GRIMALDI Raphaëlle

SAIDI Narymane

NELVA-PASQUAL Mathilde

**RAPPORT PROJET INFO : CYBER-PATH**

Notre équipe est composée de trois membres : Raphaëlle Grimaldi, Narymane Saidi (Nina) et Mathilde Nelva-Pasqual. On a choisi le sujet Cyber-Path car c’est celui qui nous semblait le plus attrayant de manière générale. Il nous a intéressé notamment parce qu’il était interactif et assez ludique.

Le fonctionnement est assez simple, on se place sur une grille composée de murs, de cibles et de robots placés aléatoirement. A chaque tour un robot et une cible sont choisis au hasard par la machine. Chaque joueur doit ensuite choisir le nombre de coups minimum dont il pense avoir besoin pour atteindre la cible avec le robot. Une fois que tous les joueurs ont choisi leurs nombres de déplacement nécessaires, c’est celui ayant choisi la plus petite valeur qui commence à déplacer le robot. 1 point est alors gagné si ce joueur atteint sa cible avec le nombre de coups prévu, sinon tous les autres joueurs prennent 1 point. La difficulté est que les déplacements sont seulement des lignes droites et que le robot s’arrête seulement quand il bute sur un obstacle (robot, murs ou cibles), il faut donc bien réfléchir à ses choix !

Pour avancer ce projet on a surtout fait les grosses parties chez nous. On a mis le document partagé de telle sorte que nous y avons toutes eu accès. On a fait seulement une première réunion toutes les trois où on a fait la structure du code en réfléchissant aux fonctions dont nous aurons besoin. Puis on s’est réparti le travail de la manière suivante : Mathilde et Nina se sont plutôt occupées de la partie déplacement de robots et déroulement de la partie. Raphaëlle à elle de son côté un peu plus travailler toute la création de la grille et l’affichage.

Il arrivait que l’on bloque sur certains morceaux du code comme sur la modélisation des cases ou celles côte à côte avait des doubles murs car les deux cases étaient collées. On a dû dans ce cas précis construire une nouvelle fonction dans le code permettant d’éviter d’avoir des doublons pour les murs. Le fait de bien afficher les cibles dans les cases et que cela soit compréhensible à était aussi un peu compliqué au début.

Le concept de murs infranchissables nous a aussi un peu déstabilisé au début et nous ne savions pas tout de suite comment les modéliser.

Pour ces parties précises on a refait une dernière réunion ou on a pu terminer ces parties dures ensemble et le code était fini. De manière générale, nous avons était plutôt efficace sur la conception de ce projet surtout grâce à Raphaëlle. Elle a tout de suite eu une bonne vision globale du travail qu’on devait fournir et c’est plutôt elle qui a pris le rôle de meneur. Elle a dispatcher les tâches de manière à gagné en efficacité et ça nous a permis de bien travailler de manière méthodique est organisé.

**Bibliothèque :**

**A/ tile.c**

***- create\_tile() :*** Initialise une case avec tous ses bords à zéro, c’est à dire qu’elle n’a aucun mur. On initialise le robot à 64 et la cible à zéro, la case est donc initialiser vide, sans obstacles.

***- better\_tile(Tile \*a, int row, int column):*** Cette procédure formate les cases pour un affichage épurer, elle attribue des valeurs -1 quand le mur est déjà matérialiser par le mur de la case d'à côté, sans cette fonction on aurait des affichages comme ceci : Case A || Case B avec 2 murs côte à côte.

***- print\_colored(char \*text, int color):*** Procédure qui permet de faire d'afficher une chaîne de caractère en couleur.

***- print\_tile1(Tile a):*** *A*ffiche la première ligne de la case a, en vérifiant que la valeur en haut est bien 1, c’est à dire que l’emplacement est disponible. Elle affiche le haut de la case.

**- *print\_tile2(Tile a) :*** Elle affiche la deuxième ligne de la case ( côtés et milieu), on vérifie que la paroi à gauche vaut bien 1 pour pouvoir l’afficher, pareil pour celle de droite. On regarde aussi si il n’y a pas de robot ou de cibles déjà présents pour l’affichage de couleurs.

**- *print\_tile3(Tile a)* :** Affiche le bas d’une case, on vérifie aussi que l’emplacement est égale à 1 et est donc disponible.

**B/ map.c**

***- create\_map(Tile \*\*map, int size\_map):*** Création de la map, on parcourt avec deux boucles imbriquées de la taille du nombre de lignes et de colonnes et ont créée une case à chaque itération pour avoir la grille complète. On vérifie avec better-tile de pas superposer deux, bords de cases l’un sur l’autre. Puis on rajoute aussi avec des boucles for les bords manquants (contour de la grille).

