Rapport Projet S5-17132
Mansuba

Durand Arthur /—/ Hamouche Luxel

Responsable de projet : Renault David
Enseignant référent : Orhan Diane

Janvier 2023

Table des matières

Introduction

Notre projet se base sur l'exemple d'un *mansuba*, considéré comme l'ancêtre Perse des échecs dont le principe majeur est de mettre l'adversaire mat. Le but est donc d'implémenter tout un système de jeu et de relation afin de produire un jeu de plateau dans l'esprit du mansuba ou des échecs.

1.1 Présentation du projet

1.1.1 Règles

Le jeu est soumis à certaines règles afin d'encadrer sa création. Pour mieux comprendre les différentes implémentations, voici les principales :

- Le monde : Il est représenté par un ensemble de positions. Il est également définit par sa longueur et sa largeur qui en les multipliants nous donnent le nombre maximum de positions.
- Les relations : C'est ce qui relie les positions entre elles en fonction des directions. On peut de ce fait définir **les voisins** comme étant toutes les positions qui sont reliées entre elles. On a donc un maximum de voisins en fonction des déplacements autorisés.
- Déplacement simple : Il est définit comme étant le changement de position pour une position voisine disponible.
- Saut simple : Il est possible, si la position voisine n'est pas disponible, de sauter par dessus l'obstacle. Cependant si la position derrière est occupée ou que la structure du monde ne le permet pas, le saut est impossible.
- Victoire : Elle définit l'arret du jeu, soit car le nombre de tour maximum est atteint, soit car un joueur a réussi à atteindre les positions de départ de l'autre joueur avec une de ces pièces.

1.1.2 Stratégie de résolution

Notre stratégie pour effectuer ce projet est basée sur les tests ainsi que sur la modularité. En effet tout au long du projet nous avons fait en sorte que toutes les valeurs utilisées puissent varier en fonctions des demandes de l'utilisateur. On a donc utilisé le moins de constantes possible pour pouvoir répondre à des changements basiques sans rajouter ou changer tout une partie du projet.
Egalement, notre méthode pour implémenter de nouvelles fonctionalités est dite par les tests. Pour cela on fait une liste d'objectifs de la fonctionnalité, chaque objectif est testé avant de passer au suivant pour être sûr que la fonctionnalité marche correctement une fois implémentée.

Environnement et jeu

2.1 Représentation du monde

2.1.1 Géométrie du monde

La premiere chose à faire était de définir des constantes, stuctures et variables qui allaient nous servir tout au long du projet telles que :

Les directions :

```
1  enum dir_t {
2 NO_DIR  = 0,       //  Direction par défault (i.e unset)
3 EAST = 1, NEAST = 2, NORTH = 3,  NWEST= 4, WEST = -1, SWEST = -2, SOUTH = -3, SEAST = -4,
4 MAX_DIR = 9,       // Total des différentes directions
5 };
6
```

Le type de couleur de la case :

```
1  enum color_t {
2 NO_COLOR  = 0,  // Couleur initiale
3 BLACK     = 1,WHITE     = 2,
4 MAX_COLOR = 3,  // Total des differentes couleurs
5 };
```

L'occupant de la case :

```
1  enum sort_t {
2 NO_SORT  = 0,   // Pièce par défault (i.e nothing)
3 PAWN_SIMPLE = 1,
4 MAX_SORT = 2,   // Total des différentes pièces
5 };
```

Elles sont définies comme ceci afin de nous servir dans les différents algorythmes comme des constances et sont grandement utiles pour les calculs .
Ces catégories sont pour certaines voué à s'agrandir au vu de l'ajout des différentes pièces et joueurs.

Nous avons également eu besoin des constantes pour définir le monde :

```
1 #define HEIGHT 10                    // Largeur du monde
2 #define WIDTH 10                     // Longueur du monde
3 #define WORLD_SIZE (WIDTH*HEIGHT)    // Taille du monde
4 #define MAX_PLAYERS 10               // Nombre de joueur maximum
5 #define UNIT_MAX WORLD_SIZE          // Nombre maximum de cases
```

Avec ces variables, nous avons créés la structure "world" qui nous a permis d'assigner à chaque case du monde une couleur et un état d'occupation, ce qui va grandement nous servir dans la suite du projet.

