Documentation : *Projet Shogi*

Ou autrement dit : Rapport de Projet

Rédaction : NoobZik

Le rapport il à été rédigé en markdown, flemme d'ouvrir Office qui met 5 ans à démarrer pour rédiger un rapport projet. Accrochez vous bien, car le rapport que vous allez lire n'est pas un rapport comme les autres.

Introduction

Le sujet projet à été présenté durant le 20 mars 2017 en cours d'amphi de programmation impérative. Le sujet est donc le **Shogi**.

Tout le projet a été développé à l'aide d'un GIT dans lequel nous avons développé de manière à pouvoir continuer à développer l'application à distance et permettre de retracer entièrement l'historique du projet (rapport de bug, état de développement des branches du programme et des différentes manipulations dans le code). L'avantage principale est de supprimer totalement la fragmentation des versions du projet. Vous voyez, j'aime pas avoir plusieurs versions.

Cependant, la méthode du projet est de faire le projet chancun de notre côté, pour qu'on puisse avoir une idée du contenue de projet.

La supervision du répertoire git est exclusivement faite par NoobZik, en même temps c'est le seul qui a des connaissance dessus et qui n'aime pas attendre les modification par clé USB ou transfert par G-Drive ou mail.

La méthode de développement est la « méthode agile ». C'est-à-dire de par adaptation systématique de l'application aux changements du besoin détecté lors de la conception-réalisation du projet et par remaniement régulier du code déjà produit (re-factoring). Avec un planning de développement à base de secteurs dans le programme sans réelle précision.

En gros, on prends une fonction au hasard et on fait avec jusqu'à qu'on rencontre un problème.

Le projet est fait en sorte qu'il écrit sous forme de MVC (ou presque ...). Un petit rappel qui ne fait pas de mal : Modele, Vue, Controleur. La priorité absolue du projet est de modéliser la vue. Une fois que c'est fait, on peut commencer à coder en free-style.

Répartition des tâches

Les membres du groupe sont :

- **Rakib Sheikh (NoobZik) : Tous les déplacements + restrictions + tout ce qui en raport avec promotion / de-promotion + debug + makefile**
- **Ibrahim Kouyate : Le contenue du game.c - piles.c - files.c**
- **Emeric Bayard : Sauvegardes et chargements de fichiers + Contenue du piece.c**

C'est toujours provisoire, il faudra revoir l'organisation du code. Me tapper tout le mouvement c'est pas équitable, il faut divisier les fonctions de déplacements valides.

Bon la faut qu'on se fasse une réunion car la sa fait pitié cette partie de répartition de tâches...

Descriptions des fichiers

La liste des fichiers

Il y a au total 8 fichiers (à l'exclusion du main).

1. piece.c
2. pile.c
3. file.c
4. mouvement.c
5. restriction.c
6. game.c
7. sauvegardes.c
8. debug.c

Description détaillé des fichiers

Cette partie détaille les fonctions qui ont été modifié du sujet ou rajouté.

Un MAKEFILE pas comme les autres

La description de l'utilisation du makefile est indiqué dans le README

Cependant, le projet avait l'objectif très strict d'avoir 0 erreurs et 0 warnings à la compilation.

De ce fait, le makefile contient tout les affichages de warnings possible. C'est pour ça que l'affichage du terminal est très surchargé à la compilation du programme. Non, non ce n'est pas des erreurs de compilation ce que vous voyez à la compilation rassurez-vous.

Mais c'est pas du tout marrant de compiler avec tout les warnings dès le début du projet. C'est comme tricher. Je trouve que c'est vraiment marrant de le faire à la fin. C'est-à-dire losque le -Wall n'affiche plus rien à la compilation. Entre nous, compiler avec seulement le warning -Wall, ce n'est pas assez jouissif pour régler tout les warnings. -Wall c'est pour les débutants. Nous ce qu'on veut c'est du zéro tolérance, c'est amusant d'avoir environ 600 Warnings d'un coups.

