Rapport du projet REV

CHARLES Mathieu, MENDES Sébastien

Sommaire

Description

Dans le cadre du module de RV (réalité virtuelle), nous avons réalisé un musée virtuel en utilisant Babylons.JS. C'est un moteur 3D temps réel sous forme de bibliothèque JavaScript permettant d'afficher une scène 3D dans un navigateur web.

Figure 1: alt text

Guide d'utilisation

Lancement

Il faut ouvrir le fichier « index.html » avec un navigateur internet. Ce projet a été en partie développé avec le navigateur Mozilla Firefox, pour avoir un rendu le plus fidèle possible au développement, l'utilisation de celui-ci est recommandée.

Configuration d'affichage

- Ouvrir un onglet et entrez « about:config » comme URL.
- Dans la barre de la recherche du panel, cherchez «security fileuri.strict_origin_policy».
- Changer cette variable à False.
- Rafraîchir si besoin la page «index.html».

Déplacement

Pour bouger dans le monde virtuel, il suffit de se servir des flèches directionnelles ou des touches `Z,Q,S,D`.

Vous pouvez également vous servir des amers dissimulés un peu partout dans le musée qui permet en cliquant dessus de se téléporter à proximité.

Pour monter à l'étage vous avez acces à plusieurs moyens. Vous avez accès à l'escalier ou à l'ascenseur. Pour pouvoir utiliser l'ascenceur vous pouvez vous presentez devant les portes de celui-ci. Si l'ascenceur est la ils s'ouvriront sinon

Figure 2: alt text

vous êtes obliger de l'appeller grâce au bouton à gauche de l'ascenseur.

Figure 3: alt text

Tableau

Vous pouvez cliquer/approcher des tableaux pour pouvoir avoir une description de celui-ci.

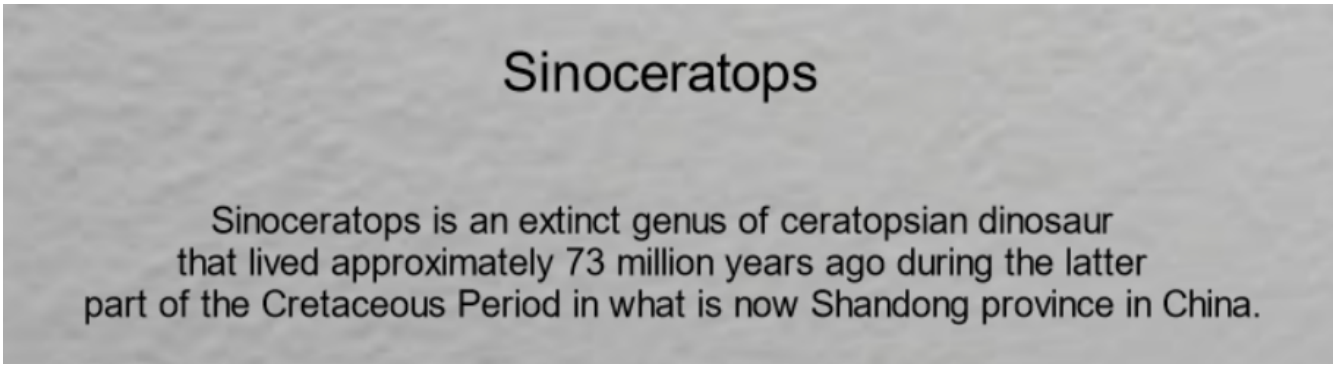

Figure 4: alt text

Thématique et plan du musée

Disposition du musée

Le musée ce décompose en 5 parties, nous avons 3 pièces de 15x5m et de 3m de haut, une entrée de 15x15m et de 6m de haut et pour finir une mezzanine de 15x15m et de 3m de haut.

