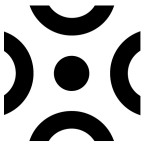
Perfectionnement à la programmation en C

Le jeu du serpent

L2 Informatique - 2024–2025

Fiche de TP 5



L'objectif de ce TP est de programmer le très célèbre *jeu du serpent*. Nous utilisons ici les notions de

- liste simplement chaînée ;
- comparaison entre représentation creuse et représentation dense ;
- interface ncurses en gestion en temps réel ;
- compilation paramétrée par fichier `Makefile`.

1 Spécification

Le principe du jeu consiste à de piloter un serpent vu du dessus pour attraper des pommes et grandir progressivement. La difficulté principale consiste en ce que le serpent est perpétuellement en mouvement et le contact avec les bords de l'écran ou avec lui-même provoque la défaite.

Voici une description de l'application à concevoir. Au lancement du jeu, un quadrillage régulier (formant 10 lignes et 16 colonnes) est affiché et trois pommes y sont disséminées aléatoirement. De plus, un serpent de petite taille figure au centre du quadrillage. Il est constitué d'une tête (qui occupe un carré du quadrillage) et d'un corps (qui occupe plusieurs carrés, un seul initialement). Les pommes sont dessinées par des disques rouges, la tête est dessinée par un disque kaki et son corps par des carrés marrons. Une illustration d'une situation de jeu est donnée par la figure 1.

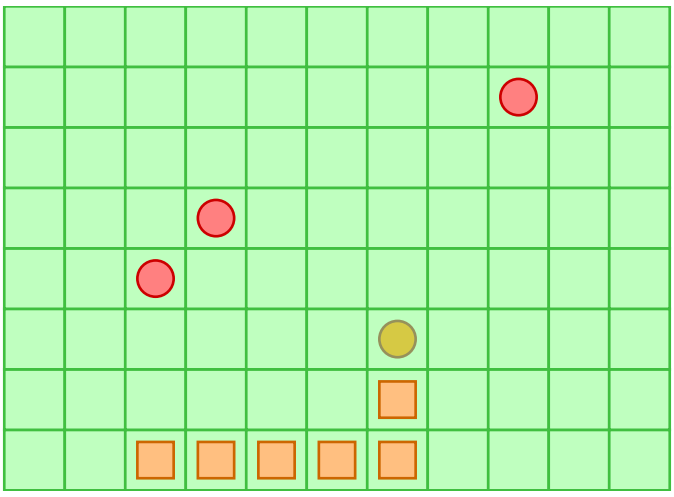


Figure 1: Une situation de jeu du serpent qui respecte les conventions graphiques établies. Le quadrillage est de dimensions 8×11.

Ces conventions de couleurs sont importantes pour la lecture des figures du sujet mais ne sont pas obligatoires dans l'implantation qui va suivre.

Le serpent se déplace à une vitesse de quatre cases par seconde selon une direction (nord, sud, est ou ouest) et l'utilisateur le pilote à l'aide des flèches directionnelles (ou `'z'`, `'q'`, `'s'`, `'d'`) pour modifier sa direction. À chaque quart de seconde, la tête du serpent se déplace vers la case indiquée par sa direction et la dernière case de son corps disparaît. La figure 2 illustre ce mécanisme de déplacement.

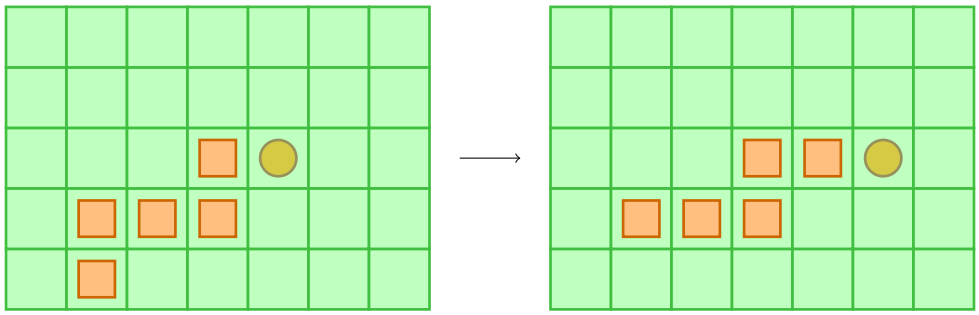


Figure 2: Le déplacement d'un serpent d'une case à une autre. Le serpent se dirige vers l'est. Le 1^{er} est la situation à un temps t et le 2^{e}, au temps $t+1$