Ca permet aussi d'avoir un code qui est propre, et optimisé pas comme dans les code des cours d'amphis.

---

piece.c

Dans ce fichier, contient tout les fonctions concernant à la gestion des pièces.

Note : Afin d'éviter toute ambiguité, Joueur 0 et Joueur 1 ont été remplacé par NOIR et BLANC.

Une structure pièce est caractérisé par trois champs :

- Un entier énuméré représentant la couleur, il y en a trois {Blanc, Noir et VIDE}.
- Un entier énuméré type représentant l'ensemble des pièces en plus de VIDE, SELECT* (Plus de détails plus bas).
- Un statut énuméré pour la représentation d'une piece pièce ou non promu.

Il est jugé plus simple de travailler sur des types énuméré que sur des type classique tel que les type unsigned int.

Les fonctions suivantes ont été rajouté ou modifié. Elles sont nécéssaire pour la suite du projet.

```
piece.c
piece_couleur(); Initialement piece_joueur(); Prend en argument une pièce piece_t et renvoie sa couleur associé.

piece_t promote_grant(); // Permet de promouvoir la piece.
piece_t demote_grant_reserve(); // Permet de dépromouvoir la piece en changeant de couleur.
piece_t demote_grant(); // Exactement la même chose qu'au dessus mais sans changer de couleur.
piece_t switch_color(); // Elle permet de changer de couleur avant de placer la pièce dans la réserve.
```

La fonction promote_grant permet de promouvoir les pièces selon les règles en vigueur. Le contenu de cette fonction est découpé en trois partie :

- La partie général qui donne le choix de promouvoir ou non les pièces sur les deux avant dernières lignes de l'adversaire.
- La partie spécifique qui impose des restrictions de promotion comme pour le cavalier et les parachutages.
- La partie pour impose une promotion direct lorsqu'on arrive à la dernière ligne de l'adversaire.

Lorsqu'on a essayé de modéliser les parachutes, on s'est rendu compte qu'il faut dé-promouvoir les pièces avant de les placer dans la réserve. On a une condition qui permet de vérifier si la pièce est à mettre dans la réserve en vérifiant son statut.

En fonction du résultat de la condition, il y a deux cas possible (soit deux fonctions) :

- demote_grant_reserve();
- switch_color ();

1. **demote_grant_reserve()**

Cette fonction permet de dé-promouvoir une pièce en changeant de couleur. Elle est appelée si cette pièce en question est promu.

2. **switch_color()**

Cette fonction permet juste de changer de couleur si cette pièce en question n'est pas promu.

3. **demote_grant()**

Cette fonction permet de dé-promouvoir une pièce sans changer de couleur. Elle est exclusivement utilisé dans la partie annuler_deplacement

---

pile.c / file.c

La *file* représente l'historique des coups joués et ses événement associé (Capture de piece et promotion).

On a une structure pour pouvoir gérer efficacement les coordonnées :

```
typedef struct movement_s {
    int movement; /* Représente le numéro du mouvement */
    coordinate_t input; /* Représente les coordonnées d'entrée */
    coordinate_t output; /* Représente les coordonées de sortie */
}movement_t;
```

Une fois qu'on à ça en tête de manière temporaire, on peut s'attaquer à la file.

En gros ce qu'un élément de la file contient :

- Un mouvement (On a vu ça au dessus au passage)
- Un booléen de promotion
- Un booléen de capture
- Et deux champs pour le mouvement suivant et précédent. C'est le principe d'une liste doublement chainée.

Cette élément de la file sera très utile lorsque l'on va traîter les déplacement expliqué plus tard dans cette documentation.

La *pile* représente l'historique brute des pièces capturés. En gros, la pile contient seulement les pièces capturés.

Pour la pile et la file, elle marche comme des liste qui sont doublement chaînée vu en TD. Cependant, deux fonctions on été introduite pour plus de clarté.