- L'entrée est sur le thème du fossil préhistorique, nous pouvons y voir différents squelettes qui sont des `.obj` et différents tableaux représentant plusieurs espèces fossilisées.
- La mezzanine est sur le thème de l'archéologie, nous pouvons y voir des tableaux de sites de fouilles, les outils nécessaires à la fouille ainsi que quelques exemples de trouvailles archéologiques.
- Les pièces ont des thèmes qui reste dans la veine de l'entrée.
  - La première et la deuxième parlent des différents groupes de dinosaures avec des exemples de ceux-ci, les saurischiens, qui sont composés des théropodes et des sauropodomorphes et les ornithischiens qui sont

composés des cératopodes et des thyréophores. Les descriptions des tableaux exprimeront les divergences entre les groupes ainsi que la description du dinosaure visualisée.
- La troisième exprime essentiellement la description de la possible disparition des dinosaures par une météorite. Nous pouvons y voir différents tableaux présentant des études sur cette hypothèse. Nous pouvons également y découvrir un système solaire en mouvement.

Problèmes rencontrés

Le premier problème rencontré a été d'exporter des objets dans la scène, nous avons réussis à exporter exclusivement des `.obj`.

Par la suite, la difficulté a été d'en trouver, en effet le thème choisi ne donne pas énormément de possibilités de trouver des contenus téléchargeables gratuitement. Par ailleurs, le thème choisi nous a permis de découvrir un monde inconnu ou les musées du monde prennent en 3D leurs expositions et donnent en libre accès leurs visualisations sur des sites spécialisés d'objet 3D, mais ils sont très rarement téléchargeable.

Pour terminer, nous aurons voulu, avoir beaucoup plus d'objet 3D pour donner un peu plus de volume à notre musée, mais nous avons préférer conserver les performances de simulation.

Entity-Component-System

Afin de pouvoir manipuler facilement les interactions entre les différents éléments du programme, nous avons mis en place une forme d'ECS (Entity-Component-System)

L'intégralité de qui a été mis en place tourne autour de l'entité

Entité

```
class Entity {
    constructor(bObject_) {
        this.bObject = bObject_; //Babylon Object associated

        this.components = {};

        entityList.push(this);
    }

    update(dt) {
        for (var key in this.components) {
            if(typeof this.components[key].update != "function"){
                continue;}

            this.components[key].update(this, dt);
        }
    }

    delete() {
        for (var key in this.components) {
            removeComponent(this,this.components[key].name);
        }
        removeFromArray(entityList,this);
    }
}
```

Laquelle est donc associée à un objet BabylonJS (`bObject`).

Chaque entité créée est ajoutée à une liste globale d'entités permettant de les actualiser. Toute entité qui est retirée de cette liste ne sera donc plus mise à jour par le système. C'est d'ailleurs de que fait la fonction `delete()` : retirer l'entité de la liste globale.

Chaque entité a la fonction `update(dt)` qui se chargera d'actualiser tous ses composants, elle est appelée par le système pour chacune des entités enregistrées.

> ***Note :*** On peut voir dans la fonction `update()` de l'entité que l'on s'assure que chaque composant a lui-même une fonction `update()`. Cela a été fait car très peu de composants utilisent en réalité cette fonction. Il semblait donc intéressant de permettre l'absence de celle-ci.

Tous les composants sont stockés dans un dictionnaire dont la clé est `ComponentInstance.name`, ceci permettant d'avoir plusieurs composants de même type sur une unique instance. L'usage d'un dictionnaire a été décidé principalement pour des raisons de performance puisque Javascript nous offrait des performances bien plus intéressantes en utilisant un dictionnaire plutôt qu'un simple tableau.

Et l'usage d'un "nom" d'instance plutôt que seulement un type de composant pour stocker les composants permet théoriquement d'affecter facilement à une entité plusieurs composants de même type.

