Rapport de Projet n°2

Option Réalité Virtuelle

Application web pour l'affichage des fichiers BIM/ifc

Entreprise Planexo

Antonio Bevilacqua, Willis Pinaud, Wei Zhou et Ye Zhou

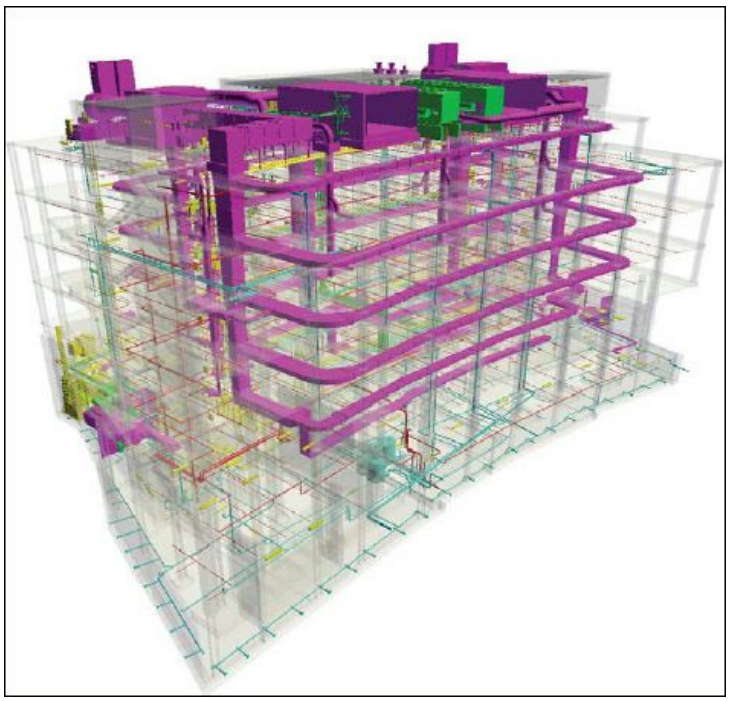


Table des matieres

Introduction

Ce projet consiste dans le developpement d'une application web pour l'affichage de fichiers de type BIM-IFC. Le sujet a été proposé par l'entreprise PlaneXo, qui a collaboré avec les membres de l'équipe dans le developpement du projet.

Contexte

Secteur d'application

Ce projet s'inscrit dans le process de transition du format papier au numerique dans le secteur des construction de batiments, qui est encore très lié à des pratiques obsolétes. Plusieurs action sont en marche pour cette cause comme le **Plan Transition Numérique dans le Bâtiment** (**PTNB**) ou le **Centre scientifique et technique du bâtiment** (**CSTB**). Le standard **BIM** (Building Information Model) est relativement recent et est à la base de ce process.

Le BIM est un modèle unique du bâtiment ou d'un ouvrage bâti, pouvant tenir dans un fichier numérique, lequel comprend toute l'information technique nécessaire à sa construction, son entretien, ses réparations, d'éventuelles modifications ou agrandissements et sa déconstruction. Le BIM, ainsi créé lors du processus de projet du bâtiment peut être utilisé jusqu'à sa démolition puis servir d'archive. Il a été utilisé pour la premiere fois à la fin des année 80, seulement dans les dernieres années il a commence à être utilisé systematiquement sur une large echelle. Le principal format qui implemente le standard BIM est le **IFC** (*Industry Foundation Classes*). C'est ce format que notre application web devait permettre de visualiser.

Entreprise

PlaneXo est une société qui propose une plateforme numérique qui se veut être un espace de travail innovant pour faciliter la collaboration et le partage d'informations entre tous les acteurs de la construction d'un ouvrage. Cependant leur produit ne permet pas l'affichage dans le navigateur des fichiers IFC. L'ajout de cette fonctionnalité serait très apprecié par les utilisateurs.
C'est pour ces raisons que nous avons été chargé de réaliser un navigateur web de fichier IFC.

Cahier des charges

La demande de l'entreprise est une application web **multi-platformes** et **Opensource**, qui permet d'afficher des fichiers IFC, ses spécificités sont (informations données par PlaneXo):

- Coté client
    - Récuperer les données du serveur (websocket)
    - Afficher – dessiner
    - Determiner framework
    - UI – mobile et ordinateur (selection et upload du fichier)
    - UI – en WebGL (pour interaction 3D et selection d'objets)

- Côté serveur
    - Lire le fichier IFC (lib existantes)
    - Parsing du fichier IFC (=> blocs d'objets)
    - Socket pour envoyer les IFC
    - Ex: IFC.send('give me the first floor')

En plus :

+ Publication du code en ligne (github) + Open source
+ Documentation de l'API de communication
+ Documentation interne du fonctionnement du viewer 3D

