Dossier rétrospectif

Boudier, Krzewinski, Levecque, Ponsdesserre, Proy

L3 informatique, Semestre 6

Table des matières

1 Introduction

Le projet consistait à réaliser différents programmes permettant de jouer au jeu de société Railroad Ink, avec comme simple base un réflecteur. Nous avons alors développé différents éléments qui nous permettent désormais de réaliser une partie. Vous pouvez retrouver les différents dépôts utilisés pour le projet en suivant ce lien : dépôts GitLab (connexion au VPN de l'Université de Lille requise). Le réflecteur est simplement un serveur Websocket renvoyant à tous les clients connectés les messages qui lui sont envoyés, un autre élément qui nous était donné était un jeu de messages dont vous pourrez retrouver plus de détails ici.

Avant de s'intéresser en détail à chaque élément que nous avons créé pour le projet, nous allons brièvement les présenter.

Tout d'abord, nous avons développé les programmes suivants :

— (Web Game Displayer) Un client permettant de jouer à l'aide d'un affichage web et qui permet aussi de regarder une partie grâce à un client spectateur.
— (Railroad) Un programme qui implémente la logique du jeu, qui inclut également le calcul du score.
— (Referee) Un client arbitral qui s'occupe d'autoriser l'envoi de certains messages, d'envoyer le score des joueurs à la fin d'une partie, de vérifier

les actions des joueurs.
— (PlacementHandler) Un programme permettant de déterminer les emplacements possibles pour une tuile donnée et un plateau de jeu donné, ce programme repose sur la logique du jeu.
— (GameMaster) Un client maître du jeu qui se charge de réaliser les lancers de dés au bon moment.
— (Client spectateur) Un client qui se charge uniquement de lire une partie depuis le réflecteur de l'afficher sur une page, il est possible d'y accéder depuis le Web Game Displayer.
— (Scripts de lancement et replay d'une partie) Un script qui permet de démarrer chaque élément de notre projet dans l'ordre qu'il faut avec les paramètres nécessaires, qui permet également de rejouer une partie à partir d'un fichier de sauvegarde.

1.1 Déroulement d'une partie et appropriation du sujet

Cette partie permet d'éclaircir quelques décisions que nous avons prises concernant le déroulement d'une partie ainsi que sur certains messages du réflecteur.

Voici comment nous avons choisi d'organiser une partie (Certaines fonctionnalités et subtilités ne sont pas précisées ici, notamment les clients de visionnage, de replay et de sauvegarde, mais détaillés dans les parties concernées dans la présentation de notre travail :

1. Le réflecteur est lancé
2. Le client arbitral se connecte au réflecteur avant tout le monde
3. Les joueurs ont un temps limité pour se connecter au réflecteur
4. A l'issue de ce temps imparti, le maître du jeu se connecte au réflecteur, et réalise un premier lancer de dés
5. Les joueurs jouent, pour indiquer qu'ils ont fini un tour, ils envoient un message *YIELDS*
6. Lorsque tous les joueurs ont envoyé un message *YIELDS*, le maître du jeu lance à nouveau les dés
7. Après sept lancers de dés, le maître du jeu envoie un message *LEAVES* pour indiquer la fin de la partie

8. Après que le maître du jeu a quitté, le client arbitral envoie les scores

Ensuite, nous nous sommes réappropriés les messages *BLAMES* et *SCORES*. En effet, dans le sujet il est indiqué qu'ils doivent être utilisés comme ceci :

— *id SCORES rang score points*
— *id BLAMES rang*

Et nous avons décidé de les modifier comme ceci :

— *id SCORES id2 points détail*
— *id BLAMES id2 détail*

Où :

— id désigne l'identifiant de l'arbitre (en l'occurrence referee)
— id2 désigne l'identifiant du joueur concerné par le message
— points désigne le nombre de points marqués
— détail désigne un message décrivant l'action illégale dans le cas de *BLAMES* ou bien décrivant la source des points dans le cas de *SCORES*

Enfin, nous avons choisi d'imposer le fait que l'arbitre et le maitre du jeu doivent respectivement avoir pour id **referee** et **gamemaster**.

