Algorithmique et Programmation 1
IMAC 1ere année

Projet

jeu stratégique

L'objectif de ce projet est de programmer un jeu de stratégie au tour par tour simplifié. Le projet est à effectuer en binômes. Le code est à rendre à la fin de semestre ; une soutenance orale de 15-20 minutes par binôme va suivre.

Description générale

Le but du projet est de programmer un jeu de stratégie/combat au tour par tour ente deux joueurs. Ceux-ci s'affrontent sur un plateau rectangulaire sur lequel sont disposés les unités des différents types, qui peuvent se déplacer, attaquer ou faire une autre action.

Lors du tour d'un joueur, ses unités remplissent leurs actions les unes après les autres. À la fin de son tour, le joueur doit indiquer la *fin de tour* pour rendre la main à son adversaire. Le jeu se termine quand l'un des joueurs perd toutes ses unités ou quand on décide d'arrêter la partie.

Le projet est à faire en plusieurs étapes. La première version du jeu communiquera avec les joueurs grâce à une interface texte. En deuxième phase, quelques améliorations, dont une interface graphique de base implémentée à l'aide de la bibliothèque `MLV` sera demandée. En troisième phase (optionelle, mais donnant accès aux meilleures notes), plusieurs améliorations additionnelles seront proposées.

Le rendu

Le projet est à effectuer en binômes. À la fin de semestre (la date sera précisée), un rendu sera à déposer. Ce rendu sera en forme d'une archive `.tar.gz` et contiendra **tous les fichiers source pertinents (bien commentés), un fichier makefile permettant de compiler le programme et un rapport en format `pdf` comme demandé ci-dessous.**

1 Le jeu

Deux joueurs (`rouge` et `bleu`) s'affrontent sur un plateau carré de dimension 12×18. Chaque joueur dispose de deux types d'unité : les `serfs` et les `guerriers`. Chacune de ces unités peut se déplacer sur une case adjacente à celle où elle est, à condition que cette case soit vide. Si la case est occupée par une unité du joueur adverse, il doit au préalable attaquer cette unité.

1.1 Mise en place

Au début du jeu, chaque joueur disposera d'un guerrier et deux serfs. Les joueurs commencent par placer ces unités avant le début du jeu.

1.2 Le plateau de jeu

Le plan est considéré comme un quadrillage de cases carrées, chaque case intérieure possède 8 voisins.

Les bords du plateau ne sont pas franchissables.

1.3 Combat

Un combat prend lieu quand une unité tente de se déplacer vers une case occupée par un ennemi. Dans ce cas, une des unités gagne (et reste dans sa case), l'autre meurt et est enlévé du jeu. Les règles sont simples : un *guerrier* gagne sur un *serf*. Si les deux unités sont du même type, c'est l'attaqueur qui gagne.

1.4 Fin de partie

La partie est terminée lorsque l'un des joueurs perd toutes ses unités, ou quand les joueurs décident de l'arrêter.

2 Implémentation

2.1 Interface texte

Le plateau sera représenté en console de la façon suivante :

```
---------------------------------
| ** | ** |    |    |  ...
--------------------------------
|    |    |    | ** | ...
-----------------------------
|    |    | ** |    | ...
-----------------------------
  ...
```

où les étoiles indiquent que la case est occupée par une unité. A la place de l'étoile, vous mettrez des symboles permettant de récapituler les informations de l'unité : son type, à quel joueur elle appartient, etc...

Il y aura également un message éxpliquant ce qui se passe :

2.2 Interface graphique

On demande également d'avoir une interface graphique minimale : chaque case sera colorée, chaque joueur aura une couleur et chaque unité une forme (par exemple, triangle pou un serf et rond pour un guerrier). L'interface est mise à jours à chaque déplacement.

2.3 Structures et représentations de données

Les unités seront réprésentés par une structure `unite` contenant toutes les informations pertinents à elle :

- Deux `int` pour stocker les coordonnées de la case sur laquelle se trovue l'unité.
- Un `char` valant `'g'` ou `'s'` selon que l'unité soit un guerrier ou un serf.
- Un `char` permettant de déterminer le joueur possédant l'unité.
- Un pointeur sur une `unite` permettant d'enchaîner toutes les unités de chaque joueur dans une liste.

