

2PROJ – DOCUMENTATION TECHNIQUE

Une présentation en détail des composantes du jeu

Supinfo international
university

Table des matières

Introduction	1	Les éléments répliqués	13
Les bases	1	Les éléments non répliqués	14
Le jeu	2	Transférer des informations entre les Maps	15
La Map	2	Entre en jeu, la Game instance	15
Le Game Mode	3	Particularités de la Game Instance	15
Le Game state.....	4	Comptes et données.....	16
Le Player State.....	5	Le backend	16
Le Player Controller	6	Base de données.....	16
Parenthèse sur les Widgets.....	6	API	17
Le Pion (Pawn).....	7	Sécurité	17
Les modèles 3d.....	7	Tester le backend.....	17
La position du pion, et la « position » du pion.....	8	ELO et Match Making.....	18
Comment attendre le pion.....	8	UI et direction artistique.....	19
Les autres acteurs	9	Création des ressources visuelles	20
Dé et boîte à dés	9	Expérience utilisateur	20
Le plateau et les cases.....	10	Quand masquer des informations ? ...	20
D'un jeu local à un jeu en ligne	11	Désactiver au lieu de masquer	21
Les conventions Unreal Engine pour un jeu en ligne	12	Coloration par joueur.....	21
		Conclusion.....	22

Introduction

Assessment - Rendering[Soumettre un Devoir](#)

À rendre le 20 Juin par 1:59


Points 500

Soumission en cours

un téléchargement de fichier

Types de fichiers zip

Disponibles 11 Janv à 20:00 - 20 Juin à 1:59 5 mois



RÉGION
NORMANDIE

1 - Introduction

In order to promote Normandy, the region wishes to create a Monopoly game based on the different cities. This desktop game will have to be playable online, implement AI, be available in several languages for the convenience of tourists, and be supported on different operating systems.

Your team is in competition with several other subcontractors to do the development, the best project will win the contract.

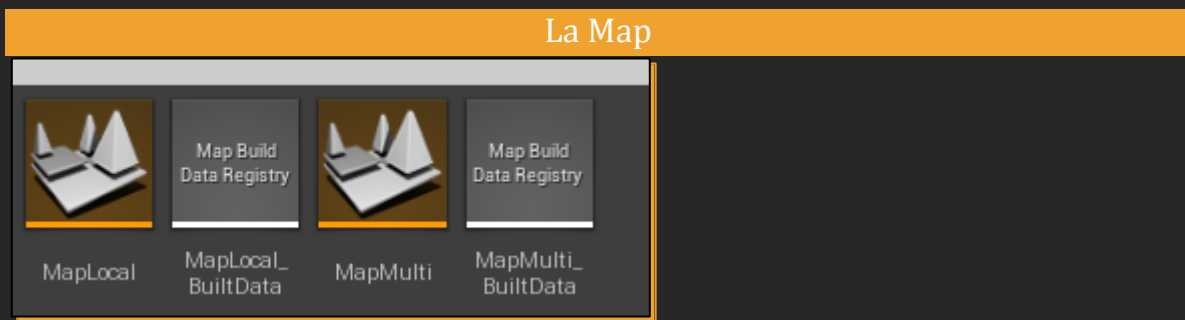
Notre projet de 2^e année à Supinfo International University consiste en la réalisation complète d'un jeu vidéo, celui du Monopoly. De nombreuses choses sont demandées dans ce sujet, et notre objectif a été de répondre au mieux à ces demandes par leur réalisation ou la présentation de nos pistes de réflexion.

Les bases

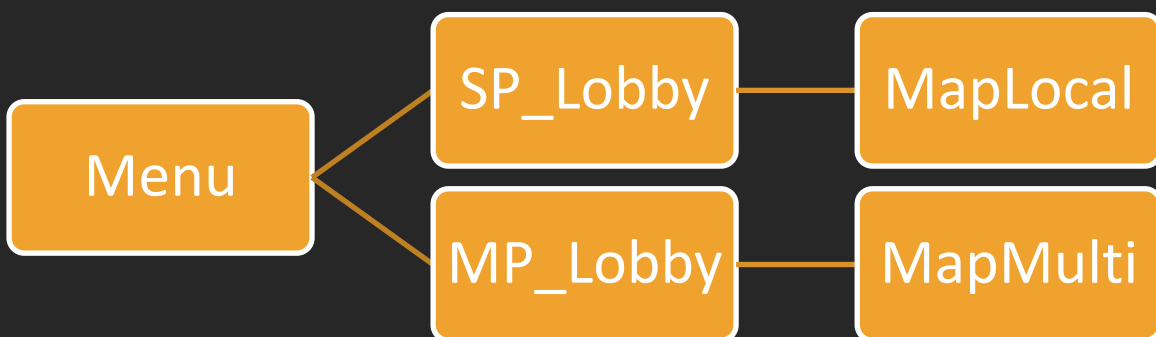
Le jeu a été réalisé sur Unreal Engine 4.27.2. Outil que chaque personne de notre groupe à du apprendre de 0, ce défi ambitieux a mené à plusieurs complications dont une réécriture complète du projet, mais cela a permis d'obtenir un jeu de meilleure qualité que si nous avions utilisé des technologies dans notre zone de confort.

Le jeu

Cette partie vise à présenter les différents composants du jeu et la manière dont ceux-ci interagissent. Les engrenages d'Unreal Engine sont différentes classes spécifiques, chacune d'entre elle remplissant une fonction bien particulière.



Représente le monde physique sur lequel les joueurs interagissent. Dans notre projet, Les maps suivantes ont été créées.



Le menu permet de choisir un jeu en ligne ou une partie locale, les maps « lobby » servent de menu de transition où les joueurs sélectionnent leurs pions, une fois cela fait, le jeu est lancé sur la map de jeu correspondant.

La gestion des tours et des joueurs ne fonctionne pas de la même manière entre multijoueur local et multijoueur en ligne. C'est pour cela qu'une séparation des modes de jeu en différentes maps est nécessaire, même si ces dernières sont des copies conformes.

A chaque map est associée un **Game Mode**.

