Rapport d’étude : Choix de la Technologie

# Introduction

Ce rapport va tenter de donner une vue globale de ce que sont les technologies Babylon.js et Three.js qui ont été choisies pour faire l’objet d’études avancées dans le cadre de la réalisation d’un synoptique 3D. Dans la conclusion de ce rapport doit se trouver une préconisation de la technologie la plus adaptée à la réalisation du projet.

# Rappel des technologies utilisées

L’étude s’est déroulées sur deux technologies qui sont Babylon.js et Three.js.

Babylon.js est un moteur de jeu présenté comme une bibliothèque JavaScript.

Three.js est un moteur 3D présenté également comme une bibliothèque JavaScript.

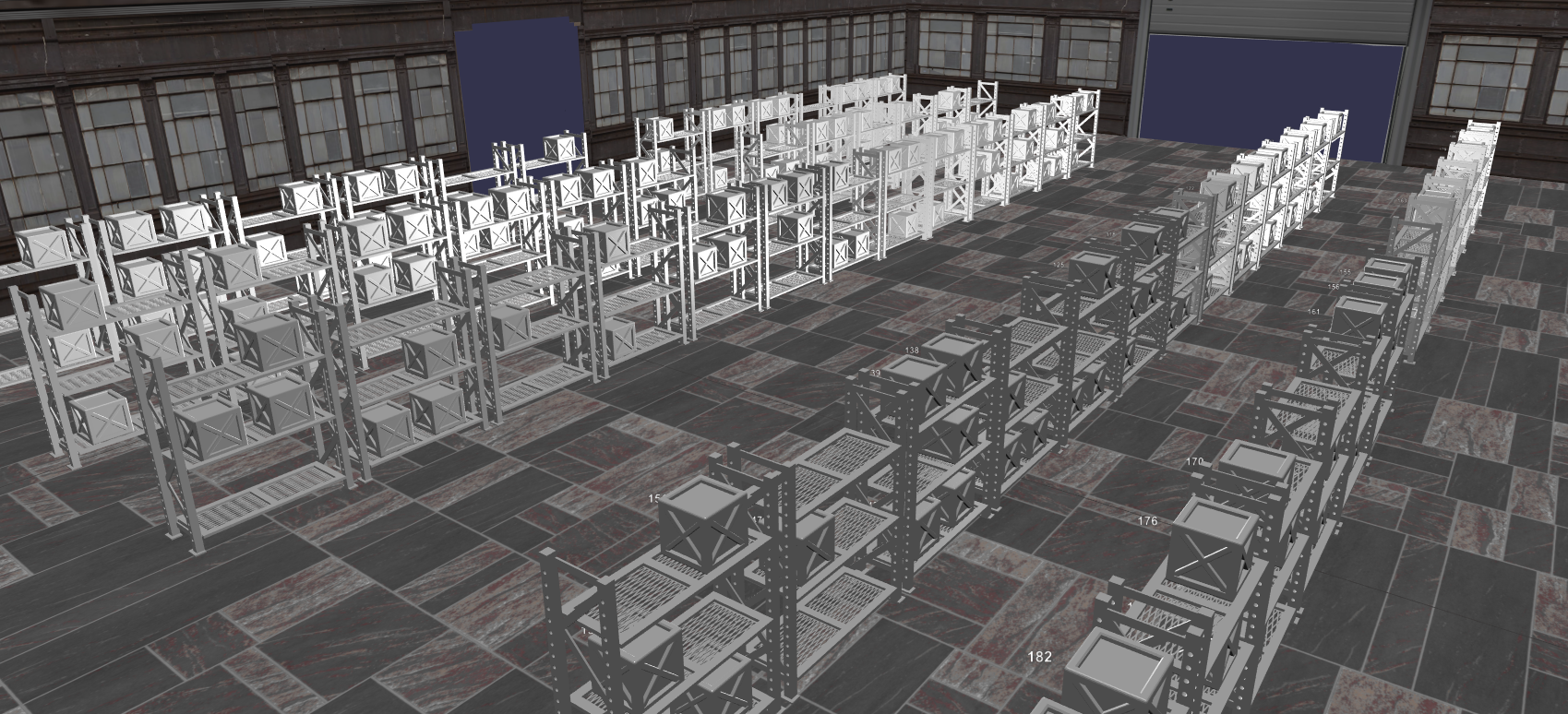
Tous deux sont basés sur WebGL qui est une API de 3D dynamique JavaScript apparue en 2011 créée par le Khronos Group. Elle permet d’utiliser les standards d’OpenGL ES dans une page web et sur bien d’autres supports. Elle permet également d’utiliser les pilotes d’OpenGL ES et d’utiliser la puissance que peuvent fournir les processeurs graphiques du terminal informatique.

