.targets.get(actor.name).forEach((component) =>
component[0].forceGenerator[0].arriveOn(component[1]));
}
that.targets.get(actor.name).forEach((component) =>
{component[0].forceGenerator.forEach((g)=>{g.eval();})});
that.targets.get(actor.name).forEach((component) =>
{component[0].act(dt);});
```

La différence est dans la manière que la caméra et les NPC changent ce drapeau

Pour la caméra se calcule la distance euclidienne au centre de chaque paire des portes, si la distance ("dist" dans l'image ci-dessus) est inférieure à 6 alors le drapeau change à 'vraie' et les portes s'ouvrent.

Pour chaque NPC on utilise des collisions. Il y a une fonction asynchrone associée à chaque porte et chaque NPC qui vérifie à chaque frame si une collision se produit. Dès que la collision passe le drapeau est changé à 'vraie' et les portes s'ouvrent. Car cette méthode ne détecte que le moment où la collision se produit mais pas le moment où les objets arrêtent de collisionner. Donc une horloge est initialisée au même moment que la collision est détectée. Quand l'horloge dépasse 5 unités du temps, les portes se ferment.

Ci-dessous sont illustrés les mesh dès collision pour chaque porte:



2.5. Téléporteurs

Pour faciliter les déplacements dans le musée, nous avons ajouté 4 téléporteurs. Les téléporteurs se présentent sous la forme de sphères semi-transparentes rouges sur lesquelles l'utilisateur peut cliquer pour se téléporter à l'endroit où se trouvent les sphères. Nous avons placé une sphère sur la mezzanine, en haut des escaliers (afin qu'elle soit accessible/cliquable depuis le bas), deux dans le hall à côté des entrées des petites salles, assez haut pour pouvoir les cliquer depuis la mezzanine, et une à l'entrée du musée. Nous avons choisi de ne pas les placer à l'intérieur des salles car elles ne seraient ni visibles ni cliquables depuis l'extérieur, ce qui les rendrait inutiles. Les 4 téléporteurs sont clairement visibles sur les Figures 1 et 2.

Pour mettre en œuvre le mécanisme de clic, nous avons envisagé deux options. Une option consiste à ce que le visiteur puisse les cliquer où que le pointeur se trouve. Le pointeur n'est pas nécessairement au centre de l'écran, il peut être n'importe où et il est défini au début lorsque l'utilisateur clique sur l'écran pour verrouiller le pointeur. Cela n'a pas beaucoup de sens et n'est pas intuitif car le pointeur n'est pas au centre et n'est pas visible.

L'autre option est d'utiliser le “ray-casting” à partir de la caméra. Cela lance un rayon en utilisant la position et la direction de la caméra. Ce rayon est toujours positionné au centre de l'écran de l'utilisateur. On peut ensuite vérifier si ce rayon touche les maillages. Si lorsque l'utilisateur clique, le rayon touche l'une des sphères, il se téléporte à cet endroit. C'est ce que nous avons décidé d'utiliser. Ainsi, si une sphère se trouve au centre de l'écran du visiteur (et qu'elle est visible, c'est-à-dire qu'elle n'est pas bloquée par un autre objet), et qu'il clique, il se téléporte.

```
// Listen for clicks
this.canvas.addEventListener("click", function(event) {
    // console.log("click");

    // Cast ray
    var ray = that.camera.getForwardRay(200);

    // Save object that was hit
    var hit = that.scene.pickWithRay(ray);
    console.log(hit);

