Ecole Polytechnique Universitaire
de Lille

Programmation Structurée

Rapport Projet Jeu de Cavalier

Corto Callerisa Sébastien Dardenne

January 10, 2019

1 Initialisation

Votre programme permettra d'initialiser une grille de taille 16x16 qui contient 1 chevalier, 8 obstacles,2 bonus et 1 cible.

Ici, on initialise une matrice 16x16 avec les éléments nécessaires pour le bon fonctionnement du jeu. C'est pourquoi on choisit de créer une fonction avec plusieurs boucles.
Problèmes:

- Le placement doit être aléatoire.
- Ne pas supprimer un élément déjà positionné.
- Pouvoir modifier facilement le nombre d'éléments.

Solutions: Dans un premier temps, nous faisons l'implémentation de la fonction *initialise_rand*() et *hazard*() qui permettent de générer des positions différentes à chaque lancement de programme. Puis dans la fonction de création de la grille *init_grid*(),on utilise des boucles "for" et "while" pour positionner chaque élément en vérifiant toujours que l'emplacement choisit aléatoirement est vide. Enfin pour faciliter la suite du programme, on passe en paramètre le nombre des différents éléments. Pour plus de modularité, les constantes sont implémentées par l'utilisation de la macro #define.

2 Affichage

On mettra en place l'affichage en prenant soin d'afficher la grille de manière stable et compréhensible. A chaque itération du jeu, il faut afficher le contenu des cases visibles et les numéros des cases dans lesquels le cavalier peut se déplacer.

Pour cette partie, nous affichons la grille en prenant en compte la vision du cavalier. On utilise donc une fonction qui permet l'affichage à chaque tour de jeu.
Problèmes:

- Ne pas cacher des éléments par les informations de déplacement (1,2,3...).
- Adapter la vision en fonction du bonus de vision du joueur.

Solutions: Dans la fonction *output_grid*(), on regarde les cases proches du cavalier, si la case est comprise dans la vision par rapport à la position du cavalier alors nous affichons les cases. Pour le tour suivant, on affiche les déplacements possibles en calculant le déplacement par rapport à la position du cavalier. Lors de l'affichage des numéros on vérifie qu'il n'y a rien dans la case sinon on affiche le contenu de celle-ci. Pour le reste de la grille qui est hors de la vision du cavalier nous affichons des carrés. Nous avons inclus une variable globale VISION_BONUS pour adapter la vision du cavalier en fonction du nombre de bonus acquis par le joueur. Enfin à chaque itération, nous effectuons un clear de la console. Pour la périodicité nous testons les dépassement et modifions les coordonnees de façon adéquate.

3 Diriger le Cavalier

Entre chaque itération du jeu, l'utilisateur doit saisir un chiffre entre 1-8 pour diriger le cavalier. S'il y a un obstacle dans la case visée, l'utilisateur gaspille un mouvement et reste dans sa position courante.

Ici, nous déplaçons le chevalier si possible et on applique les différents effets si nécessaire.
Problèmes:

- Appliquer l'effet de la case si la case lors du déplacement n'est pas une case vide.
- Déplacer le cavalier.

Solutions: Dans la fonction *move*(), on sauvegarde la position actuelle et la position future puis on regarde le contenu de la case de la position future donnée par ChoixToXY afin d'appliquer l'effet si nécessaire. Ensuite si le déplacement est effectué, on déplace le cavalier donc on supprime le cavalier se situant à la position actuelle et on ajoute le cavalier aux coordonnées de la position future. De plus la variable globale ITER compte le nombre de déplacements effectués par le joueur même si le déplacement est impossible à cause d'une case X, mais pas si le joueur essaie de traverser un mur dans la version classique.

4 Case Bonus / Malus

Il existe trois types de bonus: Bonus vision: Si le cavalier arrive sur la case d'un bonus, son champ de vision augmente d'une case (6x6, 7x7). Warp: Repositionne le cavalier aléatoirement. Duplicata: Multiplie le nombre de cibles par 2.

Ici nous ajoutons les cases pour les bonus/malus avec l'initialisation dans *init_grid*() et l'application des effets dans la fonction *move*().
Problèmes:

- Appliquer l'effet de la case et que la case soit utilisable qu'une seul fois si nécessaire.
- Modification de la grille en fonction des bonus appliqués.