Une introduction à chaque technologie testée avant de choisir Babylon.js et Three.js est disponible dans le premier rapport d’étude des différentes solutions disponibles.

# Présentation du projet

Le projet de test est bien entendu le même pour les deux technologies afin de définir plus clairement - non seulement au niveau du code mais également au niveau des performances et du rendu - laquelle est la plus adaptée aux besoins du synoptique 3D.

Le test doit mettre en scène un entrepôt industriel dans lequel sont disposées des étagères. En ligne les unes à la suite des autres, les étagères doivent avoir une place définie dans l’entrepôt. Plusieurs rangées doivent alors être générées. Sur chaque étagère des boîtes sont entreposées aléatoirement sur les emplacements prévus à cet effet. La photo si dessous illustre le rendu désiré. On peut constater que les étagères ne sont pas remplies de la même manière : chaque rechargement du contexte donnera une nouvelle scène avec des boîtes à des emplacements différents.



Chaque boîte doit être numérotée par un ID unique qui est affiché et qui permet de la rechercher via une barre de recherche. La photo ci-dessus laisse apparaître les numéros assignés aux boîtes.

La recherche d’une boîte doit zoomer sur cette dernière et permettre un déplacement orbital à 360° autour. Une fois la recherche terminée, on doit être capable de retourner à la caméra libre qui permet de se déplacer sans contrainte, via la souris et le clavier, à travers la scène.

Voici la liste de ce qui devait être implémenté en fonction de l’importance :

Impératif :

* Mettre en place la scène avec l’entrepôt, les étagères et les boîtes.
* Donner un ID aux boîtes, visible et unique.
* Générer les boîtes de manière aléatoire.
* Être en mesure de se balader dans la scène de manière libre via le clavier et la souris.

Optionnel :

* Pouvoir faire une recherche qui zoome sur la boîte et donne une vision orbitale autour d’elle.
* Pouvoir faire bouger une boîte, et si elle est taguée par la recherche, faire en sorte que la caméra suive son mouvement.

Les boîtes, les étagères et l’entrepôt doivent être des modèles 3D à charger grâces aux loaders de la bibliothèque.

Pour finir, il faut que le POC génère la scène à partir d’un modèle (à la MVC).

# Babylon.js

## Mise en place du projet

Il est aisé de commencer un projet Babylon.js de par le fait que des modules NPM sont déjà tous prêts à l’emploi et que le tutoriel de la documentation est bien clair quant à la mise en place d’un serveur à l’aide de Node.js. On regrette toutefois l’absence d’un tutoriel similaire pour mettre en place un serveur Web de type Jetty ou autre solution plus light et moins dépendante de Node.js.

Au niveau de la structure du projet, ce n’est pas très compliqué, il faut choisir avec NPM quel fichier sera le point d’entrée de l’application et lancer le serveur avec une commande simple : npx webpack-dev-server. Le reste est à l’appréciation du développeur, les dossiers, fichiers, ressources en tout genre ne sont pas tenus d’avoir des formats ou disposition particulière dans le projet.

