TP « Serpent »

N. Delestre

Objectif

L'objectif de ce TP est de développer une implémentation du jeu du serpent. Au lancement du jeu on précise en paramètre la largeur (l), la hauteur (h) du terrain de jeu et la longueur initiale du serpent. Le terrain de jeu est torique, c'est-à-dire que si le serpent sort par un côté il réapparaît de l'autre côté. Au départ du jeu la queue du serpent se trouve au centre du terrain de jeu et le serpent se dirige vers la droite. Le joueur peut changer la direction du serpent en utilisant les flèches du clavier. À chaque fois que le serpent mange un fruit, il grandit de 1, il accélère, un nouveau fruit apparaît à un endroit aléatoire et le joueur marque un point. Le jeu s'arrête lorsque le serpent se mord la queue. La figure 1 est un exemple d'affichage du jeu en mode texte. Les coordonnées des éléments du jeu sont des naturels tels que l'origine (0,0) est en haut à gauche du terrain de jeu, l'abscisse augmente vers la droite et l'ordonnée. Les éléments de la bordure du terrain ont des coordonnées du type $(0, j)$, $(l + 1, j)$ avec $j \in 1..h$ et $(i, 0)$ et $(i, h + 1)$ avec $i \in 1..l$.

Cette implémentation utilisera une interface graphique en mode texte, mais la conception doit bien séparer logique métier et interface homme-machine.

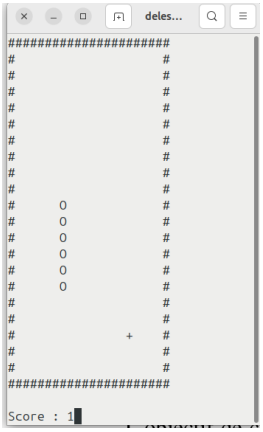

FIGURE 1 – Exemple d'affichage du jeu en mode texte

Analyse

Une analyse de ce jeu fait apparaître les TAD suivants :

TAD Direction

Ce TAD représente la direction du déplacement du serpent.

Nom: Direction

Opérations:

HAUT:	$\rightarrow$ Direction
BAS:	$\rightarrow$ Direction
GAUCHE:	$\rightarrow$ Direction
DROITE:	$\rightarrow$ Direction
directionOpposee:	Direction $\rightarrow$ Direction

Axiomes:

- directionOpposee(HAUT)=BAS
- directionOpposee(BAS)=HAUT
- directionOpposee(GAUCHE)=DROITE
- directionOpposee(DROITE)=GAUCHE

TAD Coordonnee

Ce TAD représente une coordonnée dans le terrain de jeu torique.

Nom: Coordonnee

Utilise: **Naturel**, Direction

Opérations:

coordonnee:	**Naturel** $\times$ **Naturel** $\rightarrow$ Coordonnee
abscisse:	Coordonnee $\rightarrow$ **Naturel**
ordonnee:	Coordonnee $\rightarrow$ **Naturel**
voisin:	Coordonnee $\times$ Direction $\times$ **NaturelNonNul** $\times$ **NaturelNonNul** $\rightarrow$ Coordonnee

Axiomes:

- abscisse(coordonnee(x,y))=x
- ordonnee(coordonnee(x,y))=y
- voisin(c,HAUT,l,h)=coordonnee(abscisse(c),(ordonnee(c)-2) mod h + 1)
- voisin(c,BAS,l,h)=coordonnee(abscisse(c),(ordonnee(c)+1) mod h + 1)
- voisin(c,GAUCHE,l,h)=coordonnee((abscisse(c)-2) mod l + 1,ordonnee(c))
- voisin(c,DROITE,l,h)=coordonnee((abscisse(c)+1) mod l + 1,ordonnee(c))

TAD Serpent (sans les axiomes)

Ce TAD représente le serpent, l'élément central du jeu.

