Rapport de Projet S5 : Mansuba

Groupe 17127

Rousere Joris
Tabouret Alexandre

Sommaire

Introduction

Présentation du sujet

Le jeu

Ce projet de programmation nous invite à coder des jeux de plateaux génériques en utilisant divers algorithmes et structures. En effet, l'objectif est de coder le déroulement d'une partie d'un jeu de plateau tour par tour opposant des pièces blanches à des noires. L'ordinateur déplace lui-même les pièces des deux camps les unes après les autres de manière aléatoire. Le sujet nous invite à développer des versions de plus en plus développées du jeu, à travers des "achievements" numérotés, en ajoutant petit à petit des nouvelles pièces, modifiant le plateau, et ajoutant des règles...

Les pièces d'un joueur sont placées sur la première colonne et les pièces de l'autre joueur sur la dernière colonne. Ces positions sont appelées les positions de départ de chaque joueur. Le but de chacun est de déplacer ses pièces pour les positionner sur les positions de départ de l'adversaire. On distingue deux types de victoires :

- les victoires simples : un joueur obtient une victoire simple lorsque l'une de ses pièces occupe l'une des positions de départ de l'adversaire.
- les victoires complexes : un joueur obtient une victoire complexe lorsque toutes ses pièces occupent toutes les positions de départ de l'adversaire.

De plus, un nombre maximal de tours est imposé. Si à l'issu de ce nombre de tours aucun joueur n'a gagné (le type de victoire étant fixé à l'avance) il y a alors égalité (*draw*).

Le plateau

Le jeu se déroule sur un plateau. Un plateau est un espace plan fini, divisé en différentes parties appelées cases. Lors d'une partie, les pièces sont disposées sur des cases du plateau, à raison d'une case par pièce. Ainsi une case peut être soit vide, soit occupée par une pièce.
Dans un premier temps, le sujet nous invite à nous intéresser au plateau à cases carrées. C'est un plateau disposant d'un nombre fixé de lignes et de colonnes, où les cases forment un quadrillage comme le montre la figure1 ci-dessous.

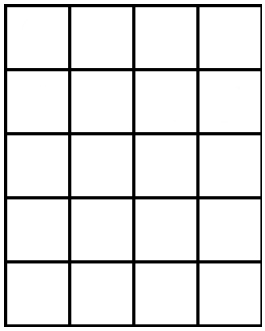

Figure 1: Plateau à cases carrées

L'achievement 2 nous propose alors d'implémenter deux nouveaux plateaux différents. Ainsi nous avons choisi d'implémenter un premier plateau à cases triangulaires et un second à case hexagonales. La figure 2 à la page suivante représente ces deux plateaux : le premier à cases hexagonales et de taille 4 x 5 et le second à cases triangulaires et de taille 6 x 4. Pour les case hexagonales, leur disposition reste très similaire à celle du plateau de base, la seule différence étant dans le décalage des colonnes. Les lignes et colonnes restent inchangées. Le plateau triangulaire est un peu plus complexe. En effet, on conserve l'organisation des cases hexagonales mais les lignes sont composées d'une alternance de triangle à l'envers puis à l'endroit.

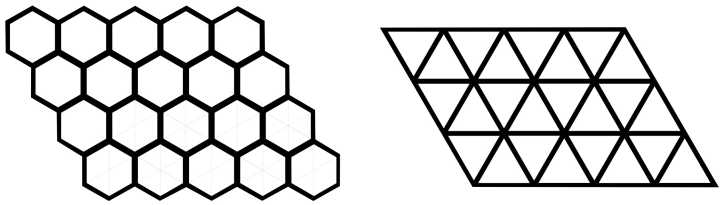

Figure 2: à gauche cases hexagonales | à droite cases triangulaires

Les pièces

Les pièces sont les seuls éléments mobiles du jeu. Chaque pièce possède une couleur qui l'associe à l'un des joueurs, et un type qui détermine les mouvements qu'elle peut faire.
Le sujet demande dans un premier d'implémenter le pion PAWN. Les déplacements de cette pièce se divisent en 3 catégories:

- le déplacement simple : le pion peut se déplacer sur l'une de ses cases voisines,
- le saut simple : si l'une des cases voisines est occupée par une autre pièce, le pion peut sauter au dessus de celle-ci et donc se déplacer de deux cases dans une direction,
- le saut multiple : après avoir effectué un saut, le pion peut continuer à faire des sauts simples autant de fois qu'il le souhaite à condition de ne pas retourner sur sa case de départ.

Ensuite, l'achievement 1 propose d'implémenter deux nouvelles pièces, la tour *ROOK* et l'éléphant *ELEPHANT* :

- la tour : peut se déplacer d'autant de cases qu'elle le souhaite dans l'une des directions cardinales (elle ne peut pas dépasser une autre pièce),
- l'éléphant : se déplace deux fois dans une direction cardinale (sans retourner sur sa position de départ). Autrement dit, l'éléphant se déplace d'une case en diagonale ou de deux cases dans une direction cardinale (peut importe si la case intermédiaire est occupée ou non).