Lors de l'execution du programme, la pile et la file marchent comme des pile. C'est à dire qu'on fait que d'empiler les deux listes.

Cependant à l'enrengistrement, la pile et la file marchent comme des file. C'est à dire qu'on extrait le premier élément, jusqu'au dernier élément.

Les fontions suivantes on été rajouté pour plus de clarté dans le code:

```
file.c

void file_thread();
// Permet d'ajouter un mouvement, un état de promotion et un état de capture dans l'historique.
void file_unthread();
// Fait le contraire de file_unthread, sans libérer la mémoire, car on a besoin les données de cette élément extrait.

pile.c
void pile_stacking(); // Ajoute une pièce dans la pile.
void pile_unstacking(); // Retire la dernière pièce ajouté dans la pile.
```

---

mouvement.c

Les mouvement (autrement dit les déplacement dans le jargon du sujet) sont répertoriés dans ce fichier.

Il s'agit de vérifier si les déplacements sont possible ou pas, de manière très naïve.

Ce fichier est décomposé en 5 blocs :

1. **Validation de coordonnées d'entré et de sortie :**

- Pour les coordonnées d'entrées, on vérifie qu'il sont bien dans l'échiquier 11x11
- En revanche pour les coordonnées d'arrivées, on doit vérifier qu'il sont bien dans l'échiquier 9x9 (c'est-à-dire sans la réserve visuel). Une valeur de coordonée est créée pour désélectionner la case, qui est (42,42), si aucun mouvement n'est possible (Notamment lors des parachute de pions).

2. **déplacement_valide et valide_win**

- déplacement valide est composé de *switch case* pour appeler la fonction de déplacement valide adéquate. Ici il y a aussi un truc pour déselectionner la pièce, il suffit de mettre les mêmes coordonées de départ et d'arrivée.
- valide_win Permet juste de savoir que si les coordonnées d'arrivé pointe un roi, bah la partie se termine. (**Remarque**: apparemment d'après plusieurs application shogi, il n'y a aucune annonce d'échec ou échec et mat dans le monde de compétition, en conséquence, ces deux fonction ne sont pas implémenté).

3. **deplacement_valide**(Inserer type de piece)

- Ce bloc contient tout les déplacement des pièces non promu et des pièces promu

Je vous invite donc à regarder dans les commentaire pour leur fonctionnenement respectif.

4. **movement_valid_helper**

- Ce bloc sert essentiellement pour la restriction qui sera abordée plus en bas.

5. **deplacement_apply**

La fonction deplacement_apply est un peu complexe à comprendre.

A la base, deplacement_apply permet justement d'appliquer de manière naïve les déplacements dès qu'elles sont vérifiés. D'inscrire ce déplacement dans la file d'historique et s'il y a capture, retirer la pièce capturé de l'échiquier et le rajouter dans la pile de capture de pièce.

Mais dès l'introduction de la réserve de parachute, il a fallu le refacturer. Puisqu'il faut afficher cette pièce capturé dans la réserve. Et en plus de cela, il faut récupérer les données de promotion, s'il y a eu ou pas.

Avant de commencer à commenter la réserve, **il faut avoir en tête que**:

- La réserve du haut et de la gauche sans la case 0,9 constitue la réserve noir
- La réserve du bas et de la droite sans la case 10,0 constitue la réserve blanc Il y a une image d'illustration plus bas de cette documentation.

On commence donc à initialiser la valeur de promotion par

```
int promoted_v = is_promoted();
```