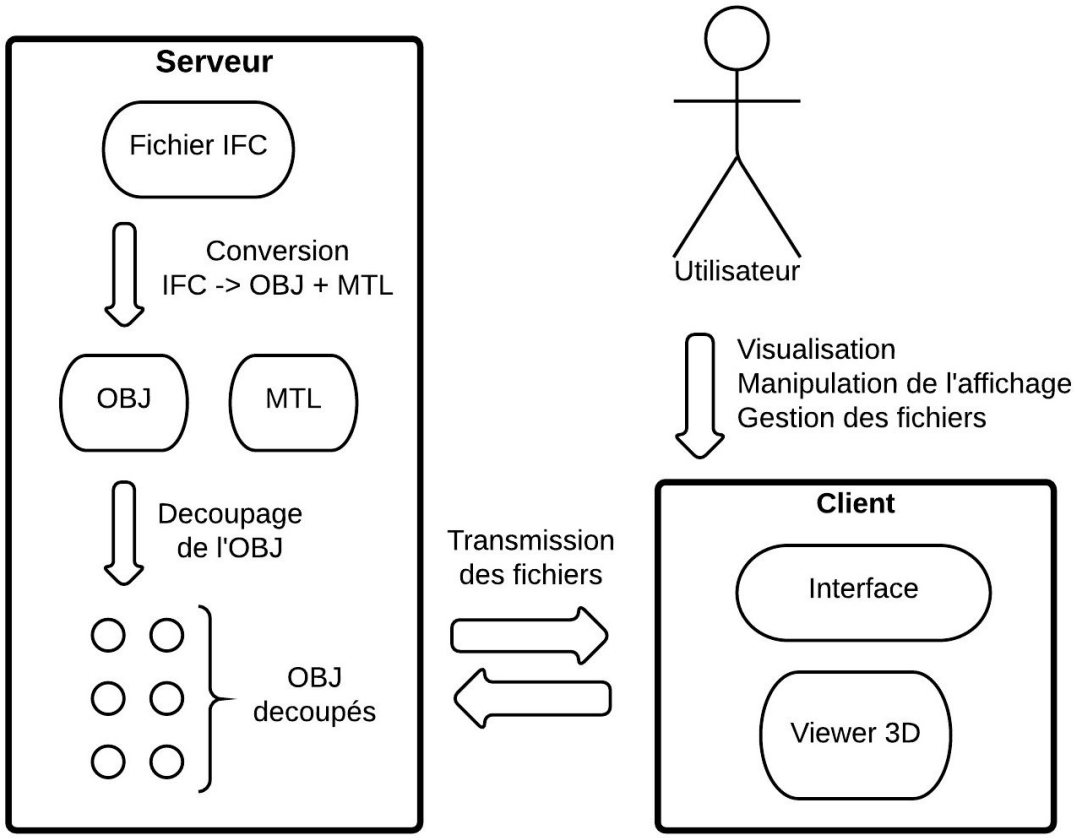
Probléme

Nous avons noté ici les principaux obstacle à la réalisation du projet, qui nous ont influencés dans nos choix techniques.

- **Efficience et performance** : le téléchargement des fichiers ifc entre le serveur et le client peut entrainer de long temps temps de chargement et nécessite une gestion des fichiers adaptée
- **Compatibilité mobiles** : la compatibilté mobile n'est pas evidente en affichage 3D, soit pour des contraintes de performance, puissance de calcul et espace de stockage limité
- **Comprehension et Parsing des fichiers IFC** : le format IFC est très complexe à comprendre à cause de la grande variété de données qu'il contient, la structure orienté objets et une manque de documentation publié

Solutions Choisies

Stratégie de résolution du problème



- *Decoupage de l'OBJ* : L'OBJ est decoupé selon les groupes de materiaux
- *Viewer 3D et Manipulation* : L'utilisateur peut selectionner des objets, les cacher ou les afficher et naviguer

Client

Viewer 3D

La technologie choisie pour l'affichage est la librairie Three.js, une librairie Opensource qui permet de rajouter une couche d'abstraction à WebGL.

Ça a permit de nous faciliter plusieurs taches, comme la gestion des Interactions 3D, l'affichage de fichiers OBJ et MTL, puis les fonctions graphiques de base comme la camera et les lumiéres.

La premiere raison de notre choix c'est la grande diffusion de la librairie et la compatibilité qui rentre dans notre cahier des charges.

Framework de l'application

Dans un premier temps nous avons effectué nos tests directement en javascript.
Or nous avons rapidement rencontré un problème du à la communication entre la partie html du site web et la partie javascript : pour mettre à jour les données sur la vue du client il fallait forcement rafraichir la page. Or nous ne voulions pas que l'experience utilisateur soit diminuée par cette difficulté technique.

Notre solution a été d'employer un framework supportant l'écriture direct entre html et javascript (double data binding).

Parmis la diversité des choix nous avons opté pour le framework de google : AngularJS.
Il dispose d'une grande communauté et surtout d'une documentation très complète et toujours accompagnée d'exemples. Cela nous a permis de nous approprier rapidement.

Serveur

Communication (nodeJS)

Nous avons choisi nodeJS parce que:

- NodeJS est écrit en JavaScript, et il est facile d'apprendre
- Il marche vite, avec une très haute efficacité.

Conversion (IfcOpenShell)

(répondre à la question : dans quel but a t on utilisé IfcOpenShell ? Pourquoi avoir choisi cette solution ? Quelles sont ces fonctions?

- Nous avons besoin de lire les datas dans les fichiers **ifc** et les transformer en ceux qu'on peut arranger; **IfcOpenShell** est un outil opensource pour lire les fichiers **ifc** qui est facile d'utiliser. Nous avons utilisé l'un de ses outils : **IfcConvert** pour transformer les fichiers **ifc** en **obj**.

Démarche

1) Documentation

Dans un premier moment nous avons fait des recherches sur les solutions techniques à choisir en fonction de notre cahier des charges.

- Pour l'**affichage 3D** notre choix a été entre X3D, Vulkan et Three.js (qui a été choisi). X3D était une bonne option, pour performance et compatibilité, mais ça nous aurais fallu apprendre une nouvelle technologie sans assez de temps pour la maitriser. Vulkan est très nouveau comme logiciel et nous avons pas trouvé assez de documentation pour l'evaluer. La choix est tombé sur Three.js, parce qu'il colle nos besoins et un plus nous avions déjà de la familiarité avec ce type de pipeline de visualisation 3D.
- Nous avons fait beaucoup de recherches de documentation pour la **gestion BIM et IFC**. Mais du au temps reduit et à la complexité de la question, nous avons choisi d'utiliser une application externe opensource, IfcOpenShell, qui nous a permit de mettre à coté la difficulté.

2) Répartition des tâches

Suite au choix de stratégie de résolution du cahier des charge nous avons réparti les tâches en deux grands pôles, la partie client avec Antoine et Willis et la partie serveur avec Wei et Ye.

a) Client

i) Réflexion sur fonctionnalités attendues

À partir du cahier des charges nous avons reflechis à quelles fonctionnalités implementer dans le client, donc l'interface et l'affichage.
Les modalités de communication et de gestion des fichiers du client influencent fortement l'interface, donc avant de établir un dessin d'interface il fallait prevoir les aspect plus techniques de fonctionnement.
Nous avons dessiné des modèles préliminaires de l'interface web et que nous allons presenter dans la suite.

ii) Mise en place du viewer 3D

Afin de lire et de visualiser les fichiers OBJ dans le navigateur web nous avons utilisé threeJS.
ThreeJS est un simple script javascript. Il a une logique d'arbre de scène, comme OpenSceneGraph.
Il a donc été naturel pour nous d'utiliser cette bibliothèque.
Dans un premier temps il a fallu regarder les exemples proposés par les développeurs sur le site de threejs, ainsi nous avons pu identifier les principales fonctions :

Instancier un nouveau moteur de rendu (WebGL) : new THREE.WebGLRenderer()

Créer une caméra : new THREE.PerspectiveCamera(FOV, ratio, min, max);

Créer une lumiere de type pointLight de couleur blanche : new
THREE.PointLight(0xFFFFFF);
On accède à la localisation de cette lumière à l'aide de l'attribut position qui est un vecteur 3D.