    // Check name of hit mesh. If it's a teleport, then teleport to position
    if (hit.pickedMesh.name == "teleporter1") {
        that.camera.position.x = 10;
        that.camera.position.z = -7.5;
        that.camera.position.y = 2.3;
    }else if(hit.pickedMesh.name == "teleporter2"){
        that.camera.position.x = -10;
        that.camera.position.z = -7.5;
        that.camera.position.y = 2.3;
    }else if(hit.pickedMesh.name == "teleporter3"){
        that.camera.position.x = 0;
        that.camera.position.z = -12.5;
        that.camera.position.y = 2.3;
    }else if(hit.pickedMesh.name == "teleporter4"){
        that.camera.position.x = 0;
        that.camera.position.z = 2;
        that.camera.position.y = 7.3;
    }
})
```

2.6. Noms et Descriptions des Tableaux

Pour donner au visiteur plus de contexte sur les tableaux qu'il voit, nous avons ajouté des noms et des descriptions à ceux-ci. Cependant, ils ne sont pas tous affichés en permanence. Afin de ne pas surcharger le visiteur d'informations et de textes, ils sont d'abord masqués. Lorsque le visiteur s'approche d'un tableau, le nom apparaît pour attirer son attention. Ensuite, s'il regarde le tableau pendant quelques secondes, l'auteur, l'année et une brève description apparaissent.



Figure 4 : Vue du tableau avec son nom et description.

Pour mettre en œuvre les noms et les descriptions, nous avons créé deux nouvelles fonctions dans `factory.js`, `Factory.prototype.description` et `Factory.prototype.title`. Elles prennent toutes deux en arguments le nom que l'objet aura, le tableau parent et la scène. Le tableau parent est important car les noms et les descriptions sont positionnés par rapport au tableau qu'ils décrivent. La fonction `Factory.prototype.title` prend également bien sûr le nom du tableau et la fonction `Factory.prototype.description` prend l'auteur, l'année et la description. Ces quatre éléments sont tous des chaînes de caractères.

Étant donné que c'est un aspect dynamique du monde, sa gestion est effectuée dans le fichier `museum.js`. L'objet `World` a comme attributs une liste de noms et une liste de descriptions, dans lesquelles les noms et les descriptions sont ajoutés lorsqu'ils sont créés. À chaque boucle de la boucle de rendu, la distance entre le visiteur et chaque tableau est vérifiée. La distance est calculée comme la somme des distances sur les axes `x` et `z`, et non comme la racine carrée de la somme de leurs carrés, afin d'économiser les ressources de calcul. Si la distance est inférieure à 7,5, les noms sont affichés et un timer est déclenché (en fait, on enregistre le temps de départ puis nous le soustrayons du temps actuel pour déterminer combien de temps s'est écoulé). Si la distance reste inférieure à 7,5 pendant 2,5 secondes, la description est également affichée. Si le joueur s'éloigne, le nom et la description sont à nouveau masqués. Une autre option pour la description aurait été de ne lancer le minuteur que si le joueur pointait directement le tableau, cependant cela aurait nécessité de faire constamment le ray-casting, à chaque boucle de rendu, ce qui entraîne d'énormes problèmes de performances (contrairement aux téléporteurs, où le ray-casting n'est effectué que lorsque le joueur clique), donc à la place, le timer est lancé lorsque le joueur est près du tableau.

```
that.titles.forEach((title) =>{
    // Calculate distance between player and painting
    that.distx[it] = that.camera.position.x - title.getAbsolutePosition().x;
```

```

that.distz[it] = that.camera.position.z - title.getAbsolutePosition().z;
that.dist[it] = Math.abs(that.distx[it]) + Math.abs(that.distz[it]);

// Check if it's less than 7.5
if(that.dist[it] < 7.5){

    // Make title visible
    title.isVisible = true;

    // Save time if it hasn't been saved
    if (that.start[it] == 0) {
        that.start[it] = new Date();
        if([it]==0)console.log(that.start[it]);

        // Check how much time has passed by subtracting start time from current
        time
    } else {
        that.now = new Date();
        that.time = that.now - that.start[it];

        // Make description visible if it is over 2500 ms
        if (that.time >= 2500) {
            that.descriptions[it].isVisible = true;
        }
    }

    // Make title and description not visible if player is further.
} else{
    that.start[it] = 0;
    title.isVisible = false;
    that.descriptions[it].isVisible = false;
}

// Increment counter for description list
it += 1;
});

// Reset counter
it = 0;

```

Ce code se trouve à l'intérieur de la boucle de rendu.

Un défi que nous avons rencontré était que les descriptions étaient trop longues pour tenir sur une seule ligne. Babylon ne permet pas de définir une longueur maximale de ligne, nous avons donc dû manuellement découper les lignes lorsqu'elles atteignaient un certain nombre de caractères (la première ligne avait même moins de caractères car elle commençait par "Description: " avant la description réelle). Babylon n'accepte pas non plus le symbole \n pour couper les lignes, nous avons donc dû placer manuellement chaque ligne en dessous de celle d'avant.

```

// Check if length of description is over 43 characters
if (desc.length > 43){

    // Add first line with first 43 characters

```

```

textContext.fillText("Description: "+desc.substring(0,43), 20, 100);

// Calculate how many additional lines are needed
let lines = Math.ceil((desc.length - 43)/ 55);

for (let i = 0; i < lines; i++) {

    // Write 55 characters on each line and place it below the line before
    if (i<lines-1){
        textContext.fillText(desc.substring(43+55*i,43+55*(i+1)), 20, 110+10*i);

        // For the last line, write however many characters are left (could be less
than 55)
    } else {
        textContext.fillText(desc.substring(43+55*i), 20, 110+10*i);
    }