Le jeu sera représenté par une structure `monde` contenant les informations suivantes :

- Un tableau de pointeurs sur `Unite` : chaque case du tableau d'indice (i, j) pointe sur l'unité présente sur cette case. Si aucune unité n'est présente, le pointeur correspondant vaut `NULL`.
- Un `int` permettant de compteur les tours effectués.
- Deux listes d'unités permettant de stocker les unités possédées par chacun des deux joueurs.

On obtient donc :

```
/* dimension du monde en nombre de cases */
#define LONG 12
#define LARG 18
/* l'origine est en haut a gauche */

#define ROUGE 'R' //identifiant du premier joueur
#define BLEU 'B' //identifiant du deuxième joueur

/* les genres d'unites */
#define SERF 's'
#define GUERRIER 'g'

typedef struct unite{
  int posX, posY;   /*pour stocker les coordonnées de l'unité*/
  char couleur;     /* ROUGE ou BLEU */
  char genre;       /* GUERRIER ou SERF*/
  struct unite *suiv;   /* liste des unités suivantes*/
} Unite;

typedef Unite* UListe;

typedef struct monde{
  Unite *plateau[LONG][LARG];
  int tour;   /* Numero du tour */
  UListe rouge, bleu; /*Listes des deux joueurs*/
} Monde;
```

En fonction des niveaux et améliorations réalisés, certains champs peuvent ne pas être utilisés ou bien ajoutés (e.g. pour être plus efficaces, les listes d'unités de chaque joueur

et occupant chaque case listes peuvent bien être implémentés en doublement chaîné). Ces modifications devraient être décrites dans le rapport du projet.

Néanmoins, en générale on demande de respecter le schéma d'implémentation.

3 Implantation du jeu

On demande de programmer le jeu par étapes. À la fin de chaque étape, sauvegardez votre code et incluez-le dans le rendu (en tout cas une version de jeu avec interface texte doit être inclue dans le rendu). Méfiez-vous de procéder avec une nouvelle étape tant que la précédente ne marche pas sans problèmes.

Même si vous vous sentez capables de programmer le jeu directement avec toutes les fonctionnalités et améliorations, on vous conseille de suivre la division en étapes (donc commencer avec la base et ajouter les fonctionnalités après). Cette approche risque d'être plus facile, plus adaptée à la partage des tâches. De plus, savoir produire du code qui peut facilement subir des modifications et savoir modifier du code existant sans le devoir complètement réecrire sont en soi des compétences utiles.

3.1 Niveau 1 – jeu de base

En première phase, une implémentation basique de jeu uniquement avec l'interface texte (comme décrite dans les sections 1 et 2 est demandée). Pour les aspects de jeu qui ne sont pas imposés pour le moment (e.g. ordre de jeu des unités pendant un tour), vous avez la liberté de choix. À cette étape là, on conseille de garder les affaires les plus simples que possible (e.g. les unités vont jouer dans l'ordre dans lequel il se trouvent dans la liste globale du joueur, sans que le joueur puisse le contrôler).

Si vous avez du mal à commencer, consultez l'appendix où l'on propose dans quel ordre attaquer l'implémentation du jeu de base.

3.2 Niveau 2 – couche graphique de base

Prenant le jeu créé dans la première phase (qui devrait fonctionner sans problèmes à ce point), implémentez une interface graphique de base comme decrite dans section 2. Le premier objectif est que le jeu et l'interface fonctionnent correctement. Ne gachez pas votre temps sur l'aspect visuel avant que la fonctionnalité soit là.

Si vous n'êtes pas du tout à l'aise avec la bibliothèque `MLV`, n'hésitez pas à faire le stricte minimum demandé et passer votre temps à améliorer le jeu de base. Consultation des exemples de code fournis avec la bibliothèque (par défaut dans `/usr/share/doc/mlv/examples`) et sa documentation sur son site web où accessibl par la commande `man` devrait suffire pour les fonctionnalités de base que l'on demande.

Pour plusieurs raisons, l'interface graphique doit être implémenté à l'aide de la bibliothèque `MLV`.

