Un rapport de développement est exigé en plus du rendu (à rendre sous forme de pdf dans le projet lui même).
Il devra contenir un descriptif de toutes les étapes de réalisation, des choix faits, des difficultés, accompagné pourquoi pas d'extraits de
code pour expliquer le fonctionnement.

Les contrôles

Mouvement de caméra : Souris sur les bords de l'écran ou ZQSD (petit toggle pour activer la souris)
Sélection d'unité : Clique gauche sur une unité / maintenir pour le rectangle de sélection
Ciblage : Clique gauche sur la cible avec des unités sélectionnées
Déplacement : Clique droit sur le sol avec des unités sélectionnées

I - Structure Globales

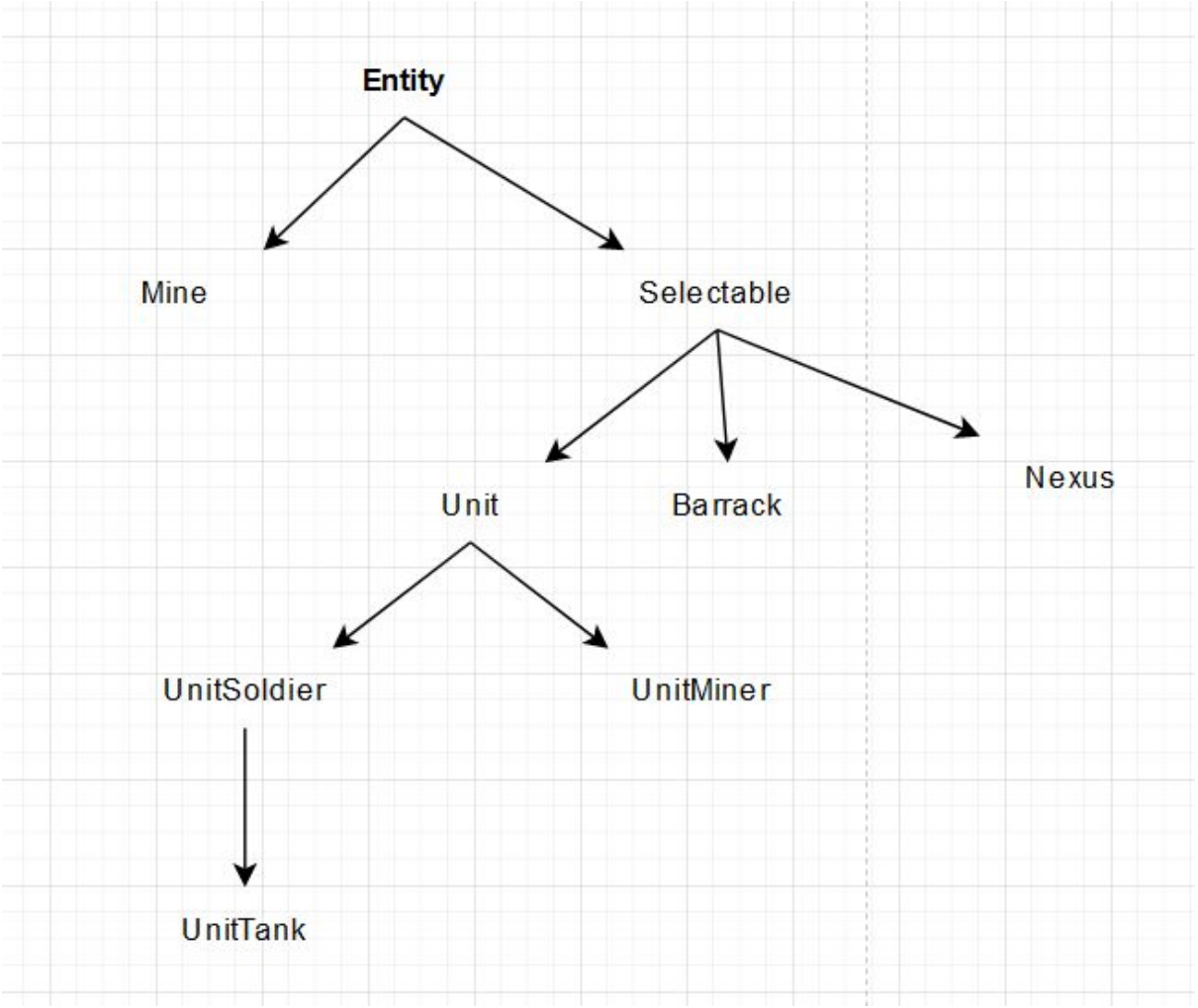
Gestion du lobby	Gestion Globale Versus	Gestion des entités au cas par cas	Gestion de la caméra
OnlineManager	**GlobalManager (singleton)**	**Entity**	**CameraController**
LobbyManager	OnlineManager	HealthBar	MiniMapCam
LoginManager	EntityManager	MapIcon	
	UnitSpawner		
	UnitSpawner		
	UIManager		
	FXManager		