Le Game Mode



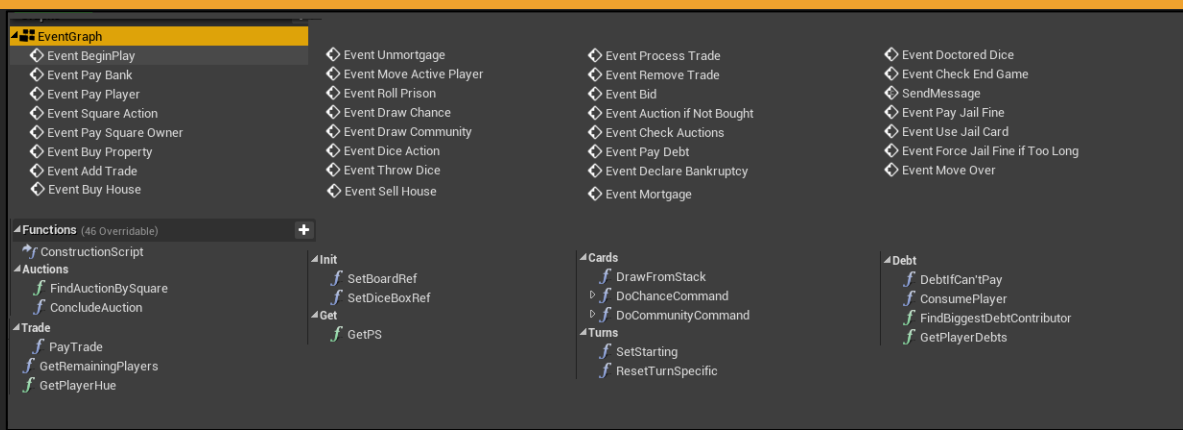
Le Game mode se trouve au cœur du jeu. Il initie la partie, gère les tours et décide du joueur actif. Dans le Game mode se trouvent les fonctions définissant le déroulement du jeu pouvant varier en fonction du mode de jeu. Mais comme celui-ci est dupliqué, il faut éviter d'y mettre les fonctions communes aux deux modes de jeu afin d'éviter la répétition.

Le Game mode possède un certain nombre de « classes par défaut ». Ces classes peuvent être surchargées pour que nous puissions mettre en place des fonctionnalités customisées.

Classes	
Game Session Class	GameSession ◀ 🔍 ✕
Game State Class	GS_Game ◀ 🔍 + ↗
Player Controller Class	PC_Game ◀ 🔍 + ↗
Player State Class	PS_Game ◀ 🔍 + ↗
HUD Class	HUD ◀ 🔍 + ✕
Default Pawn Class	None ◀ 🔍 + ✕ ↗
Spectator Class	SpectatorPawn ◀ 🔍 +
Replay Spectator Player	PlayerController ◀ 🔍 +
Server Stat Replicator	ServerStatReplicator ◀ 🔍

Les classes par défaut d'un gamemode, les surcharges étant représentées par les flèches dorées.

Le Game state



Le Game state contient toute notre logique de jeu, celle-ci étant la même que nous soyons en local ou en ligne.

C'est le Game State que nous appelons lorsque nous voulons lancer le dé, payer un joueur, effectuer un échange, enchérir, tirer une carte, acheter une maison, etc.

Si le Game Mode répond à la question « comment le jeu se déroule », c'est le Game State qui effectue tout le travail lors du déroulement de la partie.

Il contient les variables propres à la partie en cours (stack de cartes chance/caisse de communauté, etc...). Il contient aussi un tableau de tous les **Player state** du jeu.

Le Player State

C'est la représentation logique d'un joueur, toutes ses données, ainsi que quelques fonctions le concernant.

Variable	Description
Name	Le nom du joueur
Square	La case sur laquelle il se trouve
Money	L'argent du joueur
Properties[]	Les cases que possèdent le joueur
Chance[]	Les cartes chance que possèdent le joueur
Community[]	Les cartes caisse de communauté que possèdent le joueur
bInJail	Si le joueur est en prison
bDidLap	Si le joueur a effectué un tour du plateau durant son tour
bBankrupt	Si le joueur a déclaré faillite
bOutOfJailByRoll	Le joueur est sorti de prison en jouant un double, un second tour ne lui sera pas accordé s'il joue de nouveau un double
Status	Les différents statuts du joueur au cours de son tour
TurnsInJail	Le nombre de tours passés en prison, ne devant dépasser 3 sous peine de payer l'amende de force.

Bien qu'il n'y ait qu'une seule carte « libéré de prison » dans les decks chance et caisse de communauté, et que ces cartes sont les seules pouvant être possédées par un joueur, le choix a été fait de les stocker en tant que tableau afin de rendre le code plus modulaire, dans le cas où de nouvelles cartes seraient ajoutées et que le joueur puisse posséder celles-ci.

Les statuts d'un joueur peuvent être les suivants.

Status	Description
Waiting	Dans l'attente de son tour
Ready	Peut lancer le dé
HasLaunchedDice	A lancé le dé, doit attendre la fin de son déplacement
HasMoved	S'est déplacé, peut terminer son tour ou acheter la case sur laquelle il se trouve
ExtraTurn	A joué un double, peut terminer son tour, acheter une case, ou lancer de nouveau le dé

Le Game State appartient à un **Player Controller**

Le Player Controller

C'est la représentation logique dans le jeu du joueur lui-même, il n'a pas d'informations de jeu car celles-ci sont stockées dans son **Player State**. Il peut posséder un pion (**Pawn**), et effectue des appels d'évènements au Game State pour jouer.

Le Player Controller peut créer et afficher différents **Widgets**, qui sont les éléments d'interface graphique tels que des boutons/listes/panneaux. Ses widgets lui sont propres et non accessibles par les autres Player Controller.