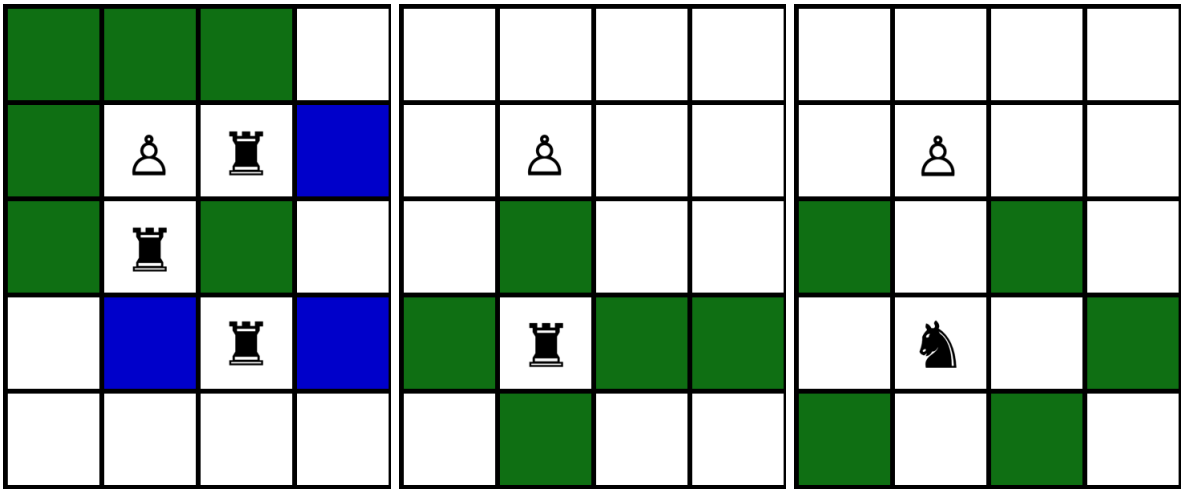

Figure 3: à gauche déplacements du pion, au centre ceux de la tour et à droite ceux de l'éléphant

Les différents plateaux de la figure 3 ci-dessus exposent les déplacements possibles sur un plateau à cases carrées pour les 3 types de pièces dans des configurations précises. Par exemple pour le premier plateau, le pion peut se déplacer sur les cases vertes ou alors sauter au dessus des tours adverses et atteindre les cases

bleues.

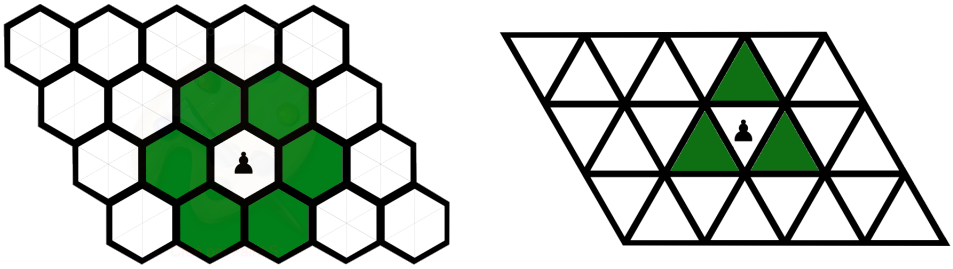

Figure 4: déplacements du pion sur un plateau hexagonale à gauche et triangulaire à droite

La figure 4 ci-dessus montre les déplacements possibles pour un pion seul sur les plateaux à cases hexagonales et triangulaires.

Organisation du rapport

Dans ce rapport, nous expliciterons le fonctionnement global de notre code final écrit en langage c, correspondant à l'achievement 2. Dans un premier temps, nous expliquerons l'implémentation réalisée en détaillant la division du code et les dépendances entre les différents fichiers. Puis nous développerons les rôles et fonctionnement de ces différents fichiers vis à vis de la réalisation du projet jusqu'au fichier final qui contient la boucle de jeu. Enfin, nous reviendrons sur différents aspects du projet tel que le développement en binôme.

1 Présentation globale de l'implémentation

1.1 Division du code

Nous rappelons que le but est de créer un jeu de plateau. Il faut donc modéliser un plateau, des pièces, et des règles qui permettent le bon déroulement d'une partie. Pour cela, nous avons divisé le code en différents fichiers, correspondant à ces différentes parties du jeu. Ces fichiers sont dépendants les uns des autres comme nous l'expliquons dans la partie 1.2.
L'intérêt de diviser le code est d'améliorer sa lisibilité et son organisation. En effet, il est plus simple de travailler sur un fichier contenant quelques fonctions qui concernent un point spécifique d'un projet, plutôt que d'avoir un fichier avec toutes les fonctions, devoir trouver la bonne partie du code... De plus, cela permet de pouvoir remplacer des parties du code simplement en changeant le fichier, ce qui peut être intéressant dans du développement à plusieurs par exemple. Néanmoins cela amène un problème de dépendances. Par exemple, la fonction d'un fichier peut nécessiter une fonction se trouvant dans un autre fichier. Il faut donc inclure celui-ci dans le premier, d'où l'utilisation de fichier d'extension **.h* qui contiennent les prototypes des fonctions nécessaires dans d'autres fichiers.