}

// If the description is less than 43 characters, simply add one line with it
} else {
    textContext.fillText("Description: "+desc, 20, 100);
}

```

Ce code se trouve à l'intérieur de la fonction `Factory.prototype.description` que nous avons créée.

2.7. Signalement

Pour guider le visiteur, nous avons ajouté des signes de flèches qui indiquent la direction de la visite. Elles sont normalement cachées et s'éclairent progressivement lorsque le visiteur s'en approche. Nous avons également ajouté du son. Lorsque le lecteur est très proche de chaque flèche et qu'il est à pleine luminosité, un signal sonore est émis.

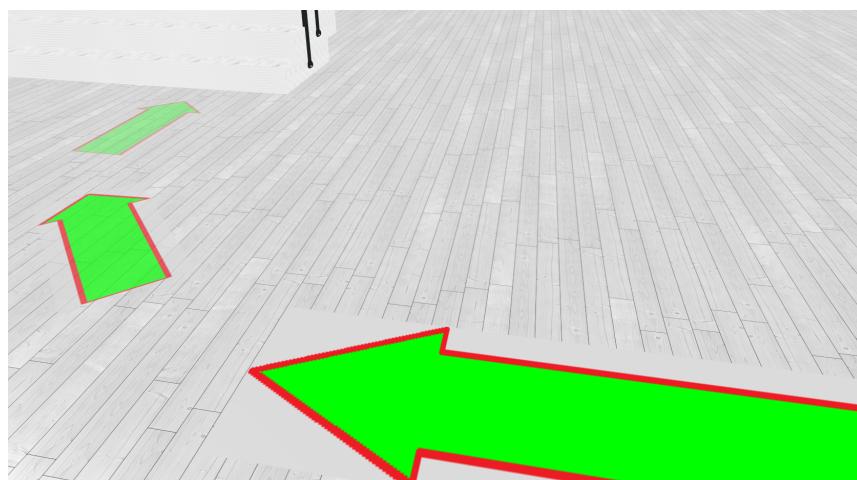


Figure 5 : Vue des signaux. On peut observer que les flèches les plus éloignées semblent plus ternes.

Comme les autres fonctionnalités dynamiques, cela doit être géré dans la boucle de rendu (render loop), faisant partie de la classe World, même si les flèches sont créées dans la classe

Factory. Ainsi, comme les titres et descriptions des tableaux, elles sont ajoutées à une liste qui est un attribut de la classe World. Ensuite, dans la boucle de rendu, nous vérifions la distance entre chaque signe et le joueur (comme pour les titres et descriptions des tableaux, nous calculons la distance comme la somme des distances sur les axes x et z pour économiser des ressources). Si le joueur se trouve à plus de 12 mètres, le signal de la flèche est complètement caché. Si le joueur s'approche à moins de 12 mètres, il devient progressivement moins opaque jusqu'à ce que le joueur se trouve à moins de 2 mètres, où il est complètement visible, et le son est joué. On contrôle que le son n'est joué qu'une seule fois lorsque vous vous approchez de chaque flèche afin qu'il ne soit pas constamment rejoué en raison de la proximité de la flèche.

```
// Loop through all arrow signs
that.lightSignals.forEach((lightSignal) => {

    // calculate distance between sign and visitor
    var distx = that.camera.position.x - lightSignal.position.x;
    var distz = that.camera.position.z-1 - lightSignal.position.z;
    var dist = Math.abs(distx) + Math.abs(distz);

    // If distance is less than 4, make the sign completely visible
    if(dist < 4){
        lightSignal.getChildren()[0].material.alpha = 1;
        if (light != its){
            that.sound.play();
            light = its;
        }
    }

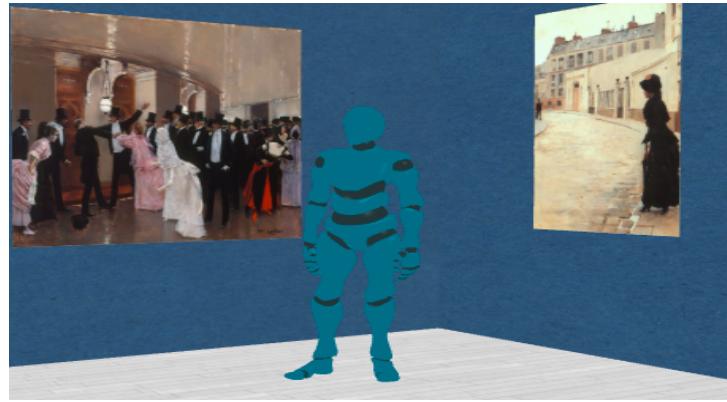
    // If the sign is between 4 and 12, make the sign semi-transparent, with
    // degree of transparency proportional to distance
    }else if (dist > 4 && dist < 12){
        lightSignal.getChildren()[0].material.alpha = - 0.125*dist+1.5;

