Immersive Factory Simulator

Guide développeur

Projet RICM5

POLYTECH GRENOBLE

Adèle BERTRAND-DALECHAMPS
Guillaume HAMMOUTI
Yacine NDIAYE
Florian POPEK
Bin SUN
Zilong ZAHO

Table des matières

Introduction

IFS est un serious game permettant la visite immersive d'une zone industrielle. Le projet consiste à simuler cette visite interactive virtuelle à l'aide de Google VR. L'utilisateur a une vue FPS (First Person Shoot) et se déplace à travers la Map à l'aide d'une manette Xbox 360. Il peut lire ou ajouter des étiquettes liées à des objets de la centrale (ex: Tuyaux), consulter des valeurs de capteurs placés sur le monde réel. Plusieurs fonctionnalités telles que la visualisation de vidéos courtes (pour aider dans les gestes) sont implémentées.

Pour clarifier la définition du projet et définir la structure de l'application, nous avons réalisé ce manuel développeur qui permettra de facilement implémenter de nouvelles fonctionnalités.

L'environnement de l'application est basé sur Unity 3D, WebRTC, Google VR et Amazon aws.

Le lancement de l'application se fait sur Unity et cette dernière se chargera de récupérer le contenu stocké dans la base de données et gérer l'authentification des utilisateurs.

Dans la suite, l'architecture générale et le modèle de données seront présentés, ainsi que les technologies utilisés.

Architecture Générale

Le diagramme de la page suivante montre une vue simplifiée de l'architecture du projet.

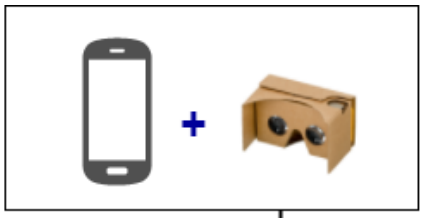
On peut voir que l'ensemble est constitué de trois grandes parties: une partie Unity, une partie Serveur et une partie IoT avec les capteurs.

La partie Unity contient le coeur de l'application (moteur et front end). Elle se base sur le runtime Unity et utilise l'API Rest WWW de Unity pour communiquer avec la partie serveur par HTTP.

Ensuite, il y'a la partie Serveur, qui contient le backend de l'application. Elle contient principalement la base de données et les fonctions de contrôle de l'API.

Enfin on a la partie Capteurs, dans laquelle on a toutes les configurations du réseau de capteurs ainsi que l'implémentation de la communication (push des données vers le serveur).

Nous allons dans la suite, développer chaque partie plus en détail.



WebRTC

UNITY

Runtime

REST API

HTTP

SERVEUR

Sails js Framework

generate API

models

controllers

ProjectAPI

MongoDB Connector

ORM

MongoDB

EC2 INSTANCE

CAPTEURS

OpenHab 2

Script Nodejs

Zwave Binding

Capteur MultiSensor 6

Capteur Oregon

Unity 3D

1. WebRTC

On utilise WebTV pour projeter l'écran d'ordinateur sur le téléphone. Avant de lancer les commandes suivantes, il faut mettre les deux entités dans le même réseau (attention avec Wifi Campus ça ne marche pas!)

- **Partie ordinateur**

Pour connecter la partie portable avec l'ordinateur, d'abord lancer le serveur dans /webrtc/server:

```
./run.sh
```

Cela va lancer un serveur qui écoute le port 9000 de l'adresse IP locale, puis lancer la page screen.html dans un navigateur. Attention, il faut ouvrir la page dans un chemin localhost, par exemple:

```
localhost/webrtc/client/screen.html
```

Puis une page suivante va apparaître, et il suffit de remplir des champs 'Connect as' avec un nom, IP où on lance le serveur. Puis cliquer sur 'Connect'.