Note : A la suite d’une incompréhension et de mon manque d’expérience, j’ai réalisé ce POC v2 sans mettre au point de modèle d’où charger les données. Je n’ai élaboré ce modèle que pour le POC v2 de Three.js et après ce rapport un POC v3 de Babylon.js avec le modèle devrait être mis en place.

Je ne serai donc pas en mesure de rapporter la partie adaptation du modèle au Framework pour Babylon.js.

## La mise en place de la scène

Les étapes de mise en place de la scène sont assez simples, il faut instancier le moteur, l’objet scène et ensuite ajouter le minimum, à savoir : une caméra, une lumière, éventuellement un « background ».

Que ce soit Babylon.js ou Three.js, les deux se basent sur la même technologie HTML pour effectuer le rendu : il s’agit de l’objet canvas qu’il suffit de créer dans un index.html. Il faut ensuite le passer au moteur de jeu pour qu’il puisse l’utiliser comme espace de rendu.

Il faut pour terminer créer la boucle qui permettra de rendre la scène en rafraichissant l’image un certain nombre de fois par seconde (environ 60) grâce au moteur de rendu.

Jusque-là, rien de complexe car tout est dit, clairement expliqué, documenté et des exemples de codes explicites sont fournis. Ainsi en quelques lignes, il est possible de poser les fondements de la scène.

En plus on peut déjà voir que la syntaxe n’est pas particulièrement difficile à appréhender.

La seule chose avec laquelle il ne faut pas se mélanger, est le fait qu’ils ont fait le choix de mettre le Y pour la hauteur là ou Three.js à choisis de mettre Z. A prendre en note.

## Les grandes lignes du projet

Babylon.js est fournis avec un ensemble de caméras assez diverses aux multiples et différentes utilités. Après une recherche très rapide sur internet, un forum annonce que le meilleur moyen pour implémenter des contrôles semblables à des contrôles à la Première Personne, est d’instancier une « FreeCamera ». Ceci fait, il faut changer son paramètre « attachControl » pour le mettre à « true » et lui passer l’objet « canvas ». Les contrôles à la souris et au clavier sont terminés et près à être utilisés.

### Les loaders

Les loaders Babylonjs sont au nombre de 3 pour le moment, un loader .OBJ, un loader .STL et un loader .GLTF. Il est possible de fait appel à ces loaders en ajoutant « babylon-loaders » aux « node-modules » du projet. Une fois présent dans le projet, ce module permet via un code assez simple de charger n’importe quel fichier 3D dans le projet, du moment que c’est un .obj/stl/gltf (qui sont les formats les plus répandus). Pour le reste des formats, Babylon.js propose dans la [documentation](https://doc.babylonjs.com/) des moyens de transformer les fichiers en .babylon via des exporter et des plugins.

L’importation d’un fichier : après l’appel du loader, on lui passe le chemin d’accès au fichier, puis une fonction callback (qui vient avec son lot d’options toutes détaillées sur la documentation). Elle permet notamment d’adapter le modèle 3D à la scène via des modifications de position initiale, d’échelle, etc.

|  |
| --- |
| BABYLON.SceneLoader.ImportMesh("", WAREHOUSEMODEL\_PATH, WAREHOUSEMODEL\_NAME,  scene, function(newMeshes){  *//newMeshes est le paramètre qui permet d'accéder au tableau où sont tous les imports que vient d'effectuer le loader*              warehouse = newMeshes[0]  *//Positionnement de l'objet à l'emplacement 0;0;0*              warehouse.position = BABYLON.Vector3.Zero();          }); |

Attention, il n’est pas possible de modifier les propriétés d’un objet via la variable qui l’a hébergé pendant le chargement, en dehors de la fonction.

Afin de contourner cette restriction, il est recommandé d’utiliser la méthode « getMeshByName() » qui permet comme son nom l’indique de récupérer un objet via le nom qu’on lui a donné. On le stocke ensuite dans une variable et on peut lui appliquer tous les changements désirés.