Le **Game Mode** gère les joueurs grâce à leurs Player Controller, il définit le contrôleur actif et demande alors au **Game State** de changer le statut de son **Player State** pour qu'il puisse effectuer des actions

Parenthèse sur les Widgets

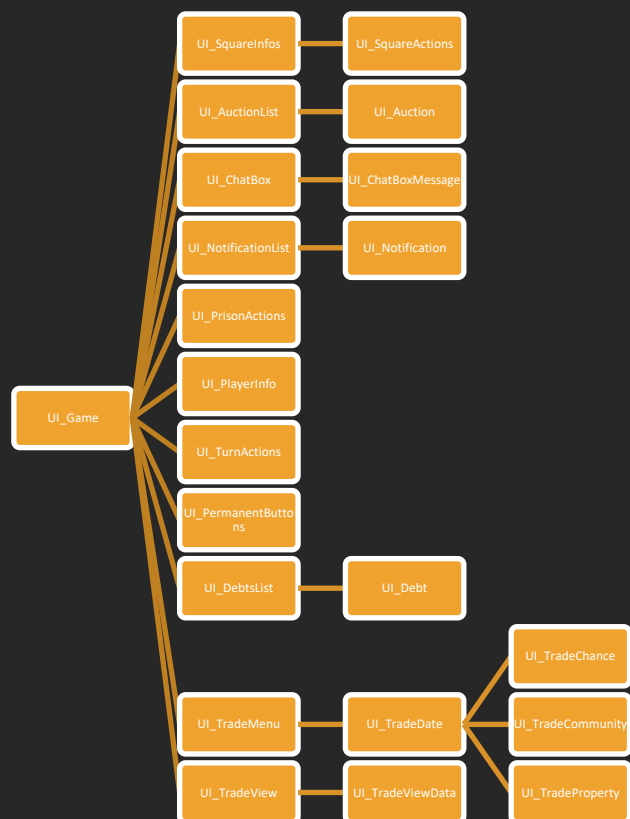
Un widget contient différents éléments tels que du texte, des boutons, des listes déroulantes, mais il peut aussi contenir d'autres widgets. Ainsi, en séparant les différentes fonctionnalités du jeu en leur propre widget, on obtient une hiérarchie.

Cette séparation facilite aussi leur gestion, par exemple, **UI_TurnActions** n'apparaît que lorsque c'est le tour du joueur, tandis que **UI_PrisonActions** n'apparaîtra que lorsque celui-ci est en prison.

Les widgets n'agissent pas sur le **Game State** ou **Player State** directement. Ils peuvent lire leurs valeurs, mais pour y effectuer des fonctions, ils passent par le **Player Controller**.

Nous avons ainsi

1. Bouton « lancer le dé »
2. Fonction « lancer le dé » du Player Controller
3. Fonction « lancer le dé » du Game State
4. Dé lancé



Le Pion (Pawn)

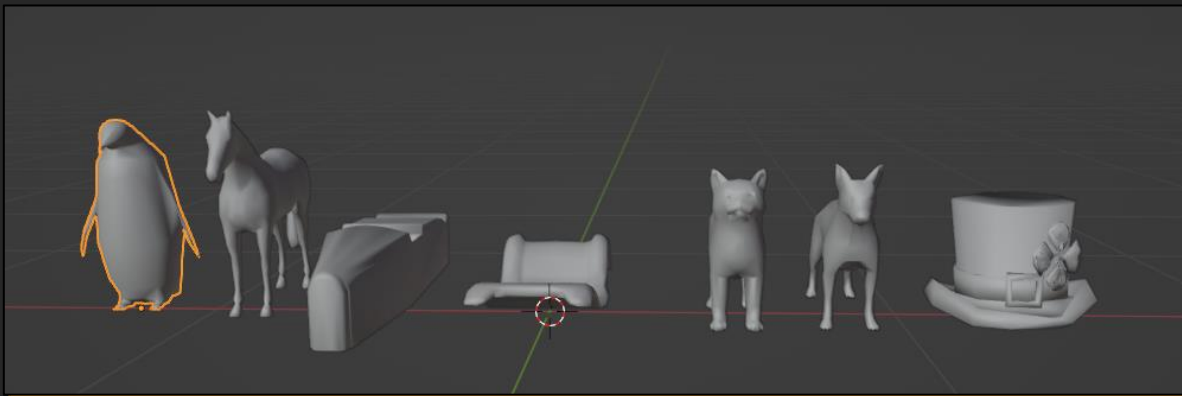
Un pion peut être possédé par un Contrôleur et représente l'entité physique du joueur, il apparaît sur la Map et peut se déplacer.

Les modèles 3d

Dans ce projet, le Pion représente le pion du joueur, il peut prendre différentes apparences.



Compte tenu de leur petite taille en jeu, le nombre de polygones de chacun des modèles a été réduit à la main pour optimiser les performances.



Ceux-ci provenant de sources différentes, leur taille a aussi de l'être normalisée.

La position du pion, et la position du joueur

Le pion possède sa propre variable *position*, celle-ci ne définit pas la position logique du joueur mais plutôt la position physique du pion. En effet, la position logique du joueur change instantanément lors de la lecture du dé, mais celle du pion doit s'incrémenter petit à petit pour permettre les animations. Cela explique le besoin d'une deuxième variable de position propre au pion.

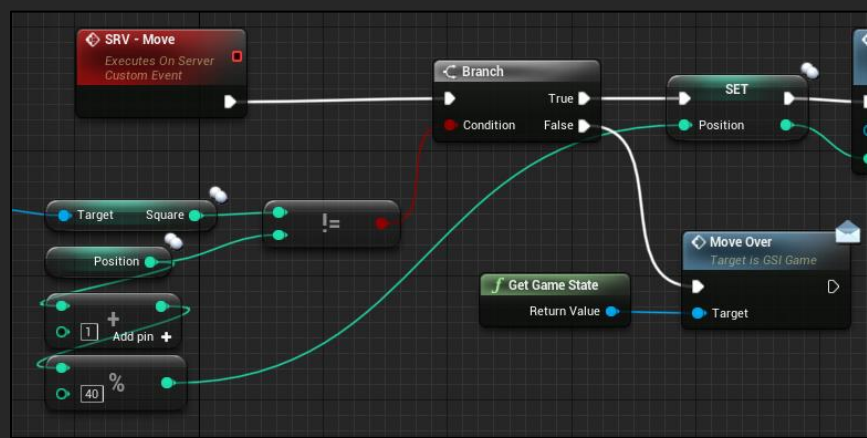
Celle-ci lui permet de sélectionner la bonne case, puis obtenir sa position sur la Map, et s'y rendre.



Comment attendre le pion

Comme expliqué précédemment, la nouvelle position du joueur dans le **Player State** est reflétée instantanément, or, l'animation du pion prend du temps, par conséquent, les « actions de case » telles que la possibilité d'acheter ou le tirage d'une carte doivent attendre la fin de l'animation.