1.2 Dépendances

Les dépendances sont donc un point important du développement du projet. La figure 5 à la page suivante expose le graphe des dépendances. Dans celui-ci, les fichiers correspondent à des sommets et les flèches représentent les inclusions. Une flèche pointant d'un fichier A vers un fichier B signifie que le fichier B est inclus dans A.
En considérant qu'un fichier $A.c$ nécessite une fonction du fichier $B.c$, il faut créer un fichier $B.h$ qui contiendra un prototype de la fonction à importer. Puis on inclut le fichier $B.h$ dans $A.c$. Cela évite de faire des *"includes"* de fichiers **.c* et donc de recopier du code entièrement ailleurs. Pour notre projet, le fichier *play.c*

est celui contenant ce qu'on appellera la boucle de jeu : c'est le fichier principal qui gère le déroulement d'une partie. Ce programme fait appel à de nombreuses fonctions que nous avons séparées dans différents fichiers comme le montre le graphe. De plus, les autres fichiers peuvent eux-mêmes en inclure d'autres. C'est le cas de *world.h* qui est inclus dans quatre fichiers.
Enfin, la figure 6 ci-dessous expose les fonctions principales de chaque fichier, que nous expliquerons dans les parties suivantes.

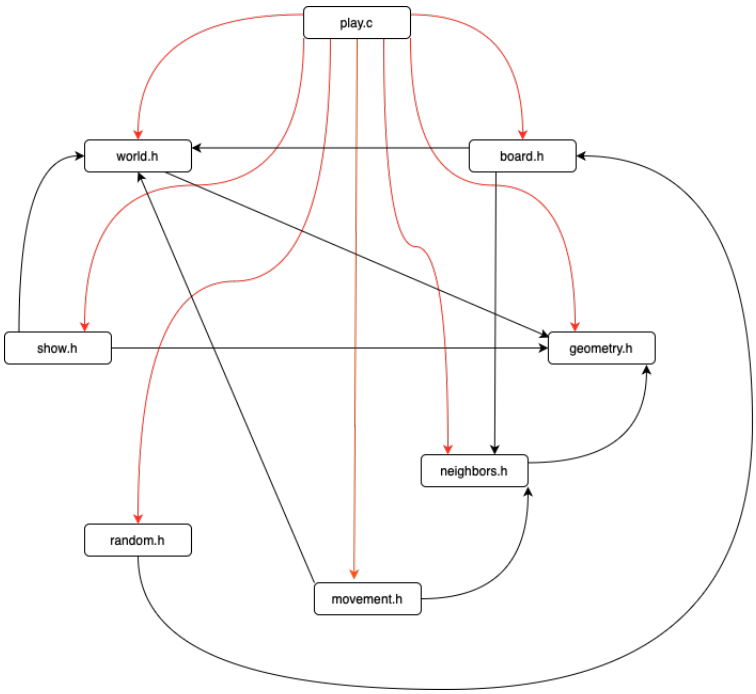

Figure 5: Dépendances des différents fichiers

Figure 6: Fonctions principales des différents fichiers

2 Plateaux

2.1 world.c et world.h

Pour pouvoir jouer sur un plateau, il faut un plateau. Il faut donc pouvoir le créer et le modifier. C'est à cela que servent les fichiers *world.c* et *world.h*. Plus précisément, ils implémentent une structure *world_t* qui sera notre plateau. Cette structure se compose d'une liste de structure appelée *square*, elle-même composée de deux *enum*. Le premier correspondant à une couleur, et le second à un type de pièce. Ainsi, notre plateau est représenté par une liste, où chacun des éléments correspond à une case. La présence ou non d'une pièce sur une case est derterminée par les 2 enums *COLOR* et *SORT* qui peuvent par exemple indiquer un pion noir (via *BLACK* et *PAWN*) ou rien (via *NO_COLOR* et *NO_SORT*). On note que les cases sont donc numérotées de 0 à *WORLD_SIZE-1* (où *WORLD_SIZE* est le nombre total de cases). Le plateau est donc représenté comme un unique tableau où chaque élément correspond à une case (les cases étant dans l'ordre croissant). Cela revient en quelques sorte à "déplier" le plateau de jeu et à l'agencer sur une seule ligne. Ainsi en considérant un plateau rectangulaire de 5 lignes et 4 colonnes, les cases de la première ligne sont numérotées de 0 à 3, celles de la seconde ligne de 4 à 7, celles de la troisième de 8 à 11...
La figure 7 suivante illustre un tel plateau.