La nomenclature des paramètres des objets est simple est claire, elle permet la modification assez aisée des objets importés dans la scène.

### La génération aléatoire des boîtes

Il s’agit là d’un algorithme qui va donc très certainement différer d’un développeur à un autre. La problématique a été résolue en définissant les 6 points de l’étagère où peuvent apparaitre les caisses. Une fois en connaissance des points relatifs à l’étagère, il est simple de faire un tirage aléatoire pour savoir si on instancie une caisse à cet endroit ou non.

Par ailleurs le Framework propose une méthode faite pour la génération d’un grand nombre d’objets copiés les uns des autres : la méthode « createInstance() » qui permet de renvoyer une nouvelle instance de l’objet sur lequel la méthode est appelée. Cela permet de ne pas faire appel plusieurs centaines de fois au loader, qui est une entité gourmande en ressources.

### L’affichage des ID sur les boîtes

Ici il fallait choisir entre un texte, ou une interface graphique utilisateur. L’interface graphique utilisateur étant cliquable, le choix s’est donc porté sur le module npm « babylon-gui » à ajouter aux modules déjà présents. Le bouton est positionné sur un plan transparent qui sert d’interface cliquable.

A noter une mécanique particulière au monde de la 3D, l’objet « parent » d’une entité : le parent d’une entité est son référentiel. Par exemple dans le cadre du bouton créer plus haut, la boîte est définie comme parent du plan. Ainsi, la position du plan est relative à la boite et non plus à la scène.

Sur le bouton a été ajouté un écouteur d’évènement qui permet de lancer une fonction au choix quand on clique sur l’ID de la boîte.Une question persistait à ce niveau : comment faire en sorte que les ID soient toujours face à la caméra qui peut bouger ? Simple, le Framework vous donne ceci :

plane.\_billboardMode = BABYLON.Mesh.BILLBOARDMODE\_ALL;

Il suffit de cette ligne pour faire en sorte que le plan, et donc le bouton qu’il héberge soient toujours face à la caméra.

### La problématique des caméras

Une fois les boutons faits, il fallait mettre au point le système de zoom et de vue à 360°.

Deux options : soit faire un script qui se lance une fois qu’on clique sur le bouton, et paramétrer la caméra pour la bloquer sur l’objet.

Ou bien créer une deuxième caméra, cette fois de type ArcRotate qui permet presque sans paramétrage de visionner un objet qu’elle prend pour cible à 360°. Et ensuite jouer avec la propriété de la scène « activeCamera » qui permet de définir quelle caméra est active.

La deuxième solution est la plus rapide et la plus simple à implémenter avec un écouteur sur le bouton. Par la suite, il a simplement suffi de mapper le retour à la camera Free sur la touche ‘Echap’ du clavier et le tour était joué.

Une fois la deuxième caméra créée, il suffisait de la récupérer pour être en mesure de mettre en place la fonction de recherche.

### La fonction de recherche

Il est simple dans Babylon de rechercher un objet qui a un nom. Comme mentionné plus haut il suffit d’utiliser la méthode « getMeshByName() ». Une fois qu’on a l’objet, on a accès à toutes ses propriétés. De cette façon, on prend la position qui n’est autre qu’un vecteur 3 et on le donne en cible à la caméra orbitale qui se chargera de faire le focus toute seule. Pour la barre de recherche c’est le module « gui » installé auparavant qui permet d’avoir accès à un « InputText ». Une fois instancié, il n’est pas difficile de faire une fonction capable de prendre le texte que l’on rentre dans la barre de recherche et de le passer à la fonction de recherche d’objet. Un écouteur d’événements sur la touche entrée était de mise pour valider le choix sans passer par un bouton qui aurait pris un peu plus de place dans l’interface. Quand on a l’objet, on fait le changement de caméra et le tour joué.