Les lumières, les mesh sont tous des objets. Ces objets doivent être ajoutés à la scène pour pouvoir être visualiser lors de l'appel du renderer.

On peut facilement créer une nouvelle scène avec : new THREE.Scene();
Puis ajouter un élément à cette scène avec scene.add(element)
Enfin, pour visualiser notre élément dans cette scene on fait appel au renderer via la commande :
renderer(scene,camera); Cela permettra de dessiner la scene vue par la caméra.

Nous avons donc créé une nouvelle scene avec plusieurs lumières afin de pouvoir accueillir nos objets.
Ajouté cette scene au moteur de rendu.
Ajouté un module permettant de capter les ordres recu par l'utilisateur : cliques de souris, scroll avec la mollette. (Nous avons utilisé plusieurs librairie au cours du développement de l'application web afin d'optimiser la navigation aussi bien sur ordinateur que sur les terminaux mobiles.)

Pour finir créé une fonction animate() comprenant un callback requestAnimationFrame récursif (prenant animate en argument). Cela permet à la scene d'être dynamique.

Enfin à l'aide d'angularJS nous avons pu utiliser une fonction OnWindowsResize() appellée automatiquement lors de redimensionnement de la fenetre pour mettre a jour la taille de la scene et ainsi ne jamais déborder hors de la fenetre du navigateur.

iii) Communication avec le serveur et chargement des OBJ

Le serveur est connecté au client via un système de socket. Ainsi il peut demander à tout instant un fichier et le serveur lui envoie.
La requete pour demander un fichier est mySocket.emit('client_data', {'letter': request}); où request est le fichier demandé.

Lors de la récéption du fichier on recoit un document brut inaffichabe dans threejs, il faut le parser.
Nous avons donc utilisé la bibliotheque MTLloader pour charger les MTL (fichier de définitions des materiaux) et OBJloader pour charger les OBJ.

Suite a de nombreuses recherches nous avons fini par comprendre le fonctionnement des méthodes de parse de ces deux librairies. Il existe peu de documentation sur le sujet car la fonction principale est load, elle permet de charger un fichier depuis un path ou une url or nous ne disposions que du fichier brut issu du serveur.

La fonction parse de MTLloader retourne un objet spécifique, celui ci doit etre mis en forme avant d'être appliqué à un modele OBJ à l'aide la fonction preload.

Le modele OBJ se parse de la meme maniere, il retourne un objet pouvant être directement ajouté dans la scene a l'aide de la fonction scene.add(objet).

La communication avec le serveur nous permet aussi de lancer la conversion et le découpage d'un fichier ifc automatiquement si celui ci n'a pas déjà été converti ou découpé. Ensuite il nous renvoie la liste des composants de ce fichier.

iv) Ajouts de fonctionnalités

Nous avons ajouté des fonctionnalités complémentaires dans l'interface suites aux tests :

- Chargement automatique du fichier mtl lors de sa sélection dans la liste les fichiers, ainsi il ne reste qu'à cliquer sur loadOBJ pour afficher l'objet dans la scene. Nous n'avons pas pu réaliser la méthode de loadMTL et loadOBJ en meme temps car cela entrainait soit l'impossibilité d'afficher l'obj du au temps de réponse du serveur, soit une boucle infinie du aux particularité structurelle du langage javascript.
- Effacer tous les fichiers de la scene d'un clique, la fonction parcours tous les objets de la scene et ne supprime que les mesh (ils sont de type "Group").
- Effacer uniquement un mesh : lors de la création d'un mesh on lui donne un nom : le nom de fichier, puis lors de la demande de suppression il suffit de retrouver ce nom dans l'arbre et de le supprimer. Pour une raison encore inconnue cela ne fonctionne pas toujours.

v) Version mobile

Le code CSS du site a été adapté pour etre responsive.
De plus nous avons ajouté l'apparition automatique d'un menu hamburger dans lequel les clients mobiles peuvent retrouver la liste des fichiers chargés par l'application.
Enfin la taille de la scene est modifié automatiquement lorsque l'on se trouve sur un appareil mobile pour pouvoir etre en plein écran.

b) Serveur

i) Conversion IFC vers OBJ

Nous avons utilisé **IfcConvert** dans **IfcOpenShell** pour réaliser cet étape:

ii) Fonction pour découper les OBJ

Nous avons écrit une fonction qui s'appelle **DevideObj()** pour découper les fichiers **.obj**; dans cette fonction, nous utilisons **FS** de Node.js pour lire et écrire les fichiers **.obj**, et utilisons des expressions régulières pour les découper.

iii) Mise en place d'un socket

Nous avons utilisé **socket.io** pour réaliser la transmission des datas entre le serveur et le client. Il peut supporter plus de équipements et navigateurs,et très facile d'utiliser.

iv) Envoie les données vers le client

Nous utilisons **socket.io** et encapsulons les datas transmis en classe de javascript pour fournir les informations dont le client a besoin.