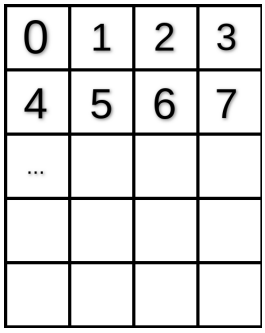

Figure 7: Plateau de base de taille 5 x 4

Le reste des fichiers *world.c* et *world.h* permet d'initialiser un plateau et de renvoyer un pointeur qui sera utilisé par les fonctions suivantes qui renvoient ou modifient les informations d'une case du plateau.

2.2 Différents modèles de plateaux

Le sujet présente d'abord le plateau comme étant rectangulaire et à cases carrées. Il possède un nombre de lignes HEIGHT et un nombre de colonnes WIDTH. Par exemple, en prenant WIDTH=8 et HEIGHT=8, on obtient un échiquier.

Les plateaux à cases hexagonales et triangulaires rajoutés par l'achievement 2 diffèrent légèrement. Le premier demande de décaler successivement d'une demi-case vers la droite les lignes. En revanche, pour le plateau triangulaire, la disposition des cases est un peu plus complexe : on retrouve le décalage des colonnes, auquel on ajoute une segmentation des lignes. En effet, visuellement une ligne est représentée par des triangles à l'endroit (pointe vers le haut) et à l'envers (pointe vers le bas) qui alternent. Nous avons choisi, par soucis d'affichage et de simplicité, de diviser une ligne en deux : la première ligne formée est composée de triangles à l'envers et la seconde de triangles à l'endroit. Par exemple, dans la figure 8 à la page suivante, si visuellement nous n'avons que 3 lignes, en réalité nous en considérons 6 (il est à noter que ce choix impose un nombre pair de lignes). Nous avons choisi de commencer la première ligne par un triangle à l'envers pour les mêmes raisons.

Il est à noter qu'avec les nouveaux plateaux choisis, la difficulté de l'implémentation réside dans le déplacement des pièces et les relations entres elles. En effet, nos deux nouveaux plateaux sont eux aussi ordonnés en lignes. Ce qui change réellement est bien le fonctionnement des pièces : pour le premier plateau

rectangulaire à cases carrées, une case possède 8 cases voisines (les cases adjacentes et les cases en diagonales) tandis que pour le plateau à cases hexagonales, une case a 6 cases voisines (les cases adjacentes).
Nous reviendrons ultérieurement sur les déplacements.

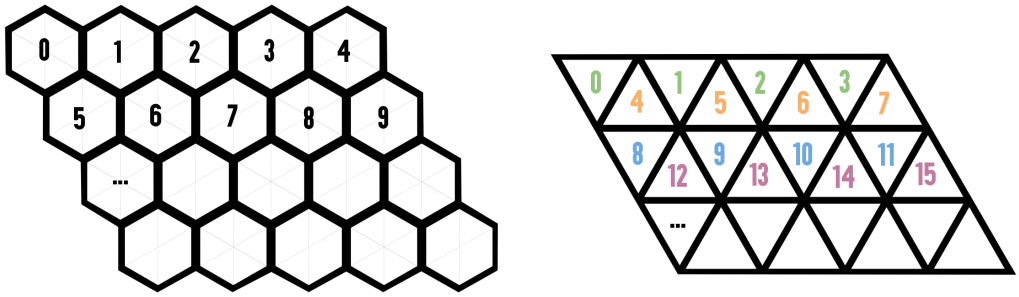

Figure 8: à gauche un plateau à cases hexagonales 4 x 5 | à droite un plateau à cases triangulaire 6 x 4

3 Cases voisines, pièces et déplacements

3.1 Cases et voisins

Maintenant que nous disposons d'un plateau, encore faut-il avoir des pièces pour pouvoir jouer. Mais avant même d'ajouter des pièces, il faut s'intéresser aux cases et leurs voisines. Ainsi, *neighbors.h* et *neighbors.c* s'occupent de ce point. Les deux fichiers implémentent diverses fonctions et structures permettant d'obtenir les voisins d'une case. En effet, en partant de l'index d'une case (son numéro sur le plateau) et d'une direction, la fonction *get_ neighbor* renvoie l'index de la case voisine dans ladite direction.
Néanmoins, il faut faire attention à deux choses, le plateau et ses limites. En effet, pour les limites, si l'on demande la case voisine à l'est de la case d'index 15, il suffit de renvoyer 15+1=16 sauf si la case 15 se trouve sur la dernière colonne et donc n'a pas de voisine à l'est. De plus, le type de plateau est important car pour les plateaux à cases triangulaires ou hexagonales, on considère que les voisines d'une case sont celles adjacentes. Ainsi, les cases hexagonales ont 6 voisines et les cases triangulaires en ont 3 :

- Les case hexagonales ont 6 voisines : par construction du plateau, elles n'ont pas de voisines au Nord ni au Sud.
- Les cases triangulaires ont 3 voisines : une à l'Est, une à l'Ouest et la dernière dépend de la ligne. Si la case est sur une ligne pair, c'est un triangle à l'endroit et donc la dernière case voisine est celle au Sud. Sinon elle est sur une ligne impair et donc c'est un triangle à l'envers. Sa dernière case voisine est alors au Nord.