II - Specs

Lobby
Créer une partie (1v1)
Rejoindre une partie
Créer des Unités
Contrôler des Unités
Miner des ressources

III - Création des entités

la classe **Entity** est utilisée comme base pour tous les objets avec lesquels le joueur peut interagir.



Entity contient

- un index in game afin de pouvoir répertorier toutes les entités présentes dans le jeu
- une liste des Unit qui l'ont prise pour cible

Au début il y avait uniquement des unités pouvant être sélectionnées et donc les classes **Entity** et **Selectable** étaient regroupées en une seule classe qui contenait notamment

- des HP
- une équipe

Mais lorsque j'ai voulu ajouter les **Mine** (bien plus tard) pour les ressources, je me suis rendu compte qu'il allait falloir distinguer ces 2 types de classes (les mines n'ayant ni HP, ni équipe).

De même pour les **UnitMiner** et **UnitSoldier** qui bien que partageant presque les même capacités n'ont pas la même logique puisqu'un mineur ne peut pas attaquer et un soldat ne peut pas miner.

Pour cette étape je me suis d'abord intéressé aux **interfaces** qui me semblaient être le bon choix mais le problème était que pour effectuer la logique, les unités ont besoin d'accéder à des **variables** qui se retrouvent plus haut dans la hiérarchie comme leur **transform** ou leur **range** qui sont déclarés dans **Unit**. J'ai donc opté pour la création de nouvelles classes qui héritent de cette dernière.

IV - Sélection et interaction

Maintenant qu'on a des unités il faut pouvoir les sélectionner et leur donner des ordres. Pour ça j'ai progressé par petites étapes simples.

- sélectionner une unité en cliquant dessus
- désélectionner une unité en cliquant dessus
- déplacer une unité sélectionnée sur la map
- faire en sorte qu'une unité retire de la vie à une autre entité
- limiter la distance à laquelle une unité peut attaquer
- sélectionner plusieurs unités et leur donner toutes le même ordre
- sélectionner une entité type barrack pour effectuer des actions spécifiques (grâce à un menu contextuel)

Encore une fois avec l'ajout de la gestion des ressources j'ai dû ajouter une dernière étape qui est de différencier les mineurs des soldats en fonction de la cible désignée car je ne leur envoie pas le même type de données.

V - Mise en réseau

Pour la parti réseau je me suis simplement servi de l'exemple fourni

- j'ai commencé par essayer de comprendre l'utilité de chacune des fonctions présentes dans les scripts de l'exemple
- j'ai ensuite clean le code de l'exemple en retirant tout ce qui ne me semblait pas pertinent pour l'exercice

Et je suis donc parti de cette base clean pour le **OnlineManager** et pour le code server.