3.3 Niveau 3a – améliorations simples

(!!! Cette section sera encore légérement modifiée. On a commencé par travailler avec un énoncé plus complex que l'on a finalement décidé de simplifier. Les améliorations proposées ne sont pas encore à jour avec cette change, donc parfois font allusion aux choses que l'on

enlevé entre temps. Ils seront mises à jour au cours de quelques jours. En tout cas il s'agit seulement des propositions (pas d'impositions) et pour la plupart il s'agit d'améliorations assez naturelles.

Notre idée d'origine a été d'inclure dès le début :

— un système de points d'attaque/défense/vie dans le combat ;
— la possibilité pour plusieurs unités d'un joueur d'occuper la même case (dans ce cas, un case de plateau correspondra à une liste chaînée des unités y présentes ; chaque unité fera donc partie de deux listes : celle de toutes les unités de son maître et celle de toutes les unités occupants la même case)
Dans ce cas, on demande exceptionnellement de tenir à cette implémentation sauf si vous arriverez à trouver et défendre d'arguments assez forts contre elle.
— la notion de production des unités : une des unités, la *reine* peut recruter d'autres unités (cependant, la production prend quelques tours pendant lesquels la reine reste inactive)

Continuez à améliorer le jeu. Faites cépendant attention à tout bien tester (après tout, les améliorations doivent améliorer). Un code plein d'erreurs ne sera pas apprécié. Toutes les fonctionnalités ajoutées doivent bien être décrites dans le rapport.

Voilà quelques propositions d'améliorations de mechanique du jeu de base qui ne concernent que très peu l'interface et peuvent alors être implémentés avec le jeu dans le régime graphique aussi bien que texte. Elles ne devraient pas être trop difficiles à mettre en place. On s'attend que pour la plupart des projects, quelques améliorations de ce groupe (ou vos propres propositions de niveau similaire) soient implémentées.

— *(!!! seulement si vous êtes à l'aise avec les listes doublement chaînées)*
Les listes doublement chaînées offrent en générale une meilleure performance quand ils sont souvent modifiés. Ré-implémenter donc les listes en doublement chaîné, modifier le code en accordance.
— Si une case sous attaque est occupée par unités de plusieurs types, le défenseur sera choisi dans l'ordre de préférence suivant : 1/ guerrier, 2/ serviteur, 3/ reine (ordonner les listes pertinents est donc conseillé).
— À chaque tour, choisir à l'aléatoire le joueur qui joue en premier.
— Ajouter la possibilité de gérer les points de mouvement. Les serviteurs et les guerriers se pourront désormais déplacer de jusqu'à deux cases. Une attaque termine leur tour. Il peuvent se déplacer d'une case et puis attaquer. (Implémenter de telle manière que le nombre de points de mouvement de chaque unité puisse être facilement modifié.)
— Permettre les joueurs de déterminer l'ordre de jeu des unités de manière triviale. Une unité pourra désormais être commandé d'attendre. On continuera alors à donner les ordres aux toutes unités restantes et on revient vers les unités en attente une fois toutes les autres desservies.
— Permettre de sauvegarder une partie en cours dans un fichier et de la reprendre plus tard : Au début du tour, après le tirage aléatoire d'ordre de jeu, le clan jouant en premier peut choisir de sauvegarder la partie dans un fichier. La couleur devant agir, ainsi que tous le éléments, devront être mémorisés.
(Surtout ne pas inclure les adresses des zones de mémoire dans le fichier de sauvegarde!!! Les listes doivent être construites chaque fois à nouveau à la reprise d'une partie.)

3.4 Niveau 3b – améliorations plus avancées

On propose ici quelques améliorations qui risquent de prendre légérement plus de temps ou de reflexion, mais qui devraient rester faisables. N'hésitez-pas à les attaquer, mais ne sentez pas vous obligés si vous avez déjà trouvé la section précédente trop difficile. Mieux vaut un programme plus simple qui fonctionne qu'un tas de tout et n'importe quoi plein de bug.

Voilà quelques améliorations de mechanisme du jeu proposées :

— Considérer également les points de vie pour déterminer l'unité qui va défendre une case sous attaque. Sur le premier plan, on choisira toujours selon le type d'unité (guerrier > serviteur > reine). Parmi les unités du même type, on selectionnera toujours celle possèdant actuellement le plus de points de vie.