2 Organisation du groupe

Pour organiser le travail tout au long du projet, nous nous sommes répartis le travail, nous avons utilisé Discord pour communiquer et nous avons pris l'habitude de faire un récapitulatif à chaque fin de séance afin d'éclaircir l'avancement du projet et de déterminer ce qu'il fallait faire la prochaine fois. Cela nous permettait également de se tenir au courant du travail effectué par chacun afin que chaque membre connaisse les différentes parties du projet et leur utilité.

FIGURE 1 – Extrait des récapitulatifs

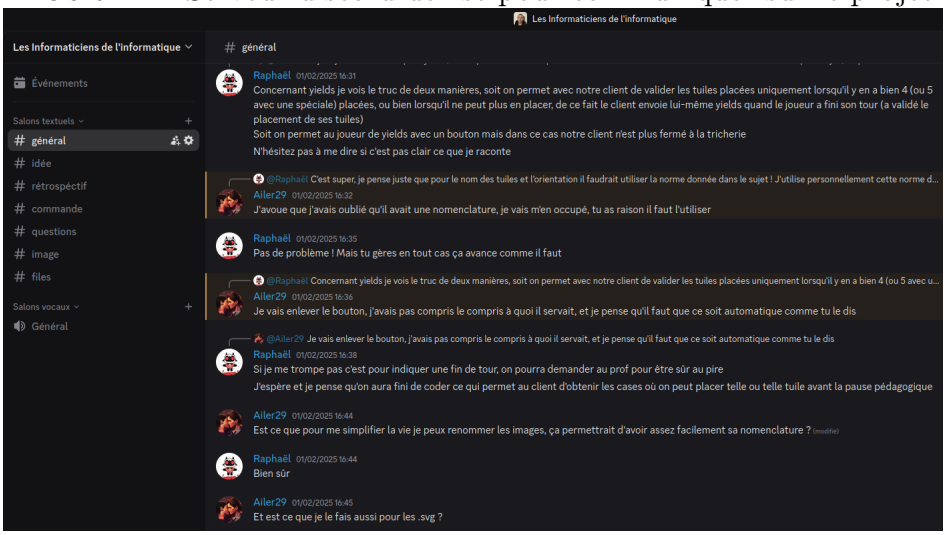
SEMAINE 13/01/25

- Approfondissement de la réflexion sur la suite de logiciels que nous allons créer
- Création d'un programme python qui permet simplement d'envoyer des messages au réflecteur via le terminal
- Début de la programmation de la logique du jeu en Java :
- Début du codage de l'affichage pour le client : reproduction de la grille en HTML et CSS, la suite consistera à programmer la possibilité de placer des tuiles dans la grille en javascript

SEMAINE 20/01/25

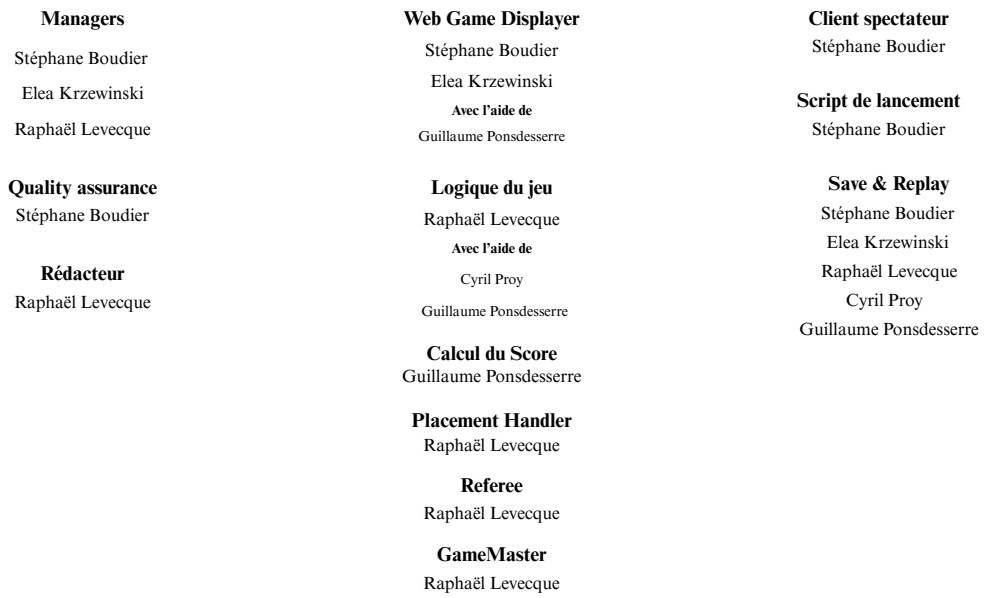
- Réflexion autour de la communication entre la partie graphique du jeu et la logique du jeu → Décision d'avoir une homogénéité de langage pour communiquer plus facilement
- Décision de créer un client maître du jeu en Java
- Avancement sur la partie graphique, amélioration du style, ajout de communication avec le réflecteur lors d'une connexion
- Réflexion sur l'algorithme du calcul du score
- Test du code java qu'on a décidé de supprimer