Nom: Serpent

Utilise: **Naturel**

Opérations:

serpent:	Coordonnee $\times$ **NaturelNonNul** $\times$ Direction $\times$ **NaturelNonNul** $\times$ **NaturelNonNul** $\rightarrow$ Serpent
positionTete:	Serpent $\rightarrow$ Coordonnee
positionQueue:	Serpent $\rightarrow$ Coordonnee
avancer:	Serpent $\rightarrow$ Serpent
direction:	Serpent $\rightarrow$ Direction
changerDirection:	Serpent $\times$ Direction $\nrightarrow$ Serpent

	accroissement:	Serpent × **Naturel** → Serpent
	longueur:	Serpent → **NaturelNonNul**
	seMord:	Serpent → **Booleen**
	estUneCoordonneeDuSerpent:	Serpent × Coordonnee → **Booleen**
	coordonneesDuSerpent:	Serpent → Liste<Coordonnees>
Préconditions:	changerDirection(s,d):	d ≠ direction(s) et d ≠ directionOpposee(direction(s))

Analyse descendante

L'algorithme du jeu ne nécessite pas d'analyse descendante puisque le problème est assez simple : après avoir initialiser le serpent, il avance dans la direction courante tant qu'il ne se mord pas la queue. S'il mange un fruit, il grandit, le score augmente de 1 et la vitesse de jeu augmente. Toutes ces opérations sont proposées par les TAD précédents.

Conception préliminaire

Type Direction

— **fonction** directionOpposee (d : Direction) **:** Direction

Type Coordonnee

— **fonction** coordonnee (x,y : **NaturelNonNul**) **:** Coordonnee
— **fonction** abscisse (c : Coordonnee) **: Naturel**
— **fonction** ordonnee (c : Coordonnee) **: Naturel**
— **fonction** voisin (c : Coordonnee, d : Direction, largeurTerrain, hauteurTerrain : **Naturel-NonNul**) **:** Coordonnee

Type Serpent

— **fonction** serpent (
  positionInitialeQueue : Coordonnee,
  longueurInitiale : **NaturelNonNul**,
  directionInitiale : Direction,
  largeurTerrain : **NaturelNonNul**,
  hauteurTerrain : **NaturelNonNul**
) **:** Serpent
— **fonction** positionTete (s : Serpent) **:** Coordonnee
— **fonction** positionQueue (s : Serpent) **:** Coordonnee
— **procédure** avancer (**E/S** s : Serpent)
— **fonction** direction (s : Serpent) **:** Direction
— **procédure** changerDirection (**E/S** s : Serpent, **E** d : Direction)
  ⌊**précondition(s)** d ≠ direction(s) et d ≠ directionOpposee(direction(s))
— **procédure** accroissement (**E/S** s : Serpent, **E** longeur : **Naturel)**
— **fonction** longueur (s : Serpent) **: NaturelNonNul**

— **fonction** seMord (s : Serpent) **: Booleen**
— **fonction** estUneCoordonneeDuSerpent (s : Serpent, c : Coordonnee) **: Booleen**
— **fonction** coordonneesDuSerpent (s : Serpent) **:** Liste<Coordonnee>

Conception détaillée

Type Direction

Puisqu'il n'y a que quatre directions possibles, on peut utiliser une énumération.
— **Type** Direction = {HAUT,BAS,GAUCHE,DROITE}

Type Coordonnee

Une coordonnée est un couple d'entiers naturels.
— **Type** Coordonnee = **Structure**
x : **Naturel**
y : **Naturel**
— **finstructure**

Type Serpent

La longueur du serpent varie au cours de l'exécution du programme. Nous allons donc utiliser une structure dynamique de données pour le concevoir, plus exactement la liste chaînée. Nous avons besoin de référencer la tête du serpent et la queue du serpent, nous allons donc nous inspirer de la conception de la `File` vue en cours :
Type Coordonnee = **Structure**
tete : ListeChainee<Coordonnee>
queue : ListeChainee<Coordonnee>
finstructure
Pour avoir des opérations en $O(1)$, le champ `queue` doit référencer le début de la liste chaînée et le champ `tete` la fin de la liste chaînée (le dernier élément), comme l'indique la figure 2.