Lorsque le serpent arrive sur une case qui contient une pomme, celle-ci disparaît et le corps du serpent grandit en évitant de faire disparaître la dernière case de son corps lors du déplacement. La figure 3 illustre ce mécanisme de croissance.

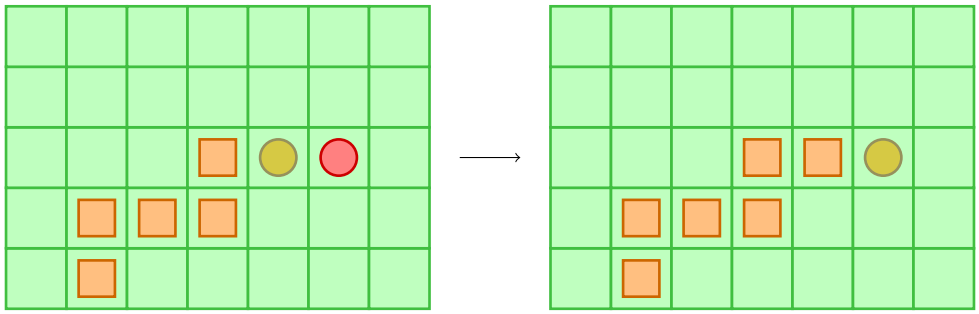


Figure 3: La croissance d'un serpent provoquée par la consommation d'une pomme. Le serpent se dirige vers l'est. Le 1^{er} quadrillage est la situation à un temps t et le 2^{e}, au temps $t+1$.

Lorsque le serpent arrive sur une case occupée par son corps ou bien s'il sort du quadrillage, la partie se termine et le score (qui est le nombre de pommes mangées) est affiché. La partie s'achève aussi lorsque le serpent vient de manger une pomme et qu'il est aussi grand qu'il n'y a plus de case libre pour créer une nouvelle pomme.

2 Écriture du programme

Exercice 1 (Conception du projet)

L'objectif de cet exercice est de concevoir une architecture viable pour le projet présenté.

1. Lire attentivement **l'intégralité du sujet** avant de commencer à répondre aux questions. La description du sujet constitue une spécification de projet.
2. Une fois ceci fait, proposer un découpage en modules cohérent du projet. Pour chaque module proposé, décrire les types qu'il apporte ainsi que ses objectifs principaux.
3. Au fil de l'écriture du projet (dans le travail demandé par les prochains exercices), il est possible de se rendre compte que le découpage initialement prévu n'est pas complet ou adapté. Si c'est le cas, mentionner l'historique de ses modifications.
4. Maintenir un `Makefile` pour compiler le projet.

Exercice 2 (Déclarations des types)

Les types suivants sont obligatoires (à déclarer et à utiliser). Il est possible (et même obligatoire en pratique) de déclarer des types supplémentaires qui ne sont pas mentionnés ici. Il n'est pas autorisé pour le moment ici d'utiliser des tableaux dynamiques. Seuls des tableaux statiques doivent être utilisés.

1. Déclarer un type `Case` qui permet de représenter une case d'un quadrillage. Un case est simplement une coordonnée dans le quadrillage.

2. Déclarer un type `Direction` qui permet de représenter les quatre directions cardinales. Il est pertinent ici d'utiliser un type énuméré.