2.1.2 Plateau de jeu

Il était nécessaire de définir un plateau de jeu pour l'utilisateur. Nous avons décidé de faire celui-ci de forme torique afin de rendre le jeu plus modulable. Il a donc fallu nous affranchir des effets de bord à l'aide de modulos. Ce choix a aussi été fait pour des raisons d'antcipation de futures modifications, qui seront plus faciles à implémenter en partant d'un modèle très généraliste tel que le tore (Figure **2.1**).

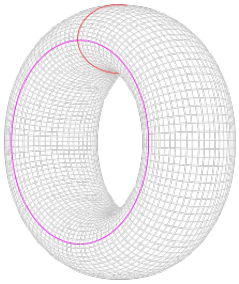
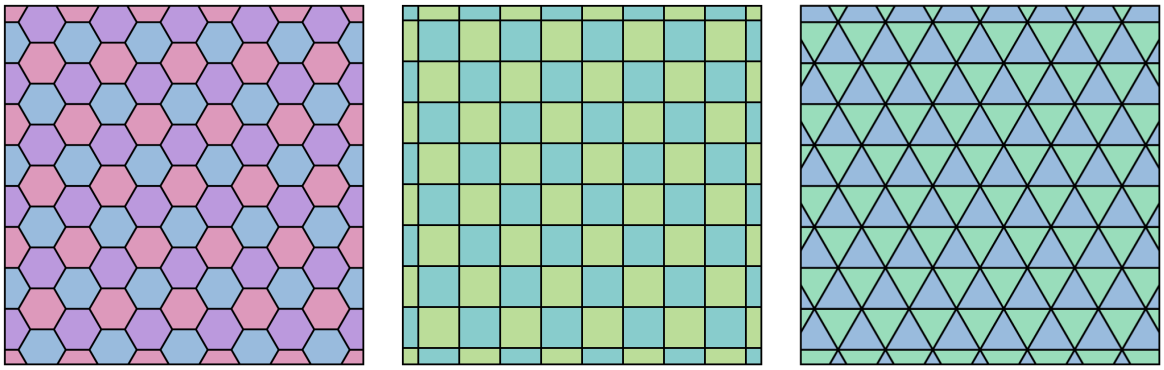


FIGURE 2.1 – Tore de jeu[1]

Par défaut, nous avons implémenté des relations entre cases de type hexagonal (Figure **2.2a**), comme il nous l'était imposé en début de projet. Cependant, dans le cadre d'un achievement nous avons également implémenté des changements de terrains qui impactent les relations faisant passer notre monde d'un pavage hexagonal, à un pavage carré (Figure **2.2b**) ou triagulaire (Figure **2.2c**). Ces changements interviennent au bout d'un certain nombre de tour, nombre qui peut être paramétré par le joueur.



(a) Pavage hexagonal[2] (b) Pavage carré[3] (c) Pavage triangulaire[4]

FIGURE 2.2 – Les différents pavages disponibles.

2.2 Pièces et joueurs

2.2.1 Pièces

Premièrement, nous avons choisi d'implémenter de simples pions, pouvant se déplacer d'une seule case (seulement si celle-ci est vide), pour des raisons de facilité principalement. Par la suite, nous avons incorporé au jeu de nouvelles pièces, le rendant plus intéressant. Chaque pièce à un déplacement particulier. C'est pour cela que nous avons créé une structure qui rend les pièces modulables. Chaque pièce appartenant à cette structure est définie par un certain nombre d'arguments.

```
1 /** A struct representing a piece */
2 struct pawns_t {
3     int player_index;        // Numéro du joueur propriétaire du pion
4     int max_dep;             // Nombres maximum de déplacements du pion
5     enum color_t color;      // Couleur du pion
6     enum sort_t type;        // Type du pion
7     int position;            // Position du pion
8     int captured;            // Etat du pion
9 };
```