    // If distance is more than 12, make sign completely transparent/invisible
    }else{
        lightSignal.getChildren()[0].material.alpha = 0;
    }
    its += 1;
});

its = 0;
```

2.8. Acteurs Virtuels

Pour les NPC nous avons utilisé le skeleton qui se trouve dans le site web de BABYLON s'appelle **dummy3.babylon**.



Il sont initialisés et associés avec une array qui contient l'ubication de chaque tableau que l'acteur virtuel doit visiter. Après il est enregistré une fonction asynchrone qui change le target de chaque acteur après une temps déterminé en sélectionnant de manière aléatoire le prochain tableau à visiter dans la liste assignée à chaque NPC. L'animation qui simule la marche est partie du fichier **.babylon** téléchargé.

```

var waitingTime = 0;
    scene.registerAfterRender(function () {

        let target = features.get('initialPos');

        if(!features.get('collisionState')[0]) {
            if (features.get('changePos')){
                let newTarget = 0
                while(true){
                    newTarget = Math.floor(Math.random() *
features.get('targets')[1].get('posterPoints').length);
                    if (newTarget != features.get('targets')[0]) break;
                };
                features.set('newPos',
features.get('targets')[1].get('posterPoints')[newTarget]);
                features.get('targets')[0] = newTarget;
                features.set('changePos', false);
            }

            target = features.get('newPos');
        }

        else {

            if (!mesh.intersectsMesh(features.get('collisionState')[1].mesh,
false)){
                features.set('collisionState')[0] = false;
                features.set('collisionState')[0] = null;
            }

            var directionTemp = new BABYLON.Vector3(features.get('newPos').x -
mesh.position.x, features.get('newPos').y - mesh.position.y, features.get('newPos').z -
mesh.position.z);
            directionTemp.normalize();
            var angleInRadians = (Math.PI / 2);
        }
    }
}

```

```

        var cosAngle = Math.cos(angleInRadians);
        var sinAngle = Math.sin(angleInRadians);
        var x = directionTemp.x * cosAngle - directionTemp.z * sinAngle;
        var z = directionTemp.x * sinAngle + directionTemp.z * cosAngle;
        target = new BABYLON.Vector3(x, features.get('newPos').y, z);
    }

    var distance = new BABYLON.Vector3(target.x - mesh.position.x, target.y - mesh.position.y, target.z - mesh.position.z).length();
    var direction = new BABYLON.Vector3(target.x - mesh.position.x, target.y - mesh.position.y, target.z - mesh.position.z);
    direction.normalize();
    const deltaDistance = 0.03;
    if(distance > 0.1){
        mesh.translate(direction, deltaDistance, BABYLON.Space.WORLD);
        if (walkRangeFlag) {
            mesh.lookAt(target);
            scene.beginAnimation(skeleton, walkRange.from, walkRange.to, true);
            idleRangeFlag = true;
            walkRangeFlag = false;
        }
    }
    else if(waitingTime++ > 200) {
        features.set('changePos', true);
        waitingTime = 0;
    }
    else if (idleRangeFlag){
        mesh.lookAt(target);
        scene.beginAnimation(skeleton, idleRange.from, idleRange.to, true);
        idleRangeFlag = false;
        walkRangeFlag = true;
    }
    else{
        mesh.lookAt(features.get('targets')[1].get('lookAtPoints')[features.get('targets')[0]]);
    }
});

```

Dans la création du chaque NPC est associé aussi une autre fonction asynchrone qui surveille si une collision se produit.

Une problématique technique dans ce partie qu'on a trouvé c'est qu'on n'a pas été capable de détecter que le collision avec les portes coulissantes. Les problèmes avec la détection de collisions entre deux NPC a été l'assignation d'un "physcallImpostor" à chaque NPC. Pour être capable de détecter des collisions, chaque "physcallImpostor" associée doit avoir une masse différente à 0. Au moment de changer ça caractéristique nous avons constaté que des comportements indésirables avaient lieu pendant les déplacement des NPC d'un tableau à l'autre

```

world.colliders.forEach((collider) => {
    collider.physicsImpostor.registerOnPhysicsCollide(mesh.physicsImpostor,
function(collideObject, collidedAgainst) {

    if (collideObject.object.name.includes('npc')) {
        console.log("collision NPC")
    }
});

```

```
        }

    else {
        world.colliderCollisions.set(collider.name, [true, world.clock]);
    }
}) ;
}) ;
```

3. Conclusion

Pour conclure on considère que ce projet nous a permis d'avoir une introduction suffisamment complète à Babylon.js et au développement de la réalité virtuelle basée sur le web. Cela nous a permis de commencer avec les concepts de base et de passer à l'échelle pour mettre en œuvre des choses plus avancées comme la gestion des collisions.