Il est important de faire attention aux limites du plateau qui diffèrent du plateau classique à cases carrées. La fonction *get_ neighbor* renvoie la position UINT_MAX si la case dans la direction demandée n'existe pas. Finalement, on obtient des fonctions permettant d'obtenir les voisines d'une case, ce qui sera utile pour le déplacement des pièces.

3.2 Déplacements

Le déplacement des pièces est une composante importante du jeu et dispose de deux fichiers dédiés, *movement.c* et *movement.h*. Lors du déplacement d'une pièce, il faut dans un premier temps noter l'ensemble de ses déplacements possibles, ce qui est réalisé par la fonction *déplacement_ possible*. Cette fonction permet d'obtenir un tableau de taille *WORLD_ SIZE*. La case numéro n du tableau correspond à la case n du plateau, et un booléen indique si la pièce peut se déplacer ou non sur cette case. Une autre fonction permet de récupérer un tableau contenant uniquement les numéros des cases où le déplacement est possible. La fonction est divisée en fonctions auxiliaires correspondant aux différents types de pièces, que nous expliquerons

dans les sous-parties suivantes.
Puis, parmi ces déplacements possibles, l'un est choisi aléatoirement. Enfin, la fonction *move_piece* réalise le déplacement en modifiant le plateau : elle copie les informations *COLOR* et *SORT* de la case de départ sur la case d'arrivée puis supprime les informations de la case de départ (qui deviennet *NO_COLOR NO_SORT*).

3.2.1 Le pion

Le pion possède trois mouvements différents. La fonction auxiliaire *deplacement_possible_pawn* détermine ces trois déplacements possibles d'un pion. Pour le déplacement simple d'une case dans une direction, on fait appel aux fonctions de *neighbors.c* qui permettent directement d'obtenir les cases voisines du pion et donc celles cherchées. On doit vérifier si les cases voisines ne sont pas déjà occupées (sinon le pion pourrait écraser une autre pièce). Puis, pour chaque case voisine occupée, on vérifie si la case suivante dans la même direction est libre. Si oui, on peut y faire un saut simple. Enfin, pour chaque case obtenue par un saut simple, on vérifie si l'on peut faire d'autres saut simple. On doit faire attention à ne prendre en compte que les sauts simples qui n'ont pas encore été découverts. Pour cela on garde en mémoire les emplacements des saut simples dans un tableau et on vérifie si l'on n'a pas déjà réalisé le saut qu'on est en train d'étudier.

3.2.2 La tour

La tour est plus simple que le pion. La fonction *deplacement_possible_rook* retient en mémoire toutes les cases vides dans les directions cardinales jusqu'à atteindre une case occupée.

3.2.3 L'éléphant

Enfin, la fonction *deplacement_possible_elephant* regarde si les huit deplacements possibles pour ce type de pièce sont possibles, et garde en mémoire ceux qui le sont.

4 Initialisation du plateau et conditions de victoires

Nous avons décidé de créer les fichiers *board.c* et *board.h* pour y placer des fonctions en rapport avec la disposition du plateau. On y retrouve notamment 2 fonctions qui conditionnent le début et la fin de la partie. La première permet d'initialiser le plateau en plaçant les pièces aux positions de départs des joueurs. La seconde fonction est *victory* qui pour un joueur donné, renvoie le nombre de ses pièces se trouvant sur des positions de départ de l'adversaire. Ainsi, si cette fonction renvoie 0, le joueur n'a pas gagné, si *victory* renvoie au moins 1 il a donc au moins une victoire simple, et si *victory* renvoie le nombre de pièces possédées par chaque joueur, il a une victoire complexe.

5 L'aléatoire

La partie se déroule toute seule : l'ordinateur joue seul en déplaçant les pièces aléatoirement. C'est notamment pourquoi nous avons écrit les fichiers *random.c* et *random.h* qui contiennent les quelques fonctions dédiées à l'aléatoire. Plus précisément, une première fonction choisit quel joueur jouera en premier (blanc ou noir). Une seconde choisit une pièce à déplacer (ici on considère que si la pièce choisie ne peut pas être déplacée, le joueur passe son tour). Enfin la troisième et dernière fonction est celle qui choisi le déplacement effectué parmi ceux possibles.