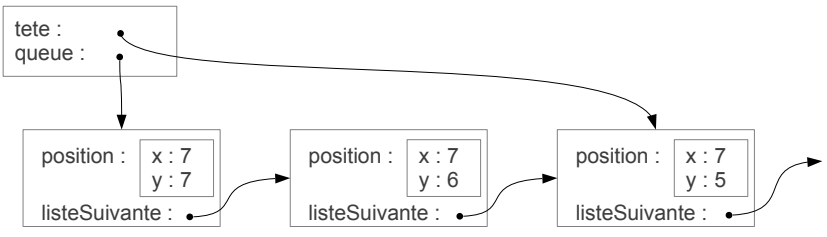

FIGURE 2 – Représentation du serpent

En effet, l'opération `avancer` consiste à ajouter une coordonnée à la fin de la liste et à supprimer, ou pas, le premier élément.

Afin d'éviter de parcourir la liste chaînée pour obtenir sa longueur, nous pouvons ajouter un champ `longueur`.

Lorsque le serpent mange un fruit, il grandit. Nous allons donc ajouter un champ `accroissement` qui contiendra le nombre de fois où le serpent doit avancer sans supprimer le premier élément de la liste chaînée.

Nous devons connaître la direction courante du serpent, nous allons donc ajouter un champ `direction`.

Enfin lorsqu'il avance, le serpent ne doit pas sortir du terrain de jeu. Nous allons donc ajouter deux champs `largeurTerrain` et `hauteurTerrain` qui contiendront les dimensions du terrain de jeu.

Finalement le type `Serpent` est conçu de la manière suivante :

Type Serpent = **Structure**
  tete : ListeChainee<Coordonnee>
  queue : ListeChainee<Coordonnee>
  longueur : **NaturelNonNul**
  accroissement : **Naturel**
  direction : Direction
  largeurTerrain : **NaturelNonNul**
  hauteurTerrain : **NaturelNonNul**
finstructure

La seule fonction nécessitant un algorithme est la procédure `avancer` :

procédure avancer **(E/S** serp : Serpent**)**

  Déclaration coordTete, nelleCoordonnee : Coordonnee
  nelleTete : ListeChainee<Coordonnee>

debut
  coordTete ← positionTete(serp)
  nelleCoordonnee ← voisin(coordTete, direction(serp), serp.largeurTerrain, srp.hauteurTerrain)

  nelleTete ← listeChainee()
  ajouter(nelleTete, nelleCoordonnee)
  fixerListeSuivante(serp.tete, nelleTete)
  serp.tete ← nelleTete
  si serp.accroissement = 0 **alors**
    supprimerTete(serp.queue)
  sinon
    serp.accroissement ← serp.accroissement - 1
    serp.longueur ← serp.longueur + 1
  finsi
fin

Enfin l'initialisation du serpent au début du jeu peut être vu comme au départ un serpent de longueur 1 avec un accroissement égal à la longueur initiale du serpent suivi d'autant d'avancements.

Séparation logique métier et IHM

Afin de bien séparer la logique métier (le fait de jouer au jeu du serpent) et l'interface homme machine (l'affichage du jeu et la gestion des entrées utilisateur), nous allons utiliser le mécanisme des *callback*. En plus des informations nécessaires à l'initialisation du jeu, le sous programme principal de la logique métier (procédure `jouer`) prendra en paramètre quatre sous programmes :

— une procédure pour afficher le terrain de jeu ;
— une procédure pour mettre à jour l'affichage du terrain de jeu ;
— une fonction pour récupérer la direction choisie par l'utilisateur. Cette opération n'étant pas bloquante, si l'utilisateur ne choisit pas de direction, la fonction retourne la direction courante du serpent ;
— une procédure pour demander à accélérer le jeu.

Le premier paramètre de ces sous-programmes est l'interface (de type `Interface` non connu) sur laquelle vont être effectuées les actions demandées.