1. Source : https://uk.wikipedia.org/wiki/%D0%9E%D0%B4%D0%BD%D0%BE%D0%B7%D0%B2'%D1%8F%D0%B7%D0%BD%D0%B0_%D0%BE%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D1%8C
2. Source : https://fr.wikipedia.org/wiki/Pavage_hexagonal
3. Source : https://fr.wikipedia.org/wiki/Pavage_carr%C3%A9
4. Source : https://fr.wikipedia.org/wiki/Pavage_triangulaire

Nous avons implémenté plusieurs types de piéces :

♟ Le Pion simpe :
Le pion est la pièce basique du jeu, il possède un seul mouvement dans la direction de son choix. Cependant il est possible d'en faire une dame d'échec en augmentant simplement son nombre de mouvements maximum afin qu'il puisse se déplacer sur de plus grandes distances.

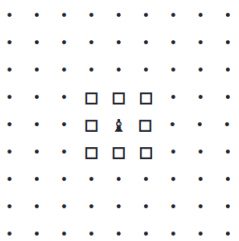


FIGURE 2.3 – Déplacement d'un pion

♜ La Tour :
Elle peut, comme aux échecs, se déplacer seulement en direction des points cardinaux. Nous avons décidé de limiter ses déplacements au Max(longueur du plateau, largeur du plateau). En effet, sans cette condition, étant donné l'apparence torique de notre plateau, ses movements pourraient être infinis.

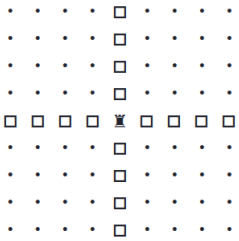


FIGURE 2.4 – Déplacement d'une tour

♞ L'éléphant :
Il se déplace uniquement suivant les 4 directions cardinales, dans le cas de base, il dispose de deux déplacements successifs, mais cette valeur peut être modifiée.

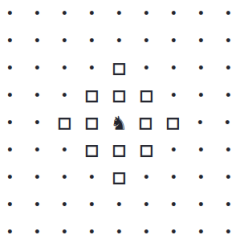


FIGURE 2.5 – Déplacement de l'éléphant

♚ Le Roi premier :
Il se téléporte directement sur les cases portant des numéros premiers.

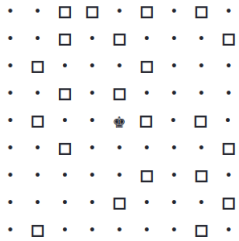


FIGURE 2.6 – Déplacement du Roi

2.2.2 Joueurs

Un minimum de deux joueurs est nécessaire pour lancer une partie. Chaque joueur est associé à un index unique. Ils possèdent aussi une couleur, un nombre de pièces ainsi qu'un tableau qui les contient. La structure se présente comme ceci :

```
1 /** A struct representing a player */
2 struct players_t {
3     int index;                          // Numéro du joueur
4     int pawns_nb;                       // Nombre de pièces
5     struct pawns_t pawns[WORLD_SIZE/2]; // Tableau des pièces du joueur
6     enum color_t color;                 // Couleur du joueur
7 };
```

Le jeu peut également être joué à plus de deux joueurs en fonction de la taille du monde. L'implémentation de joueur tient à respecter l'équilibre du jeu. Donc si le plateau ne permet pas de répartir un certain nombre de joueurs différents à l'initialisation alors cet équilibre est rompu.

2.2.3 Formations de départ

Pour plus de diversité, nous avons mis en place deux formations de départs différentes. Une première, classique, ressemblant aux échecs (mais toujours en prenant en considération la forme torique de notre plateau).

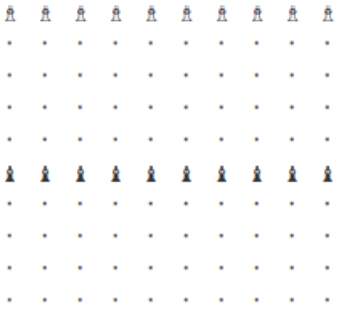


FIGURE 2.7 – Départ classique

La deuxième formation que nous avons implémentée est plus particulière et présente le plateau sous forme de *champs de bataille*, avec des énormes blocs de pièces séparés par des tranchées. Cette formation avait à l'origine pour but de tester la réaction des pièces lorsqu'elles sont entourées de pleins d'autres.