L'aléatoire est géré par la fonction *srand* de *<random.h>* (à ne pas confondre avec notre fichier *"random.h"* qui est à inclure avec les guillemets et non les opérateurs "<" et ">") qui peut prendre un paramètre générant l'aléatoire. Ce paramètre peut être fixé lors de l'exécution du programme en ajoutant *-m* suivi d'un nombre entier à la ligne de commande. Par exemple : *./project -s 154*. Cette option est facultative.

La boucle de jeu que nous verrons plus tard fait appel à ces différentes fonctions afin de faire se dérouler la partie.

6 L'affichage

On dispose d'un plateau et de pièces pouvant être bougées sur celui-ci. Néanmoins il est intéressant d'afficher le plateau pour pouvoir observer le déroulement de la partie (et aussi débugger). Le fichier *show.c* s'occupe de cela et plus précisément la fonction *show_world*. Celle-ci parcourt le plateau world et récupère les informations contenues dans chacune des cases : les enums COLOR et SORT. Ensuite, elle les convertit en *string* grâce à la fonction *place_to_string* issue du fichier *geometry.c*.
show_world utilise une fonction auxiliaire pour chacun des plateaux disponibles (il faut indenter correctement chaque ligne et mettre les bons espaces).

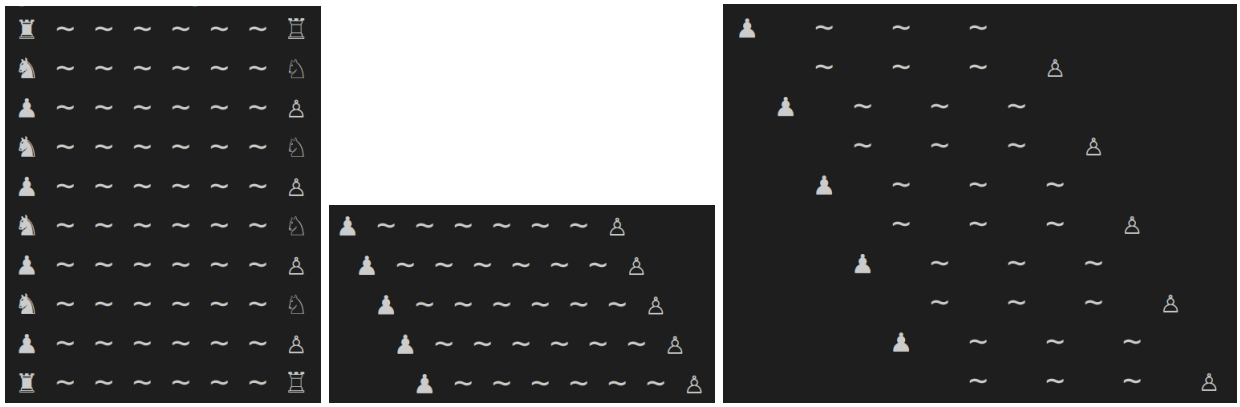

Figure 9: à gauche plateau carré | au milieu plateau hexagone | à droite plateau triangle

La figure 9 ci-dessus montre l'affichage des différents plateaux dans la console en début de partie. Pour les plateaux à cases carrées et hexagonales l'affichage est simple à comprendre, néanmoins pour le plateau triangulaire, l'affichage est plus difficilement compréhensible. Cela est dû à l'imbrication des lignes de triangles à l'endroit et à l'envers. Pour faire simple, on considère l'une des cases : si une autre case se situe juste en dessous, on peut y aller ainsi qu'à celle au dessus en diagonale, sinon une autre case se situe juste au dessus et on peut y aller ainsi qu'à celle en dessous en diagonale.

7 Boucle de jeu

7.1 L'initialisation de la partie

Pour que la partie ait lieu, il faut tout d'abord créer un monde et l'initialiser avec la fonction *world_init*. Il faut ensuite placer les différentes pièces à leur positions initiales avec *placement_initial* et *position_init*. Pour terminer, on choisit quel joueur a le trait en premier aléatoirement avec *get_random_player*.

7.2 Le déroulement de la partie

Le déroulement d'un tour correspond à l'action d'un joueur puis du suivant et ce, jusqu'à ce qu'un joueur gagne ou que le nombre maximal de tours fixé soit atteint. Nous avons donc choisi d'utiliser une boucle *while* qui ne se termine que lorsque l'une des conditions précédentes est atteinte. Plus précisément, chaque tour de boucle correspond à l'action d'un seul joueur. Ainsi, un tour réel de jeu correspond à deux tours de la boucle de jeu (la boucle *while*).
Dans la boucle, dans un premier temps il faut récupérer les informations du joueur qui joue (ses pièces et leur positions) que l'on stocke dans des variables adaptées. Puis on choisit aléatoirement l'une de ses pièces grâce aux fonctions du fichier *random.c*. Enfin, on choisit aléatoirement l'un des déplacements possibles pour cette pièce (s'il y en a un) et on l'effectue, en utilisant les fonctions dédiées du fichier *movement.c*. Enfin, on met à jour le compteur de tours en l'incrémentant de 1, les informations dans les variables du joueur et

la variable correspondant au joueur qui joue (de sorte qu'au prochain tour de boucle ce soit l'autre joueur qui joue).

