WebGL - TP

Deshors Yann / Perignon Axel

January 9, 2018

WebGl nous permet de manipuler et d'afficher des artefacts en 3D dans une page html. Cependant pour un novice en 3D la prise en main peut s'avérer laborieuse, c'est pourquoi nous vous présentons une bibliothèque s'appuyant sur webGL mais plus simple à manipuler. Nous utiliserons dans ce TP la bibliothèque Javascript Three.js.

Le but d'aujourd'hui sera d'obtenir une sorte de vitrine montrant ce que l'on peut faire avec Webgl (une liste non exhaustive bien sûr). Les codes sources vous sont fournis. A vous de les modifier afin d'obtenir le résultat désiré. Attention, modifiez seulement les fichier Javascript!!! Les liens vers les documentations se situent à la fin de chaque sous partie, veillez à lire TOUTE la sous-partie concernée avant de commencer à coder. Bande de noobs!

Mise en place d'une scène Faire de la 3D c'est bien, encore faut-il pouvoir observer le fruit de son travail.

2 Fonctions seront nécessaires:

- Une fonction d'initialisation
- Une fonction d'animation (celle-ci sera appelée à chaque nouvelle image, donc en moyenne 60 fois par secondes)

La fonction d'initialisation devra avoir:

- Une caméra
- Une scène
- Un renderer

Il existe différents types de caméras, nous utiliserons celle du type 'perspective'.

Documentation Vous trouverez ici les liens pouvant vous aider:
Caméra: `https://threejs.org/docs/index.html#api/cameras/Camera`
Scène: `https://threejs.org/docs/index.html#api/scenes/Scene`
Renderer: `https://threejs.org/docs/index.html#api/renderers/WebGLRenderer`

Un objet avec une texture Le fichier à modifier est cube_texture.js.
Maintenant que nous avons notre scène nous allons y ajouter un objet. Cette objet sera un cube de dimension 1. Vous trouverez la texture en question dans le dossier "textures" sous le nom de crate.gif.

Les objets d'une scène sont des Mesh, des maillages. Ils héritent de la classe Object3D permettant principalement pour nous de manipuler leur position, leur orientation et leur dimension. Un Mesh a 2 composantes importantes:

- Sa forme géométrique, déterminée par un ensemble de triangles.
- Un matériel à appliquer sur cette forme géométrique. Le matériel n'est pas qu'une texture, il peut être défini par de nombreuses données (couleur, texture, transparence, réflexion, normal map, height map, déplacement/répétition de la texture, etc). Ici nous ne nous intéresserons qu'à l'application d'une texture.

Il va donc nous falloir une instance de la classe Mesh. Nous lui donnerons ensuite la géometrie de notre objet ainsi que la texture voulue. Bien évidemment, il faut ajouter cet objet à notre scène pour voir le résultat.

Nous allons maintenant animer notre objet sur 2 axes: x et y. Ces modifications s'effectueront sur les coordonnées de notre objet (le Mesh), dans la fonction animate. En effet, nous voulons que le cube bouge de manière continue et sans action ne notre part.

Les processeurs n'étant pas tous cadencés à la même fréquence, il ne faut pas que notre objet aille plus ou moins vite en fonction de l'ordinateur/smartphone. Nous utiliserons une "horloge" pour remédier à ce problème.

Petit indice: regardez du coté de la fonction delta.

Sphère et lumière Maintenant que vous savez créer un cube avec une texture, créez une sphère en 3D texturée elle aussi mais avec un material de type phong et non pas basic. Placez votre sphère 2 mètres au dessus de votre cube sinon vous aurez du mal à la voir, par défaut un objet se place en 0,0,0.

Vous aurez remarqué que votre sphère est bien moins visible, c'est à cause du material utilisé. Jusque là vous avez utilisé un material qui ne nécessite pas de lumière dans votre scène, sa "luminosité" est fixe et il ne projette pas d'ombre, très pratique pour un rendu rapide.

Pour voir votre sphère il va falloir ajouter une lumière à votre scène. Le plus simple est de commencer avec ambientLight qui n'est pas à considérer comme une ampoule, il s'agit plutôt d'une luminosité ambiante pour votre scène. Il n'y a pas de notion d'obstacles entre cette source de lumière et l'objet à éclairer, tout objet de la scène avec le material adéquat recevra cette lumière sur toutes ses faces.