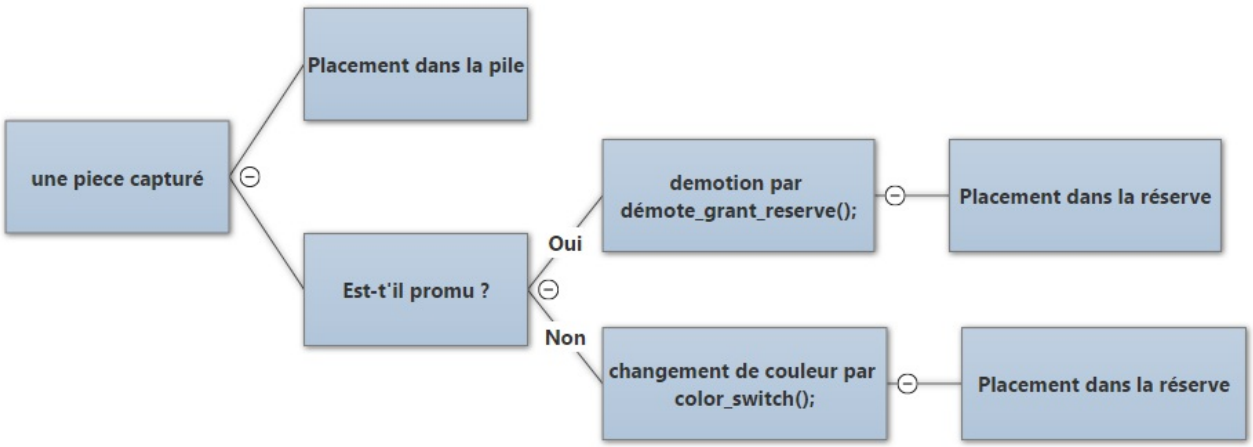
is_promoted est une fonction qui permet de vérifier si la pièce en question peut être promu ou pas. Le résultat est renvoyé sous forme de booléen.

Ensuite, on doit vérifier si les coordonnées d'arrivées contiennent une pièce. Dans le cas où les coordonnées d'arrivées contiennent bien une pièce, on place la pièce dans la capture des pièces qui est la pile, sans la retirer de l'échiquier.

Vient ensuite le placement dans la réserve.

Avant de continuer, normalement, vous devez remarquer que nous avons stocké la pièce directement dans la pile mais pas dans la réserve. La méthode qu'on a employé est le dédoublement de la pièce capturé. On place d'abord la pièce dans la pile, et ensuite on la modifie pour la placer dans la reserve visuel.

Diagramme décision : Cas d'une pièce capturé



Avant de commencer les boucles while tant attendu, il se peut que la pièce capturé est une pièce promu, dans ce cas, les conditions de promotions sont vérifiés. Si c'est vrai, la piece est dépromu en changeant de couleur. Dans le cas contraire, on change juste la couleur.

Les modifications liée à cette pièce est stocké dans une variable temporaire *p_tmp*

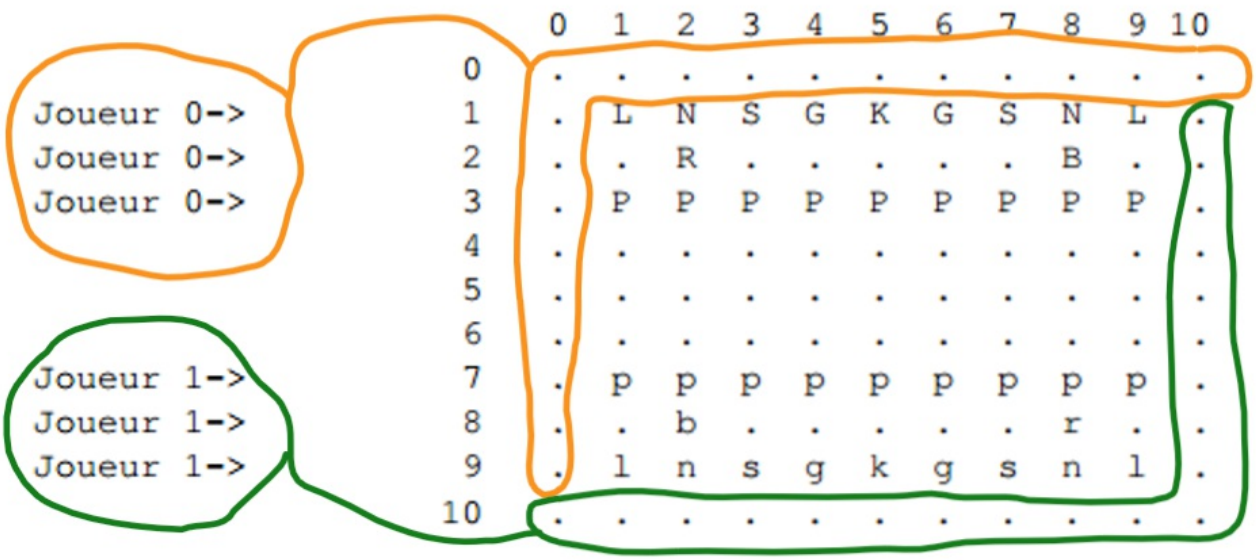
Cette condition est vérifé par la fonction :

```
piece_t demote_grant_reserve();
// Sinon
piece_t color_switch();
```