v) Ajouts de fonctionnalités

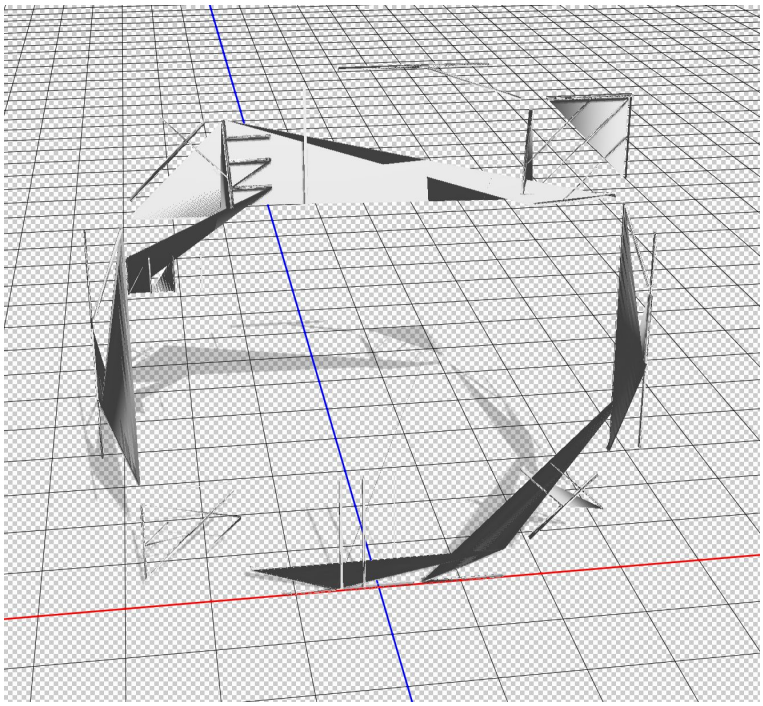
Nous avons réalisé les fonctions ci-dessous sur le serveur:

- Permet le client d'envoyer une demande du fichier **ifc**, et puis le serveur va juger si le fichier existe ou pas: si oui, le serveur retournera une liste des sous-fichiers du fichier **ifc**, et le client peut renvoyer une demande pour obtenir un sous-fichier; mais sinon, une erreur va être retourné.
- Permet le client de téléverser un fichier **ifc** sur le serveur.
- Après avoir reçu un fichier **ifc**, le serveur utilisera **ifcconvert** pour le transformer en un fichier **obj,** et puis le dévider **obj** en quelques sous-fichiers.

vi) Test et résolution

Nous avons pu tester l'application en version locale, donc sur la meme machine ou est en execution le serveur. Ça nous a permit de verifier que le serveur marchait bien en soit. Pour le tester sur d'autres plateformes nous avons du attendre de mettre l'application en ligne.

Aussi, la découpe des fichiers a été une étape importante. Il nous a fallu comprendre complètement la structure de ces fichiers. La première version de notre fonction cote serveur nous renvoyait des objets invalides ou bien incohérents comme le montre l'image ci dessous.



Le problème venait du decallage de la numerotation des face issue du reassemblage des differentes partie de l'ensemble dans les sous fichiers obj.
Il necessitait une mise à jour des indices des index des faces après le découpage.

c) Déployement de l'application sur le web

Cette étape ne rentrait pas dans le cahier des charges de l'entreprise mais comme nous avions une application fonctionnelle nous avons voulu la mettre en ligne pour de "vrai" et ainsi voir quels étaient les problemes rencontrés et aussi pouvoir la faire tester par le plus grand nombre sans a avoir a installer les dépendances : en résumer pouvoir réellement profiter les fonctionnalités de l'application.

i) Configuration du serveur web

Nous avons donc acheté un serveur web chez Ovh pour une durée d'un mois. C'est un serveur appelé VPS (virtual private server).
La premiere étape a été de cloné le github sur celui ci puis de procéder à l'installation des dépendances via les fonctions npm install et bower install.
Ensuite il a été nécessaire de modifier les adresses ip indiquées dans le serveur et le client pour qui passe bien par le serveur web et non pas par l'adresse local du client qui tenterait d'y acceder.
Malheureusement à ce niveau là il était toujours impossible d'acceder à l'application depuis l'exterieur.

ii) Ouverture des ports

Après quelques recherches nous avons identifié le problème : les ports utilisés par notre application étaient fermés par le serveur.

Nous avons pu les ouvrir grace à l'outil iptables.

Ensuite nous avons rencontré des problemes avec le serveur nodes js qui ne voulait pas se connecter au client à cause de règles de sécurité empechant les requetes entre deux sites distance. Nous avons donc du modifier les headers des requetes envoyées entre le serveur et le client.

iii) Configuration du routage des ports avec Ngnix

Enfin, l'application ne fonctionnant toujours pas car le socket n'arrivait pas à se connecté nous avons utilisé Nginx comme proxy interne afin de rerouter les connections.

L'utilisateur se connecte donc via http sur le port 80 à notre client, il est rerouté vers le client angularJS sur le port 8000.
Puis lorsqu'il fait appel au socket sur le port 8080 ces requetes sont reroutées vers le port du serveur nodeJS, le port 8000.