Documentation Vous trouverez ici les différents liens pouvant vous aider.
Cube: `https://threejs.org/docs/index.html#api/geometries/BoxBufferGeometry`
Mesh: `https://threejs.org/docs/index.html#api/objects/Mesh`
Horloge: `https://threejs.org/docs/index.html#api/core/Clock`
Sphere: `https://threejs.org/docs/index.html#api/geometries/SphereBufferGeometry`
Material: `https://threejs.org/docs/index.html` et écrivez Mesh dans la barre de recherche

Lumière: `https://threejs.org/docs/index.html?q=loading#api/lights/AmbientLight`

Un objet déplaçable Toujours dans le même fichier.
Nous allons maintenant essayer d'interagir avec notre objet. Le cube et la sphère créés précédemment devront être déplaçables. La caméra devra l'être aussi. Plutôt que de vous faire coder le déplacement de la caméra relatif à un objet de la scène ou déterminer le plan orthogonal à la direction de la caméra sur lequel déplacer le cube, nous allons voir un exemple qui fait déjà cela et plus encore.

Il y a de nombreux exemples sur la page de Three.js qui vous montrent ce qu'il est possible de faire. Nous allons nous intéresser à celui-ci : `https://threejs.org/examples/?q=dra#webgl_interactive_draggablecubes`

Regardez ce qu'il est possible de faire, clic droit, clic gauche, clic/drag sur un objet ou dans le vide.

Cliquez sur le bouton "view source" en bas à droite de l'écran. La partie qui nous intéresse en particulier est THREE.DragControls. Implémentez cette fonction en adaptant les paramètres pour votre scène et ajoutez les 2 lignes suivantes pour gérer les évènements de début et fin de drag. Portez aussi votre attention sur la fonction render de cet exemple.

Une pause Vous qui êtes aptes, par vos nombreuses années d'étude, à comprendre les limites matérielles de l'informatique, je vous conseille fortement d'observer l'animation (environ 1 minute et demi) de cet exemple, cela vous donnera une idée des possibilités de Three.js, ne touchez pas à la molette de votre souris : `https://threejs.org/examples/?q=depth#webgl_camera_logarithmicdepthbuffer`

Désormais si vous voulez expliquer à quelqu'un la notion de logarithme/exponentialité, vous savez quoi leur montrer.

Importer un objet Fichier importation.js

Dans le monde du développement 3D, il n'est pas rare d'utiliser des modèles 3D créés par d'autres développeurs. Cela permet entre autre un gain de temps. Avec three.js c'est la classe OBJLoader2 qui permet ce genre d'opération (il y a aussi OBJLoader). Attention, même si cette classe est référencée dans la doc de three.js, il s'agit d'un script javascipt à part entière disponible dans le dossier exemples de leur github. Vous trouverez dans le dossier "3d files" un modèle à ajouter dans votre scène.

Ici c'est un peu délicat, il y a plusieurs fichiers pour un objet. Un fichier obj contient les éléments déterminant notre objet en 3D et un fichier mtl contient les materials de notre objet.

Vous n'avez pas à vous en occuper mais sachez que dans le fichier mtl il y a une référence à un fichier image png dont des portions seront appliquées sur notre objet 3D pour le "peindre". Ce fichier png doit rester dans le même dossier que le mtl et doit conserver son nom tel quel.

Le fichier sera donc chargé en 2 fois, comme on donne un material à un objet Mesh il vaut mieux charger le material d'abord.

Il nous faut plusieurs classes, heureusement pour vous les fichier js les contenant sont déjà téléchargés, placés dans le bon dossier et ajoutés au hmtl. OBJLoader2 dérive de LoaderSupport donc il nous faut ces 2 fichiers. Pour que certaines fonctions d'OBJLoader2 relatives au material fonctionnent il nous faut aussi MTLLoader.

Plutôt que de charger notre fichier mtl avec MTLLoader, nous allons utiliser la version d'OBJLoader2, fonction .loadMtl qui n'a pas de doc. Pour remédier à ça on va creuser, ligne 1347 du fichier OBJLoader.js.