### Finalisation et bonus

A ce stade, le POC est presque terminé et il ne reste plus que la boîte en mouvement à mettre en place. Pour ce faire, il faut opérer un changement sur le vecteur 3 qui définit la position de la caisse. La seule petite particularité de ce changement est qu’il doit s’effectué dans la boucle de rendu, afin que la position soit mise à jour à chaque image. En donnant un nom à la boîte lors de son instanciation, on s’assure qu’elle puisse faire l’objet d’une recherche.

Pour le déplacement le choix s’est porté sur une utilisation de la fonction cosinus qui permet une oscillation et donc un aller-retour de la caisse. Une fois que tout ceci est en place, tout se fait seul, aucunement besoin de renseigner quoi que ce soit à la caméra. Si la caisse en mouvement fait l’objet d’une recherche via son nom ou si l’on clique sur son ID, la caméra suit automatiquement son mouvement. Par la suite, la caisse volante a été remplacée par un chariot qui transporte cette caisse pour plus de réalisme.

## Fin du POC V2 Babylon.js

Globalement, ce projet de POC version 2 fût assez simple et le plus long à chaque fois était d’aller vérifier quoi faire, quelle notion utiliser dans telle situation. Le plus difficile et ce qui ralentissait le plus l’avancée du développement, était de ne pas connaître le monde de la 3D mais surtout de ne pas connaître le Framework. On sent la volonté de faire du monde complexe de la 3D, quelque chose de simple à appréhender. On n’a globalement pas à se soucier de grand-chose de très « 3D » sur ce genre de projet et chaque fois qu’une question est venue, non seulement les forums ou la documentation bien fournie y répondaient, mais en plus, un grand nombre de questionnements sont souvent adressés très simplement et par des commandes, propriétés déjà prêtes. On sent la volonté d’affranchir le développeur des questionnements relatifs à la 3D pour laisser ce dernier se concentrer sur la partie fonctionnelle et sur ce qu’il a vraiment à faire : mettre en place sa scène, son jeu, sa démonstration, etc.

### Les points forts

* La simplicité de ses commandes et de sa syntaxe
* Des problèmes complexes sont souvent adressés par de simple paramètres ou fonctions prêtes à l’emploi
* La communauté active et chaleureuse, toujours prête à aider
* La documentation en deux parties : une partie tuto/explicite et une partie purement technique avec l’ensemble des propriétés, méthodes relative à chaque élément
* Les forums bien fournis pleins de conseils pratiques
* Un code relativement compact
* Des chargements dans les formats standards de 3D simples et efficaces

### Les points faibles

* L’acclimatation avec la 3D est toujours un peu déroutante au départ (pas forcément lié à Babylon.js)
* La variable utilisée dans le loader n’est pas utilisable par la suite en dehors de la fonction pour continuer de paramétrer l’objet, et ce même si elle est déclarée avant.
* Le positionnement des caisses sur les étagères qui demande de faire du pas à pas pour faire en sorte que cela rende bien
* Uniquement trois loaders, ce qui est peu, même s’ils fournissent de quoi exporter le reste des fichiers dans leur format

BabylonJS est un framework facile à prendre en main et ce même pour un débutant (JavaScript ou 3D) et il offre une très bonne expérience sur le sujet. Il oriente véritablement le développement sur une approche fonctionnelle du projet avec très peu de technique pure.

# Three.js

## Mise en place du projet

Contrairement à Babylon.js qui conseille l’utilisation de Node.js, Three.js préconise un serveur Web pour faire tourner les projets. Bien qu’ils aient des modules pour Node.js, il faut noter qu’il est appréciable qu’ils montrent la démarche à suivre pour aller vers une alternative complète à Node.js. Bien que pratique, rapide et complet Node reste parfois un obstacle au développement à plusieurs, ou à l’intégration du projet sur un serveur de production qui embarque une autre technologie.