Afin de permettre cela, ce n'est pas le **Game State** qui initie de lui-même l'action (tirage de carte, paiement, etc), mais c'est le pion lui-même qui demande au Game State d'effectuer son action lorsque l'animation est terminée.



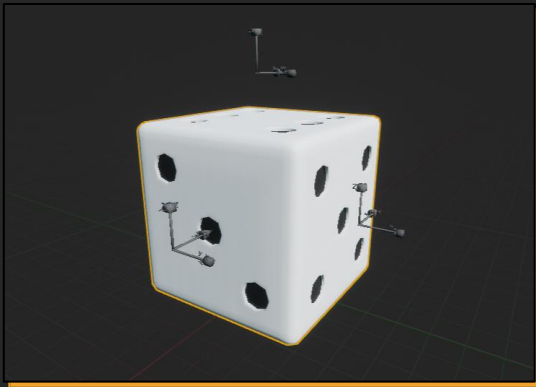
Une fois sa destination atteinte, le pion appelle le Game State

Les autres acteurs

En plus des classes « primaires » proposées par Unreal Engine, le jeu repose sur de nombreuses classes dites « actrices », il s'agit de classes posées ou générées sur la Map, et possédant les comportements que nous programmons en elles.

Dé et boîte à dés

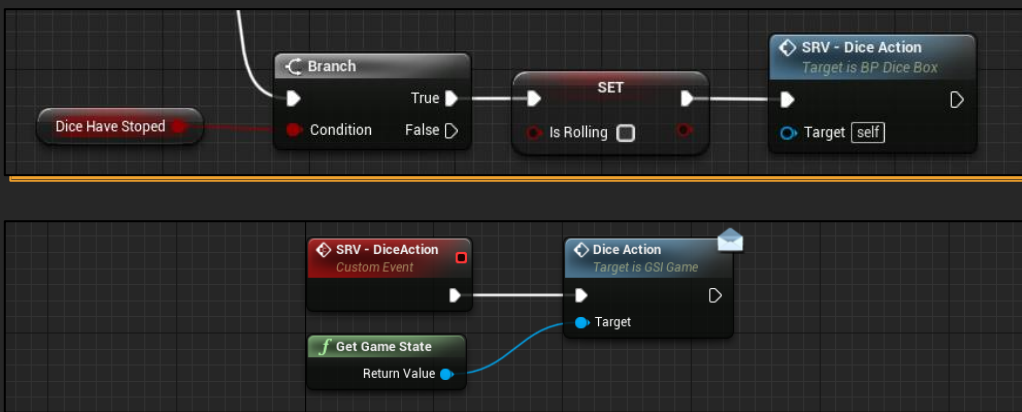
Ces deux acteurs fonctionnent ensemble pour permettre le lancer de dé. Le dé (*BP_Dice*) possède un modèle 3d de dé ainsi qu'un « capteur » sur chaque face, le capteur le plus haut à un moment précis indique alors la face du dé à lire.



Le dé, entouré de ses capteurs (invisibles en jeu)

La boîte à dés (*BP_DiceBox*) est à la fois un acteur physique et logique. Il est composé de murs invisibles permettant de restreindre les dés sur le plateau, mais il possède aussi des fonctions permettant de créer de dés et les lancer à une vitesse aléatoire.

C'est lorsque les dés deviennent immobiles que cet acteur appelle alors le **Game State** pour effectuer le mouvement du joueur.



Le plateau et les cases

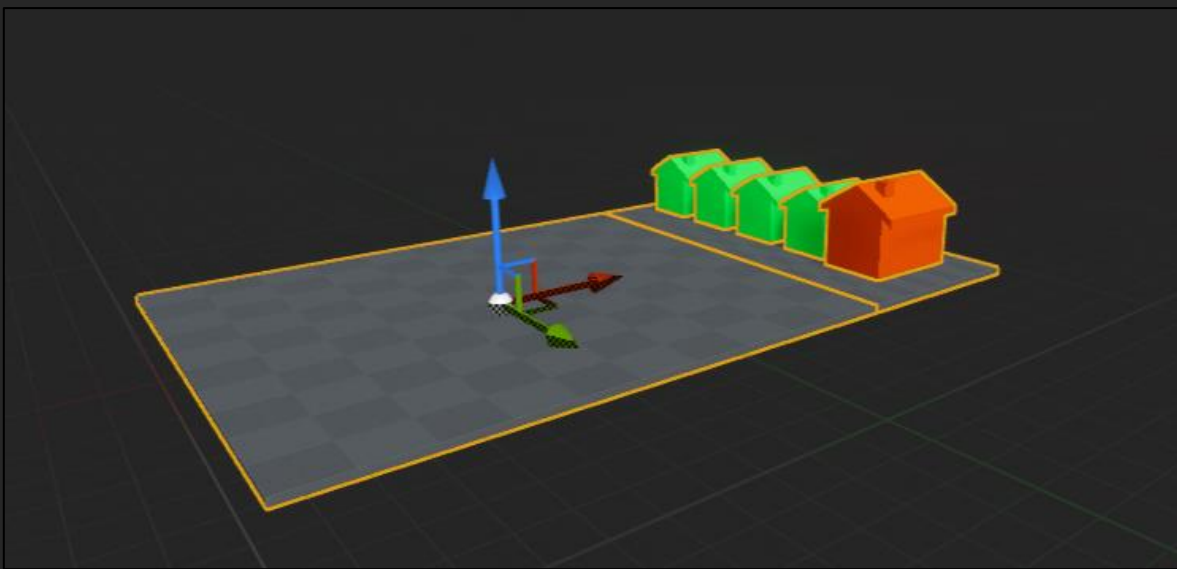
Le plateau (*BP_Board*) est un acteur logique, sa fonction est de contenir dans un tableau toutes les cases du jeu placées sur la Map, afin de servir de référence directe à ces cases pour le reste du code.

Il contient aussi les valeurs par défaut permettant d'initialiser toutes les cases.

Les cases (*BP_Square*) sont des acteurs placés directement sur la Map, ils contiennent de nombreuses variables telles que leur *nom*, *nombre de maisons*, *hotels*, *index du joueur qui la possède*, et *couleur*.