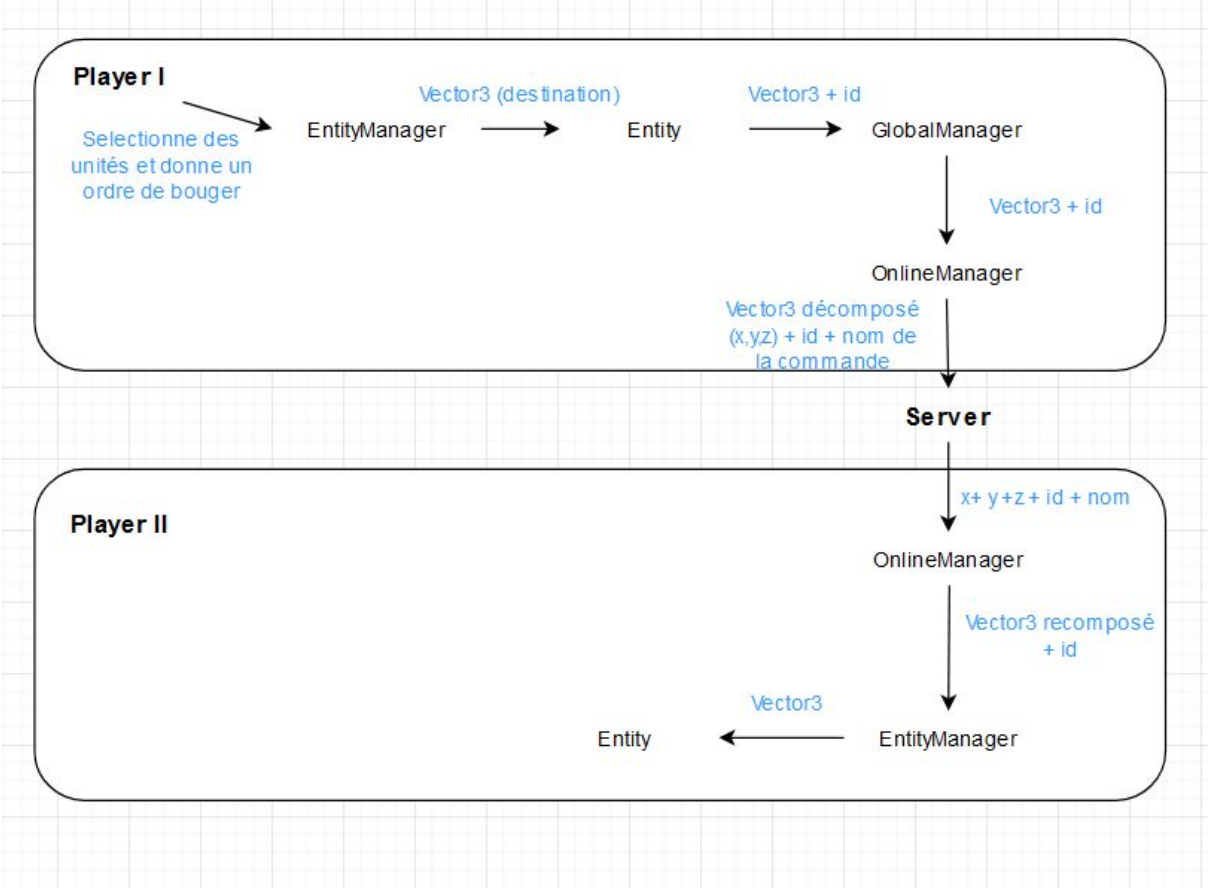
Les différentes infos que j'ai choisi de faire passer aux autres joueurs sont

- les déplacements des unités
- les unités ciblées
- les unités spawn

C’est à cette étape là que **l’index in game** des entités devient util. Car de ce que j’en ai déduit on ne peut envoyer que des types de **variables Simple et des string** au serveur.
Ce qui implique que je ne pouvais logiquement pas envoyer une **Entity** ou un **GameObject**.

J’ai donc commencé par stocker toutes les **Entity** de la carte dans une **List** déclarée dans le **EntityManager**.
Je me suis assuré que le **“recensement” de ces Entity** se fasse toujours dans le même ordre sans quoi l’index est inutilisable.
Une fois que j’avais ma liste complétée et un index correspondant sur chacunes des entités je pouvais commencer à passer des infos concernant des entités d’un côté à l’autre du serveur.

Pour illustrer la démarche on va prendre l’exemple du déplacement. Le principe est ensuite le même pour les différentes infos énoncées plus tôt avec des variables différentes envoyées selon le besoin.



Une fois ce système en place on a des unités qui bougent et qui attaquent.