Système

L'actualisation de toutes les entités se fait via le "système". On utilise le `BABYLON.Engine` et sa "render loop" pour appeler périodiquement une fonction qui va actualiser toutes les entités :

```
var tlastCall = Date.now();
engine.runRenderLoop( function(){
    scene.render();
    var tnow = Date.now();
    updateEntities(tnow-tlastCall);
    tlastCall = tnow;
});
```

> ***Note :*** Ceci est définit dans l'initialisation du monde

La fonction `updateEntities(dt)` que voici :

```
function updateEntities(dt){
    for(let i = 0; i < entityList.length; i++)
    {
        entityList[i].update(dt);
    }
}
```

Se contente de parcourir toutes les entités enregistrées et de les actualiser.

Composant

Un composant de base aurait une structure de ce type :

```
class CTemplate {
    static get type(){
        return "Template";
    }
    constructor(name_= "Template") {
        this.type = CTemplate.type
        this.name = name_;
    }
```

```
    onComponentAdded(entity) {
        this.entity = entity;
    }

    update(entity,dt) {

    }

    onComponentRemoved(entity) {

    }
}
```

Éléments obligatoires d'un composant

Parmi ces éléments, les suivants sont **obligatoires** :

- `static get type()`

Cet accesseur est utilisé par les différentes fonctions de l'ECS pour permettre d'identifier facilement le type d'une instance d'un composant d'une entité. Il est donc obligatoire pour le bon fonctionnement de l'ensemble.

- `onComponentAdded(entity)`

Cette fonction est obligatoire, en particulier car c'est dans cette fonction que doit être faite l'association d'une entité à un composant (toujours en parlant d'instances). Il est donc particulièrement important de ne pas oublier l'instruction `this.entity = entity;`

Le choix de ne pas associer l'entité au composant lors de son instanciation a été motivé par plusieurs raisons comme le fait de pouvoir dissocier l'instanciation et l'association d'un composant. Ce qui peut être utile si on souhaite dupliquer un composant par exemple. Un autre avantage est de pouvoir instancier un composant, le modifier puis l'instancier, ce qui peut être pratique si l'on souhaite modifier une valeur avant l'association sans pour autant modifier le constructeur du composant. Mais cela permet aussi tout simplement de mieux distinguer les étapes en programmant.

Cette fonction est appelée lorsqu'un composant est ajouté à une entité.

- `constructor(name_= "Template")`

Le constructeur est bien sûr obligatoire puisqu'il permet d'instancier le composant. Mais surtout car il permet d'exécuter les instructions

```
this.type = CTemplate.type
this.name = name_;
```

La première permettant d'accéder à la fonction permettant de déterminer le type d'un composant (statique) depuis une instance et la seconde permettant d'affecter un nom à l'instance du composant. On notera d'ailleurs que la plupart des composants ont un nom codé "en dur" puisque destiné à n'être utilisés qu'une seule fois par entité.

Éléments optionnels

- `update(entity,dt)`

Cette fonction permet d'exécuter à une fréquence relativement élevée du code lié au composant associé à une entité. Le paramètre `dt` permet d'obtenir le temps écoulé entre deux appels à la fonction `update(entity,dt)`

- `onComponentRemoved(entity)`

Cette fonction permet d'exécuter une action lorsque le composant est supprimé d'une entité.

Composants créés pour le projet

- `CCollisions` : Active les collisions Babylonjs lorsqu'ajouté à une entité. Les désactives lorsque supprimé d'une entité. >***Note :*** Ce composant nous a obligé à écrire une fonction qui permet d'appliquer un changement à un objet BabylonJS et tous ses enfants
- `CCanInteract` : Ce composant permet d'ajouter une action à exécuter lors d'un clique sur un objet. A partir du moment où ce composant est ajouté, **l'objet sur lequel il l'a été sera mis en avant en affichant un cadre blanc autour de la zone clickable** >***Note :*** Ce composant base son usage sur le système événementiel classique du navigateur. Nous reviendrons plus tard sur son utilisation
- `COnPlayerNearby` : Permet d'exécuter une actions lorsque le joueur rentre/sort d'une zone définie par un point et un rayon (une zone sphérique donc).
- `CAnimation` : Permet d'animer un objet plus facilement qu'en utilisant directement l'API proposée par BabylonJS et plus efficacement qu'en utilisant uniquement l'ECS et en programmant les animations à la main.
- `COnStoppedMoving` : N'a pas été utilisé. Permet théoriquement de détecter quand une entité arrête de se déplacer.
- `CPeriodicAction` : Permet simplement d'exécuter une fonction périodiquement (avec un délai en ms)
- `CSlidingDoor` : Permet de créer facilement une porte coulissante
- `CRotatingDoor` : Permet de créer facilement une porte battante
- `CElevator` : Ce composant a été créé uniquement pour répondre aux besoins particuliers du système d'ascenseur.