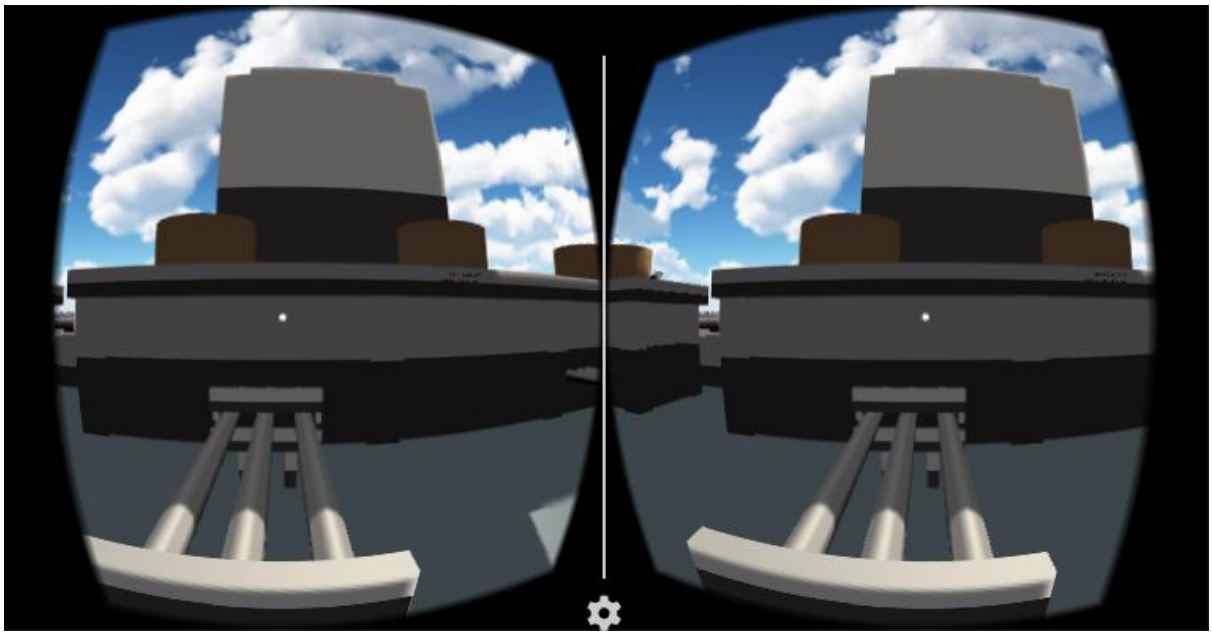
- **Partie portable**

Il faut Installer org.xwalk.webrtc-0.0.1-debug.armeabi-v7a.apk sur un portable android, ouvrir l'application, remplir l'adresse IP du serveur, et cliquer sur 'Connect'

A présent, les deux parties sont connectées avec le serveur, alors du côté ordinateur, il faut remplir 'client' dans le champ 'Make call to', et cliquer sur Call, l'écran de l'ordinateur sera projeté sur le portable.

2. Google VR

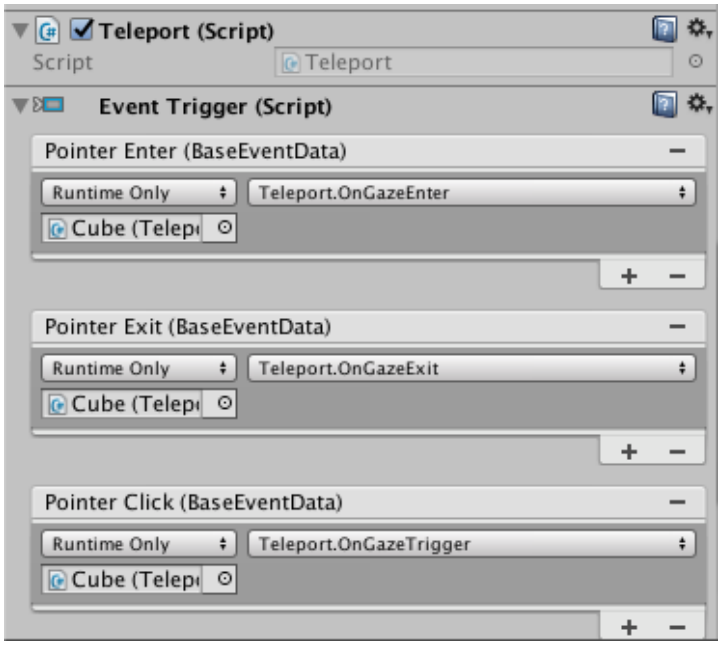
Télécharger le package GoogleVRForUnity.unitypackage sur le site de Google VR, importer tous les dossiers du package dans le projet, ajouter le prefab GvrViewerMain dans la scene, attacher aussi le prefab GvrReticle sous Main Camera, lancer le projet, on arrive à avoir une vue semblable à celle-ci:



Il possède un viseur mais aussi une vue en VR. Et pour que le viseur puisse activer ou désactiver un trigger (capteur ou étiquette), il faut ajouter un script 'PhysicsRaycaster' dans Main Camera et un script 'GazeInputModule' dans EventSystem.

- **Créer un trigger**

Pour qu' un cube se comporte comme un trigger, ajouter le script (par exemple teleport.cs) et aussi event trigger pour configurer le comportement quand on clique sur le cube:



- **Commutation de VR et viseur**

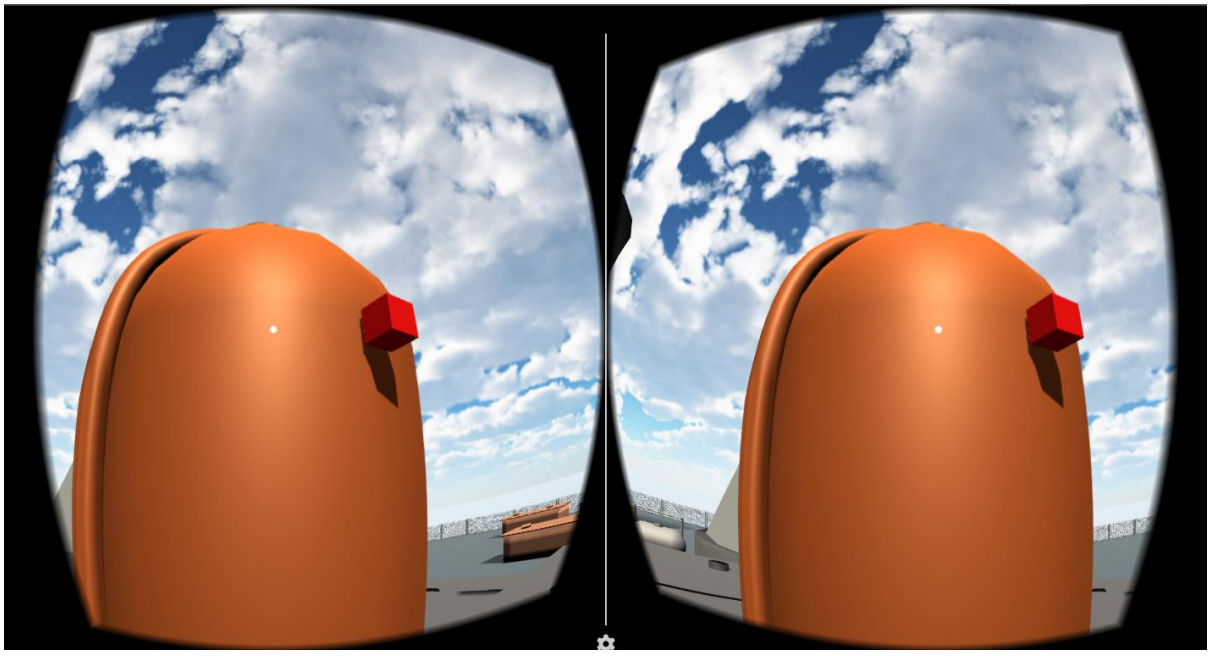
Ajouter le script 'SwitchVR' dans EventSystem,mettre le script 'GazeInputModule' dans le champ 'VR script', aussi ajouter quelques lignes de code dans la fonction update() de GvrViewer.cs, pour activer et désactiver la variable VRModeEnabled.

