Création de jeux avec BabylonJS

Support des cours

Document mis à jour le 26/01/2024 08:26

Version de brouillon

Historique des versions

Date	Version	Commentaires
25/01/2024	V1	Première version

Liste de diffusion

Arguimbau Olivier	1ex

Auteur : **Olivier Arguimbau**

Table des matières

Support 4 – Une première scene et du mouvement

Passage en JavaScript MODERNE (conversion du projet en classes).

Ajout d'un peu d'interactivité et de mouvements.

Utilisation des classes

Modification du fichier « index.js » :

On ne garde que l'initialisation du canvas et de l'engine, le reste est laissé à une autre classe.

```
import { Engine } from "@babylonjs/core";

import Game from "./game";

let canvas;
let engine;

const babylonInit = async () => {

    canvas = document.getElementById("renderCanvas");
    engine = new Engine(canvas, false, {
            adaptToDeviceRatio: true,
    });

    window.addEventListener("resize", function () {
        engine.resize();
    });
};



window.onload = () => {

    babylonInit().then(() => {
        const game = new Game(canvas, engine);
        game.start();
    });

}
```

On a modifié le code pour être plus propre lors de l'initialisation du DOM, dans la fonction « onLoad » on appelle l'initialisation de babylon (tiré d'un exemple officiel), enfin on instancie notre classe « Game » en lui passant les deux paramètres et on appelle enfin la méthode « start ».

Dans le répertoire src on va créer un fichier « game.js » et y déclarer notre classe :

En JavaScript les classes se présentent sous la forme :

```
class Game {
    #canvas;
    #engine;

    constructor(canvas, engine) {
        this.#canvas = canvas;
        this.#engine = engine;
    }
}
```

Deux propriétés en « privées » préfixées d'un hashtag (en Type Script on prefixera par le mot clé « private » et on remplacera le # par un _ (pour plus de lisibilité dans votre code).

Le mot clé « this » signifiant « mon instance courante » pour la classe.

PARENTHESE -- ATTENTION – PIEGE JAVASCRIPT –

En JavaScript tout est objet, aussi « this » même dans une classe change parfois de sens, dans une fonction par exemple « this » EST la fonction (heureusement pas dans une méthode membre.)
Exemple de piège :

```
class Game {

    start() {

        this.#engine.runRenderLoop(function () {
                  console.log(this) ; // this N'EST PAS L'INSTANCE DE Game
                scene.render();
        });
    }
```

Dans les anciennes version de JS on devait un peu bidouiller, maintenant vous pouvez utiliser les fonctions « arrows » qui réaffectent le « this » correctement.

```
class Game {

    start() {

        this.#engine.runRenderLoop( () => {
                  console.log(this) ; // this EST L'INSTANCE DE Game
                scene.render();
        });
    }
```

FIN DE LA PARENTHESE

Complétons notre classe Game :

```
    start() {


        const scene = this.createScene();
        this.#engine.runRenderLoop(function () {
                scene.render();
        });

    }
```

Ici la méthode start (appellée dans index.js) créé une scene par l'appel d'une autre méthode puis grâce à la méthode « runRenderLoop » de BabylonJS va updater notre scene, ici en ne faisant qu'un render.

```
    createScene() {
        const scene = new Scene(this.#engine);
        const camera = new FreeCamera("camera1",
            new Vector3(0, 5, -10), scene);
        camera.setTarget(Vector3.Zero());
        camera.attachControl(this.#canvas, true);
        const light = new HemisphericLight("light",
            new Vector3(0, 1, 0), scene);
        light.intensity = 0.7;
        const sphere = MeshBuilder.CreateSphere("sphere",
            {diameter: 2, segments: 32}, scene);
        sphere.position.y = 1;
        const ground = MeshBuilder.CreateGround("ground",
            {width: 6, height: 6}, scene);
        return scene;
    }
```

La méthode createScene issue du playground.

- Création de la scene
- Création de la caméra, lumière, positionnement
- Puis création d'une sphère et d'un sol

```
export default Game;
```

A la fin de notre fichier nous exportons la classe Game afin qu'elle puisse être référencée par index.js

Vous devriez pouvoir tester, pas de changement mais nous avons gagné en clarté.

Améliorations

Matériaux simples

Le gris c'est bof, aussi ajoutons un material pour le ground et la sphère.

A la fin de createScene (avant le return) ajouter :

```
        const matGround = new StandardMaterial("boue", scene);
        matGround.diffuseColor = new Color3(1, 0.4, 0);
        ground.material = matGround;
```

```
        const matGround = new StandardMaterial("boue", scene);
```

Ceci créé un Material Standard, on lui donne un nom et une scene (optionnel surtout si vous n'en avez qu'une)

```
        matGround.diffuseColor = new Color3(1, 0.4, 0);
```

Ceci affecte la couleur marron à la propriété diffuseColor (voir la doc détaillée ici : https://doc.babylonjs.com/features/featuresDeepDive/materials/using/materials_introduction)

```
ground.material = matGround;
```

On affecte le material à notre sol.