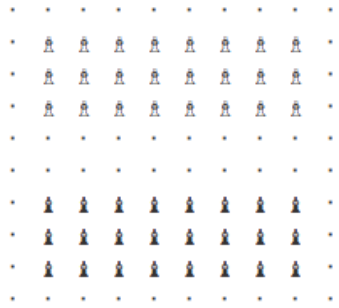


FIGURE 2.8 – Départ champ de bataille

Les types de pions qui composent ces formations sont modulables. Par défaut, la formation est remplie avec des pions simples. Néanmoins il est possible de changer cette pièce par défaut, ou encore d'ajouter un certain nombre de pièces "spéciales" en plus de la pièce par défaut.

2.3 Boucle de jeu

2.3.1 Sélection des options

Toujours dans le but d'augmenter la modularité de notre jeu, nous avons rendu un maximum d'options au choix du joueur lors de l'exécution. Les choix se font par l'intermédiaire de paramètres à ajouter lors de l'exécution du programme. Si les options sont écrites de manière incorrecte, le programme ne les prendra pas en compte et utilisera des valeurs par défaut.

2.3.2 Déroulement d'une partie

Lorsque le programme est lancé, le programme initialise un "monde extérieur", que nous verrons plus tard et qui contient le plateau de jeu, les joueurs et leurs pièces.

Au début de chaque tour, un contrôle sur le changement de terrain est effectué afin de le modifier (s'il est nécessaire, d'après les options de l'utilisateur).
Ensuite, une pièce du joueur est choisie au hasard pour se déplacer dans l'une des cases accessibles aléatoirement. Puis, on vérifie s'il y a un vainqueur ou si le max de tours a été atteint. Dans ce cas précis le jeu s'arrête et l'indique à l'utilisateur sinon on passe au tour suivant.

2.3.3 Conditions de victoire

Notre jeu admet deux conditions de victoire différentes :

— Le premier signe la fin du jeu dès lors qu'une pièce atteint les positions de départ de l'adversaire. Cette condition est dite "simple". Et permet de facilement vérifier les déplacements de nos pièces et le bon fonctionnement du jeu.

— La seconde nécessite que la totalité des pièces arrive dans les positions de départ de, ou des adversaires. Celle-ci est beaucoup plus complexe et nécessite des déplacements guidés pour l'atteindre. Cependant elle permet de faire durer plus longtemps les parties notamment en mode "champ de bataille" afin de voir beaucoup d'interactions. Cette condition est dite "complexe".

Architecture du projet

3.1 Relations

Notre jeu est basé sur les relations entre les différentes parties qui composent notre environnement de jeu.

3.1.1 Voisinage

Le voisinage d'une pièce correspond aux places du monde qui sont directement en contact avec la place de la pièce. Le nombre de voisins d'une pièce dépend donc de la géométrie de notre plateau (cf. **2.1.2**). On va déterminer ces derniers en testant l'intégralité des directions disponibles pour la pièce en n'omettant pas de vérifier si les cases sont occupées ou non. Cela nous donne donc un ensemble de cases disponibles pour cette dernière qui répond bien aux contraintes géométriques du plateau.

3.1.2 Déplacements

Nos déplacements sont réalisés en modifiant dans le monde l'état de la position ainsi que l'index associé à la pièce en question. Cela informe donc aux autres pièces que cette position est prise et permet de savoir à quel joueur elle appartient.

Pour déterminer la position exacte du déplacement, on utilise un fonctionnement de recherche de proche en proche, qui va déterminer les voisins directs, puis en fonction du nombre de mouvements de la pièce, réefectuer ce processus avec le voisin direct situé de la direction souhaitée. Le déplacement de cette dernière se fait ensuite aléatoirement parmi ces possibilités et comme dit précédemment, il est impossible pour cette dernière de choisir une case déjà occupée par une pièce alliée.
Nous avons choisi d'implémenter cette méthode de recherche de proche en proche car elle simplifie l'utilisation de variable de déplacement maximum pour chaque type de pièces.