3. Déclarer un type `Serpent` qui permet de représenter un serpent. Il s'agit d'un type structuré contenant un champ renseignant sur la direction du serpent et un champ renseignant sur les cases que ce dernier occupe. Ce deuxième champ est une liste simplement chaînée de cases où la tête de la liste est la case de la tête du serpent et le reste contient — dans l'ordre — les cases de son corps. Par exemple, le corps du serpent de gauche de la figure 3 est codé par la liste

   ```
   [(2, 4), (2, 3), (3, 3), (3, 2), (3, 1), (4, 1)]
   ```

   Ici l'origine figure en haut à gauche et les cases sont indexées par ordonnée puis abscisse (conformément à la convention adoptée par ncurses) mais rien n'est imposé là dessus dans ce TP.

   Pour représenter proprement de telles listes de cases, il est conseillé de déclarer des types permettant de manipuler des listes de cases dans le module dédié à la représentation des cases.

4. Déclarer un type `Pomme`. Une pomme est codée par la case qui la contient.

5. Déclarer un type `Monde` qui permet de représenter une situation de jeu. Elle comprend les dimensions du monde (nombre de lignes et nombre de colonnes), le serpent, les positions des pommes présentes et le nombre de pommes mangées. Les pommes du monde sont enregistrées dans une liste simplement chaînée. Ici aussi, il est pertinent de déclarer des types permettant de manipuler des listes de pommes dans le module dédié à la représentation des pommes.

Remarque 1 Dans le TP du *Chomp*, la situation de jeu était codée par un tableau à deux dimensions dont chaque case contenait un état (carré de chocolat présent ou absent). Dans le présent TP, la situation de jeu n'est pas codée globalement par un tableau : on renseigne plutôt les objets qui évoluent dessus (le serpent et les pommes). Ainsi, il est possible de considérer des mondes très grands, de dimensions 1024×1024 par exemple, puisque la taille mémoire ne dépend pas de ces dimensions (qui aboutirait tout de même à 2^{20} cases) mais plutôt de la taille du serpent et du nombre de pommes. On parle ici de **représentation creuse**, contrairement à la **représentation dense** adoptée pour le *Chomp*. Aucune n'est fondamentalement meilleure que l'autre : chacune a ses avantages et ses inconvénients.

Exercice 3 (Mécaniques du jeu)

Les fonctions suivantes sont obligatoires (à déclarer/définir et à utiliser). Il est possible (et même nécessaire) de déclarer/définir des fonctions qui ne sont pas mentionnées ici.

1. Écrire une fonction `Pomme pomme_aleatoire(int nb_lignes, int nb_colonnes)` qui renvoie une pomme créée aléatoirement de sorte qu'elle tienne dans un quadrillage de dimensions `nb_lignes` × `nb_colonnes`.

   Se référer à l'exemple de programme donné dans l'exercice 7 du TP 2 pour réaliser une génération aléatoire correcte. Ces notions de génération aléatoire seront vues en détail dans la fin du cours.

2. Écrire une fonction `void monde_ajouter_pomme(Monde *mon)` qui ajoute une nouvelle pomme au monde `mon`. Cette fonction doit utiliser la fonction précédente. Attention, il faut veiller à ce que la nouvelle pomme n'apparaisse pas sur une pomme déjà existante ou bien sur le serpent. Il faut donc appeler la fonction précédente `pomme_aleatoire` jusqu'à ce qu'elle renvoie une pomme sur une case valide. On suppose qu'une telle case existe.

3. Écrire une fonction

   `Serpent serpent_initialiser(int nb_lignes, int nb_colonnes, int taille)`

   qui renvoie un serpent dont la tête est située au centre d'un quadrillage de dimensions `nb_lignes` × `nb_colonnes` (ou dans l'une des cases centrales suivant la parité des dimensions). Le serpent doit tenir sur `taille` cases en comptant sa tête et son corps. Les cases du corps du serpent sont à gauche de sa tête et la direction du serpent est vers l'est.

4. Écrire une fonction `Case serpent_case_visee(Serpent serp)` qui renvoie la case visée par le serpent `serp`. Il s'agit de la case que la tête de `serp` va occuper au prochain déplacement. Il se peut que cette case figure en dehors du quadrillage (elle peut même avoir des coordonnées strictement négative si le serpent est en haut du quadrillage et possède le nord comme direction, ou est à gauche du quadrillage et possède l'ouest comme direction).