Solutions: Dans un premier temps, nous choisissons de distinguer chaque bonus/malus par une case spécifique (W: warp, D: Duplicate, B: Bonus vision, M : Malus). Pour le duplicate et le bonus vision nous supprimons le bonus après utilisation par le joueur, au contraire la case du warp n'est pas supprimée après utilisation. L'application du bonus s'effectue dans la fonction *move*(). Lors d'une case bonus vision d'activée, la variable globale VISION_BONUS gagne +1. Pour la case warp, on déplace le cavalier aléatoirement dans une case vide en supprimant bien sûr la position de départ du cavalier.

5 Fin du Jeu

Le jeu se termine quand le cavalier arrive sur la case de la cible. Le nombre de mouvements est affiché comme le score du joueur.

Nous voulons donc que le jeu se termine au moment où le cavalier se déplace sur une case T en affichant le nombre d'itérations effectuées grâce à une variable globale.
Problèmes:

- Ajouter une itération même si le joueur effectue un déplacement impossible.

- Détecter le restart de la partie quand le joueur veux recommencer.

Solutions: A chaque itération, on vérifie si la case du futur déplacement est T, si c'est le cas le jeu se termine en affichant la variable ITER qui donne le nombre de coups fait par le joueur pour parvenir à la victoire. Cette variable globale augmente de 1 pour chaque mouvement fait par le joueur même si le mouvement est impossible (case X). A chaque tour le joueur à la possibilité de restart en tapant R, lors de ce restart la grille est réinitialisée ainsi que les variables globales. Le joueur peut alors changer de mode et de difficulté. Pour faciliter la correction, on peut aussi entrer 'D' pour dévoiler/cacher la grille.

6 Une variante périodique

On proposera une variante de l'implemention mise en place permettant d'utiliser la grille de maniere periodique

Nous voulons que le cavalier puisse se déplacer au travers de la grille à la façon d'un tore.
Problèmes:

- Il faut savoir les cases à afficher même si elles sont plus loins que 2 cases du cavalier.
- Afficher correctement les numéros
- Gerer l'affichage des coins
- Deplacer correctement le cavalier

Solutions: On découple le traitement de la vision de la boucle principale d'affichage. On enregistre l'information de visions dans un autre tableau, cases_vision, ainsi que les numéros correspondant aux déplacements. On intègre le traitement des coins en faisant une opération modulo sur la hauteur ainsi que différents tests. Ainsi la fonction d'affichage dépend de la grille de jeu mais aussi de la surcouche vision. Le déplacement du cavalier est fait dans la fonction move(), par la sous fonction perdiodize(int x, int y) qui utilise un algorithme similaire à celui de la vision.

7 Difficulté du jeu

On fera ici en sorte de permettre à l'utilisateur de donner un pourcentage de difficulté, en fonction duquel la grille est rempli du nombre de bonus et d'obstacles, mis 'a des positions aléatoires

Problèmes:

- Il faut que les bonus s'augmentent indépendamment des malus et inversement

Solutions: Le joueur choisit une difficulté comprise entre 0 et 10, 5 étant la difficulté standart avec 2 bonus et 8 obstacles. De 0 à 4 on multiplie le nombre de bonus par 5 - diff et de 6 à 10 on multiplie le nombre d'obstacles par diff - 5.

8 Bonus : Malus

Vous pouvez imaginer et ajouter des malus. A chaque tour les malus peuvent se rapprocher d'une case en direction du cavalier.

On implémente un malus ('M') qui se cache parmi les bonus. Si le joueur tombe dessus la phrase "/!\Ce BONUS était un MALUS !" s'affiche à droite de la grille sur la ligne ou se trouve désormais le cavalier. La grille possède alors désormais un 'T' piégée qui fait perdre le joueur s'il tombe dessus, mais elle ne se déplace pas pour lui laisser une chance.
Nous avons choisi le camouflage plutôt que le rapprochement en utilisant un algorithme similaire à celui de la cible mobile par choix d'implémentation.

9 Bonus : Cible mobile

On fera ici de sorte que le cible bouge aléatoirement d'une case à chaque itération du jeu (horizontalement ou verticalement).

On fait bouger la cible dans une direction aléatoire et si ce n'est possible on regarde la position suivante dans l'ordre : haut, droite, bas, gauche. Par exemple si on a tiré bas et que la case est remplie on teste à gauche. Seule la cible originale est mobile et non les duplicate ni le piège.