— Permettre de selectionner une destination plus éloigné : si une unité est demandé de se deplacer vers une case éloigné mais valide, cette case sera choisi comme sa destination. Pendant les tours à suivre, l'unité va s'automatiquement approcher de cette destination. Elle attendra de nouveaux ordres une fois arrivé ou une fois dérangé par d'unités ennemi sur une des cases voisines.

— Ajouter un élément aléatoire au combat (ne pas hésiter à modifier les statistiques et la mechanique du combat sous la condition de tout bien décrire dans le rapport).

— ...

Voilà quelques améliorations liées à l'interface graphique

— Ajouter des textures.

— Une tappe de souris sur une case (avec l'autre bouton que celui dédié au déplacement) affichera une liste de toutes les unités y présentes avec leurs nombres de points de vie et de points de mouvement restants.

— La liste des unités du point précédent permettra de selectionner la prochaine unité à recevoir ses ordres (supposant qu'elle appartient au joueur active). Bien réfléchir pour gérer l'ordre de jeu correctement (et efficacement).

— Avec cela en place, on n'a plus vraiment besoin de diviser chaque cellule en segments pour séparer les unités selon leur type. On peut donc tenter de retravailler l'interface pour devenir plus ergonomique, plus jolie.
(Attention : concevoir une bonne interface peut prendre beaucoup de réflexion et du temps, même si vous vous inspirez par ce qui existe déjà. Si vous êtes arrivés jusqu'à là, vous êtes invités à le tenter pour gagner d'expérience, mais c'est déjà au-délà de ce qui est attendu. A ce point-là, vous pourriez aussi regarder les bibliothèques graphiques plus standards tels que GTK et SDL).

4 Consignes générales

!!! TODO : répéter une fois de plus ce qu'il faut rendre comme versions du jeu

Le rapport

Le rapport (en format pdf) doit décrire les fonctionnalités du programme :

— ce qui a été implémenté et comment s'en servir (espèce de manuel utilisateur comprehensible même sans connaître l'énoncé du projet) ;

— ce qui n'a pas été implémenté (bugs connus, points omis) ;

— ce qui serait à améliorer ;

les détails d'implémentation :
— la structure du code : comment est-il reparti entre les modules (décrire d'abord de manière générale, sans trop rentrer dans du technique) ;
— les méthodes et les algorithmes utilisés (détailler surtout sur les parties du code dont la structure n'est pas imposée par l'énoncé) ;

et toutes les autres choses pertinentes au projet :
— la répartition des travaux entre les membres du binôme ;
— points notables dans le procédé des travaux, difficultés rencontrées...

Le rapport devrait permettre de se vite orienter dans le code, présentant son organisation et les idées derrière l'implémentation. Inutile, par contre, d'inclure les détails trop techniques (comme par exemple lister les prototypes de toutes les fonctions). On les pourra toujours trouver dans les fichiers en-tête bien commentés.

Appendix : pour bien démarrer

(Cette section est bien à jour et ne va plus changer sauf de corrections éventuelles.) Si vous avez du mal à démarrer, suivez le pas suivants :

1. Définir toutes les structures et types comme proposées ci-dessus.
2. Déclarer, dans le `main`, une variable de type `Monde`.
3. Définir une fonction `void initializerMonde(Monde * monde)` qui initialisera le `monde` passé par adresse (mettant la valeur de tous ses pointeurs, y compris les cases du plateau à `NULL` et le compteur de tours à zero). L'utiliser dans le main.
4. Définir une fonction `int creerUnite(char type, LUnite * unite)` qui alloue une nouvelle unité de type `type`, et stocke son adresse (qui est de type `LUnite`) dans `*unite`. (On passe donc une adresse par adresse : bien réflechir pour comprendre ce qui se passe !). La valeur de rétour sert à la gestion d'erreurs.
5. Définir une fonction `int placerAuMonde(Unite *unite, Monde *monde, int posX, int posY, char couleur)` qui place une unité qui vient d'être créée (et dont l'adresse reçue en paramètre) à la position souhaité dans le monde sous le contrôle de son nouveau maître. La valeur de retour sert à la gestion d'erreurs (la case désigné étant déjà occupée, par exemple).
   NB : Bien mettre à jour les champs de l'unité ainsi que le liste d'unités de son propriétaire et le plateau de jeu.
6. Définir une fonction `int affichePlateau(Monde monde)` qui affiche l'état actuel du plateau de jeu (en régime texte, comme demandé ci-dessus).
   Tester avec le monde vide, puis en ajoutant quelques unités.
7. Définir une fonction `void deplacerUnite(Unite *unite, Monde *monde, int destX, int destY)` qui déplace une unité vers une case spécifiée par les coordonnées (on suppose que celle-là existe et est vide). Tester.
8. Définir une fonction `enleverUnite(Unite *unite, Monde *monde` qui enlève l'unité, dont l'adresse passée en argument, de jeu.
   NB : Bien enlever l'unité du liste des unités de son maître, du plateau de jeu et finalement libérer la mémoire occupée.

