
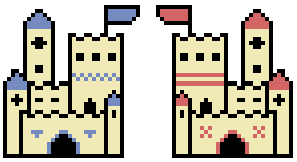


Jeu de la Sorcière : Witcher

Rapport de projet

MOREL Pierre-Jean MÉDINA Enzo LANGLAIS Hugo

3 mai 2020

Résumé

Le jeu de la sorcière est un jeu de plateau très peu connu avec des règles simples, mais il demande de faire preuve de stratégie et de réflexion. Ce jeu au tour par tour est donc facile à implémenter sous forme de programme informatique. Pour ce faire nous avons utilisé la bibliothèque réseau *PodSixNet* qui permet de réaliser des jeux multijoueurs en Python, ainsi que *Tkinter*, la bibliothèque graphique intégrée à Python, pour réaliser l'interface utilisateur et le jeu en lui même.

Implémentation

Le serveur, au centre du tournoi

Le serveur a le rôle de gérer toutes les transmissions d'informations entre les clients. Dès qu'un client a une information à donner ou demander, il envoie un message au serveur. Il s'assure de la connexion au tournoi des différents joueurs et fait le lien entre eux.

Une première classe `GameServer` hérité de la classe `Server` de PodSixNet sert uniquement à gérer les connexions/déconnexions de clients ainsi qu'à stocker la liste des objets `ClientChannel` (clients connectés). Il possède deux états `WAITING` et `TOORNAMENT` pour en attente de joueurs et tournoi lancé. Les joueurs peuvent se connecter mais ne peuvent pas se défier tant que le serveur est dans l'état `WAITING`. Une fois le serveur dans l'état `TOORNAMENT` les joueurs peuvent enfin se défier et plus personne ne peut entrer dans le tournoi. Ensuite, toutes les informations concernant les clients côté serveur sont stockées sous forme d'attributs de la classe `ClientChannel` hérité de la classe `Channel` de PodSixNet. Ces attributs sont : un booléen `available` qui indique si le joueur est disponible pour une partie ou s'il est en jeu, une chaîne de caractère `nickname` pour le pseudo du joueur, un entier `rating` représentant le score et un objet `other` du type `ClientChannel` qui représente l'adversaire ou qui vaut `None` s'il n'est pas en match. Ainsi chaque joueur coté serveur a un accès direct à son adversaire par cet attribut et n'a pas besoin de l'objet `GameServer`. Les `ClientChannel` sont ainsi autonomes et peuvent gérer directement les messages envoyés par leur client en mettant à jour les scores ou état des joueurs côté serveur. De cette manière, les informations sur les joueurs sont toujours à jour sur le serveur et peuvent être envoyés aux clients à tout moment.

Le client, informateur du jeu

Le client s'occupe de gérer le tableau des scores et transmet toutes les informations concernant les parties qu'il reçoit à la fenêtre de jeu. Quand un client se connecte il envoie son pseudo et après vérification de celui-ci par le serveur (pseudo trop long, nom déjà utilisé, etc.), le client demande alors la liste des joueurs qui sera envoyé à tout le monde. De même, quand il reçoit la liste des joueurs, peu importe la raison, elle est mise à jour dans la fenêtre en fonction des états de joueurs. Ensuite le client répond aux messages de demande de match en fonction du ranking des joueurs. Il est pour l'instant dans l'état `WAITING`. Quand un match est accepté par deux joueurs, une fenêtre `GameWindow` est alors créée et tous les messages concernant le jeu lui sont transmis en appelant ses méthodes. Il passe alors dans l'état `IN_GAME` et ne répond plus aux clicks sur la fenêtre du tournoi mais continue de recevoir la liste des joueurs.

La GameWindow, affichage du jeu

C'est dans cette fenêtre que le jeu est affiché et que les entrées de l'utilisateur sont gérées. Pour représenter le plateau de jeu, nous avons utilisé un dictionnaire d'objets `Case` avec des tuples de deux entiers comme clés, représentant les coordonnées des cases sur le plateau. Nous avons également créé deux objets `Player` représentant les deux joueurs sur le plateau. Les cases peuvent avoir plusieurs statuts : `"empty"` pour les cases vides, `"occupied"` pour les cases occupées par des rochers ou des joueurs ainsi que `"enemy_caste"` et `"castle"` pour les châteaux des deux joueurs. Les cases ont ainsi une image et des propriétés différentes en fonction de leur statut. Quand un joueur se déplace sur une case elle prend le statut `"occupied"` et quand il la quitte, elle reprend le statut `"empty"`.

Le plateau est affiché sur un `Canvas` avec un fond vert, il fait 9x7 et les cases 40x40. Le canvas est plus grand pour laisser de l'espace en haut du plateau de façon à afficher les noms des joueurs ainsi qu'une image représentant leur état : "bouger", "poser un rocher" ou "en attente". De plus, le joueur qui commence voit ses éléments (sorcier, rocher et château) colorés en bleu et le second joueur, colorés en rouge. Nous avons choisi de mettre des couleurs au rocher de façon à ce que les joueurs voient quels rochers ils ont posé. Cela ne change rien au jeu (il n'y a pas de différence dans le gameplay entre un rocher adverse ou non) mais permet améliorer la visibilité et d'adapter sa stratégie.

Si l'utilisateur clique sur une case non appropriée ou s'il est est en attente, rien ne passe. Enfin, à chaque nouveau déplacement ou rocher posé on vérifie si les joueurs ont perdu. Si aucune case n'est `"empty"` ou `"castle"` autour d'un joueur alors celui ci a perdu et l'autre a gagné.

Un diagramme récapitulatif des actions principales entre clients et serveur est disponible en annexes.

Expérience

Pour ce projet nous avons dû faire preuve d'une grande organisation étant en groupe de trois. Nous avons dû nous répartir les tâches afin d'être efficace au maximum. Pour cela nous avons notamment utilisé Github pour gérer le code.

D'autre part, nous avons beaucoup appris sur la réalisation d'un serveur orienté tournoi. Cela implique de gérer de bien des cas, dû aux nombre d'états d'un joueur, si l'on veut que toute les situations soit gérées et qu'il n'y est pas de bugs si par exemple deux joueurs demandent le même joueur en même temps. Nous ne pensions pas qu'il fallait autant réfléchir à la construction d'un tel programme et avons donc appris l'importance de réaliser des schémas représentant les différentes interaction entre toutes les parties du programme (comme celui fourni en annexes).

Enfin, il nous a fallu être très attentif dans la gestion de toutes les erreurs possibles : s'assurer qu'un joueur n'attende pas indéfiniment un joueur qui se serait déconnecté, s'assurer qu'on ne puisse pas défier un joueur qui vient de se déconnecter et inversement ne pas pouvoir accepter une invitation d'un joueur qui se déconnecte, etc. Faire attention à tous ces petits bugs nous a beaucoup appris !

Annexes

- ❏ Le code du projet est hébergé sur Github à cette adresse https://github.com/pjdevs/witchertornament .

- ❏ Schéma explicatif des relations client/serveur :