Mécaniques du programme

Interactions cliquées

Surlignage / mise en valeur de l'objet cliquable

Nous avions besoin d'un moyen de mettre en valeur les objets cliquables afin de permettre à l'utilisateur de savoir si une interaction est possible ou non.

Pour cela, nous affichons la "hitbox" d'un objet sur lequel nous pouvons cliquer dès que le curseur au centre de l'écran passe ledit objet. Dès que la souris sors de la zone cliquable, la dernière "hitbox" affichée est masquée.

Afin de savoir quand est-ce qu'un objet est au centre de l'écran, nous utilisons une technique visant à projeter un "rayon" invisible dans l'axe de la caméra et si celui-ci rencontre un objet cliquable, nous le récupérons.

Tout cela est fait à l'aide d'un composant écrit pour la caméra :

```
class CCameraRayCaster {
    constructor() {
        this.name = "CameraRayCaster";
        this.lastSelected = null;

        this.delayFromLastCall = 0;
```

```
        this.UPDATE_DELAY = 200;
    }

    onComponentAdded(entity) {
        this.entity = entity;
    }

    update(entity,dt) {
        this.delayFromLastCall += dt;
        //cast a ray for highlighting selection every this.UPDATE_DELAY ms instead of always
        if(this.delayFromLastCall < this.UPDATE_DELAY)
        {
            return;
        }
        this.delayFromLastCall = 0;

        if(this.lastSelected)
        {
            this.lastSelected.showBoundingBox = false;
        }
        var pickResult = pickMesh(true);
        if (pickResult && pickResult.canInteract){
            this.lastSelected = pickResult;
            pickResult.showBoundingBox = true;
        }
    }
}
```

On notera que pour des raisons de performance, un rayon n'est projeté que toutes les 200ms. Cela n'entache pas l'expérience utilisateur mais permet d'économiser un peu les ressources.

Chaque rayon projeté étant une opération relativement coûteuse en terme de ressources.

Interaction

C'est de cette manière que l'on peut cliquer sur un élément et exécuter une action :

```
function pickMesh(testCanInteract=true){
    function pickPredicate(mesh){
        if(mesh.canInteract)//property defined by the component "CCanInteract"
        {
            return true;
        }
        return false;
    }

    var ray = camera.getForwardRay(40);

    var pickResult=scene.pickWithRay(ray);

    if (pickResult.hit){
        if(testCanInteract && !pickResult.pickedMesh.canInteract)
            return null
        return pickResult.pickedMesh;
    }
    return null;
```

```
}

window.addEventListener("click", function () {
    var pickResult = pickMesh(true);
    if (pickResult){
        pickResult.onInteraction();
    }
});
```

En réalité, lorsque l'on ajoute un composant "CCanInteract", nous donnons aux mesh associés (pluriels car on parle d'un mesh et de tous ses enfants) la propriété "canInteract" qui est ensuite utilisée par une fonction projetant un rayon lors d'un click pour savoir si le mesh obtenu en résultat doit avoir la possibilité d'intéragir ou non.

Si oui, nous cherchons à exécuter la fonction du mesh "onInteraction" qui est à nouveau une propriété rajoutée par le composant "CCanInteract".