Cependant c’est là qu’arrive la désynchronisation. Des unités détruites chez un joueur ne l’étaient pas chez l’autre ou des barres de vie n’étaient pas au même montant des 2 côtés.
J’ai donc ajouté 2 infos à passer

- les unités détruites

- les montants des barres de vie

Pour les entités détruites on reprend simplement le même principe que plus haut à savoir : une **entité est détruite**, elle notifie le **EntityManager**, puis le **OnlineManager**, **server**, **OnlineManager**, **EntityManager** et enfin **destruction de l'entité de l'autre côté**.

Pour ce qui est de la vie, chaque unité notifie une méthode 1 fois par seconde en lui envoyant son montant de vie, l'info est envoyée de l'autre côté et le **EntityManager** de l'autre côté se chargent de remettre les infos à jour pour toutes les unités de la carte.

Pour les infos qui sont envoyée j'ai utilisé le même système à chaque fois pour éviter que les fonctions boucles.

```
public void SetDestination(Vector3 _desti, bool _fromNet)
{
    _navMesh.SetDestination(_desti);
    _navMesh.isStopped = false;
    arrived = false;

    if(!_fromNet)
        GlobalManager.Instance.OnlineManager.SetUnitDestination(indexInGame, _desti);
}

public virtual void SetTarget(Entity _target, bool _fromNet)...
```

Si la fonction est appelée depuis le **OnlineManager** le **boolean _fromNet** est passé à vrai et la fonction ne renvoie pas d'ordre au **OnlineManager**.

VI - Lobby

Pour le lobby j'ai créé une nouvelle scène sur laquelle le joueur arrive au lancement du jeu. Il a le choix entre

- créer une partie
- rejoindre une partie

Dans les 2 cas le joueur entre un **nom** et le valide. Le **nom** en tant que string est envoyé au **OnlineManager** du lobby qui vérifie si le nom est déjà attribué à une partie.
Si le joueur veut rejoindre une partie et que cette partie existe bien, une dernière vérification est effectuée pour s'assurer que la partie ne compte pas déjà le nombre de joueur nécessaire.
En cas d'échec de l'une de ces vérifications un message s'affiche pour le joueur

VII - Création de compte et BigData

Pour ce qui est de l'utilisation de **BigData**, la doc de Player.io est plutot bien faite donc ce n'était pas très compliqué..

J'ai commencé par créer une **Table** et un **DataObject** à la main dans cette table et récupérer les infos de cet objet dans le jeu.
Une fois que j'avais réussi à faire ça je pouvais donc comparer les identifiants entrés dans le jeu et ceux contenus dans la base de donnée et connecter le joueur si ces dernières correspondaient.

L'étape suivante était de créer un objet dans la base de donnée à partir du jeu j'ai donc simplement suivi l'exemple de la doc.

```
DatabaseObject car = new DatabaseObject();
car.Set("Name", "OldSpeedy");

//Create nested stats object
DatabaseObject stats = new DatabaseObject();
stats.Set("Acceleration", 3.2);
stats.Set("TopSpeed", 50);
stats.Set("Steering", 5);
stats.Set("WorkingBreaks", false);
car.Set("Stats", stats);

//Create laptimes array
DatabaseArray laptimes = new DatabaseArray();
laptimes.Add("20:34");
laptimes.Add("22:03");
laptimes.Add("22:30");
car.Set("LatestLapTimes", laptimes);

//Save car to table "Cars" under key "car20312"
PlayerIO.BigDB.CreateObject("Cars", "car20312", car, null);
```

```
DatabaseObject newAccount = new DatabaseObject();
newAccount.Set("PassWord", passWord);
newAccount.Set("Victories", 0);
newAccount.Set("Deafeats", 0);
```

Une fois le compte créé on teste si c'est possible de s'y connecter et tout est bon.

J'ai aussi ajouté une fonction pour vérifier que le nom de compte entré lors de la création n'est pas déjà pris et une autre pour que je joueur confirme son mot de passe en le tapant une deuxième fois.

Un ajout basique que j'aurais pu implémenter est le fait de rester connecté une fois le mot de passe entré en séparant les scènes lobby et login ou en ajoutant un objet persistant à travers les scènes qui contiendrait une structure avec toutes les infos nécessaires (id, compte) au lieu de les passer d'un manager à un autre.