On en a parlée dans la partie piece.c

Comme il y a deux bloc de réserve pour chaque joueur, il y aura donc au totale 4 boucles while.

Cas universel: (Noir et vous devinerez pour les blanc qui est presque la même chose).



La première boucle va parcourir la réserve du haut, de la droite vers la gauche. Pendant ce temps, on teste si la case est vide :

- Si elle est vide : On place la pièce capturé modifié qui est stocké dans p_tmp à cette case, et on change la valeur de boucle pour sortir.
- Sinon on passe à la case suivante.

Dans le cas ou toute les cases sont occupée, on change la variable de la deuxième boucle pour qu'on puisse entrer dans la deuxième boucle.

Donc on parcours maintenant la réserve de gauche, du haut vers le bas. C'est le même algorithme que ci-dessus. Mais on parcourt jusqu'à l'avant dernière case (Car la dernière case c'est pour l'autre joueur).

Une fois qu'on à fait tout ça, il ne reste plus qu'à ajouter les données dans la file et de changer de joueur.

Les données de la file sont :

- Le mouvement.
- Le booléen de test de promotion.
- Le booléen de capture de pièce.

S'il n'y pas de pièce à l'arrivée, on applique tout simplement les déplacement.

6. **annuler_deplacement()**

Cette fonction permet de faire le chemin inverse de deplacement_apply.

On fait une extraction du dernier élément inséré de la file. Puis en fonctions des données de cette élement on peut soit :