**- *print\_row(Tile \*\*map,int row,int size\_map)* :** affiche la ligne row de la grille map de taille size\_map

***- print\_map(Tile \*\*map, int size\_map)*:** affiche toutes les lignes grâce à une boucle avec la fonction précédente.

**- *is\_wall(Tile \*\*map, int size\_map, int row, int column, int position)* :** Vérifie si il y a un mur, elle renvoie 1 si il y a un mur dans la case a la position sélectionnée dans les paramètres et 0 si ce n’est pas le cas. On vérifie pour tous les bords. Quand on vérifie le mur du haut on vérifie aussi le bas de la case du dessus. Pour le mur de gauche on vérifie aussi si la case de gauche a un mur en bas.

**- *add\_wall(Tile \*\*map, int size\_map, int row, int column, int position)*:** Elle ajoute un mur à la case souhaitée et dans la position souhaité (haut, droite, bas , gauche) = position(1,2,3,4). Ajuste aussi le mur de la case voisine.

***- is\_target\_valid(Tile \*\*map,int size\_map, int row, int column):*** Regarde si l'emplacement donnée est possible pour avoir une cible, retourne 0 (non valide) ou 1 (valide). Elle vérifie que les cibles ne soient pas en contact avec les murs aléatoires des bordures afin d'éviter tout problème. Ensuite on regarde si la case choisi ne contient pas déjà une cible. Puis si les cases voisines n’en contiennent pas non plus.

***- random\_wall(int \*array,int size\_map):*** On remplit un tableau avec des coordonnées aléatoires pour des murs, tout en vérifiant qu’il n’y est pas deux murs côte à côte qui serait inutilisables.

***- add\_map\_side\_walls(Tile \*\*map, int size\_map):*** Ajoute des murs sur les côtés. On prend le tableau des coordonnées du mur de chaque côté du plateau et on change les murs aléatoires pour chaque côté de la grille donc 4 fois. D’abord 2 murs du haut vers la gauche, puis du bas vers la gauche, de gauche vers le haut et enfin 2 murs de droite vers le haut.

***- add\_map\_target(Tile \*\*map, int size\_map):***Ajoute les cibles sur la grille, on génère d’abord des coordonnées de cibles valides et on les ajoute au plateau. Enfin, on crée deux murs autour de la cible : un autour de la cible aléatoire et le deuxième qu’on place dans la position suivante.

***- add\_map\_robot(Tile \*\*map, int size\_map, int array[4][2]):*** Ajoute les robots sur la grille, on génère d’abord des coordonnées de robots valides et on les ajoute au plateau.

**C/ game.c**

***- go\_up(Tile \*\*map, int size\_map,int array[4][2], int bot) :***Déplace un robot donné vers le haut tant qu’il n’y a pas de murs ou d’obstacles sur le chemin.

***- go\_right(Tile \*\*map, int size\_map,int array[4][2], int bot):*** Déplace un robot donné vers la droite tant qu’il n’y a pas de murs ou d’obstacles sur le chemin.

***- go\_down(Tile \*\*map, int size\_map,int array[4][2], int bot):*** Déplace un robot donné vers le bas tant qu’il n’y a pas de murs ou d’obstacles sur le chemin.

***- go\_left(Tile \*\*map, int size\_map,int array[4][2], int bot):*** Déplace un robot donné vers la gauche tant qu’il n’y a pas de murs ou d’obstacles sur le chemin.

***- print\_scores(int \*array, int size\_array):*** Affiche le tableau de scores. Il est compté sur 3 cas possibles : Si le joueur atteint sa cible dans le bon nombre d’essai gagne deux points (situation 1), si … (situation 2) ou … si … (situation 3).

***- print\_winner(int \*array\_score, int size\_array):***Affiche le gagnant, on parcours d’’abord le tableau pour trouver le maximum. On le parcours ensuite une deuxième fois pour remplir un tableau avec les indexes des gagnants. Enfin on affiche le ou les gagnants.

**D/ utils.c ( diverses fonctions utiles dans le programme)**

***- rdm(int mini, int maxi) :***Fonction qui renvoie un nombre aléatoire.

***- getint(int \*result) :*** Demande un entier à l’utilisateur et s’assure de la validité .

***- get\_positive\_int(int \*result) :*** Procédure qui demande un nombre strictement positif à l'utilisateur et s'assure de la validité de la réponse.

***- pick\_min\_array(int \*array,int size\_array,int \*player\_index) :***En cas de joueur ayant donné le même nombre de coups cette fonction en sélectionne un au hasard. On parcourt le tableau pour trouver le minimum, puis une deuxième fois pour récupérer tous les joueurs ayant donné ce nombre. Enfin on en choisit un aléatoirement parmi eux.