Documentation

Client

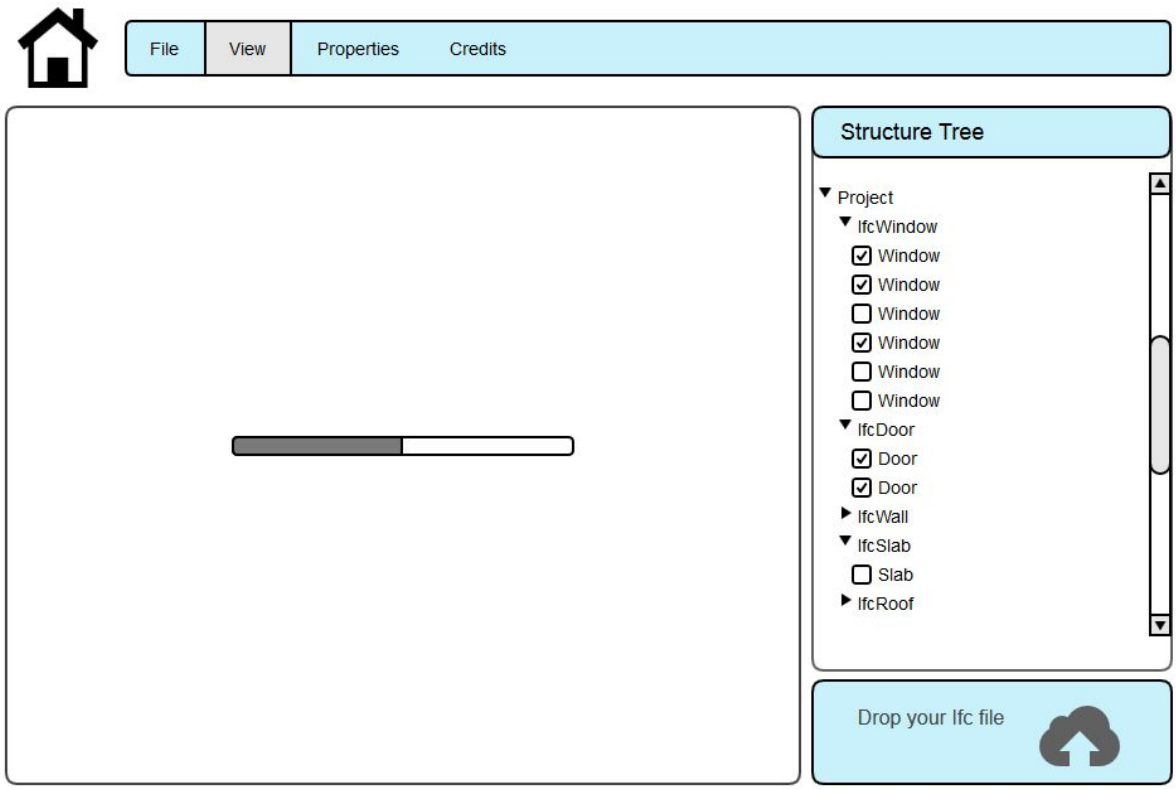
Interface Utilisateur

Cahier des charges de l'interface

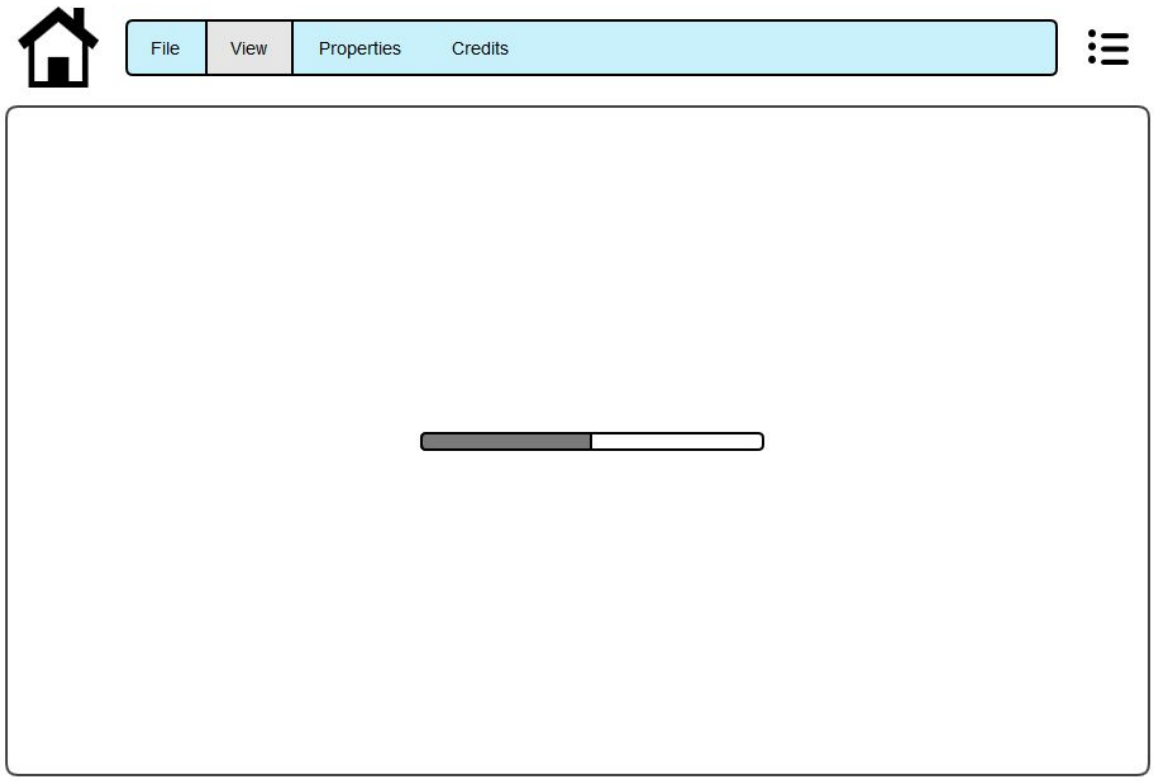
L'interface nécessite de etre fonctionnelle, donc claire et adapté à etre éxploité soit sur des grands écrans que sur des petits. Une grande section pour l'affichage et des menus comprenants le fonctions de base, le chargement de fichiers et la liste des objets qui composent le batiment affiché.

Fonctionnalités et ergonomie

Voici des models preliminaires de l'interface web de l'application sous differents platformes d'exploitation.



Version Desktop de la page d'accueil de l'application.



Version Mobile de la page d'accueil de l'application. (Menu fermé)

Version Mobile de la page d'accueil de l'application. (Menu ouvert)

Version mobile

Dans le but d'adapter le site aux mobiles nous avons du changer le style CSS

```
@media (max-width: 800px) {
 .mobile {
   display: block;
 }
 .desktop {
   display: none;
 }}
```

Grace à ce code on peut modifier le style css des class mobile et desktop uniquement pour les machines ayant un affichage inferieur à 800px. Dans ce cas on cache les objets de la classe desktop et on affiche ceux de la classe mobile. (propriété display)

```
$('button').click(function() {
 $(this).toggleClass('expanded').siblings('div').slideToggle();
});
```

Dans le code javascript nous avons ajouté une fonction permettant de dérouler le menu hamburger lorsque l'on clique dessus