5 arguments:

- url c'est le chemin de votre fichier mtl
- name le nom que vous voulez donner à votre fichier, pas très important pour nous
- content permet de charger votre fichier d'une autre manière mais cela ne nous concerne pas donc null
- callbackOnLoad le nom de la fonction qui nous permettra d'agir une fois le fichier chargé
- pas besoin du dernier argument, pour les curieux voyez ici `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`

Vous avez dû remarquer que jusque là il n'yavait rien de bien compliqué, une bonne partie du code est déjà présente, des commentaires dans tous les sens et vu qu'on ne veut pas que vous bloquiez sur des détails on vous aide et il vous reste sans doute pas mal de temps avant la fin du TP alors qu'il ne reste plus qu'une seule chose à faire.

Vous sentez venir le truc? Oui, maintenant on va attaquer des trucs plus sympas, j'espère que vous avez profité de votre pause.

Un mini jeu Il est maintenant temps de créer notre propre mini-jeux!

Vous connaissez le casse-brique? Bah on n'a pas le temps d'en faire un... Mais on peut faire des murs, une balle qui rebondit et une barre qui se déplace!! Alors on va faire tout ça.

On ne vous abandonne pas pour autant, direction fichier intersections.js. Suivez les instructions du fichier, à savoir création de la caméra, de la scène, des lumières. Des variables globales sont déjà présentes pour vous aider.

Revenez ici pour la création des boites.

Chaque boite à une géométrie similaire même si elles ne font pas forcément la même taille. Si vous créez une géométrie à chaque boite, votre GPU va inutilement devoir chercher quelle boite a quelle géométrie à chaque renouvellement de la vue.

Cependant comme nos murs et notre barre sont des parallélépipèdes rectangles, on peut se permettre de créer une seule géométrie, un cube d'un mètre de côté, puis de changer plus tard l'échelle de chacun de nos objets Mesh ayant cette géométrie afin d'obtenir nos murs et notre barre. Idem pour le material, on utilise le même pour chaque objet par soucis d'optimisation.

Le lancer de rayons On ne lance pas grand chose normalement. Pour déplacer notre barre sur un axe on veut connaitre la position relative entre notre curseur de souris/doigt sur l'écran du téléphone et la boite dans notre scène en 3D. Problème, la souris n'est pas dans la scène, en fait elle n'est même pas dans un environnement 3D.

Pire, je ne peux même pas m'adapter en disant que si je touche la gauche de l'écran je déplace la barre vers la gauche et inversement pour la droite. Si je déplace la caméra dans ma scène je peux regarder ma scène de dos, décalé, de côté, etc et mes directions seraient inversées. Il faut que l'on intéragisse vraiment avec notre scène en fonction de l'emplacement et de l'orientation de la caméra. Pour ça on a un truc très bien, imaginer une droite, plus exactement un rayon. Quand vous allez pointer un endroit de l'écran, on va faire partir un rayon depuis notre caméra qui passera par l'endroit pointé sur notre scène.

Problème, comment s'arrêter? Comment savoir si on veut que notre rayon s'arrête à 10 cm devant notre caméra ou est-ce que ja vais avoir le point xyz passant par mon rayon à 1 km de distance?

Voici venir le concept autour duquel repose ce mini-jeu, les intersections. Facile, on crée un plan ou ici une boite qu'on va considérer comme un plan. On veut que notre barre se déplace sur un seul axe et à la surface de ce plan. On va récupérer la position du point 3D à l'intersection de notre rayon et notre plan. Si on veut que notre boite ne se déplace que le long de l'axe X on peut simplement lui donner la valeur X de notre point.

Allez à la fonction startMovingBar() appelée à la fin du document, voir $("#cont_vue"). Suivez les instructions dans les commentaires.

Nouveau problème, Houston on ne voit plus bien ni la barre ni notre scène, quelqu'un a mis une grosse boite en plein milieu, soit-disant pour lancer des rayons dessus. Pas grave, on doit bien pouvoir la cacher, en effet voyez l'attribut visible true/false pour notre Mesh.
...
...
Ça alors, l'intersection ne marche plus? C'est ce qui arrive quand vous cachez un objet, c'est balot.

Bon on va utiliser un material différent alors, quelque chose de discret mais comme on est de bons informaticiens on n'a pas envie de créer encore un material. C'est là qu'on se rend compte que quand on crée un Mesh sans lui donner de material il en a un par défaut et on peut y accéder directement avec .material.