Dans le cas où toutes les cases sont occupées par une pièce alliée, le joueur ne peut pas jouer et son tour est passé.

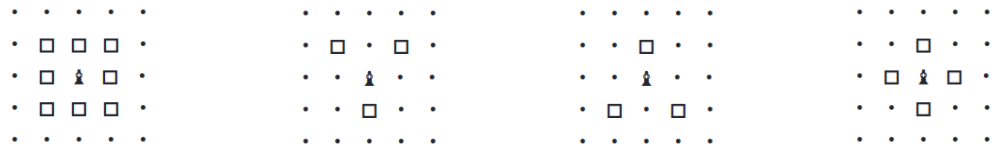
3.1.3 Changements de terrain

Dès lors que la partie commence, une initialisation de la *seed* du terrain est nécessaire.

Cette *seed* influe directement sur les interactions entre les différentes positions du monde et change donc les déplacements possibles.
Chaque pièce sera donc restreinte à un certain nombre de relations :

- Pour un terrain classique (hexagonal), toutes les directions sont autorisées.
- Pour un terrain à pavage triangulaire, une case sur deux possède les mêmes relations de voisinage, on a ici les relations (N,SE,SW) pour une partie des pièces et (S,NE,NW) pour l'autre.
- Pour un terrain à pavage carré, les seules directions autorisées sont celles des points cardinaux (N,S,E,W).

Après avoir réduit le nombre de directions possible, le système de déplacement est appliqué de manière identique, elle prend alors compte des contraintes de direction.



(a) Classique hexagonal (b) Triangulaire 1 (c) Triangulaire 2 (d) Carré

FIGURE 3.1 – Les différents déplacements en fonction des pavages.

Nous avons donc réussi à implémenter un système qui en modifiant une *seed* de terrain, modifie les relations de voisinage de la partie en cours.

3.1.4 Positions de départ

Etant donnés l'aspect torique de notre plateau, nous avons dû faire très attention aux déséquilibres possibles en début de la partie, et ceci beaucoup plus que pour un plateau classique où il aurait suffi de positionner les pièces de chaque côtés de celui-ci.

C'est pour résoudre ce problème que nous avons décidé d'utiliser un algorithme pour créer les formations en fonction des paramètres. Pour cela, nous avons implémenté un algorithme qui en fonction des types de pièces à placer et de leur nombre, choisit une formation qui soit identique pour chaque joueur et qui ne crée pas de déséquilibre entre eux, et ceci quel que soit le nombre de joueurs ou la taille du plateau de jeu. Par exemple, la figure **3.2** montre les positions de départ pour une partie lancée avec 4 joueurs, sur un monde de taille 10×10, avec 2 tours (pièce, cf. **2.2.1**) par joueur.

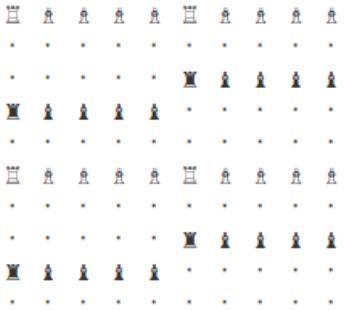


FIGURE 3.2 – Départ classique à 4

Cette facon de penser le départ du jeu nous a permis le rendre jeu encore plus modulable et de l'adapter à beaucoup plus de configurations en permettant des possibilités de départ presque infinies.

3.1.5 Capture et libération

Si une pièce atterit sur une pièce adverse, une capture est réalisée. Si plus tard, la position de la pièce capturée est libre, elle a une probabilité d'être libérée. Cette probabilité est fixée à `50/100` mais peut être modifiée par l'utilisateur. La question de la capture d'une pièce adverse par un joueur est une des raisons qui nous ont poussé à créer un "monde extérieur" (développé en **3.2.1**), qui sous forme d'une structre contient le plateau, les joueurs et les pions.