▷ Name	Text	▲ ▼ ✕
▷ Price	Integer	▲ ▼ ✕
▷ Type	EN Square Type	▲ ▼ ✕
▷ Color	EN Color	▲ ▼ ✕

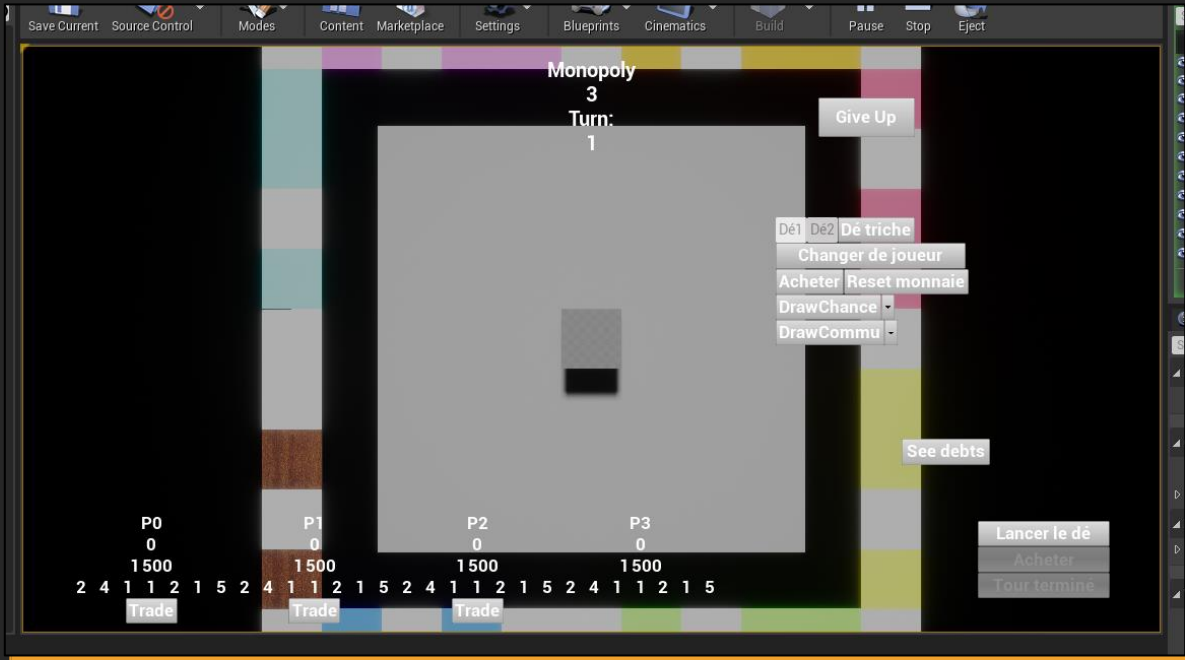
Une case renvoie ses coordonnées sur la Map pour qu'un pion s'y rende, il existe aussi une classe enfant (*BP_SquareThin*) qui contient des maisons déjà placées, celles-ci sont rendues visibles en fonction de la variable indiquant son nombre de maisons.



Là où la case possède des fonctions la concernant comme *Ajouter une maison* ou *Obtenir le prix*, le plateau contient des fonctions demandant un accès à toutes les cases, comme la *vérification de si une case appartient à un monopole*, ou la *Recherche de la prochaine case d'un certain type en partant d'une case précise*.

D'un jeu local à un jeu en ligne

Le jeu a d'abord été fait dans sa version multijoueur local.



Cette ancienne version avait la majorité de sa logique de jeu dans le **Game Mode**, et n'utilisait ni **Game State**, ni **Player State**. En recherchant comment transformer ce jeu local en jeu en ligne. Nous avons bien vite constaté qu'une restructuration entière du projet est nécessaire, la semaine du 30 mai au 6 juin a donc été consacrée à faire exactement cela, rebâtir le jeu.

En effet, les composants présentés précédemment doivent être gérés d'une certaine manière afin de pouvoir communiquer correctement en ligne.

Avec du recul, cette erreur était inévitable. En effet le fonctionnement d'un jeu en ligne demande une connaissance de base de Unreal Engine, chose que nous n'avions pas puisqu'il s'agissait de notre tout premier projet, cette version abandonnée du jeu peut alors être vue comme un « prototype nécessaire » pour se familiariser avec l'outil.

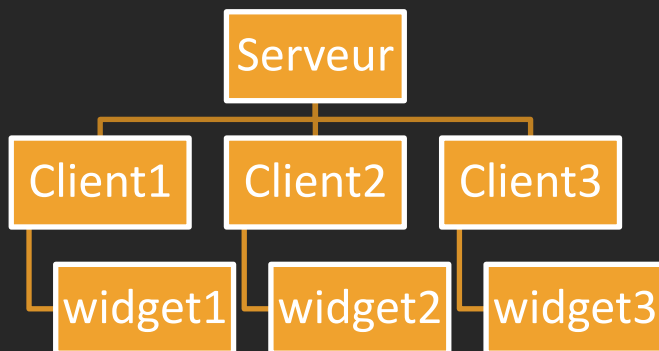
Les conventions Unreal Engine pour un jeu en ligne

Unreal engine met en place un modèle serveur-client, où l'état du jeu existe sur un serveur, et celui-ci est répliqué aux clients.

Cependant, il existe 3 différents niveaux dans cette hiérarchie.

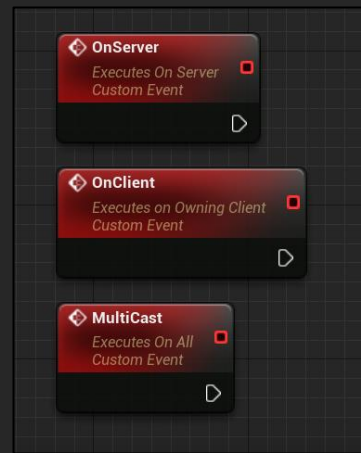
- Uniquement sur le serveur : Le **Game Mode**
- Sur le serveur et sur le client : **Player Controller**, **Game State**, **Player State**, **Acteurs**
- Uniquement sur le client : Les **Widgets**

C'est pour cela que nous avons focalisé la logique du jeu sur le Game State, et que, comme expliqué précédemment, les widgets doivent obligatoirement passer par le Player Controller pour interagir avec le jeu.



Les variables du Game State et des Player States sont répliquées si marquées comme tel, mais pour les changer sur tous les clients, alors le changement doit être effectué côté serveur. C'est pourquoi les différentes classes ont des options permettant de choisir où effectuer certains évènements.