FIGURE 2 – Serveur discord utilisé pour communiquer sur le projet



Concernant la répartition du travail, la figure suivante résume quel rôle chacun des membres a occupé et sur quels éléments ils ont travaillé. Concernant la partie *Save & Replay* qui a demandé peu d'efforts mais qui nous a tous impliqué car elle nous a permis à chacun de se familiariser avec le réflecteur dès le départ.

FIGURE 3 – Organisation du groupe



3 Présentation de notre travail

3.1 Web Game Displayer (client de jeu)

3.1.1 Description

La composante Web Game Displayer est un client sur navigateur permettant de jouer au jeu RailRoad Ink et de regarder une partie (détaillé dans la partie client spectateur). Ce client présente d'abord une page d'accueil qui propose de jouer ou bien de regarder une partie. Si l'option jouer est choisie, le client demande au joueur de se connecter tout d'abord au réflecteur en entrant un identifiant, qui sera utilisé comme pseudonyme durant toute la partie. L'utilisation du navigateur permet à plusieurs joueurs de se connecter dans la même partie. Une fois connecté le joueur ne gère pas l'envoi de commandes vers le réflecteur. Il place simplement les tuiles sur le plateau de jeu et le client se charge d'envoyer les commandes correspondantes, par exemple :

```
Raphael PLACES Hc S B6
```

Cyril YIELDS

ou encore

Elea ENTERS

Cette composante est codée en majorité JavaScript, elle utilise deux technologies, socket io et websocket. Nous utilisons Websocket pour lire et afficher les messages envoyés sur le réflecteur, mais on ne peut pas envoyer des messages. L'utilisation de socket io est indispensable pour ce client. En effet il permet de gérer un chat entre les différentes instances de ce client connectées, ou encore d'avoir un affichage des plateaux des autres joueurs, qui utilisent ce même client pour jouer, après chaque fin de tour.

Au sein de ce client, nous utilisons également 2 autres composantes que nous avons créées, le placement handler et notre calcul score. Le placement handler permet de s'assurer que les tuiles sont placées uniquement à des endroits legaux selon les règles du jeu. Ainsi, notre client interdit la triche. De plus, pour commencer à jouer, il suffit de lancer le projet avec nodemon, ce qui lance un script qui permet de lancer le réflecteur sur l'adresse IP de la machine, de récupérer les informations nécessaires pour se connecter au serveur. Pour les joueurs, il suffit de se connecter au lien suivant : AdresseIPMachineServeur :9000/.

A la fin de la partie, le client de jeu permet également de consulter un classement.

3.1.2 Packages et principales classes

Les packages :
- `image/` : les images pour l'affichage des tuiles
- `script/` : les codes JavaScripts
- `style/` : le CSS de la page HTML

Les principales classes :
- **IOController** (`ioController.js`) : Gère les differentes connexions des clients, le chat ou encore l'affichage des plateaux des autres joueurs.
- **PlacementManager** (`game.js`) : Implémente les fonctions principales pour jouer

3.1.3 Effort de développement estimé

- **Nombre de classes : 7**
- **Nombre de lignes de code :** Environ 700.
- **Nombre de commits et de personnes impliquées :** 2 personnes impliquées pour un total de 60 commits .

3.2 Railroad (logique du jeu)

3.2.1 Description

Cette partie de notre projet implémente en JavaScript la logique du jeu. On retrouve en effet le plateau de jeu et les différentes avec leur orientation. Ceci a été une base pour développer d'autres programmes qui avaient besoin d'avoir la logique du jeu afin de répondre à leurs besoins. C'est effectivement le cas du Placement Handler, de l'arbitre et du calcul du score.