L'implémentation de cette structure nous a permis d'implémenter la capture de cette façon :

- Opérations sur la pièce :
  — Lorsqu'une pièce est capturée, sont attribut `captured` (cf. **2.2.1**) prend la valeur `1`.
- Opération sur le monde extérieur :
  — La pièce est ajoutée à la liste des pièces capturées : `captured_pawns[]`, et sa position est retirée de la liste des positions prises par le joueur : `current_sets[index_du_joueur][]`.

Procéder de cette manière nous a permis de ne pas modifier la position de la pièce capturée, ainsi, elle la garde en mémoire. Lorsque les conditions de libération sont remplies, on effectue le schéma inverse, et la pièce est de retour sur le plateau.

3.2 Organisation et structure du projet

3.2.1 Organisation du monde

Afin de faire évoluer notre jeu tout en gardant un aspect modulaire, nous avons dû créer un fichier "monde extérieur", `world_ext.c`, qui englobe la totalité des objets de notre jeu (monde, joueurs, pièces, ...). Ce fichier implémente la structure suivante.

```
1  struct world_ext_t {
2      struct world_t* world;                          // Plateau de jeu
3      int nb_players;                                 // Nombre de joueurs.
4      struct players_t players[WORLD_SIZE];           // Liste des joueurs.
5      struct sets_t initial_sets[WORLD_SIZE];         // Ensembles de positions de départ des
6                                                      // joueurs.
7      struct sets_t current_sets[WORLD_SIZE];         // Ensembles de positions prises par
8                                                      // chaque joueurs.
9      int nb_captured_pawns;                          // Nombre de pièces capturées.
10     struct pawns_t* captured_pawns[WORLD_SIZE];     // Liste des pièces capturées.
11 };
```

Le fichier `world_ext.c` met en lien l'ensemble des fichiers de base du jeu et fournit des fonctions *de haut niveau* à la boucle de jeu, comme par exemple une fonction pour gérer les captures et les libérations, ou encore pour effectuer un déplacement de pièces.

3.2.2 Dépendances des fichiers

Afin de rendre le projet le plus modulable possible, nous avons séparé notre projet en plusieurs fichiers `.c`, ces fichiers contiennent les fonctions qui permettent au tout de fonctionner. Chacun de ces fichiers `.c` inclu un fichier `.h` du même nom. Ces fichiers d'entête contiennent les `header` de toutes les fonctions "publiques" qui ont pour but d'être utilisées par d'autres fichiers. Les inclusions ne se font jamais entre les fichiers `.c`, mais uniquement avec les `.h`.

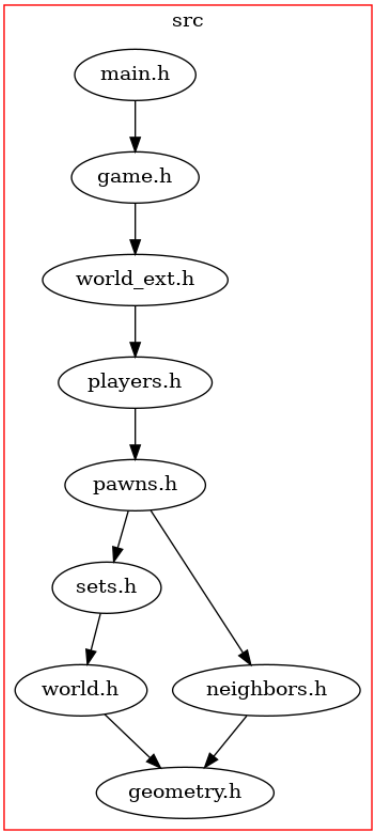


FIGURE 3.3 – Graphique de dépendance des fichiers source. *Généré avec* `Graphviz`.

Cette organisation du projet à base d'inclusions nous permet une très grande modularité. En effet, si un fichier `.c` est modifié (comme par exemple `world.c`), le projet continue de fonctionner tant que la nouvelle implémentation respecte le fichier `.h` correspondant.