Par exemple, si notre Player Controller veut lancer le dé, alors il doit effectuer l'action sur le serveur pour que celle-ci soit répliquée, c'est alors la copie du Player Controller SUR le serveur et NON celle du client qui initiera l'action. Dans le cas contraire, le lancer de dé ne sera vu que par un seul client.



Cela vaut aussi pour les changements de variables, et toute autre action importante du jeu.

Contrairement à un jeu d'action plus classique, l'aspect « tour par tour » du Monopoly fait que la quasi-totalité des Évènements sont appelés côté serveur.

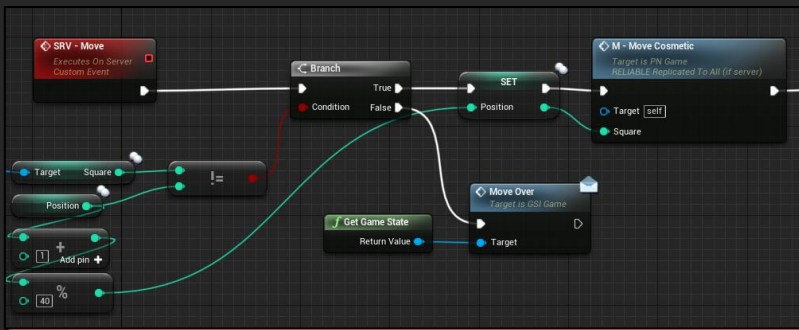
Les éléments répliqués

Sont alors répliqués pour ce jeu.

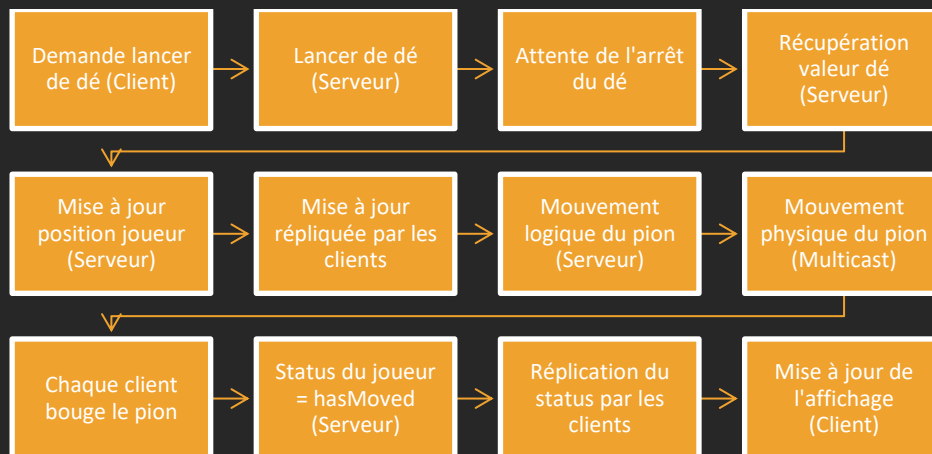
- Le Game State et ses variables
- Tous les Player States
- Les dés lancés
 - Le serveur effectue la simulation des dés, tandis que les clients ne simulent rien, et ne font que répliquer et interpoler la position des dés régulièrement

La sélection du modèle pour chaque pion, ainsi que leurs mouvements, ne peuvent pas être répétés aussi facilement. Ce n'est pas le pion lui-même qui bouge, mais plutôt un composant du pion, or, Unreal Engine ne réplique pas nativement ce genre de mouvements.

Une approche manuelle a donc été mise en place, celle-ci consiste à faire un appel multicast sur le serveur, pour que chaque client bouge individuellement le pion concerné.



La partie « logique » du mouvement est effectuée sur le serveur, mais un appel multicast est effectué pour que le mouvement soit vu par tous. Il est important de noter qu'un appel multicast n'a de sens que s'il est effectué sur le serveur, car le client n'a pas l'autorité de faire ce genre d'appels. Avec ces connaissances, un lancer de dé classique se déroule ainsi.



Les éléments non répliqués

Pas tous les composants du jeu sont sujets à la réplication. C'est le cas par exemple de la sélection d'une case.

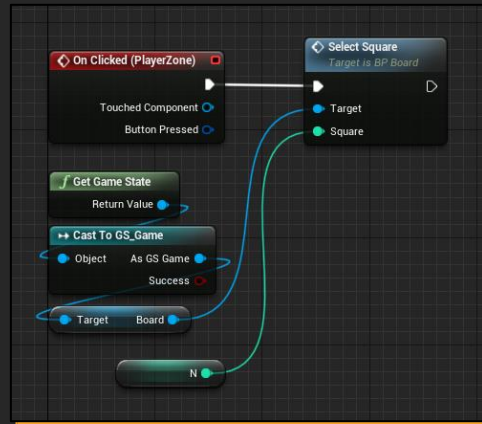


Lorsqu'un joueur clique sur une case, les informations de cette dernière s'affichent, ainsi que des boutons d'actions dans le cas où le joueur possède la case en question.

Or, les autres joueurs n'ont pas besoin de savoir sur quelle case un joueur en particulier clique, ils seraient même bien agacés si tous les joueurs partageaient la même sélection. C'est pourquoi cet élément n'est spécifiquement pas répliqué.

En effet, toutes les actions concernant la sélection d'une case sont effectuées uniquement sur le client initiateur.

Bien sûr, les actions déclenchées par les boutons d'achat/vente de maison sont, elles, effectuées sur le serveur.



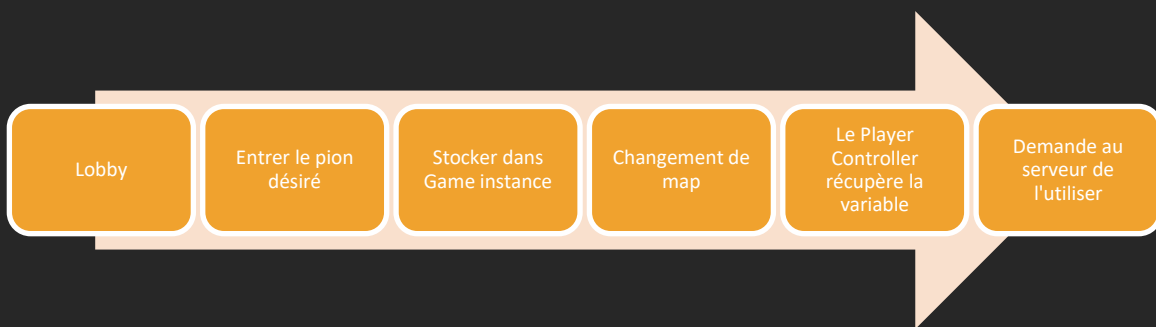
L'architecture du jeu repose donc sur une série de décisions quant à quelle action doit être effectuée où. La décision de répliquer ou non un évènement/une variable/un acteur fait partie des défis dans la mise en place d'un jeu en ligne comme celui-ci.