Lorsque le joueur est proche

Nous utilisons pour cela un composant : "COnPlayerNearby" qui lui, possède une fonction `update(entity,dt)` qui est donc exécutée périodiquement.

Cette fonction teste la distance entre la caméra (la position de l'avatar) et une position donnée. Cette position correspond à la position du mesh par défaut mais peut être définie explicitement.

Si jamais le joueur rentre ou sort de la zone ainsi définie, une fonction est appelée en lui passant en paramètre l'entité concernée et est-ce que celle-ci est entrée ou sortie de la zone.

> ***Note :*** Le choix de n'appeler la fonction à exécuter que lorsque le joueur rentre ou sort de la zone et non pas dès qu'il est dedans était un critère important pour l'optimisation et la simplicité d'utilisation du code.

Animations

Cette partie fut relativement compliqué à mettre en place puisqu'il fallait trouver un moyen de rendre accessible les animations BabylonJS facilement utilisables avec notre système ECS.

Il a été fait le choix d'utiliser les animations BabylonJS pour plusieurs raisons : + Nous n'allons pas ré-inventer l'eau chaude... + Pour des raisons de performances. Puisqu'il est plus efficace de faire appel au système intégré plutôt que de coder des animations "en dur" avec des composants et des fonction "updates" + Cela permettait d'améliorer la généricité du code tout en maintenant un temps de programmation acceptable.

Voici le composant "CAnimation" utilisé pour réaliser les animations de notre projet :

```
class CAnimation {
    static get type(){
        return "CAnimation";
    }

    //example : field_="rotation.y" to make an object turn around Y
    constructor(name_,scene_,keyframes_,framerate_,duration_,field_,animationFunction = undefined) {
        this.type = CAnimation.type; //we must set this because JS is shit
        this.name = name_;
        this.scene = scene_;
```

```
        this.keyframes = keyframes_;
        this.framerate = framerate_;
        this.duration = duration_;

        // var trueFramerate = (this.duration >= 1)?
        this.animation = new BABYLON.Animation("compAnimation_"+this.name, field_, this.framerate/this.dur
        this.animation.setKeys(this.keyframes);

        if(animationFunction == undefined){
            this.animate = this._animate;
        } else {
            this.animate = animationFunction;
        }
    }

    onComponentAdded(entity) {
        this.entity = entity;
        this.entity.bObject.animations.push(this.animation);
    }

    _animate(reverse = false) {
        if(reverse)
            this.scene.beginAnimation(this.entity.bObject, this.framerate, 0, false);
        else
            this.scene.beginAnimation(this.entity.bObject, 0, this.framerate, false);
    }

    onComponentRemoved(entity) {

    }
}
```

Et un exemple d'utilisation pour les portes :

```
CSlidingDoor.onComponentAdded(entity) {
[...]
const frameRate = 60;
const animDuration = this.animDuration;
const keyFrames = [];

keyFrames.push({
    frame: 0,
    value: entity.bObject.position[this.axis],
});
keyFrames.push({
    frame: frameRate,
    value: entity.bObject.position[this.axis]+(this.travelDist*(this.reverseOrientation?-1:1)),
});

// console.log("----------------",entity.bObject.getBoundingInfo().boundingBox);

addComponent(entity,new CAnimation("sliding",this.scene,keyFrames,frameRate,animDuration,"position."+this.
addComponent(entity,new CCanInteract(function(e){
    var slidingDoor = getComponent(e,CSlidingDoor.type);
    slidingDoor.switchOpenedClosed();
}));
addComponent(entity,new COnPlayerNearby(function(e,hasEntered){
```

```
    var slidingDoor = getComponent(e,CSlidingDoor.type);
    if(hasEntered)
    {
        slidingDoor.setDoorState(true);
    }
    else
    {
        slidingDoor.setDoorState(false);
    }
},3.5));
[...]
```

Cette configuration va permettre de lancer l'animation BabylonJS à chaque clique sur la porte ou lorsque le joueur arrive à proximité de la porte (et la ferme lorsque le joueur s'en éloigne).