3.2.3 Compilation

Pour faciliter les travaux de compilations séparés, nous avons intégré un fichier `Makefile`. Ce fichier nous a permis de déclarer des règles générales qui simplifient les commandes de compilations. Nous nous en somme par exemple servi pour compiler automatiquement tous les fichiers `.o` nécessaires, ou encore pour compiler et éxécuter tous les tests en même temps.

3.3 Tests

3.3.1 Structure des tests

Nous avons décidé de séparer les tests dans différents fichiers pour plus de flexibilité. Chaque fichier test contient les fonctions visants à tester un unique fichier source. Cette méthode nous a permis de pouvoir tester l'ensemble du code source en même temps, ou bien de lancer les tests d'un fichier en particulier.

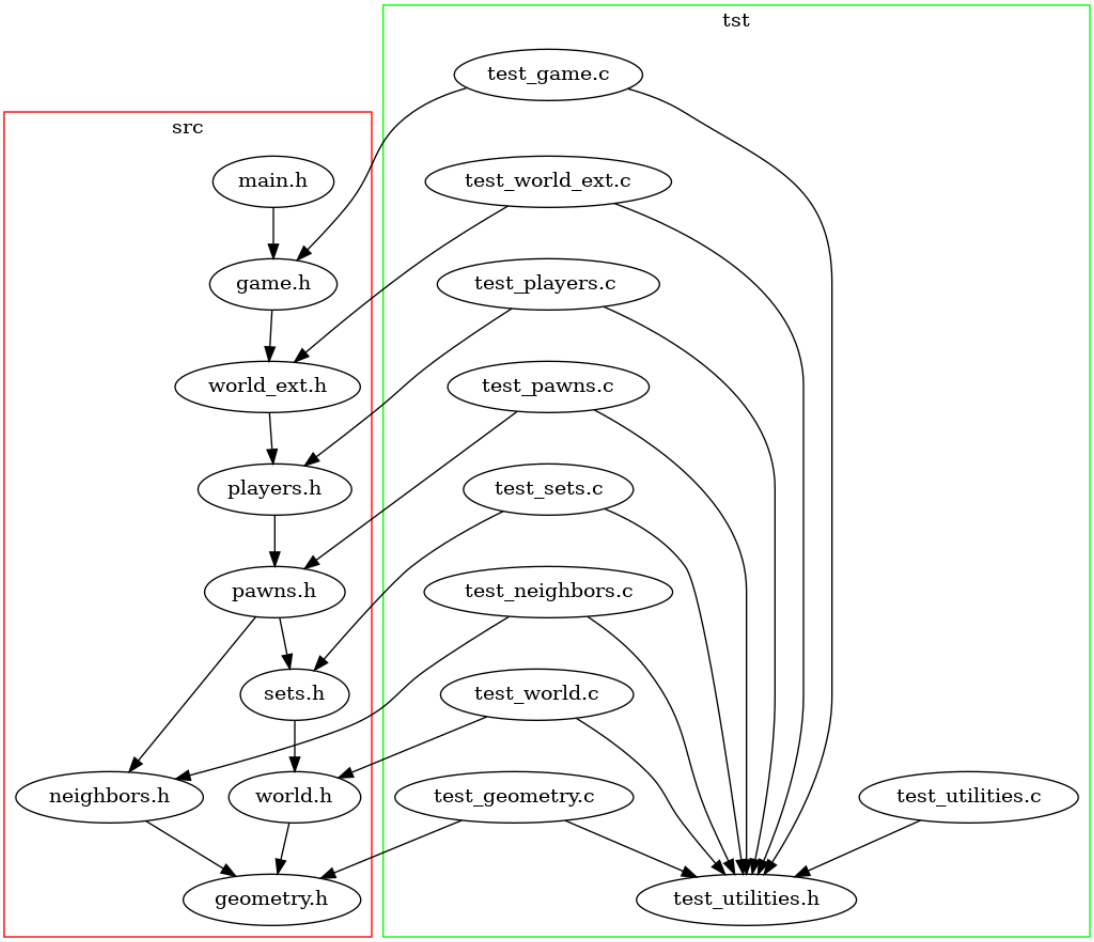


FIGURE 3.4 – Graphique de dépendance des fichiers tests. *Généré avec* `Graphviz`.