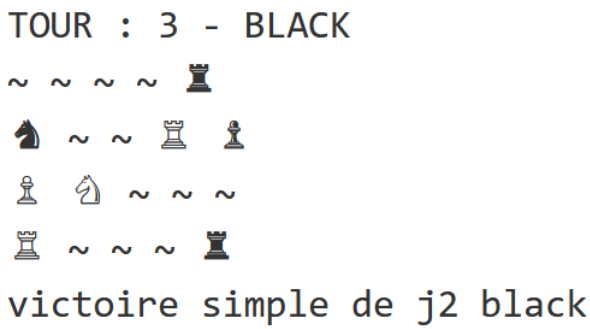

Figure 10: Victoire du joueur noir au troisième tour grâce à son éléphant sur un plateau à cases carrée de taille 4 * 5

8 Environnement

Ce projet à introduit divers outils et contraintes à gérer. Cela à demander un certain apprentissage de notre part mais surtout de l'adaptation.

8.1 Le travail en binôme

Ce projet a été réalisé en binôme. Il a donc fallu se répartir les tâches, communiquer, etc. Nous devons être honnêtes, dans un premier temps ce travail d'équipe ne fût pas simple. Le manque de communication, la répartition des tâches hasardeuse et mal pensée, les erreurs et manques de tests qui causent des erreurs dans des fonctions de l'autre binôme furent nombreuses. En effet nous avons eu beaucoup de difficultés à travailler ensemble. Néanmoins, après quelques semaines de travail, la cohésion du groupe a pu se renforcer et travailler à deux est devenu plus simple et plus efficace et ne relevait alors plus du tout du calvaire.
En général, nous nous sommes réparti l'écriture et les tests de chacune des parties importantes du codes (celles vues précédemment). Cependant, nous avons relu et corrigé les *quelques éventuels* bugs, problèmes et erreurs de segmentation rencontrés par l'un comme par l'autre. Cela nous a donc demandé d'avoir une compréhension globale du code assez pointue.

8.2 Makefile

Pour automatiser la compilation des fichiers, nous avons fait appel au langage Makefile. Celui-ci nous permet de n'avoir à taper qu'une seule ligne de commande pour compiler l'ensemble de notre projet. En effet, nous entrons *make* dans le terminal et le programme est compilé. Il est également possible de gérer la taille du plateau en ajoutant des paramètres. Par exemple, pour obtenir un plateau 5x5, nous écrivons *make HEIGHT=5 WIDTH=5*.

Le makefile requiert notamment que les dépendances lui soient définies pour pouvoir fonctionner. Celui-ci compile d'abord les différents fichiers **.o* puis fini par compiler le fichier *play.c*. Et on obtient finalement un exécutable du projet.

Le makefile permet aussi de compiler les fichiers de tests pour en obtenir un exécutable.

8.3 Utiliser git

Un nouvel outil est arrivé à notre disposition avec ce projet : git. Pour l'un comme pour l'autre, il s'agissait de la première fois que nous l'utilisions. Nous avons finalement réussi assez rapidement à nous y acclimater bien que nous ayons eu tous deux le même problème au moment de travailler sur nos ordinateurs personnels, ainsi que certains nombre de conflits à gérer au début.
Finalement, après plus de deux mois à l'utiser, git nous paraît indispensable pour travailler en groupe.

8.4 Le problème des dépendances et de l'organisation du code

Au long de ces deux mois de développement, l'organisation de notre code n'a cessé d'évoluer. Nous avons d'abord eu du mal à comprendre comment les fonctions s'articulent entre elles mais également comment fonctionnent les dépendances en C. Nous avions, dans un premier temps, inclus des fichiers **.c* dans d'autres fichiers **.c*. Suite à l'avertissement de l'un de nos professeurs encadrants, nous avions résolu le problème en regroupant tous nos fichiers secondaires dans le fichier *world.c*. Bien sur, ceci n'était pas viable, et c'est ce que nous avons compris suite à notre entretien avec notre professeur encadrant à la mi-projet. Nous avons donc réorganisés nos fonctions de manière bien plus logique, les regroupant en fonction de leur utilité et en créant les fichiers **.h* correspondants.