Transférer des informations entre les Maps

Le jeu est constitué de différentes Maps, or, nous pouvons observer des relations entre ces Maps. Puisque les lobbys servent à définir noms, pions, et autres données, cela signifie que ces données doivent être transmises entre ces niveaux. Mais aucune des ressources vues précédemment ne permet cela, en effet, le **Game Mode**, qui se trouve en haut de la hiérarchie, est lié à la Map et est donc perdu lors du changement. Comment assurer alors ce transfert ?

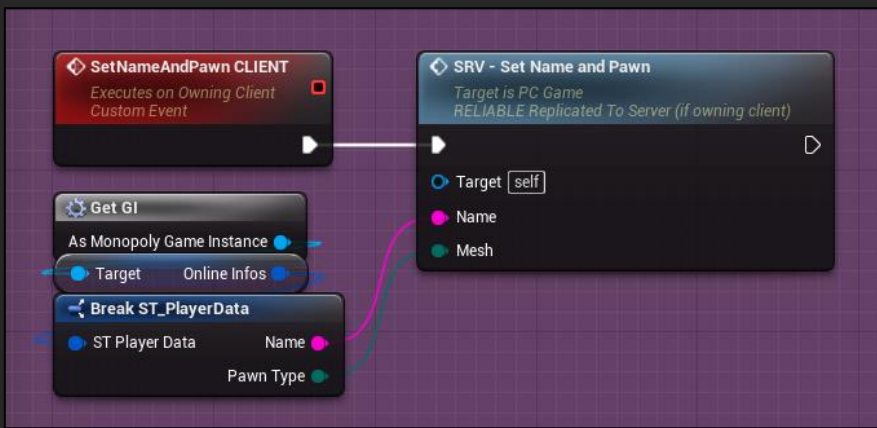
Entre en jeu, la Game instance

L'instance de jeu est une composante Unreal offrant encore plus d'abstraction, celle-ci est spécifique au projet et agit donc comme une classe globale où nous pouvons stocker des informations entre les niveaux, c'est d'ailleurs ce qui a été fait dans notre projet.



Particularités de la Game Instance

Celle-ci n'est pas à considérer comme un « Game Mode avancé », le Game Mode se trouve sur le serveur, il y en a un pour l'ensemble des clients. Or, la Game Instance est spécifique à chaque client. Par conséquent, la récupération d'une variable doit être effectuée sur le client pour ensuite être transmise au serveur.



Comptes et données

Une des demandes du sujet était la mise en place d'un système pour créer des comptes utilisateurs et leurs statistiques de jeu.

Le backend

```

1 // ...
2 // ...
3 // ...
4 // ...
5 // ...
6 // ...
7 // ...
8 // ...
9 // ...
10 // ...
11 // ...
12 // ...
13 // ...
14 // ...
15 // ...
16 // ...
17 // ...
18 // ...
19 // ...
20 // ...
21 // ...
22 // ...
23 // ...
24 // ...
25 // ...
26 // ...
27 // ...
28 // ...
29 // ...
30 // ...
31 // ...
32 // ...
33 // ...
34 // ...
35 // ...
36 // ...
37 // ...
38 // ...
39 // ...
40 // ...
41 // ...
42 // ...
43 // ...
44 // ...
45 // ...
46 // ...
47 // ...
48 // ...
49 // ...
50 // ...
51 // ...
52 // ...
53 // ...
54 // ...
55 // ...
56 // ...
57 // ...
58 // ...
59 // ...
60 // ...
61 // ...
62 // ...
63 // ...
64 // ...
65 // ...
66 // ...
67 // ...
68 // ...
69 // ...
70 // ...
71 // ...
72 // ...
73 // ...
74 // ...
75 // ...
76 // ...
77 // ...
78 // ...
79 // ...
80 // ...
81 // ...
82 // ...
83 // ...
84 // ...
85 // ...
86 // ...
87 // ...
88 // ...
89 // ...
90 // ...
91 // ...
92 // ...
93 // ...
94 // ...
95 // ...
96 // ...
97 // ...
98 // ...
99 // ...
100 // ...

```

Le backend pour ce projet est un serveur NodeJs, utilisant express pour mettre en place une API communiquant avec une base de données MySQL.

Base de données

La base de données MySQL contient une table *utilisateurs*

Variable	Type
Id	Int AUTO INCREMENT PRIMARY KEY
Username	VARCHAR(255)
Password	VARCHAR(255)
Created	DateTime
Games_played	Int
Games_won	Int

Afin de gérer un futur système d'amis, les tables suivantes peuvent aussi être considérées.

FriendRequests

Variable	Type
From	Int COMPOSITE KEY
To	Int COMPOSITE KEY

Friendships

Variable	Type
From	Int COMPOSITE KEY
To	Int COMPOSITE KEY

La séparation est nécessaire, car les demandes sont différentes des relations établies.

API

Le backend Node possède les Endpoint suivants.

Endpoint	Méthode	JSON	Description
/login	POST	username, password	Connecte un joueur, renvoie l'ID du joueur connecté
/register	POST	username, password	Inscrit un joueur dans la base de données et renvoie son ID
/game-over	PUT	username, hasWon	Incrémente le nombre de parties jouées pour le joueur concerné, incrémente le nombre de victoires si hasWon = 1
/user/:id	GET		Récupère les informations (sauf mot de passe) du joueur ayant l'ID donnée dans l'URI

Sécurité

Naturellement, le backend utilise des paramètres de requête afin d'éviter les injections sql, et les mots de passes sont hashés avant d'être stockés dans la base de données. L'obtention des données des utilisateurs ne renvoie ni mot de passe, ni donnée sensible. Cependant, l'endpoint game-over peut être exploité pour artificiellement augmenter son nombre de victoires.