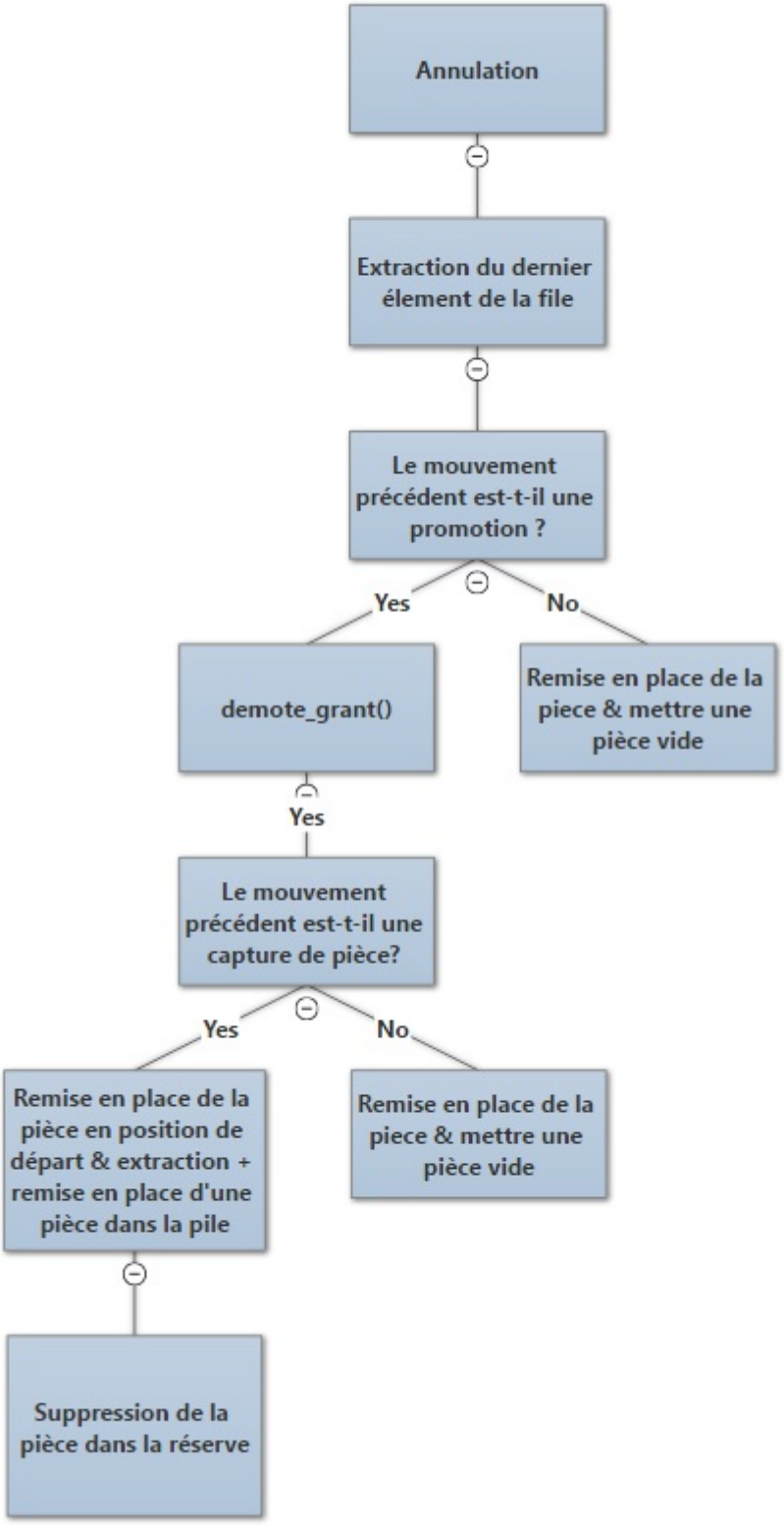
- Dépromouvoir une pièce si il y a eu une promtion de la pièce. Grâce aux information de la file, il y a un booléen qui permet de savoir si le mouvement précédent a donnée lieu à une promotion. Si c'est vrai, on utilise demote_grant()
- Restaurer une pièce capturé en prenant soin de traiter la réserve. (Il est possible qu'il est toujours buggé, Utilisez la fonction Signalement de bug sur le repo du bitbucket). Egalement avec les information de la file, il y a un booléen qui permet de savoir si le mouvement précédent donne lieu à une capture de pièce. Dans ce cas, il y aura un extraction de la pièce capturé situé dans la pile, avec traitement de la réserve.
- Et dans tout les cas, restaurer la position initiale de la piece en question avant le déplacement. Il suffit de remettre la pièce dans sa position de départ et de mettre une case vide dans la position d'arrivée.

Remarque : Dans le cas ou la pièce capturé est une pièce qui était promu, il y a pas besoin de promouvoir cette pièce. En effet, cette pièce existe déjà, elle était tout simplement caché dans la pile. Plus de détail dans la partie. Les détails sont dans deplacement_apply();

On prend aussi le soin de changer de joueur. Sinon c'est pas marrant de jouer deux fois d'affiler...

Pour résumer

Diagramme décision : Annulation



restriction.c

Cette partie du code est un peu délicate. Lorsqu'on a développé les déplacement, on s'est rendu compte que les pièces peuvent se balader sur tout échiquier même s'il y a des obstacles sur leur parcours.

Pour contrer ce comportement anormale, il a fallu introduire des restrictions de mouvement. Il y a cependant un nouveau problème. Les pièces pouvait quand se déplacer partout.