3.2.2 Packages et principales classes

Les packages :
- `tile/` : Gestion des tuiles et de leurs propriétés.
- `board/` : Gestion du plateau de jeu et de son état.
- `score/` : Calcul et gestion des scores.
- `test/` : Contient les tests unitaires avec Jest.

Les principales classes :
- **Board** (`board.js`) : Représente le plateau de jeu.
- **Score** (`score.js`) : Implémente le calcul du score selon les règles du jeu.
- **Tile** (`tile/tile.js`) : Gère une tuile du jeu (placement, type, connexions).
- **TileType** (`tile/tile.util/tileType.js`) : Définit les différents types de tuiles.
- **RailExit** et **RoadExit** (`tile/tile.util/`) : Représentent les sorties du plateau de jeu

3.2.3 Effort de développement estimé

- **Nombre de classes :** 8 (`board.js`, `score.js`, `tile.js`, plusieurs utilitaires).

— **Nombre de tests :** 3 fichiers test (`board.test.js`, `score.test.js`, `tile.test.js`)
— **Nombre de lignes de code :** Environ 1000.
— **Nombre de commits et de personnes impliquées :** 3 personnes impliquées pour un total de 128 commits .

3.3 Referee (client arbitral)

3.3.1 Description

Ce client a été développé dans le but d'arbitrer une partie, c'est-à-dire que ce client qui décide quel message peut être envoyé par quel client, c'est aussi ce client qui vérifie que les actions des joueurs sont légales (en partie grâce au Placement Handler), et c'est également lui qui attribue le score à la fin de la partie.

Premièrement, il s'approprie les messages *SCORES* et *BLAMES*. Ensuite, à chaque nouvelle connexion il autorise le nouveau client à envoyer les messages *PLACES* et *YIELDS* jusqu'à ce que le gamemaster se connecte. Lorsque le gamemaster se connecte, l'arbitre lui autorise d'envoyer des messages *THROWS*.

Ensuite, l'arbitre vérifie que les joueurs ne réalisent **pas** l'une des actions suivantes :
— Placer une tuile qui n'a pas été lancée par le gamemaster
— Envoyer d'un *YIELDS* alors qu'il reste des tuiles à placer
— Placer deux tuiles spéciales au cours d'un même tour
— Placer deux fois la même tuile spéciale au cours d'une même partie
— Placer une tuile là où il ne peut pas
S'il détecte une de ces actions il envoie un message *BLAMES* (message modifié comparé au sujet, voir partie Déroulement d'une partie et appropriation du sujet). Si un joueur est blâmé, il est enregistré comme tricheur et est ignoré pour le restant de la partie.

Enfin, à la fin de la partie, c'est-à-dire quand le gamemaster envoie LEAVES, l'arbitre envoie le score de chaque joueur. Pour ce faire il envoie des messages *SCORES* (message modifié comparé au sujet, voir partie Déroulement d'une partie et appropriation du sujet). Il envoie alors le score total suivi du détail

de ce score. Le calcul du score est réalisé grâce à la logique du jeu et notamment la classe **Score.js** développée dans la partie **Railroad**. Concernant les joueurs qui ont été blâmé, l'arbitre ne calcule pas leur score et indique simplement *disqualifié*.

3.3.2 Effort de développement estimé

— **Nombre de lignes de code :** Environ 190.
— **Nombre de personnes impliquées :** 1 personne impliquée.

3.4 PlacementHandler (librairie)

3.4.1 Description

Pour notre client de jeu, nous avons pris la décision d'interdire la triche. C'est-à-dire qu'un joueur qui utilise notre client, ne peut pas placer des tuiles qui n'ont pas été tirées, il ne peut pas placer plusieurs tuiles spéciales par tour, il ne peut pas utiliser une même tuile spéciale plusieurs fois au cours de la partie et il ne peut pas placer des tuiles qui ne relient pas d'autres tuiles de même type.