Ajouter deux champs 'Switch' et 'SwitchVR' dans InputManager avec des clés correspondantes sur la manette.

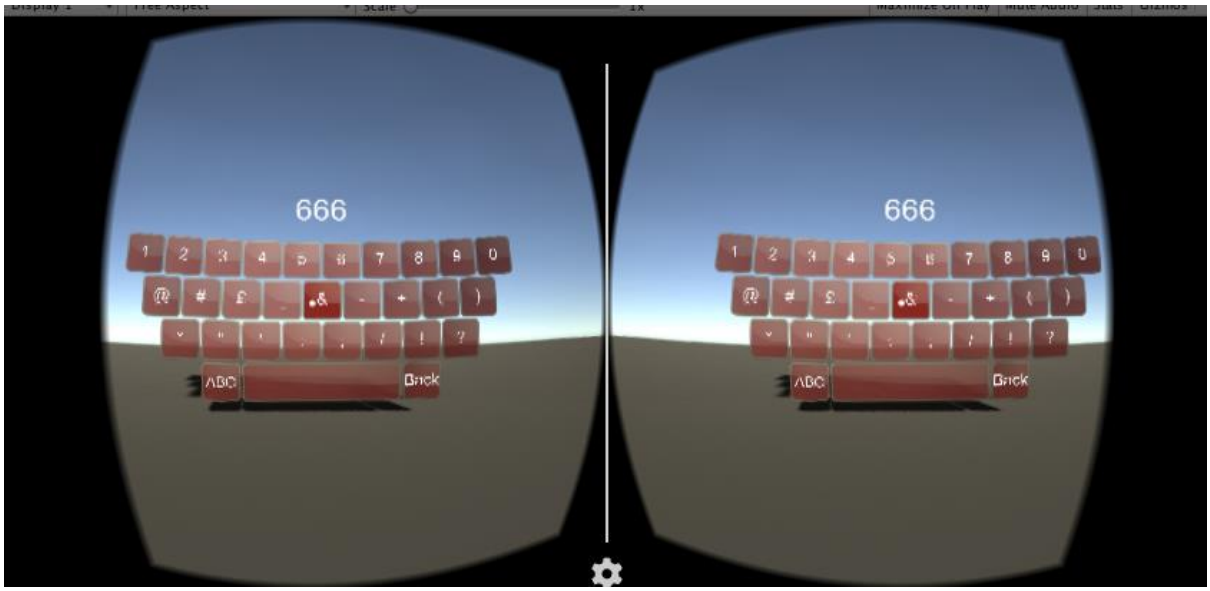
Jusqu'à maintenant, le projet peut activer et désactiver le mode VR et le viseur par la manette.

3. Communication avec le serveur

Dans notre usine, on a beaucoup de bouton rouge comme suivant, il représente un capteur validé ou une étiquette validée, on peut cliquer pour voir le message dedans, si on désactive l boutons, ils deviennent blancs, et on ne peut plus cliquer dessus.



Chaque fois qu'on clique sur une étiquette, il crée un nouveau thread pour traiter le processus qui charge les données du serveur sur Cloud (AWS) via APIRest (Ici pour effectuer des commandes Post et Get dans script, il faut installer Curl pour réutiliser le code), il va afficher un clavier virtuel en 3d avec un panneau d'information, et si on clique encore une fois sur le bouton, il va fermer le panneau et clavier, enregistrer le nouveau message dans la base de donnée via APIRest.

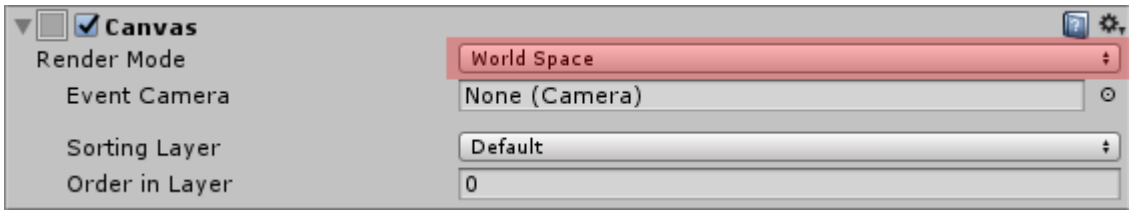


Si c'est un capteur, un graphe sera affiché (il faut importer d'abord le package 'Vectrosity') avec toutes les valeurs 24 heures avant.