Pour ce projet c’est le serveur Web Jetty qui a fait tourner le POC. Léger et rapide, Jetty n’est pas très contraignant, il permet même via un fichier xml, de déployer une application qui n’est pas présente dans son dossier webapps. En revanche, à chaque changement au niveau du code, il faut vider le cache du serveur, sans quoi ce dernier ne met pas la page à jour.

Le reste ressemble beaucoup à Babylon.js, pas besoin de structure particulière, simplement de télécharger manuellement la bibliothèque afin d’être en mesure de l’utiliser dans le code.

Le principe est toujours le même, un fichier index.html avec un canvas pour héberger le rendu, puis du JavaScript.

## La mise en place de la scène

La mise en place d’une scène avec Three.js est semblable à celle de Babylon.js. On créé une scène, une lumière, une caméra (on remarque un choix moins grand de caméras pour Three.js), on instancie un moteur de rendu et le tout dans une boucle qui va afficher toutes les images suivant le taux de rafraîchissement possible. Three.js a une syntaxe assez simple et vraiment similaire à Babylon. Pas de noms trop compliqués ni de principes coriaces, là aussi la simplicité est de mise.

En parallèle avec les quelques heures de familiarisation avec le framework, il y a eu la mise au point du modèle de données qui viendra se câbler sur Three.js via des fonctions ou classes passerelles.

Aucun problème particulier dans la mise en place de la scène. Toutes les manipulations sont assez basiques et ne requièrent pas des heures de recherche. D’ailleurs un point positif demeure la documentation qui est très agréable à utiliser, fournie et toujours liée au GitHub. De même, les forums grouillent de solutions et de questions, donc on ne se sent pas seul largué sur cette technologie.

## Les grandes lignes du projet

### Les contrôles

Avec Babylon.js, il suffit d’implémenter une caméra particulière pour ajouter des contrôles clavier à la scène. Avec Three.js, il faut implémenter l’exemple « PointerLockControls » qui permet d’assigner la souris à la caméra pour ce qui est de l’orientation de cette dernière. Pour le déplacement, il a fallu développer avec un écouteur clavier, l’action de chaque touche afin de permettre d’avoir un déplacement multidirectionnel. C’est quelque chose qui est relativement lourd à mettre en place bien que dénué de véritable défi algorithmique. Là où l’appel de la caméra et son paramétrage prennent trois courtes lignes sur Babylon.js ; implémenter tout cela avec Three.js demande plus de ligne, plus d’objets avec notamment une horloge (pour le delta qu’elle peut nous fournir entre deux frames), et rendent le code moins plaisant à produire et à lire.

### Les loaders

Les loaders ThreeJS ont été testés pendant les quelques heures de prise en main du Framework au tout début du POC v2 avec Three.js. Et le gros avantage que possède Three.js sur Babylon.js, c’est qu’une grande partie des formats connus de modélisation 3D est supportée en import direct. Il y a un loader par extension de fichier et une grosse vingtaine de loaders en tout. Pour importer le fichier c’est assez simple, cela ressemble à Babylon avec le path, une fonction callback, la modification interne à la fonction puis l’impossibilité de modifier l’objet par la suite.

Seulement une fois dans le développement de la scène, impossible de charger la scène à partir du modèle de données.

Le problème venait du l’inaccessibilité des objets une fois chargés. Même problématique donc que BabylonJS. Cependant, impossible également de leur donner de nom. Ainsi, même visibles et bien chargés dans la scène, ils demeuraient inaccessibles.

Ce bug non résolu a bloqué le développement.

De multiples tentatives de debug et de logique ont été mises en place plusieurs jours durant pour tenter de remédier à ce blocage mais sans succès.