Ici nous n'avons pas utilisé de texture (image mappée sur un objet) c'est plus simple pour commencer.

```
        const matSphere = new StandardMaterial("silver", scene);
        matSphere.diffuseColor = new Color3(0.8, 0.8, 1);
        matSphere.specularColor = new Color3(0.4, 0.4, 1);
        sphere.material = matSphere;
```

On complexifie un peu les choses.

Rappel pour Color3 et les valeurs normalisées en général :

Color3 (ou Color4) accepte des valeurs entre 0 et 1, ces valeurs représentent la proportion de Rouge , Vert , Bleu (et Alpha).

Matériaux plus complexes

Allez sur le site : https://github.com/BabylonJS/Assets/tree/master/textures et téléchargez une texture, https://github.com/BabylonJS/Assets/blob/master/textures/floor.png par exemple.

L'enregistrer dans le répertoire « assets/textures » (le créer pour l'occasion)

Modifiez le code ainsi :

Avant la déclaration de la classe Game :

```
import floorUrl from "../assets/textures/floor.png";
```

Puis dans createScene :

```
matGround.diffuseTexture = new Texture(floorUrl);
```

Enlevez la couleur diffusecolor ou modifiez la si vous souhaitez.

Bonus Matériaux : Le bump map pour ajouter de la profondeur et du réalisme

Récupérer la texture suivante ici :
https://github.com/BabylonJS/Assets/blob/master/textures/floor_bump.PNG

L'enregistrer dans le répertoire « assets/textures »

Ajouter au début du code la ligne suivante :

```
import floorBumpUrl from "../assets/textures/floor_bump.png";
```

Et plus tard dans le code en dessous de la création de la texture du floor :

```
matGround.bumpTexture = new Texture(floorBumpUrl);
```

Ombres

Avant la création de la sphère ajouter les lignes suivantes :

```
const sLight = new SpotLight("spot1", new Vector3(0, 20, 20), new Vector3(0, -1, -1), 1.2, 24, scene);
```

Ceci va créer une nouvelle source de lumière en mode « spot » (la lumière hémisphérique de notre scène ne produit pas d'ombres)

Puis pour associer un générateur d'ombres à une source de lumière :

```
const shadowGenerator = new ShadowGenerator(1024, sLight);
shadowGenerator.useBlurExponentialShadowMap    = true;
```

Après la création du ground on indique qu'on souhaite qu'il reçoive les ombres :

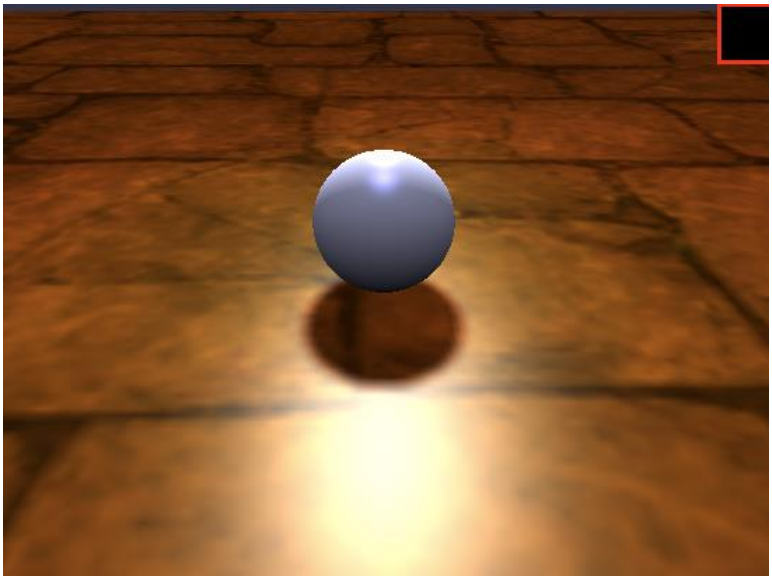
```
ground.receiveShadows = true;
```

Dernière étape on indique que la sphère projette des ombres :

```
shadowGenerator.addShadowCaster(sphere);
```

A insérer après la création de la sphère

Vous devriez avoir comme résultat :



Pour plus de détails sur les lumières :

https://doc.babylonjs.com/features/featuresDeepDive/lights

Et les ombres :

https://doc.babylonjs.com/features/featuresDeepDive/lights/shadows

Game loop

Comme évoqué dans un précédent document nous allons transformer notre scene en mini jeu en commençant par créer une game loop.

On ajoute la propriété #gameScene à notre classe (avant le constructeur) :

```
#gameScene;
```

On modifie légèrement notre fonction start :

```
    start() {
        this.initGame();
        this.gameLoop();
        this.endGame();
    }
```

La fonction start est assez simple.