4. Gestion d'un UI

Pour gérer les interfaces utilisateurs avec la réalité augmentée, il faut que les canvas soit représenté en 3D et non comme une simple image superposée au rendu comme cela est normalement fait.

On configure donc le canevas principal attaché au personnage en rendu *World Space* :



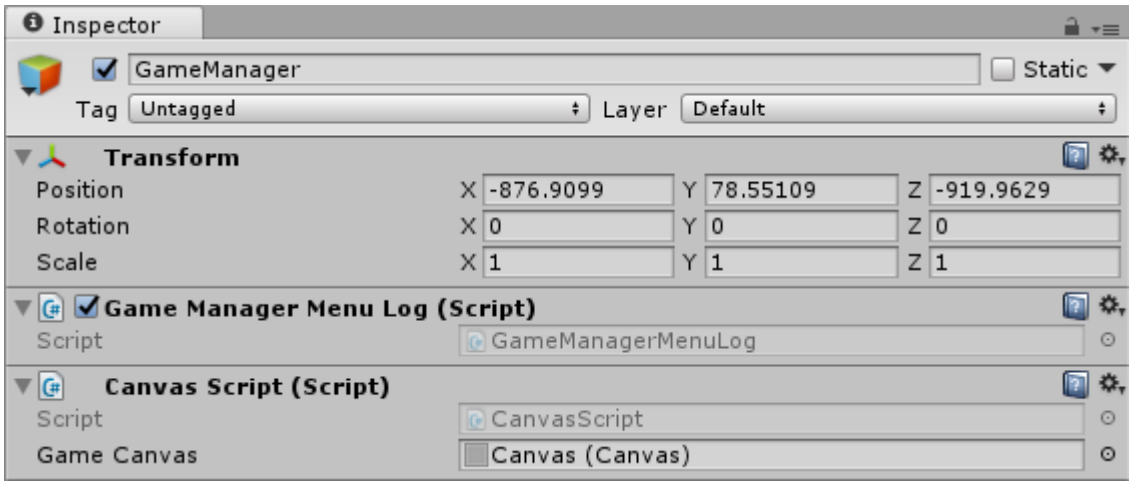
L'UI est maintenant représentée en 3D et pourra donc être visualisée en VR.

La conséquence de ceci est que l'UI est difficilement positionnable devant la caméra. Pour obtenir un placement exact, les coordonnées relatives des panneaux principaux à l'écran peuvent être inscrites dans le GameManager via l'API fournie :

```
CanvasScript.SetGameObjectPosition(CanvasScript.GetChild("MiniMap"), new Vector2(1, 0), new Vector2(1, -1));
CanvasScript.SetGameObjectPosition(CanvasScript.GetChild("MenuPausePanel"), new Vector2(0.5f, 0.5f), new Vector2(0, 0));
CanvasScript.SetGameObjectPosition(CanvasScript.GetChild("MenuHelpPanel"), new Vector2(0.5f, 0.5f), new Vector2(0, 0));
CanvasScript.SetGameObjectPosition(CanvasScript.GetChild("MenuMapPanel"), new Vector2(0.5f, 0.5f), new Vector2(0, 0));
```

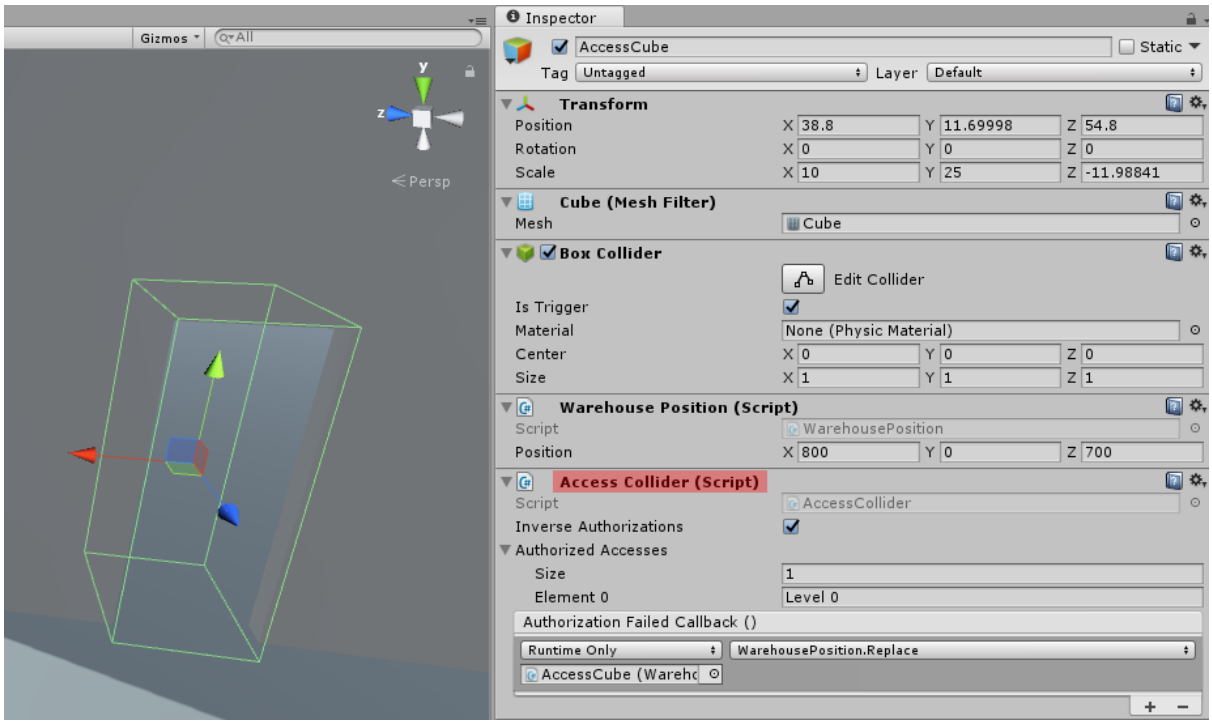
Le comportement d'un UI est écrit dans un script dénommé GameManager<ma_scene> qui est standard dans sa forme et son écrite, pour chaque scene, et dont les fonctionnalités communes sont décrites dans un le script *CanvasScript*. Une piles des appels avec retour automatique sur le dernier champs sélectionné est fournie, entre autre.