5. Écrire une fonction

   ```
   Monde monde_initialiser(int nb_lignes, int nb_colonnes,
                           int taille_serpent, int nb_pommes)
   ```

   qui renvoie un monde dans une configuration initiale pour le jeu. Le quadrillage du monde est de dimensions `nb_lignes` × `nb_colonnes`, le serpent est de taille initiale `taille_serpent` et le monde contient à chaque instant `nb_pommes`. Les pommes y sont disposées aléatoirement. Le serpent se trouve dans sa configuration initiale fournie par la fonction `serpent_initialiser`.

6. Écrire une fonction `int monde_est_mort_serpent(Monde mon)` qui teste si le serpent du monde `mon` va mourir s'il se déplace sur la case qu'il vise. Le serpent meurt s'il vise une case à l'extérieur du quadrillage ou bien une case de son corps.

7. Écrire une fonction `void monde_evoluer_serpent(Monde *mon)` qui modifie le serpent du monde `mon` de sorte à le faire avancer suivant sa direction vers la prochaine case. Si la tête du serpent arrive sur une case vide du quadrillage, le serpent s'y déplace simplement. S'il arrive sur une case occupée par une pomme, celle-ci est mangée, le serpent grandit en conséquence, et une nouvelle pomme créée aléatoirement est ajoutée à `mon`.

Remarque 2 les prototypes des fonctions demandées ici ne sont pas optimaux, notamment en ce qui concerne le passage par valeur des variables d'un type structuré. Nous adoptons ces choix ici, pour le moment, par souci de simplicité.

Exercice 4 (Graphisme)

Les fonctions suivantes sont obligatoires (à déclarer/définir et à utiliser). Il est possible (et même nécessaire) de déclarer/définir des fonctions qui ne sont pas mentionnées ici.

Toutes les fonctions d'affichage suivantes utilisent la bibliothèque graphique ncurses et agissent sur la fenêtre courante. Les conventions d'affichage sont libres : il est possible de faire tenir chaque case du monde sur une case de la fenêtre ou, à l'inverse, dessiner chaque case du monde par des carrés de côté 3 — par exemple — de la fenêtre. Toute liberté est autorisée à condition de proposer un affichage efficace, précis et minimaliste.

1. Écrire une fonction `void interface_afficher_quadrillage(Monde mon)` qui affiche le quadrillage sous-jacent au monde `mon`. Pour le moment, le serpent et les pommes ne sont pas affichées.

2. Écrire une fonction `void interface_afficher_pomme(Pomme pom)` qui affiche la pomme `pom` selon sa position. Cette fonction est vouée à dessiner la pomme dans le quadrillage créé par la fonction précédente.

3. Écrire une fonction `void interface_afficher_serpent(Serpent ser)` qui affiche le serpent `ser` selon sa position. Cette fonction est vouée à dessiner le serpent dans le quadrillage créé par la fonction de la 1 question.

4. Écrire une fonction `void interface_afficher_monde(Monde mon)` qui affiche le monde `mon`. Ceci dessine la situation de jeu au complet : le quadrillage, les pommes, le serpent et le nombre de pommes mangées.

5. Écrire une fonction `void interface_piloter(Monde *mon)` qui détecte la pression sur une touche commandant la direction du serpent du monde pointé par `mon` et qui la met à jour.

Exercice 5 (Assemblage du jeu)

Nous allons utiliser tout ce qui a été fait jusqu'à présent pour construire la boucle d'interaction principale qui permet de jouer au jeu du serpent. L'algorithme suivant est mis en pratique :

1. initialiser le monde ;
2. afficher le monde et attendre qu'une touche soit pressée pour commencer ;
3. tant que la partie n'est pas terminée (c.-à.-d. que le serpent n'est pas mort) :
    1. faire évoluer le serpent en fonction de sa direction :
    2. mettre à jour l'affichage (serpent, pommes et score) ;
    3. attendre $\frac{1}{4}$ de seconde ;
    4. mettre à jour la direction du serpent en fonction de la touche pressée ;
4. afficher que la partie est terminée et le score final.