Donc pour contrer encore une fois ce comportement anormale, la fonction

```
mouvement.c

deplacement_valide();
```

est modifié pour que les pièces peuvent se déplacer exclusivement sur des cases de type '*' (SELECT).

C'est ici que se fait le prototype de l'aide visuel, avec les cases '*' (SELECT)

L'aide visuel *permet d'afficher les déplacement possible sur l'échiquier. Elle est très efficace lors des développement des déplacements de pièces*

Détails du fonctionnement des restrictions

Nous avons atteint les limites du code, en effet, il n'est pas possible d'avoir une fonction qui possède un argument optionnel.

Nous avons donc une coordonnées par défaut qui est donc comme d'habitude :

```
coordinate_t COORDINATE_NULL = {42,42};
```

Pendant l'affichage de l'échiquier, il y a une condition qui permet d'entrer dans la condition qui vérifie si les coordonnées sont différent de 42,42 justement.

Lorsque l'utilisateur entre une coordonnée d'entrée, la condition est vérifié et c'est ici que entre en jeux les restrictions.

La fonction :

```
restriction.c

movement_restriction();
```

est appelé durant deplacement_valide. Cette fonction appel la restriction adapté en fonction de la pièce sélectionné.

Si la restriction n'exige pas de conditions particulière, on fait appel a

```
Situé à mouvement.c

movement_valid_helper();
```

par une boucle for, pour tester les cases.

Entrons dans les détails de ***movement_valid_helper.***

Cette fonction permet de savoir si la case testé par la boucle for, est valide pou effectuer déplacement ou pas. Ils font appel naturellement aux déplacements valide des pièces respectifs, **A l'exception de tour_promu et fou_promu qui font appel à roi.**

Si vous voulez comprendre pourquoi cette exception. Tout simplement car si on ne met pas ces exceptions, alors l'aide visuel va tout simplement permettre de sauter les autres pièces pendant leur déplacement. Et justement c'est le but des restrictions d'empêcher les déplacement qui saute les autre pièces on le rappel.

Note : Lorsque on a implémenté les restrictions de tour_promu et fou_promu, on les a considéré qu'ils fonctionnent de manière théorique. Mais ce n'était pas le cas, ce comportement anormale à été découvert très tardivement, par faute de temps, elle à été remplacé par les restrictions du roi, en plus des restrictions de tour et fou respectivement.

Pour en revenir à ***movement_restriction.***

Pour les quatre cas *tour, fou, lance et parachute et leurs promotions respectifs* ont leur propres restrictions spécifique, qui sont définit dans le même fichier. Je vous invite donc à regarder les commentaires pour leur fonctionnement.

- **Note** : Les parachutes, sauf le cas des pions, tout les cases sont changé en ' * '.

Après avoir effectué les restrictions, le joueur saisie les coordonnées d'arrivé. La condition dans

```
mouvement.c

void deplacement_valide(); et
int restriction_detector();
```

doivent être tout les deux vérifier pour pouvoir appliquer les déplacements.

```
restriction.c

restriction_detector();
// permet de tester si la case d'arrivée est vide ou occupé par une pièce de couleur différente.
```

Un commentaire pour pion parachute

Donc les pions ont une restriction particulière, ils ne peuvent pas se placer sur la colonne où il y a déjà son compatriote du même type.

Pour ça, il y a 3 Boucles for. Il faut aussi noter que pour acceder à la troisième boucle il faut que la condition de test soit vérifié (Elle est changé par la deuxième boucle si besoin).

- La première permet de parcourir les colonnes
- La deuxième permet de parcourir chaque ligne de la colonne actuelle, si il y a on compatriote, on change la variable conditionnel pour ne pas rentrer dans la troisième boucle
- La troisième boucle permet de changer tout les case de la colonne en case SELECT pour l'aide visuel.

game.c

Pas grand chose à commenter ici, vu que game.c se focalise exclusivement de l'interaction utilisateur et machine par une interface graphique, appel les fonctions en fonction de l'interaction de l'utilisateur.