Pour que tout fonctionne, le canevas doit être lié au GameManager :



5. Gestion des accès

La gestion des accès est faite avec une boite de collision usuelle à laquelle est lié le script *AccessCollider* :

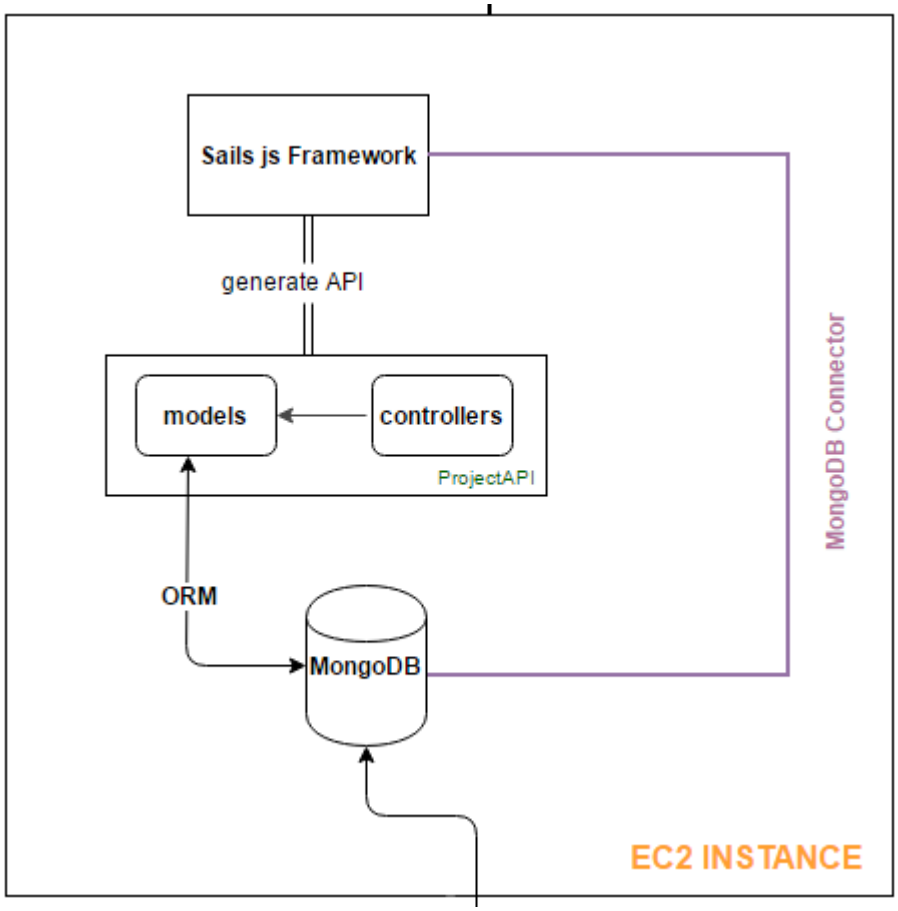


Dans ce script, il est possible d'indiquer les accréditations (sous forme littérales) requises, ou le comportement inverse (les accréditations à ne pas avoir).

Enfin, l'utilisateur donne au script une fonction callback à appeller lorsque les accréditations ne correspondent pas. Un comportement typique serait d'afficher un message d'erreur et de replacer le joueur avant la boite de collision.

Serveur EC2

Pour héberger notre serveur, nous utilisons amazon aws. Elle nous offre l'hébergement d'une instance ubuntu gratuitement pendant un an. Dans cette instance, se trouvent la base de données de l'application ainsi que l'API rest basé sur Sails.



Donc l'idée est de créer une instance ubuntu et de travailler comme on le ferait en local sur notre machine. Il faudra mettre en place l'environnement soit installer nodejs, npm, mongodb,nodejs-legacy, sails et autres prérequis. Une fois une base de données créée et un projet sails mis en place, on peut générer autant d'api qu'on veut (controllers et models) en changeant dans les fichier de configuration de sails l'adaptateur par défaut par le connecteur MongoDB (rentrer les informations de connexion à la base de données locale).

Attention dans la partie Sécurité de l'instance, il faut autoriser les ports utilisés pour sails (par défaut 1337 mais à sujet de changement pour des raisons de débug). Il est préférable donc d'autoriser une plage raisonnable disons de 1337 à 1340 et le port 27017 (mongodb) dans les connexions entrantes et sortantes.