Ensuite les trois fonctions précitées :

```
    initGame() {
        this.#gameScene = this.createScene();
    }
```

Ici un simple appel à notre fonction createScene, on stocke la valeur retournée

```
    endGame() {

    }
```

Vide pour le moment (et probablement pour toujours)

```
    gameLoop() {

        this.#engine.runRenderLoop( ()=> {
            this.updateGame();
            this.#gameScene.render();
        });
    }
```

Ici nous avons du changement, utilisation d'une fonction arrow pour pouvoir utiliser le mot clé « this » sans avoir à bidouiller, enfin un appel à la fonction updateGame restant à définir puis au rendu de la scene.

Et la fonction updateGame qui ne fait rien pour le moment :

```
    updateGame() {

    }
```

Essayez, pas de changement mais cela arrive.

Objets de jeu et mouvements

Nous allons modifier la fonction createScene pour nous rappeler des objets créés :

Après la création de la sphère ajoutez :

```
this.#sphere = sphere;
```

Et ajouter la propriété #sphere à la classe et une autre propriété :

```
    #sphere;
    #phase = 0.0;
```

Maintenant dans la fonction update :

```
    updateGame() {

        this.#phase += 0.03;
        this.#sphere.position.y = 2+Math.sin(this.#phase) ;
    }
```

En détail pour chaque ligne :

```
this.#phase += 0.03;
```

On augmente notre phase de 0.03 unités

```
this.#sphere.position.y = 2+Math.sin(this.#phase);
```

On modifie la position y de la sphere suivant une sinusoïde

Vous devriez voir la sphere monter/descendre régulièrement.

C'est fini ? Et bien non, car notre sphere se déplace mais que ce passe t'il si le PC est plus lent ? Ou si des ralentissements se produisent ? Et bien la sphere va ralentir et on ne veut pas ça dans un jeu.

Dans un jeu si un ralentissement se fait sentir on va perdre en images / secondes mais les vitesses de déplacement des objets dans le jeu doivent rester identiques, en gros si la sphere met 1 seconde à monter elle doit toujours mettre une seconde même si le PC tourne à 30 images par secondes et non 60.

C'est ce qu'on appelle une game loop déterministe ou « Deterministic Gameloop »

Détails ici : https://gameprogrammingpatterns.com/game-loop.html

Deterministic GameLoop

NOTE : Cette gestion des vitesses « adaptatives » est assez complexe au premier abord et vous pourriez être tenté de la zapper, mais votre jeu sera alors dépendant de la plateforme sur laquelle il est exécuté.

Le principe est simple : On doit adapter la vitesse de nos objets à la vitesse de dessin du navigateur. Actuellement nous augmentons notre phase de 0.03 à chaque tour de boucle.

Notre boucle est appelée 60 fois par secondes soit toutes les 16.67millisecondes

Si on détaille cela donne 0.03*60 = 1.8 unités / secondes. Donc notre souhait est de conserver cette variation de 1.8 unités/secondes quelle que soit la vitesse de rafraîchissement du navigateur.

On doit donc calculer notre variation par rapport à la durée réelle d'affichage d'une image (16.67ms idéalement).

BabylonJS nous fournit cette information grâce à la fonction `engine.getDeltaTime()`

Finalement notre variation n'est plus de 0.03 unités mais « X*delta = 0.003 »

Quand delta = 16.67 notre variation est de 0.03 on en déduit que X = 0.03/16.667 soit 0.0018 environ à 60 images / secondes.

Avec un tableau pour résumer :

Fps	deltaTime	facteur	Variation/seconde
60	16.67	0.0018	16.67*0.0018*60=1.8
30	33.33	0.0018	33.33*0.0018*30=1.79
144	6.944	0.0018	6.944*0.0018*144=1.79

On voit que la compensation marche aussi pour les machines « trop » rapides.

Notre variation idéale de 1.8 / seconde est respectée.

En code on écrit donc :

```
    updateGame() {

        let delta = this.#engine.getDeltaTime();

        this.#phase += 0.0018 * delta;
        this.#sphere.position.y = 2+Math.sin(this.#phase);
    }
```

Pour simplifier l'explication j'ai arrondi les calculs. Idéalement notre facteur serait plutôt de 0.0019 voire 0.002

Note bis : Les moteurs physiques ont eux aussi leur « deltaTime » qui peut être différent, il faudra en tenir compte.

Derniers mots sur cette solution : elle n'est pas idéale.. il en existe d'autres mais plus complexes.

Affichage des FPS

Dans la fonction gameLoop ajouter avant le runRenderLoop :

```
const divFps = document.getElementById("fps");
```

Puis après l'updateGame (toujours dans la fonction abritée par le runRenderLoop) ajouter :

```
divFps.innerHTML = this.#engine.getFps().toFixed() + " fps";
```

Avant `this.#gameScene.render();`

Prochain support : Input et intéractivité