Si certaines de ces interdictions sont gérées directement par le client, nous avons pris la décision de délocaliser la gestion des placements possibles pour un plateau donné et une tuile et son orientation données. Cela représente alors le rôle du Placement Handler. Pour le réaliser, nous avons choisi de le programmer en JavaScript afin d'assurer une certaine continuité avec le client, permettant de l'intégrer facilement. Il a d'abord fallu coder la logique du jeu, en particulier implémenter le plateau de jeu, les différentes tuiles ainsi que les orientations. À partir de là, nous avons ajouté une classe PlacementHandler qui enregistre un nouveau lancé de dés, et permet de savoir s'il reste des tuiles à placer, si une tuile avec une certaine orientation peut être placée.

Tout cela a ensuite été publié sous la forme d'un package npm, qui a pu être installé chez le client de jeu afin d'en permettre l'utilisation. De plus, cet élément est également utilisé par le client arbitral pour vérifier que certaines actions des joueurs sont légales, et pour récupérer le plateau de jeu de chaque joueur en fin de partie dans le but de calculer les scores.

3.4.2 Effort de développement estimé

— **Nombre de lignes de code :** Environ 80.
— **Nombre de personnes impliquées :** 1 personne impliquée.

3.5 GameMaster (maître du jeu)

3.5.1 Description

Pour assurer le bon déroulement d'une partie, il est nécessaire qu'il y ait un nouveau lancer une fois que chaque joueur a fini son tour. Il faut également mettre fin à la partie une fois les 7 tours terminés. Pour réaliser ceci, nous avons écrit un programme en Haskell qui permet d'enregistrer les joueurs d'une partie, c'est-à-dire, les clients connectés au réflecteur avant qu'il ne connecte lui-même (à l'exception de l'arbitre), il y a donc un ordre de connexion à respecter.

Une fois connecté, ce client attend que l'arbitre lui autorise l'envoi de messages *THROWS* avant de réaliser un premier lancer. Nous avons décidé que chaque joueur notifiait sa fin de tour en envoyant un message YIELDS au réflecteur. Ainsi, le GameMaster attend de recevoir un message YIELDS de chaque joueur connecté et préalablement enregistré, avant de lancer à nouveau les dés. Il est important de noter que le GameMaster ne prend donc pas en compte d'éventuelles triches. En effet, un joueur pourrait envoyer un message YIELDS sans avoir réalisé de placement auparavant, le GameMaster ne possède pas le rôle d'arbitrer, de vérifier les actions des joueurs. Après les 7 tours, il se déconnecte simplement du réflecteur.

3.5.2 Packages et principales classes

Les principales classes :
— **GameMaster** (`GameMaster.hs`) : Représente le GameMaster.
— **Parser** (`Parser.hs`) : Parser en Haskell.

3.5.3 Effort de développement estimé

— **Nombre de fichiers :** 2
— **Nombre de lignes de code :** Environ 200.

— **Nombre de commits et de personnes impliquées :** 1 personne impliquée pour 11 commits.

3.6 Client spectateur

3.6.1 Description

Le client spectateur a, comme son l'indique, l'objectif de permettre à chacun de regarder une partie. Pour ce faire, il lit simplement les messages envoyés sur le réflecteur pour tenir à jour le plateau de jeu des joueurs sur une page web. Cette page web est accessible depuis la page d'accueil du client de jeu, mais il est aussi possible de s'y rendre en ayant le bon URL.

3.6.2 Effort de développement estimé

— **Nombre de lignes de code :** Environ 350.
— **Nombre de commits et de personnes impliquées :** 1 personne impliquée pour un total de 9 commits.

3.7 Save Replay Client (clients de sauvegarde et de replay)

3.7.1 Description

Nous avons créé deux programmes qui permettent pour l'un d'enregistrer tous les messages qui ont été envoyés sur le réflecteur. Pour ce faire, il se connecte à la fin d'une partie, puis enregistre dans un fichier tous les messages qu'il reçoit de la part du réflecteur. L'autre programme envoie toutes les lignes d'un fichier au réflecteur, agissant ainsi comme un replay d'une partie enregistrée au préabable.

3.7.2 Effort de développement estimé

— **Nombre de fichiers :** 2
— **Nombre de lignes de code :** Environ 50.
— **Nombre de commits et de personnes impliquées :** 5 personnes impliquées pour un total de 9 commits .