1. Le modèle de données

Pour gérer les utilisateurs et l'authentification, nous stockons leurs données d'inscription dans la base de données. Nous disposons donc de 4 collections dans la base de données que sont:

- Users: ensemble de documents des utilisateurs
- Etiquette: pour stocker les étiquettes présents dans l'application et permettre la modification en temps réel
- Capteur: pour stocker les données reçues du capteur MultiSensor 6+
- CapteurOregon: pour stocker les données reçues du capteur Oregon

En utilisant le projet Variety, on arrive à avoir un aperçu du schéma de notre base de données ainsi que certaines statistiques liées aux collections présentes. Cela va nous permettre, vu qu'on a une base de donnée non relationnelle, de voir comment les données sont structurées.

- **La collection users**

```
ubuntu@ip-172-31-28-61:~$ variety Udb/users --host localhost --port 27017 --outp
utFormat='ascii'
MongoDB shell version v3.4.2
connecting to: mongodb://localhost:27017/Udb
MongoDB server version: 3.4.2
Variety: A MongoDB Schema Analyzer
Version 1.5.0, released 14 May 2015
Using collection of "users"
Using query of { }
Using limit of 12
Using maxDepth of 99
Using sort of { "_id" : -1 }
Using outputFormat of "ascii"
Using persistResults of false
Using resultsDatabase of "varietyResults"
Using resultsCollection of "usersKeys"
Using resultsUser of null
Using resultsPass of null
Using logKeysContinuously of false
Using excludeSubkeys of [ ]
Using arrayEscape of "XX"
Using plugins of [ ]
+-------------------------------------------------+
| key          | types    | occurrences | percents |
| ------------ | -------- | ----------- | -------- |
| _id          | ObjectId |          12 |    100.0 |
| accreditation | String   |          12 |    100.0 |
| createdAt    | Date     |          12 |    100.0 |
| login        | String   |          12 |    100.0 |
| nom          | String   |          12 |    100.0 |
| password     | String   |          12 |    100.0 |
| prenom       | String   |          12 |    100.0 |
| updatedAt    | Date     |          12 |    100.0 |
+-------------------------------------------------+
```

- **La collection étiquette**

key	types	occurrences	percents
_id	ObjectId	5	100.0
createdAt	Date	5	100.0
description	String	5	100.0
tags_capteur	String (4),null (1)	5	100.0
titre	String	5	100.0
updatedAt	Date	5	100.0
url	String (4),null (1)	5	100.0
video_val_capteur	String (4),null (1)	5	100.0

- **La collection capteur**

key	types	occurrences	percents
_id	ObjectId	1032	100.0
item	String	1032	100.0
realName	String	1032	100.0
timestamp	Date	1032	100.0
value	Number (860),String (172)	1032	100.0

Pour cette collection, on n'a pas eu à faire un design, les attributs de la collection ont été directement générés par le binding MongoDB de OpenHab.

- **La collection capteurOregon**

key	types	occurrences	percents
_id	ObjectId	320	100.0
createdAt	Date	320	100.0
description	String	320	100.0
item	String	320	100.0
updatedAt	Date	320	100.0
value	String	320	100.0

2. Le framework Sails

Pour faciliter la communication serveur vers l'extérieur et de l'extérieur vers le serveur de base de données, on se base sur le framework Sails qui propose une architecture MVC de base c qui rend son utilisation abordable. On a généré des api pour chaque collection et chaque api a ses propres composantes (model, view, controller).

Dans le dossier du projet sails qu'on a créé, nous avons donc généré 4 api, une pour chaque collection de notre base de données. Le modèle est une représentation de la collection, le controller contient les fonctions pour les opérations CRUD et autres fonctions si nécessaire.

Et la partie view n'a pas été exploité dans ce projet mais en gros elle permet de personnaliser en html l'interface de sails.

Dans le fichier de configuration routes.js, on ajoute les routes associées aux fonctions créées dans les controllers. Voici un aperçu de la configuration de routes.

```
 'post /update': 'EtiquetteController.update',
 'get /humidity': 'CapteurController.humidity',
'get /temperature': 'CapteurController.temperature',
```

- **Lancement automatique de sails**

Pour maintenir la session sails en dehors de la session ssh, on utilise forever (voir la doc de sails): **sudo forever start app.js**

La communication avec la base de données peut excéder le timeout de connexion de sails. Dans ce cadre, pour résoudre le problème il faut ajouter modifier le hooktimeout dans les fichiers env/production et env/development.

```
   models: {
     connection: 'myMongodbServer'
   },
hookTimeout: 60000

};
```

Capteurs

1. Capteur Multisensor 6 [Intégration avec OpenHab 2]

Le capteur Multisensor utilise le protocole Zwave, ce qui facilite son intégration avec OpenHab vu qu'on a un binding Zwave disponible. Il faut créer les items pour ce capteur qu'on utilisera dans la suite. Optionnellement, on a créé un sitemap pour afficher sur l'interface les valeurs du capteur. (Voir la doc de OpenHab 2). Ci dessous le processus d'envoi des données vers la base de données:

→ Utilisation du USB stick Zwave sur un port Usb de la machine locale

→ Détection du USB stick dans Openhab

→ Détection du capteur

→ Création des items

→ Visualisation si besoin

→ installer Binding MongoDB pas encore dispo dans OH2 (voir add-ons de OpenHab 1 et installer manuellement)

→ Configurer l'envoi vers le serveur de base de données distant

→ Changer dans le fichier de config de MongoDB du serveur l'adresse IP pour les connexions entrantes. (ici c'est localhost, il faut soit enlever la ligne soit mettre 0.0.0.0)

```
bindIp: 127.0.0.1
```

Pour débugger l'implémentation du push de OpenHab vers le serveur, regarder le fichier log de openhab aide beaucoup à situer les éventuels problèmes.