Afin de contrer ces abus, une fois une infrastructure serveur mise en place, il suffira de whitelister la range d'adresse des serveurs afin que seuls eux puissent appeler ces endpoints.

L'implémentation actuelle est liée au manque de moyens financement pour héberger une infrastructure en ligne. Mais les schémas relationnels et le code backend ne changeraient que peu entre cette version et une version tournant sur des instances AWS.

Tester le backend

Pour tester le backend, il vous faudra avoir installé MySQL et npm.

1. Une fois l'archive du backend extraite, lancer mysql dans l'invite de commande, puis taper la commande `MySQL>source script` où « script » est le chemin du fichier `createDB.sql`.
2. Se rendre dans le dossier extrait et y exécuter la commande `npm install` pour télécharger les dépendances.
3. Une fois les dépendances installées et la base de données mise en place, utiliser la commande `node index.js` pour lancer le serveur.

Le serveur sera alors en mesure de répondre aux requêtes http.

ELO et Match Making

Les victoires et défaites peuvent être utilisées pour calculer le score ELO d'un joueur et le mettre en compétition avec des joueurs de niveau similaire.

$$ELO = 500 + \left(\text{victoires} - \frac{\text{défaites}}{x} \right) * y$$

x Est un coefficient de réduction du poids des défaites, son impact doit être moins fort que celui d'une victoire pour créer un sentiment de progression chez les joueurs, même s'ils ne gagnent pas tout le temps. Ainsi, une valeur de 3 ou plus pourrait fonctionner, celle-ci est à ajuster en fonction des retours des joueurs.

y Sert de coefficient de ségrégation, il définit l'impact des victoires, donc la séparation entre les joueurs de différents niveaux. Il peut être une constante.

Pour le matchmaking, nous pouvons exclure les sessions avec une différence d'ELO supérieure à un certain nombre (> 100).

UI et direction artistique

Unreal Engine étant un outil extrêmement puissant, il aurait été tentant de vouloir miser sur un style réaliste et haut en détail. Or, compte tenu du grand nombre de demandes du projet, il serait irréalisable de modéliser un environnement entier basé sur les paysages de Normandie.

L'approche stylistique est donc une simpliste, visant à faire quelque chose de beau dans le spectre de nos compétences plutôt que viser trop haut pour un résultat médiocre.

Nous sommes donc arrivés à cette interface.



Création des ressources visuelles

Excepté le plateau de jeu et les pions. Tous les visuels du jeu ont été faits à la main. Notamment l'interface utilisateur.



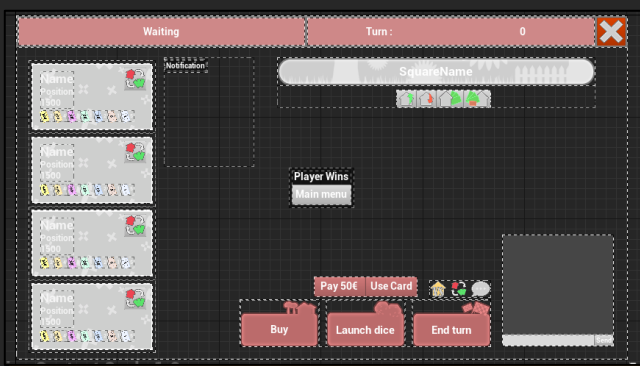
Expérience utilisateur

Un problème rencontré au début de la phase UI-Design a été de trouver un moyen de ne pas surcharger l'utilisateur, jouer au Monopoly demande de nombreuses informations quant à nos ressources et celles des autres joueurs, il est donc très difficile de trouver un bon équilibre.

Le décalage du plateau vers la droite a permis dans un premier temps de placer les cartes informations des joueurs sur la gauche et l'utilisation de boutons stylisés permet de communiquer l'action sans texte.

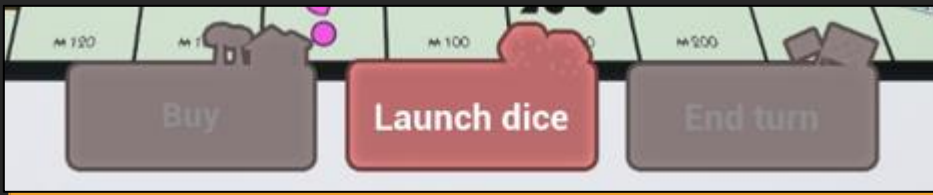
Quand masquer des informations ?

L'interface utilisateur serait trop chaotique si tous les boutons étaient toujours visibles, il faut donc prioriser.



Les cartes infos-joueurs sont importantes et restent visibles, mais les infos de case et les boutons d'actions peuvent être masqués lorsque ceux-ci ne sont pas nécessaires, de même pour les actions de prison.

Désactiver au lieu de masquer



Quand c'est le tour du joueur, tous les boutons s'affichent, mais certains sont désactivés, cela permet au joueur de comprendre qu'une action est nécessaire avant de pouvoir par exemple terminer son tour.

Il est aussi important de noter que désactiver les boutons au lieu de les masquer réduit la perte d'attention. Si ceux-ci disparaissaient, alors l'équilibre de la zone basse de l'écran serait en changement constant et casserait l'attention du joueur s'il regarde une autre partie de l'écran.

Coloration par joueur

Pour le multijoueur local, celle-ci est plus importante, les boutons et le cadre sont colorés ainsi afin de savoir immédiatement quel est le joueur actif (en utilisant comme référence les cartes infos à gauche de l'écran).

Ce même système de coloration permet de savoir à quel joueur appartient une case sans utiliser de texte pour écrire son nom.



Conclusion

En conclusion, ce projet de taille conséquente, combiné au choix ambitieux d'utiliser Unreal Engine, a créé une charge de travail bien supérieure à ce qui était imaginé.

De plus, il est difficile de bien s'organiser lorsque tout le monde est novice dans une technologie et que tout le monde gagnerait à toucher à tout (chose nécessaire lorsqu'on apprend un nouvel outil).

Mais malgré cela, ce projet nous a permis de repousser nos limites et explorer de manière assez poussée un potentiel outil de travail pour notre carrière.