3.8 Script de lancement et replay d'une partie

3.8.1 Description

Ce script bash a pour objectif de lancer une série de programmes nécessaires à l'exécution du jeu. Il s'assure que les dépendances node sont installées et suit un ordre précis pour le bon fonctionnement des services. Voici ce que réalise le script :

1. Installation des dépendances node
2. Lancement du serveur de jeu - Web Game Displayer
3. Récupération de l'adresse IP du serveur
4. Lancement du réflecteur avec l'adresse IP récupérée
5. Récupération du port WebSocket du réflecteur
6. Lancement du client save
7. Lancement de l'arbitre - Referee
8. Lancement du spéctateur - Spectate
9. Pause de 60 secondes durant laquelle les joueurs doivent se connecter
10. Lancement de GameMaster avec l'adresse IP et le port récupérés précédemment

A partir de ce script il est aussi possible de rejouer une partie et de regarder cette partie en ajoutant une option. Voici ce que réalise le script avec l'option en question :

1. Installation des dépendances node
2. Lancement du serveur de jeu - Web Game Displayer
3. Lancement du client replay
4. Lancement du spéctateur - spectate

3.8.2 Effort de développement estimé

— **Nombre de fichiers :** 2
— **Nombre de lignes de code :** Environ 150.
— **Nombre de commits et de personnes impliquées :** 1 personne impliquée pour un total de 15 commits .

3.9 Retour d'expérience et difficultés rencontrées.

Tout d'abord, ce projet a été une nouvelle expérience de travail collaboratif. L'importance pour nous résidait dans la bonne organisation et communication afin que l'avancement soit plutôt efficace. Par ailleurs, ce projet nous a particulièrement appris sur la réflexion autour de l'assemblage de différentes briques dans le but de créer un produit final qui répond au cahier des charges. Pour réaliser ceci, nous avons pris plaisir à travailler avec des langages de programmation différents, des librairies différentes, tout en devant se familiariser avec un élément qui nous était donné, le réflecteur, ainsi qu'avec les règles du jeu Railroad Ink.

Le processus de développement du projet n'a pas été sans embûches. En effet, au départ il a été compliqué de déterminer précisément ce qu'il était attendu. Les visions des différents membres s'entrechoquaient mais avec le temps nous nous sommes mis d'accord sur ce qu'on allait produire. Enfin, il y a eu quelques difficultés techniques, notamment lors du calcul du score et du maitre du jeu. Toutefois nous avons su surmonter ces difficultés afin de mener à bien le projet.

4 Améliorations

Bien que nous soyons globalement satisfaits du travail que nous avons produit, nous pensons tout de même que quelques éléments de notre projet pourrait être améliorés.

Par exemple, dans notre client de jeu, il y a une fonctionnalité qui permet à chaque joueur de voir le plateau de jeu des autres joueurs à chaque fin de tour. Toutefois, cela affiche uniquement les plateaux des joueurs qui utilisent notre propre client, on pourrait alors faire en sorte que cela affiche également le plateau de jeu des personnes qui jouent sans notre client.

Ensuite, à plusieurs reprises nous lisons des messages qui sont envoyés sur le réflecteur pour le bon fonctionnement de certains programmes (Maître du jeu, arbitre, client de jeu, spectateur), toutefois la lecture des messages n'est pas uniformisé et on écrit à chaque fois du nouveau code pour une fonctionnalité similaire. Ainsi, nous pourrions simplifier cela en créant une librairie

qui lit les messages envoyés sur le réflecteur de manière standardisée afin de l'utiliser dans les différents programmes qui en ont besoin.

D'autre part, nous avons indiqué qu'un joueur qui triche est blâmé. Malgré cela, le maître du jeu attend tout de même à ce qu'un joueur blâmé envoie un message YIELDS pour indiquer une fin de tour et ainsi assurer les lancers de dés. Nous pourrions de ce fait faire en sorte que lorsqu'un joueur est blâmé il est ignoré par le maître du jeu, en plus de l'arbitre.

Enfin, l'arbitre autorise les joueurs à envoyer des messages *PLACES* et *YIELDS*. Cela a pour but que chacun des clients sache quels clients sont les joueurs. En revanche, cela n'est pas exploité et les clients considèrent plutôt les joueurs comme étant les clients qui se connectent entre l'arbitre et le maitre du jeu. Nous pourrions alors faire en sorte que les joueurs soient reconnus grâce à ces messages afin de permettre une meilleure flexibilité dans l'ordre de connexion des clients.