2. Capteur Oregon

On utilise le capteur oregon pour recevoir les données du capteur . On Utilise un script node.js pour recevoir et émettre les données. Les données sont sous forme JSON et on utilise un proxy Http: Get pour organiser et envoyer les données. Voici un exemple de stockage dans la base de données.


```
[
  {
    "item": "Id",
    "value": "57090",
    "description": "57090",
    "createdAt": "2017-03-13T14:22:48.084Z",
    "updatedAt": "2017-03-13T14:22:48.084Z",
    "id": "58c6ab38cd58fae83f8cb894"
  },
```


```
var rfxcom = require("./rfxcom.js");

var http=require('http');

var querystring=require('querystring');

rfxcom.on("open", function (){

  rfxcom.on('data', function(dataanalyse)
```

Le script rfxcom.js est le script qui permet de se connecter avec le capteur , on importe le script avec la fonction “rfxcom.on(‘data’,function(dataanalyse))” afin de recevoir les données sur le capteur .

```
item_value = data['analyse'][items]['value'];

item_string=data['analyse'][items]['toString'];
```

On utilise le code ci-dessus pour récupérer les données dont on a besoin .

Ensuite on fait une forme de JSON pour stockage les noms des items ,les valeurs,et les description.

```
var data_out={
                    item:items,
                    value:item_value,
                    description:item_string
                    };
                    data_out=querystring.stringify(data_out);
```

Après on etablit une requête Http get pour envoyer les données vers notre serveur comme le code suivant:

```
var req =http.get({
                    host:"52.10.23.69",
                    port: 1337,
                    path:"/capteurOregon/create?"+data_out,
                    method:"GET"
                    },function(error,response,body){
                    if(!error && response.statusCode==200)
                    {
                              console.log(body);
                    }else{
                              console.log('ok');
                    }
                    });
                    req.end();
          }
```

A la fin, on doit indiquer quel port USB connecter avec le capteur , on peut trouver un port libre par la commande de linux ‘ ls /dev/tty.* ‘ Après, on peut connecter avec le capteur comme le code suivant:

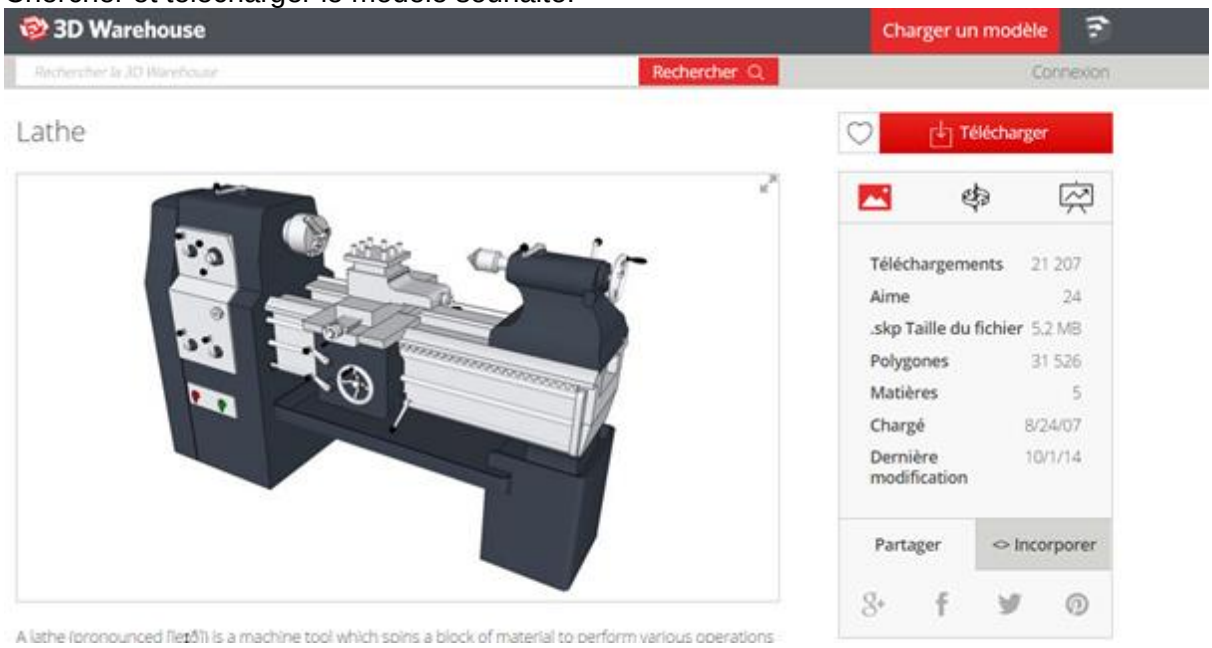
```
rfxcom.open('/dev/ttyUSBo');.
```

Modélisation 3D

Sketchup est un logiciel de modélisation et d'animation 3D. Les gros avantages de ce logiciel sont sa compatibilité avec Unity3D mais surtout son repository entièrement gratuit de modèles 3D. On a pu y trouver la plupart des modèles dont nous avions besoin.
https://3dwarehouse.sketchup.com/