---

Enfin il a aussi été nécessaire de modifier la taille du viewer 3D lors du passage en mode mobile. Quand le menu de selection des fichiers est caché (mode mobile) on va agrandir la taille du renderer (passage de la taille du menu "rightmenuratio" à la taille d'une légere bordure blanche : 20px

```
if($window.innerWidth<800){
 _camera.aspect = ($window.innerWidth  - 20 ) /
($window.innerHeight - headerSize);
 _camera.updateProjectionMatrix();

 _renderer.setSize( $window.innerWidth  - 20 , $window.innerHeight
- headerSize);
}
 else{
 _camera.aspect = ($window.innerWidth  -rightMenuRatio ) /
($window.innerHeight - headerSize);
 _camera.updateProjectionMatrix();

 _renderer.setSize( $window.innerWidth  -rightMenuRatio ,
$window.innerHeight - headerSize);
}
```

Affichage et traitement des données avec ThreeJS

Lecture des OBJ

Voici la fonction permettant de lire les OBJ depuis les données envoyées par le serveur.

```
var loaderOBJ = new THREE.OBJLoader();
mtl.preload();
loaderOBJ.setMaterials(mtl);
var objModel = loaderOBJ.parse(objData.data);
clearScene();
objModel.name = filepath;
_scene.add( objModel );
```

On commence par instancier un loader d'obj, puis on preload les données de materiaux qui ont été chargée dans la variable mtl. On applique ces materiaux aux loader, ainsi ils seront automatiquement copiés dans les propriétés du fichier chargé.

Enfin à l'aide de la fonction parse du loader on peut récuperer notre objet. Il ne reste plus qu'à l'ajouter à notre graph de scene.

Lecture des MTL

```
var mtlLoader = new THREE.MTLLoader();
mtlData = mtlLoader.parse(data_mtl.data);
mtlData.preload();
```

On instancie MTLloader puis on utilise la meme fonction que précédement (parse), elle nous retourne un objet complexe devant etre preloadé avant son utilisation.

Affichage de la scène (fonctions utiles)

Instancier un nouveau moteur de rendu (WebGL) : new THREE.WebGLRenderer()

Créer une caméra : new THREE.PerspectiveCamera(FOV, ratio, min, max);

Créer une lumiere de type pointLight de couleur blanche : new THREE.PointLight(0xFFFFFF);
On accède à la localisation de cette lumière à l'aide de l'attribut position qui est un vecteur 3D.

Les lumières, les mesh sont tous des objets. Ces objets doivent être ajoutés à la scène pour pouvoir être visualiser lors de l'appel du renderer.

On peut facilement créer une nouvelle scène avec : new THREE.Scene();
Puis ajouter un élément à cette scène avec scene.add(element)
Enfin, pour visualiser notre élément dans cette scene on fait appel au renderer via la commande :
renderer(scene,camera); Cela permettra de dessiner la scene vue par la caméra.

Fonction de gestion des éléments de la scène

Voici le gestionnaire des entrées utilisateurs. Nous en avons essayé plusieurs et celui ci répond le plus à nos attentes. Il prend naturellement la caméra en argument car il va agir dessus pour donner l'impression de zoom (en changeant la fov par exemple)

```
_trackball = new THREE.TrackballControls(_camera, container);
_trackball.rotateSpeed = 3.5;
_trackball.zoomSpeed = 2.0;
_trackball.panSpeed = 0.5;
_trackball.noZoom = false;
_trackball.noPan = false;
_trackball.staticMoving = true;
_trackball.dynamicDampingFactor = 0.3;
_trackball.minDistance = 1;
_trackball.maxDistance = 200;
```

Autres

Cette fonction permet de remettre la scene a zero tout en gardant les lumières.
Elle dépend du scope car elle est utilisée par un bouton dans le code html (le $scope est l'ensemble des variables étant partagées entre le code html et le code javascript dans le langage angularJS)

```
$scope.cleanTheScene = function(){
 for( var i = _scene.children.length - 1; i >= 0; i--){
   var obj = _scene.children[i];
   //select only the meshes
   if (obj.type == "Group"){
     _scene.remove(obj); //and kill it!
   }
 }
};
```

Communication avec le serveur

Reception des fichiers

Pour recevoir les fichiers depuis le serveur on commence par lui demander de récuperer le fichier via la commande (fp est le pseudo path du fichier):

```
 sendRequest(fp);
```

Puis on appel le socket avec le mot “server_data” pour qu’il nous envoie les donnees qu’il a préparé pour nous. Ces données sont stockées dans la variable data passée au callback.
Si les data sont nulles alors on log une erreur.

```
 mySocket.on("server_data", function(data){
   var OBJData = data.data;
   if(data.data === 0){
     console.log('this ifc does not exist!!');
   }else{
     console.log(data.data);
     for (var part in data.parts){
       var currentList = $scope.layers;
       var newList = currentList.concat({name:data.parts[part]});
       $scope.layers = newList;
     }
   }});
```

Envoie des fichiers

Pour envoyer un fichier on stock les données qu’il contient et le nom du fichier qui a été déposé (drag and drop upload) dans un json qui sera par la suite envoyé au serveur grace au mot “upload” et la fonction emit() du socket.

```
var reader = new FileReader();
       reader.readAsBinaryString(file);
       reader.onload = function (event) {
var data = {data:event.target.result,
     name:evt.dataTransfer.files[0].name};
           mySocket.emit('upload', data);
       };
     }
```

Déployement de l’application

Pour déployer l’application il a été nécessaire de faire ces modification dans le serveur et l’application.
Dans app.js

```
var myIoSocket = io.connect('http://164.132.225.122:8080');
```

Dans server.js

```
webSvr.listen(8080)
```

Serveur

Conversion IFC vers OBJ

```
//Appelez la console pour utiliser IfcConvert.
var cmd = './IfcConvert '+ pathifc + letter +'.ifc';
exec(cmd, function(error, stdout, stderr) {
    console.log("[createOBJ] transformed ifc to obj!");
    ReadFiles(letter);
})
```