9. Définir une fonction `int attaquer(Unite *unite, Monde *monde, int posX, int posY)` qui gère le combat. L'unité passée en argument attaquera la case spécifiée par `destX` et `destY` (on suppose qu'elle est occupée par un ennemi). Le résultat du combat est déterminé suivant les règles de jeu décrites ci-dessus. Si une des unités meurt (c'est toujours le cas avec les règles actuelles), elle est correctement enlevée du jeu). La fonction retourne 1 en cas de victoire, 0 en cas de perte de l'unité attaquante.
10. Définir une fonction `int deplacerOuAttaquer(Unite *unite, Monde *monde, int destX, int destY)` qui gère le déplacements et le combat. La valeur de retour sert à informer du résultat.
    — Si les coordonnées ne sont pas valides, la fonction ne fait rien et retourne -1.
    — Si la case marquée par les coordonnées n'est pas voisine à celle où l'unité en question se trouve, la fonction ne fait rien et retourne -2.
    — Si la case marquée par les coordonnées est déjà occupé par une unité alliée, la fonction ne fait rien et retourne -3.
    — Si la case destination est vide, valide et adjacente, l'unité se déplace, la fonction retourne 1.
    — Si la case de destination est valide, adjacente, et occupée par un ennemi, un combat prend lieu. La fonction retourne 2 si l'attaquant a gagné ou 3 s'il a perdu.
11. Définir une fonction `void gererDemiTour(char joueur, Monde *monde)` qui gère toutes les actions d'un joueur pendant un tour. Elle parcourt toutes les unités du joueur `joueur`. Pour chacune d'eux :
    — l'état actuel du plateau de jeu ansi que les informations concernant cette unité sont affichés ;
    — les ordres pour l'unité sont *clairement* demandées
    (direction de déplacement/attaque ou ne rien faire) ;
    — l'ordre est effectuée et l'utilisateur est informé du résultat
    (si l'ordre n'est pas valide – donc la fonction `deplacerOuAttaquer` renvoie une valeur négitive – l'utilisateur en est informé, mais ne reçoit pas l'opportunité de se corriger ; l'unité ne fera rien ce tour) ;

    Une fois toutes les ordres distribuées et effectuées, le joueur est demandé d'indiquer la fin de son tour, ce que l'on attendra avant de sortir de la fonction (*par exemple à l'aide d'une boucle avec `scanf(" %c", ...)`*).
12. Définir une fonction `void gererTour(Monde *monde)` qui gère un tour de jeu. Donc pour le moment qui lasse les deux joueurs jouer (toujours dans le même ordre) et qui met à jour le compteur des tours.
13. Définir une fonction `void viderMonde(Monde *monde)` qui va proprement vider le monde passé en argument. Toute la mémoire allouée pour les unités sera libérée et le monde, en tant que structure sera réinitialisé (nottament tous ses pointeurs remis à `NULL`.
14. Définir une fonction `void gererPartie(void)` qui sert à effectuer une partie de notre jeu. Elle :
    — prépare le plateau de jeu ;
    — positionne les unités initiales sur le plateau (créer une ou plusieurs fonctions pour garder le code propre) ;
    — laisse les joueurs prendre leurs tours tant qu'un un d'eux ne gagne ;
    — propose après chaque tour d'arrêter la partie sans gagnant ;

— à la fin de partie annonce son résultat et vide correctement le monde.