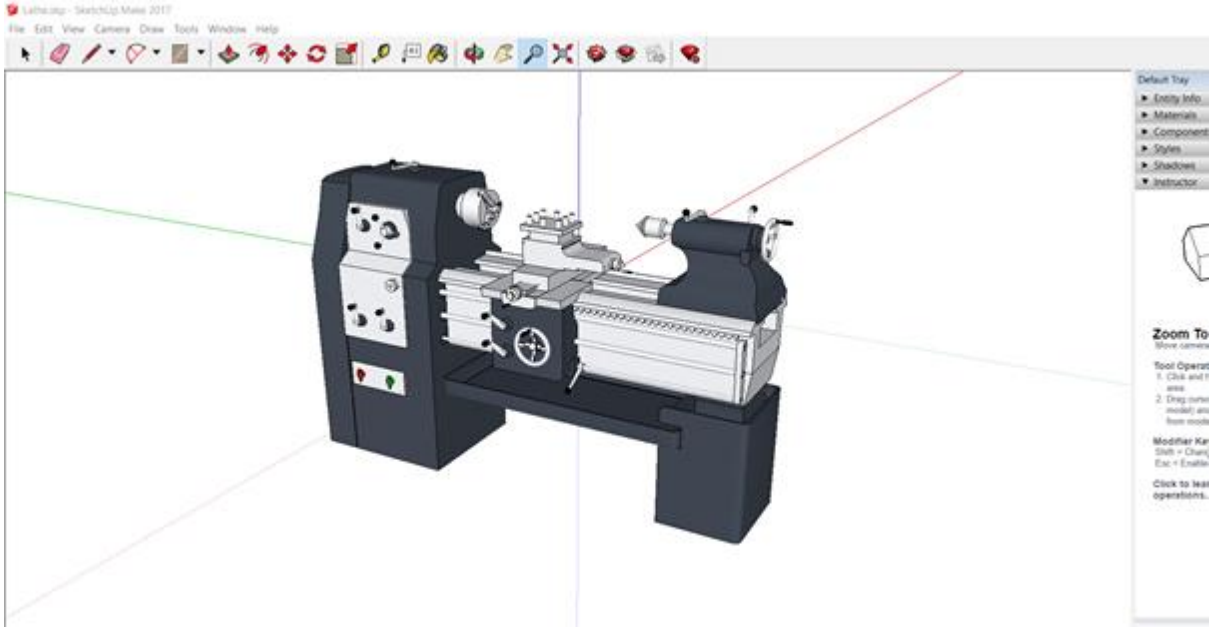
Guide en vidéo : https://www.youtube.com/watch?v=vGkdNlpYgzs

Etape 1 :
Chercher et télécharger le modèle souhaité.



Etape 2 :

Ouvrir et modifier si besoin le modèle à l'aide de SketchUp.

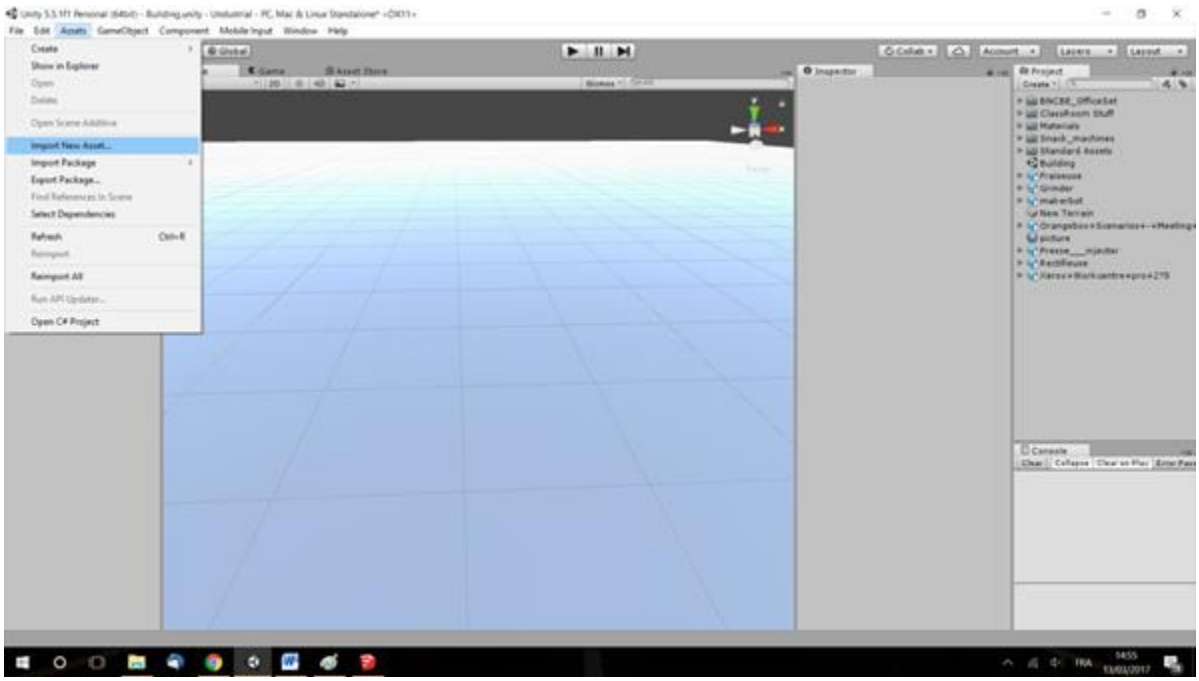


Etape 3 :

Exporter le modèle au format FBX avec les options suivantes cochées : Triangulate all faces, Export two sided faces, Export texture maps

Etape 4 :

Sous Unity, importez le fichier fbx que vous venez de créer.
Asset -> Import New Asset…



Etape 5

Ajouter la boite de collision.
Add component -> Mesh Collider