Il suffit de lire les commentaire dans le fichier game.c associé. Je ne vois pourquoi je détaillerai cette partie dans ce document...

---

sauvegardes.c

Donc déjà on commence très fort par :

```
struct stat st = {0};
```

Va nous servir à vérifier plus tard, l'existence du dossier de sauvegare par la fonction :

```
int stat();
```

La fonction :

```
void partie_sauvegarder();
```

est divisé en deux sous fonctions

```
void game_save_board(); Qui permet de sauvegarder l'échiquier
void game_save_meta(); Qui permet de sauvegarder le contenue de la pile et file.
```

Aussi, cwd permet de récupérer le chemin du projet actuelle pour pouvoir créer un dossier qui n'existe pas et de sauvegarder dans ce dossier.

Le fonctionnement de game_save_meta et de game_save_board sont assez similaire au vu de la structure du code.

Ne soyez pas choqué si vous voyez des return qui sont tout seul. C'est normal, on veut quitter la fonction dès qu'il y a une erreur. La première partie du bloc des sauvegardes est tout simplement une partie de concatenation. On va concatener dans l'ordre suivant:

- cwd
- /partie
- /
- "nom du ficher"
- Extension du fichier

1. game_save_board

Il suffit d'écrire PL au début du fichier. Et ensuite il faut juste afficher l'échiquier dans le fichier avec fputc.

2. game_save_meta

Il suffit d'écrire PR au début du fichier. Et ensuite on commence à écrire tout les éléments de la file dans le fichier (respectivement pile).

Un commentaire pour partie charger :

J'avais eu un problème de chargement de partie. Apparament, il y a eu des problèmes touchant à la bibliothèque string qui à entraîné pour la première fois les erreurs de segmentations et les erreurs qui sont directement pointé les fichiers includes du système (En même temps, si on code comme un dieu, on casse le code source du compilateur). Un mail à été envoyé au chargé de TD du groupe, mais resté sans réponse. J'ai du réfléchir à un autre moyen de coder pour eviter de re-casser le code source du compilateur.

L'erreur associé se trouve sur ces deux liens :

- Les logs du terminal
- Le code en question

Je m'en rappel plus comment je l'ai réglé ce bug d'ailleur, vu que je l'ai recodée à l'aide des cours (Non pas de Galilée mais de openclassroom).

Aussi ,les pièces ont été affecté a la mauvaise position lors du chargement du fichier. Il s'avère qu'en changeant la boucle de 0 à 12 (Initialement de 0 à 11) cela règle le problème mais je n'ai pas compris pourquoi... Ne me demandez pas pourquoi, je repondrai 'proute' avec les deux paumes de main en l'aire.

debug.c

C'est peut être la partie la plus fun de tout le projet. Il permet de moduler le programme en fonction de ce qu'on fait.

C'est-à-dire :

- Inspecter les cases de l'échiquier. Il se peut que des case de sont pas complètement vide. Inspecter une cellule permet de vérifier les champs de cette case.
- Prendre connaissance du contenue de la pile et file. Lorsque on à développé les pile
- Créer un plateaux SANS toucher au game.c

Le debug permet aussi de créer une partie sans toucher au game.c De cette manière, il est alors très simple de créer des situations de partie, pour pouvoir finir de programmer les fonctions. Par exemple, pour modéliser les déplacements du fou, on mettait un fou vide en plein millieu de l'échiquier. Lorsque l'on selectionne la pièce, l'aide visuel s'affiche. Et en fonction de l'aide visuel, on définit les conditions de déplacement.

C'est vraiment cool pour programmer le contrôleur. Plus besoins de refaire une partie à zéro pour tester un truc en rapport avec les déplacement. Tout ça, sans toucher au code principale.

Annexes

C'est la fin de cette documentation ou rapport de projet.