Vous pouvez réduire l'opacité à 0, comme ça elle sera toujours présente mais transparente. Puis dans la doc vous verrez que ça ne suffit pas, il faut aussi passer à true un certain attribut du material. `https://threejs.org/docs/#api/materials/Material.opacity`

Suivez le reste du fichier, créez une sphère pour notre balle d'1 mètre de rayon et ajoutez nos objets à la scène.

Intersections, des maths? Un peu de théorie, si je vous donne une instance d'une classe cube ayant une position xyz et dimensions largeur longueur profondeur, imaginez ce que vous auriez à faire pour déterminer quand un cube de votre scène rentre en collision avec un autre cube. Vous les voyez tous ces if

avec x, y, z supérieur inférieur + ou - largeur/2.

Rassurez-vous, on a des objets de catégorie math avec three.js qui font "presque" très bien le boulot, il y a une fonction pour vérifier si une instance de Box en intersecte une autre. Problème, vous avez vu le "presque" entre guillemets? Il se trouve que ces objets mathématiques de three.js sont assez variés, vous avez des Box2, des Box3, des plans, des cylindres et des sphères. Ces objets ont des fonctions tel que instersectsBox qui prend en paramètre une box et renvoie true ou false. Sphere a cette fonction, il y a aussi Box3 qui a la fonction intersectsSphere(). J'ai tout essayé, impossible de faire cohabiter ces fonctions d'intersection entre Sphere et Box, de plus on dirait que personne sur internet n'est au courant de l'existance de ces fonctions dans three.js.

Je n'ai pas envie de vous faire coder une fonction pour détecter l'intersection entre une sphère et un parrallélépipède rectangle même si je sais que vous en mourrez d'envie MAIS... Il se trouve que notre balle rebondit sur des éléments ayant tous des faces parralèles ou à 90°les unes des autres...

Notre sphère va rester une sphère mais quand on va détecter les collisions avec nos objets mathématiques on va considérer notre sphère comme une Box3. Aussi comme vous l'aurez peut-être déjà remarqué, nos Mesh ne dérivent pas de ces éléments, on va donc associer chaque Mesh de notre scène avec des Box3.

Créez autant de Box3 que de murs, barre et balle. Ensuite on va placer nos objets dans la scène, c'est déjà fait? Tant mieux, maintenant on est sûr que tout le monde a la même scène. Par contre vous ne vous demandez pas comment on fait pour que nos objets mathématiques aient les même coordonnées et dimensions que nos objets dans la scène? Retournez sur la doc de Box3 et voyez setFromObject()

Oui mais on a dit que notre sphère mathématique est un Box3 et dans la scène le Mesh correspondant dérive d'une géométrie de sphère. Pas grave le setFromObject détermine la boundingBox de votre Mesh tout en prenant en compte les hypothétiques changements d'échelle appliqués quelle que soit la géométrie utilisée.

Maintenant, fonction animate.

Déplacez vos objets en fonction de leur vitesse.

Mettez à jour les données de vos objets mathématiques.

Vérifiez s'il y a eu des collisions.

Ajustez la direction de la balle selon les collisions qu'il a pu y avoir. Cas particulier avec une collision sur la barre.

Félicitations, vous avez terminé! ou pas. S'il vous reste du temps et que vous êtes déjà arrivés ici, modifiez votre code pour créer un petit moteur physique. Vous avez vu comment déplacer votre sphère en fonction de sa vitesse.

Ajoutez une gravité à votre scène. La gravité est une accélération, elle influe sur la vitesse de votre balle.

Appliquez une accélération fixe sur votre balle dans la direction -y, testez différentes valeurs. Pour cela, à chaque renouvellement de la vue, variez la vitesse de la balle en y. newVitesse = oldVitesse -+ acceleration * temps écoulé.

Surtout, n'oubliez pas de limiter la vitesse maximale de votre balle. Comme votre vitesse est sur les deux axes x et y, on la représente par un vecteur. La taille du vecteur donne votre vitesse réelle, normez le à la vitesse maximale

désirée.

Ajoutez du code pour que cliquer sur la balle lui donne une impulsion/accélération sur y.