Fonction pour découper les OBJ

```
var DevideObj = function(letter){
    ObjDiffParts = [];
    //utiliser des expressions régulières pour détecter les
    //sous-parties dans le fichier mtl.
    //détecter le mot "newmtl"
    var patt = /newmtl .+/g;
    var parts = mtlData.match(patt);
    var npart = parts.length;
    console.log("[obj split]there are " + npart + "objs");
    var tempstr = (objOrigin + 'g').replace(/\ng /g, "\ng g ");
    //Créer le dossier pour les sous-parties
    if (!libFs.existsSync(pathobjs)) {
        libFs.mkdirSync(pathobjs);
    }
    //Pour chaque partie, nous générons un nouveau obj.
    for(var i = 0; i< npart; i++){
        parts[i] = parts[i].substr(7);
        console.log("[obj split]creating" + parts[i]+".obj");
        //utiliser des expressions régulières encore une fois
        patt = eval("/usemtl " + parts[i] + "[\\s\\S]*?\\ng/g");
        var subparts = tempstr.match(patt);
        var nsubpart = subparts.length;

        //générons un fichier obj
        var filename = letter +parts[i]+'.obj';
        var filenameMtl = letter +parts[i]+'.mtl';
        ObjDiffParts.push(filename);
        ObjDiffParts.push(filenameMtl);
        objData = "mtllib "+ letter +".ifc.mtl\n";
        //coller tous les messages detecté dans le fichier
        for(var j = 0; j < nsubpart; j++){
            objData += "g " + (i+1) + "\ns 1\n";
```

```
            objData += subparts[j].substr(0,subparts[j].length-1);
        }
        console.log("success create");
    };
}
```

Mise en place d'un socket

```
io.sockets.on('connection', function(socket){
    //send time informations to client
    setInterval(function(){
        socket.emit('date', {'date': new Date()});
    }, 1000);

    //get request from client
    socket.on('client_data', function(data){
        soc = socket;
        console.log("client get " + data.letter);
        ReadFileAndSendData(data.letter);

    });
    socket.on('upload',function(data){
       soc = socket;
       console.log("J'ai recu un IFC");
       libFs.writeFile("WebRoot/files/"+data.name,data.data);
    });
});
```

Envoie les données

```
//envoyer un objet de javascript.
soc.emit('server_data', {'data': letter + '.obj',
                         'number': ObjDiffParts.length,
                         'parts': ObjDiffParts});
```

Conclusion

Les objectifs atteints

L'application finale comprends un viewer Web 3D fonctionnel avec toutes les fonctionnalités demandées dans le cahier de charge.
En global nous avons developpé un viewer Web, qui réalise la conversion et l'affichage de fichiers ifc, leur chargement sur le client et stock les données.
L'interface de visualisation compatible mobile et permet des interactions clavier souris ou tactile.
Nous avons géré les problémes de taille à transferer entre le client et le serveur des fichiers grace au decoupage.
Les frameworks utilisés et les modules de notre application sont tous open source, comme demandé par l'entreprise.
Nous avons mis en ligne l'application web, qui est active (pour le mois de mars 2016) et est fonctionnel à l'adresse http://164.132.225.122/ (ne fonctionne pas sur le réseau de l'école).

Developpements envisagés

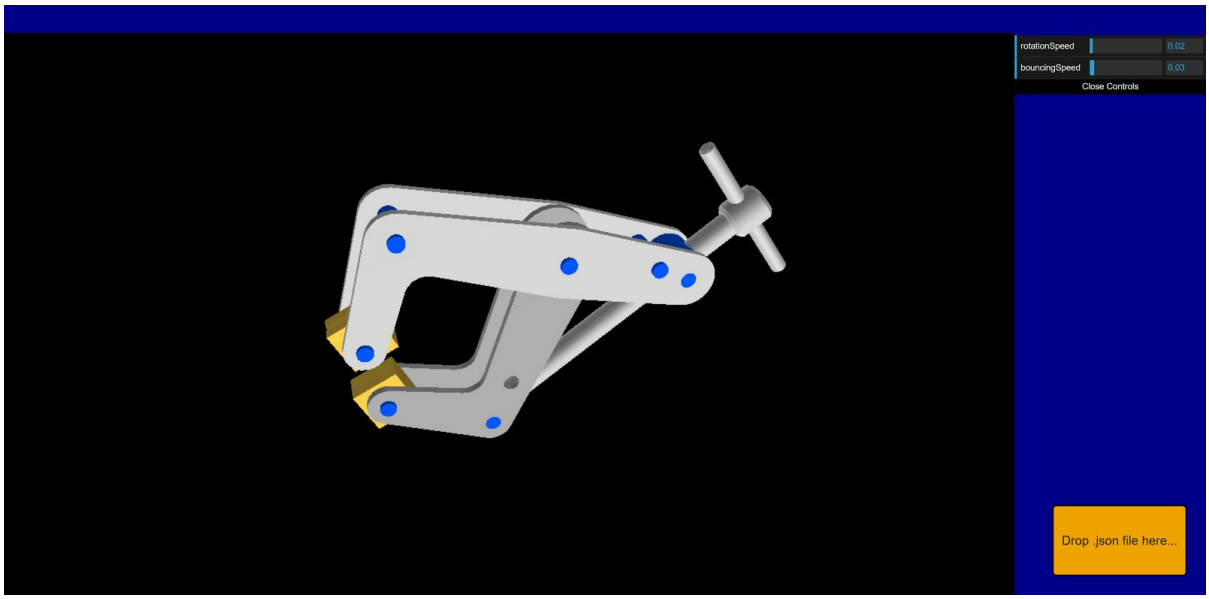
Nous avons pensés aux ameliorations qu'on aurait fait au projet avec du temps supplementaire.
Nous avons remarqués que le systéme de conversion des fichiers ifc en obj est assez lourd et rends difficile l'utilisation de ce système directement (lenteur du traitement), mais il faudrait etudier une differente chaine qui implique une conversion preliminaire et la disponibilité des fichiers obj déjà traités au lieu de l'utilisation.
La librairie de contrôle des input de l'utilisateur (TrackballControls) que nous avons utilisé pour l'interaction avec le modèle 3D est adaptée pour des petits modèles, mais pas pour des grandes structures. Il serait interessant d'ameliorer la navigation 3D avec des commandes plus complexes et flexibles.
Enfin, comme nous avons intégré plusieures modules complémentaires il serait utile de tester les fonctionnalités de l'application de façon systematique et structuré pour trouver les points d'amelioration necessaires.