---

3 Améliorations

Exercice 6 (Améliorations)

Plusieurs améliorations sont proposées ici. Elles sont accompagnées d'indices de difficulté ((⋆) : facile, (⋆⋆) : moyen, (⋆ ⋆ ⋆) : assez difficile).

Choisir dans cette liste des améliorations à apporter au programme de sorte à totaliser au moins **sept** ⋆.

Conserver une version sans amélioration du projet avant d'apporter des améliorations.

1. (⋆) Mettre au point un système pour mettre en pause la partie en appuyant sur une touche dédiée.

2. (⋆) Ajouter la possibilité de créer des pommes empoisonnées qui provoquent la défaite si mangées. Au moins une pomme parmi celles présentes dans le monde ne doit pas être empoisonnée.

3. (⋆) Ajouter la possibilité de créer des pommes particulières qui apportent deux points au lieu d'un seul si mangées.

4. (⋆) Ajouter un mode de jeu particulier dans lequel le joueur doit réaliser le plus grand score possible pendant une durée limitée imposée (et à choisir).

5. (⋆⋆) Ajouter la possibilité de créer des pommes anxiogènes qui provoquent un doublement de la vitesse du serpent pendant une durée déterminée. Si le serpent mange par exemple deux telles pommes dans un court intervalle de temps, sa vitesse est donc multipliée par 4.

6. (⋆⋆) Mettre au point un système de sauvegarde permanent des meilleurs scores. Au lancement du jeu, figurera un menu permettant de visualiser les meilleurs scores, ligne par ligne. Une ligne contient le nom du joueur, le score obtenu et éventuellement le nom du mode de jeu concerné.

7. (⋆⋆) Ajouter des sons qui sont joués à chaque déplacement du serpent, lorsqu'il mange une pomme et lorsqu'il meurt.

8. (⋆⋆) Ajouter un mode de jeu dans lequel apparaissent de manière aléatoire des murs dans le monde. Un mur est un obstacle qui provoque la perte de la partie si rencontré. Les murs ne doivent évidemment pas apparaître sur le serpent ni sur les pommes, et, pour ne pas rendre la partie trop compliquée, non plus sur l'une des quatre cases voisines à la tête du serpent.

9. (⋆ ⋆ ⋆) Rendre le programme paramétrable : le joueur peut choisir les dimensions du quadrillage, le nombre de pommes, la taille initiale du serpent et la durée d'un tour de boucle (exprimée en ms). Ces paramètres doivent figurer dans un fichier de configuration `Serpent.ini` situé dans le même répertoire que l'exécutable. Il est constitué de lignes de la forme

```
largeur = 128
hauteur = 96
nombre_pommes = 4
taille_serpent = 7
duree_tour = 2500
```

10. (⋆⋆⋆) Même objectif que celui du point précédent à la différence près que les options sont accessibles depuis un menu d'options à incorporer dans l'application. Attention : la programmation de ces deux améliorations combinées ne rapporte pas 6 ⋆ mais 4 uniquement.

11. (⋆ ⋆ ⋆) Concevoir un mode de jeu à deux joueurs où s'affrontent dans un même monde deux serpents pilotés chacun par un joueur. Les mêmes conditions de défaite que dans le mode classique à un joueur s'appliquent, avec en plus le fait qu'arriver sur une case occupée par un serpent ennemi provoque la défaite. Le joueur gagnant est celui qui a réalisé le score le plus élevé une fois que les deux serpents sont morts (car ils le seront nécessairement si la partie dure suffisamment longtemps).

12. (⋆ ⋆ ⋆⋆) À la condition que l'amélioration précédente ait été apportée, faire en sorte d'ajouter un mode de jeu humain contre machine dans lequel l'un des serpents est dirigé par l'ordinateur. Le jeu de l'ordinateur doit être cohérent et compétitif.