La figure **3.4** montre les inclusions de nos fichiers tests. On remarque que chaque fichier test inclu uniquement le fichier source qu'il teste. De plus, nous avons décidés de séparer deux fonctions générales des tests dans un fichier `test_utilities.c`. Ce fichier est le seul des fichiers tests à avoir un fichier d'entête, car c'est le seul qui est inclu dans d'autres fichiers. Il contient les deux fonctions suivantes :

```
1 void str_test(const char str1[], const char str2[]) // Compare 2 strings
2 {
3     (!strcmp(str1, str2)) ? printf("\t\tPASSED\n") : printf("\t\tRecieve %s instead of %s
      .\n", str1, str2);
4 }
5
6 void int_test(const int int1, const int int2) // Compare 2 integers
7 {
8     (int1 == int2) ? printf("\t\tPASSED\n") : printf("\t\tRecieve %d instead of %d.\n",
      int1, int2);
9 }
```

Ces deux fonctions nous ont été très utiles dans le cadre de la **programmation par le test**. En effet, appeler celles-ci dans nos fichiers test nous a permis de très facilement comparer les retours des fonctions testées avec ce que nous attendions. En cas de réussite, le programme affiche : `PASSED`, tandis que si le test ne passe pas, le programme affiche : `Recieve <valeur_reçu> instead of <valeur_attendue>`.

Nous avons donc pu écrire nos tests, puis implémenter nos fonctions jusqu'à ce que toute nos lignes de tests affichent : `PASSED`.

Conclusion

4.1 Difficultés rencontrées

Nous avons rencontré quelques difficultés lors de la réalisation de ce projet, en voici quelques-uns.

- Lors des premiers affichages il nous était impossible d'afficher le monde en raison d'un problème d'initialisation des structures. Nous passions par une variable globale ce qui empêchait de déclarer plusieurs variables de chaque structure.
- Nous avons également remarqué des problèmes en lien avec la taille de notre monde, en effet, pour des plateaux très très grands (de l'ordre de 10000 cases, et avec moitié moins de pions), notre jeu ne fonctionne plus pour ce qui nous semble être des raisons de gestion de mémoire. Ce problème est querellé avec le nombre maximum de joueurs. Nous pensons que ce problème pourrait être résolu en utilisant des `mallocs()` et des `free()`.

4.2 Bilan du Projet

Nous avons réussi à faire un jeu qui fonctionne de 2 à 10 joueurs (voir plus en fonction de la taille du monde), sur un plateau torique de 9 à 25 000 cases, avec 4 types de pièces différentes.
Le jeu admet 2 possibilités de départs et de victoires. Il comprend aussi des changements de terrain en cours de partie (avec 3 géométries disponibles) et un système de capture et de libération de pièce. La partie peut se jouer toute seule en choisisant aléatoirement les pièces et leurs déplacements. Le plateau est affiché tour par tour et affiche aussi à l'utilisateur les coups joués et des informations sur les événements du jeu.

Il nous resterait maintenant à implémenter des déplacements guidés pour permettre aux pièces de se déplacer en direction des objectifs de fin de partie.

4.3 Ce que le projet nous a apporté

Au cours de ces dernières semaines, nous avons appris à travailler ensemble sur un même code et à se corriger mutuellement. Ainsi, nous nous sommes amélioré en `C` et avons compris certaines suptilitées du language qui nous étaient encore inconnues. Le projet nous a forgé à l'utilisation d'outils comme `Git`, indispensable pour travailler en groupe de manière efficace, ou encore `Makefile`, qui facilite grandement les tâches répétitives des projets de développements. Ces compétences nous seront indispensables dans la suite de notre parcours.
Ce projet nous a également appris à produire un rendu qui respecte des demandes précises tout en ayant un code lisible et compréhensible.
Pour finir, l'autonomie laissée lors de ce projet nous aussi a permis de répondre au sujet de manière assez libre. Ceci nous a fait réfléchir à la manière de lier les différentes structures et données du projet pour créer par nous-mêmes notre architecture de jeu.