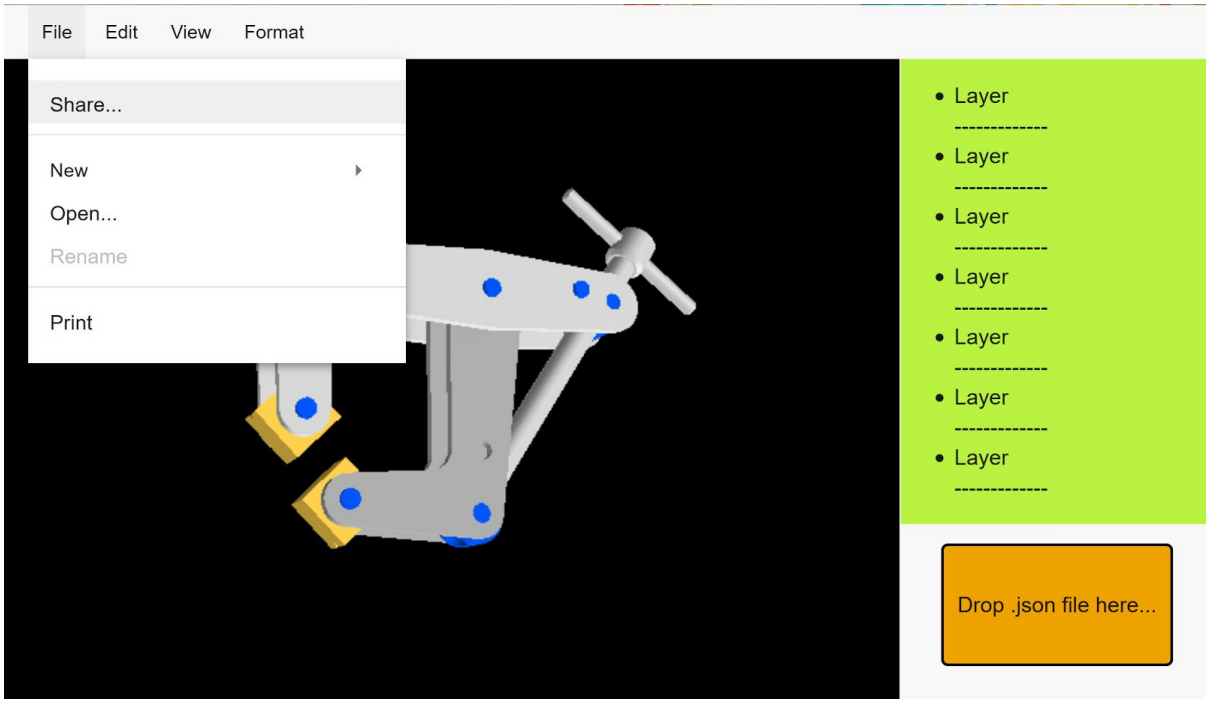
Considérations

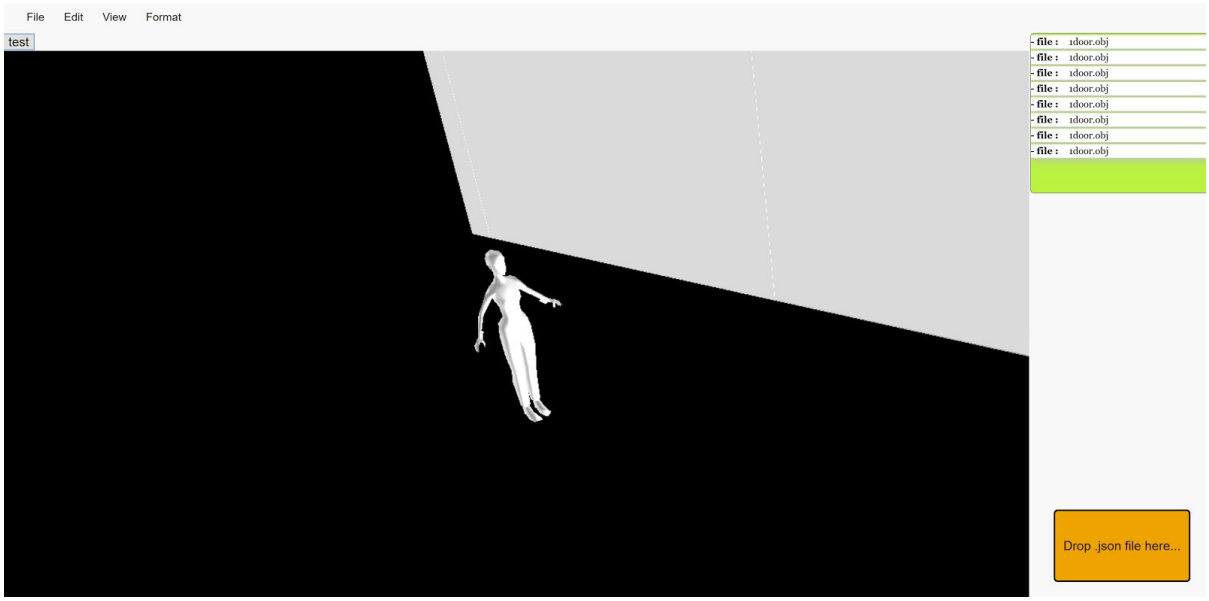
Globalement nous sommes satisfaits des resultats de ce projet, dont nous avons reussi à atteindre la presque totalité des objectifs malgré le peu de temps à notre disposition.
Nous avons reussi à bien nous organiser et partager les taches entre tous les membres du groupe, tous ont donné leur contribution. La communication avec l'entreprise et le professeur referent ont été fréquents et efficaces.
En soi ce projet nous a permis d'améliorer nos compétences en developpement web, qui n'est pas inclus dans notre specialisation.

Annexes

Evolution de l’interface






File
Edit
View
Format
test
- file : 1door.obj
- file : 1door.obj
- file : 1door.obj
- file : 1door.obj
- file : 1door.obj
- file : 1door.obj
- file : 1door.obj
- file : 1door.obj
Drop .json file here...

Renderer sans antialiasing mais premiere réussite de chargement des mtl ET obj depuis le serveur