9 Les tests

S'il y a un aspect du projet que nous avons longtemps négligé, ce sont les tests. En effet, durant les premières semaines du projet, nous n'avons tout simplement pas fait de tests. Et ceux-ci nous ont manqué lorsque notre projet ne compilait pas alors que nous avions déjà fini la boucle de jeu. C'est à ce moment là que nous avons développé des tests pour pratiquement chacune de nos fonctions. Nos tests sont regroupés dans différents fichiers qui correspondent chacun à une partie du projet (comme les déplacements, le plateau etc) et un *main* qui les regroupe.
Pour ces tests, nous avons lancé les fonctions dans plusieurs cas définis dont nous connaissons le résultat et nous comptons le nombre de succès et d'échecs. Une seule fonction est testée par fonction de test. Cette fonction de test revoit le nombre de succès et le nombre de succès est affiché avec le fichier contenant le *main* des tests.
Ces tests sont compilés et lancés automatiquement avec le makefile. Pour cela, il faut taper *make test* dans le terminal pour qu'un exécutable test soit créé.

10 A propos de nos algorithmes

10.1 Terminaisons et corrections des algorithmes

Ce projet contient un certain nombre d'algorithmes très différents, s'occupant du fonctionnement des différentes parties du jeu comme les déplacements, le plateau... Néanmoins, il est intéressant d'étudier d'une part la terminaison de ces algorithmes et d'autre part, leur correction. La première correspond au fait qu'un algorithme termine peu importe les arguments valides en entrée. La seconde correspond au fait qu'un algorithme réalise effectivement ce qu'on veut qu'il fasse. Néanmoins au cours de ce projet nous avons en partie délaissé ces deux aspects, notamment par manque de temps et d'organisation. On peut donc se demander si ces algorithmes sont bel et bien fonctionnels. Pour ce qui concerne la correction, nous nous en sommes en réalité un peu occupé. En effet, nous avons testé les différents algorithmes pour au moins savoir s'ils réalisent ce que l'on veut dans des cas particuliers. Cependant, ils ne sont pas nécessairement corrects pour certaines entrées "non valides". En effet, nous avons écrit nos algorithme dans un contexte précis, en sachant quels arguments pourraient être pris en entrée et donc nous n'avons pas vérifié que les entrées respectaient les conditions fixées.Par exemple, la fonction *move_piece* permet de déplacer une pièce de la case i, jusqu'à la case j, mais elle ne vérifie pas que ces deux cases existent bien. Par conséquent, si l'algorithme qui calcule les positions de déplacement possibles pour une pièce renvoie des positions qui n'existent pas, *move_piece* peut modifier des éléments qui se trouve hors du plateau, quelque part dans la mémoire (ou alors l'executable

plante). Ici, le bon fonctionnement de *move_ piece* dépend du bon fonctionnement de la fonction précédente. Concernant la terminaison, nous n'avons utilisé que de l'itératif et les boucles *while* utilisées possèdent toutes des cas d'arrêt bien atteint.

10.2 Complexité des algorithmes

Un autre aspect intéressant à étudier est la complexité de nos algorithmes. Cela correspond aux ressources nécessaires (en temps et en espace) au fonctionnement de l'algorithme. Concernant la complexité temporelle, nos algorithmes sont globalement une succession de boucles *while* qui ont au plus *WORLD_ SIZE* tours de boucle, dans la boucle de jeu qui a au plus *MAX_ TOUR* tours de boucle. On obtient donc une complexité temporelle relativement correcte O(*WORLD_ SIZE*MAX_ TOUR*). De plus il est intéressant de noter que les statistiques du dépôt git exposent les complexités cyclomatiques de nos fichiers **.c*. Celle-ci correspond au nombre de chemins possibles au sein d'un algorithme. L'un des fichiers montre une complexité cyclomatique élevée : *neighbors.c*. Cela est en fait dû à la fonction *get_ neighbor* qui contient beaucoup de chemins possibles. En effet pour chaque type de plateau, et pour chaque direction possible dans ces plateaux elle calcule la case voisine d'une position dans une certaine direction. Enfin, la complexité spatiale de nos algorithmes se constitue de la structure *world_ t* représentant le plateau de *WORLD_ SIZE* cases et des quatre tableaux correspondant aux pièces des joueurs (deux pour chaque joueur, le premier étant les positions initiales, la position des pièces au fur et à mesure que la partie se déroule).

Conclusion

Ce projet avait pour but de coder un jeu de plateau tour par tour. Il fallait gérer la création et la gestion d'un plateau, ainsi que des pièces des joueurs.
Au travers de ce projet, nous avons mis en relief les nouveaux langages que nous avons découvert en ce début d'année. Le langage C est alors devenu beaucoup plus naturel qu'avant. Ce projet nous a également fait découvrir des utilitaires tels que *git* ou *makefile* ainsi que des fonctionnalités du c comme *getopt* qui nous serviront durant les prochains projets et années. Il nous a aussi fallu gérer un système de dépendances de nos fichiers. Enfin, il nous a également permis de nous organiser en équipe quand nous codions jusqu'alors seuls.
Au final, bien plus qu'un jeu de plateau, nous avons mis en place une nouvelle manière de travailler pour tous nos futurs travaux.