Le pseudo code de ce sous programme est :

procédure jouer **(**
    E/S interface : Interface,
    E afficherTerrain : procédure(**E/S** Interface, Serpent, Coordonnee, **Naturel**),
    E mettreAJour : procédure(**E/S** Interface, Serpent, Coordonnee, **Naturel**),
    E obtenirDirection : procédure(**E** Interface, Direction, **S** Direction,
    E accelerer : procedure(**E/S** Interface),
    E largeurTerrain, hauteurTerrain, longueurInitialeSerpent : **NaturelNonNul**,
    E directionInitialeSerpent : Direction,
    S score : **NaturelNonNul**
)

Déclaration serp : Serpent,
coordFruit : Coordonnee
nelleDirection : Direction

debut
score ← 0
serp ← serpent(coordonnee(largeurTerrain div 2, hauteurTerrain div 2), longueurInitialeSerpent, directionInitialeSerpent, largeurTerrain, hauteurTerrain)
nelleCoordonneeFruit(largeurTerrain, hauteurTerrain, serp, coordFruit)
afficherTerrain(interface, serp, coordFruit, score)
tant que non seMord(serp) **faire**
    obtenirDirection(interface, direction(serp), nelleDirection)
    si nelleDirection ≠ direction(serp) et nelleDirection ≠ directionOpposee(direction(serpent))
    alors
        changerDirection(serp, nelleDirection)
    finsi
    avancer(serp)
    si positionTete(serp) = coordFruit **alors**
        score ← score + 1
        accroissement(serp, 1)
        nelleCoordonneeFruit(largeurTerrain, hauteurTerrain, serp, coordFruit)

```
        accelerer(interface)
      finsi
      mettreAJour(interface, serp, coordFruit, score)
   fintantque
fin
```

Le programme C

Le programme est composé des fichiers suivants :

— `include/direction.h` et `src/direction.c` le module qui implante le type `D_Direction` ;
— `include/coordonnee.h` et `src/coordonnee.c` le module qui implante le type `C_Coordonnee` ;
— `include/serpent.h` et `src/serpent.c` le module qui implante le type `S_Serpent` ;
— `include/jeu.h` et `src/jeu.c` le module qui implante la fonction `J_jeu` ;
— `include/interface.h` et `src/interface` le module qui définit le type `I_Interface` et les prototypes des fonctions de gestion de l'interface en mode texte en utilisant la bibliothèque `ncurses` ;
— `src/main.c` le programme principal qui utilise les modules précédents pour faire fonctionner le jeu.

À ces fichiers, s'ajoutent des fichiers de tests unitaires dans le répertoire `src` (pour les modules `direction`, `coordonnee` et `serpent`). Le fichier `makefile` permet de générer le programme principal (`make`) ou les tests unitaires (`make test`).

Il y a quelques modifications de la conception préliminaire dû aux spécificités du langage C

— pour être indépendant du type implantant l'interface, cette dernière est passée à la logique métier sous la forme d'un `void*` ;
— afin de simplifier son utilisation, la fonction `S_coordonneeDuSerpent` ne retourne pas une liste mais un tableau dynamique de coordonnées.

Travail à réaliser

1. Le projet utilise le SDD liste chaînée disponible sur le gitlab du cours `https://gitlab.insa-rouen.fr/delestre/algo-exemples`. Pour pouvoir l'utiliser dans ce projet et pourquoi pas dans d'autres projets, nous allons l'installer en local :
   — clonez le projet gitlab dans un répertoire ;
   — compilez le sous projet sdd (`algo-exemples/CM/09-sdd`) : vous obtenez une bibliothèque statique `libsdd.a` ;
   — si vous ne les avez pas encore, créez les répertoires suivants :
      — `~/.local/include`
      — `~/.local/lib`
   — créez les liens symboliques pour tous les fichiers d'entêtes (`arbreBinaire.h`, `listeChainee.h`, etc.) depuis `~/.local/include` vers les fichiers du projet git ;
   — faites de même pour le résultat de la compilation (depuis `~/.local/lib`) ;
   — configurez dans votre `.bashrc` les variables d'environnement `C_INCLUDE_PATH`, `LIBRARY_PATH` et `LD_LIBRARY_PATH` pour indiquer à gcc où aller chercher des fichiers d'entête et des bibliothèques non systèmes mais qui ne font pas partie du projet ;
2. Développez les fonctions de `src/serpent.c` afin que tous les tests unitaires fonctionnent ;

3. Développez le fichier `src/jeu.c`.