### La génération aléatoire des caisses

Pour cette partie-là c’était relativement simple, il faut le faire directement dans le modèle. Il faut passer les objets du modèle à des fonctions passerelle entre le Framework et le modèle, et créer une boîte 3D dans la scène calquée sur la boîte du modèle. Et c’est notamment ici que devait être instancié le nom de la boîte 3D via le nom du modèle. Il aurait suffi ensuite d'utiliser la méthode « getObjectByName() » similaire à BabylonJS. Mais s’il est impossible de mettre en place un nom pour le modèle 3D nouvellement chargé, alors cela compromet complètement l’avancée du développement.

### Le reste du projet

Comme Three.js n’est pas allé plus loin, les réflexions sur les caméras, les boutons, le module de recherche, etc, n’ont pas spécialement été approfondies. En revanche il est à noter que ThreeJS est généralement plus verbeux, plus capricieux. Un exemple serait au niveau de la rotation des objets : sur Babylon.js, les objets peuvent être orientés en ne renseignant qu’un seul des trois axes. Sur Three.js il faut forcément renseigner les 3 axes, même si c’est pour effectuer des rotations de 0 sur deux des trois axes. Et au final c’est un ensemble de petits détails comme celui-ci qui mis les uns avec les autres font une différence notable dans la lisibilité, la praticité, l’accessibilité, etc.

## Fin du POC v2 Three.js

Globalement le framework n’est clairement pas dénué de sens pour un projet comme celui-ci. Il est un peu plus verbeux et capricieux certes mais cela reste une bibliothèque agréable à utiliser. La communauté est active et disponible de même que les possibilités sont infinies. En revanche il est très certainement moins recommandable pour les débutants et sera peut-être à la longue, un peu compliqué à maintenir.

### Les points forts

* La communauté, la documentation, les forums : énormément de ressources très bien faites
* La syntaxe globalement agréable et simple d’utilisation
* La présence d’exemples parlants
* Les nombreux modules qui facilitent grandement le développement (PointerLockControls…)
* Les nombreux loaders
* La facilité de déploiement

### Les points faibles

* La mise en place des contrôles
* Les loaders impossibles à faire fonctionner
* L’arborescence du projet : les exemples sont considérés comme des bibliothèques mais sont quand même dans le dossier exemples…
* Trop verbeux par moment
* Parfois trop complexe pour les débutants

# Conclusion

Ma recommandation parait je le pense évidente : il s’agit pour moi de Babylon.js qui remplit tous les critères haut la main et va même au-delà. Je ne pense pas que Three.js soit mauvais pour autant, mais cela dit je doute que cette technologie soit parfaitement adaptée à ce que l’on recherche pour la mise en place du synoptique 3D. Je ne suis pas un professionnel de la 3D et je ne pense pas que ceux qui reprendront mes travaux seront forcément du métier, alors pour cette raison je pense qu’il vaut mieux s’orienter vers Babylon.js qui rend une copie remarquable. Un bon ensemble cohérent entre le paramétrage à souhait et cette idée de Framework qui nous éloigne vraiment bien des problématiques poussées et trop complexes que soulève la 3D.

# Annexes

Site de Babylon.js : <https://www.babylonjs.com/>

Documentation Tuto : <https://doc.babylonjs.com/>

Documentation API : <https://doc.babylonjs.com/api/>

Documentation « HowTo ? » : <https://doc.babylonjs.com/how_to/>

Forum 1 : <https://forum.babylonjs.com/>

Forum 2 Ancien Forum : <https://www.html5gamedevs.com/forum/16-babylonjs/>

Forum 3 : <https://stackoverflow.com/questions/tagged/babylonjs>

GitHub : <https://github.com/BabylonJS/Babylon.js>

Site de Three.js : <https://threejs.org/>

Documentation : <https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene>

Aide : <https://threejsfundamentals.org/>

Forum 1 : <https://discourse.threejs.org/>

Forum 2 : <https://stackoverflow.com/questions/tagged/three.js>

GitHub : <https://github.com/mrdoob/three.js/>

Jetty : <https://www.eclipse.org/jetty/>

Node.js : <https://nodejs.org/en/>