Élements du projet

Les portes

Il existe 2 types de portes et un composant a été créé pour chacuns d'eux : `CSlidingDoor` et `CRotatingDoor`. Ces composants permettent de créer facilement plusieurs portes qui partagent le comportement de s'ouvrir/fermer lors d'un click ou lorsque le joueur s'approche/s'éloigne de la porte.

L'ascenseur

Cet élément a été un réél défi à réaliser. Il était nécessaire de coordonner les animations et de proposer une expérience "proche de la réalité" afin d'éviter une gêne lors d'une utilisation potentielle en réalité virtuelle.

Un composant non-générique "CElevator" a donc été créé pour permettre la gestion et la synchronisation de toutes les animations.

L'ascenseur est en réalité constitué de 4 portes, 2 groupes de boutons et le corps de l'ascenseur (lequel possède des boutons).

4 portes car elles ne se déplacent pas avec l'ascenseur mais sont attenantes au bâtiment. Les 2 groupes de boutons correspondent aux boutons pour appeler l'ascenseur à l'étage et au rez-de-chaussé.

Le corps de l'ascenseur se déplace avec le joueur (il s'agit en réalité d'une "téléportation") si celui-ci est dedans.

Les escaliers

Cet élément a posé problème puisqu'il était nécessaire de permettre à l'utilisateur de les monter.

Or, en utilisant les fonctionnalités de base de BabylonJS, cela peut être permis en appliquant à la caméra une certaine valeur de gravité. En effet, BabylonJS autorisera alors à la caméra de monter sur des petits rebords si la valeur de la gravité est adaptée à l'environnement.

Problème : cette valeur de la gravité est dépendante du framerate. Donc selon la machine sur laquelle tournait le projet, nous pouvions parfois nous retrouver dans des situations où la gravité était trop élevée et ne permettait pas de monter les escaliers ou l'opposé où la gravitée était trop faible et l'avatar pouvait alors voler simplement en avançant en regardant vers le haut.

Pour régler ce problème, nous avons mis en place un moyen d'actualiser la valeur de la gravité dépendamment du framerate :

```
function createScene(){
    var scn = new BABYLON.Scene(engine) ;
    scn.collisionEnabled = true;

    //refresh scene gravity adapting to framerate every 3s
    var scnEntity = new Entity(scn);
    addComponent(scnEntity,new CPeriodicAction(function(entity){
        const earthGravity = -9.81;
        var fps = engine.getFps();
        entity.bObject.gravity = new BABYLON.Vector3(0.,earthGravity/fps,0.);
    },3000,"refreshGravity"));

    return scn ;
}
```

Le composant `CPeriodicAction` utilisé de cette manière nous permet de rafraîchir la gravité appliquée à la scène toutes les 3 secondes. Puisque nous récupérons le framerate actuel, nous pouvons la calculer dynamiquement et nous assurer d'avoir toujours une valeur correcte, proposant un fonctionnement stable du programme.

Affichage du titre du tableau proche

Cela nécessité plus de modifications que prévu. Nous utilisons une fois de plus le composant `COnPlayerNearby` mais certaines zones se recoupaient (en particulier à travers les murs).

Pour chacun des tableaux ayant cette fonctionnalité, nous l'avons donc relié à une sorte de "sémaphore" permettant de n'afficher que la première zone dans laquelle le joueur est entré. De cette manière, nous avons pu obtenir un résultat assez convaincant.

Pour éviter de surcharger l'utilisateur d'informations, nous n'affichons que le titre du tableau lorsque l'on s'en rapproche. Pour afficher la description, il suffit alors de cliquer dessus.

Ce choix a été fait afin de respecter le conseil donné dans l'énoncé de ne pas afficher trop de choses à l'écran en même temps. Et en particulier lorsque le joueur ne fait que se déplacer dans l'environnement.