A-Maze-Ing

Projet 3/3 pour le module de programmation fonctionnelle

Koalab koala@epitech.eu

Abstract: A-Maze-Ing est un projet proposant la réalisation d'un petit jeu vidéo basé sur un labyrinthe qu'il devra est être possible de résoudre. Le choix du gameplay est libre et la bibliothèque graphique utilisée est OCamlSDL.

Ce projet est librement inspiré d'un projet du cours de programmation fonctionnelle de L3 de M. Vincent Balat, Maître de Conférence à l'université Paris Diderot, qui a donné son aimable accord pour l'utilisation et l'adaptation libre de ses contenus pédagogiques.

KOALA

Table des matières

Chapitre I

Introduction

Ce projet marque la fin du module de programmation fonctionnelle de deuxième année. Dans cette optique une certaine liberté dans sa réalisation et dans vos choix techniques vous est laissée.

Votre but est de coder un petit jeu vidéo ayant pour thème un labyrinthe. Vous aurez cependant les contraintes suivantes :

- Le rendu graphique ET la gestion des évènements DOIVENT être fait avec la bibliothèque graphique OCamlSDL. Pas de dérogation possible.
- Chaque étape de la partie obligatoire DOIT pouvoir être compilée et exécutée de manière indépendante des autres. L'organisation de votre code en conséquence est à votre discrétion.
- L'utilisation d'un Makefile pour compiler votre projet est obligatoire.

Consultez la section "Consignes" de ce sujet pour plus de détails.

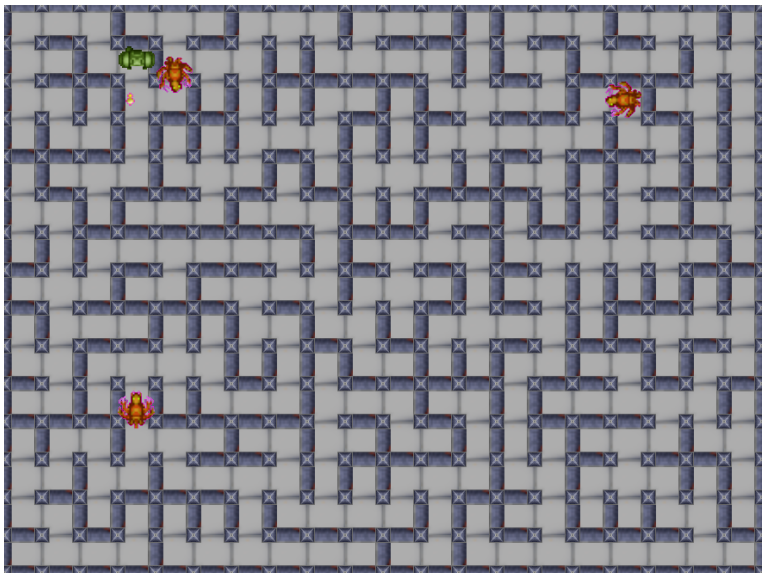

FIGURE I.1 – `A-Maze-Ing`

Chapitre II

Partie obligatoire

II.1 Etape 1

Nous allons dans un premier temps construire des labyrinthes dit "parfaits".

Utilisation :

- Vous DEVEZ compiler un exécutable nommé "step1" (ou tout nom équivalent) prenant en argument la largeur et la hauteur du labyrinthe à générer. Par exemple :

```
1        >./step1 10 10
```

- En cas d'arguments invalides ou en nombre incorrect, vous DEVEZ afficher un usage et quitter proprement.

Nous allons considérer le labyrinthe comme une grille de cases carrées. Chaque case sera composée de 4 portes pouvant être soit ouverte soit fermée indépendamment. Une porte ouverte correspondra a une communication possible entre les deux cases adjacentes et une porte fermée correspondra à un mur.

- Vous DEVEZ proposer un module pour représenter un labyrinthe à cases carrées.
- Vous DEVRIEZ proposer également un module pour représenter une case d'un labyrinthe.
- Vous DEVRIEZ proposer également un module pour représenter une porte d'une case d'un labyrinthe.

Privilégiez au maximum la programmation modulaire et les types abstraits !

Un labyrinthe est dit "parfait" lorsqu'il y a un **unique** chemin reliant tout couple de cases. Afin de réaliser un tel labyrinthe, nous allons utiliser l'algorithme suivant :

- Fermer toutes les cases du labyrinthe et les "colorier" toutes de façon différente.
- Choisir une porte au hasard entre deux cases (donc entre deux cases de couleurs différentes) et l'ouvrir si elle ne l'est pas déjà.
- Colorier la seconde zone de la même couleur que la première.
- Répéter l'opération jusqu'à ce que le labyrinthe ne comporte plus qu'une seule couleur.

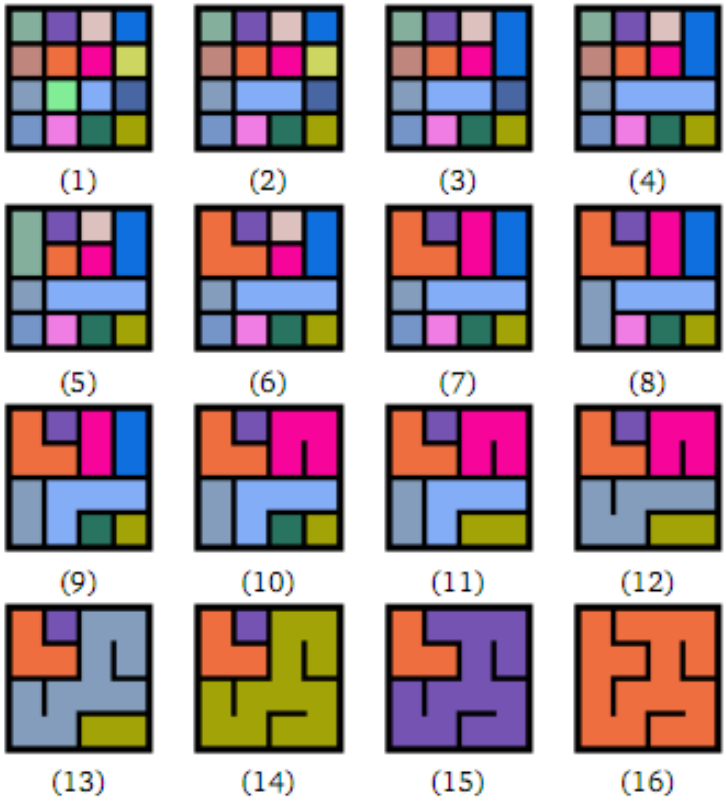

FIGURE II.1 – Génération d'un labyrinthe parfait par coloration

L'utilisation de "couleurs" n'est bien sur que pour visualiser l'algorithme. N'importe quel type dénombrable fera parfaitement l'affaire.

Avec cet algorithme, on est assuré de ne faire communiquer deux parties du labyrinthe que par une seule porte, et on a bien un labyrinthe parfait.

Une implantation naïve de cet algorithme conduit à des performances désastreuses ! Il est très important de l'implanter intelligemment. Une génération de 100x100 ne devrait pas prendre plus de quelques instants, surtout sur les machines de l'école !

- Vous DEVEZ afficher votre labyrinthe sur la sortie standard une fois celui-ci généré. Cette consigne ne s'applique plus dans les étapes suivantes.

II.2 Etape 2

Maintenant que vous pouvez générer un labyrinthe "parfait", nous allons le dessiner ! Utilisation :

- Vous DEVEZ compiler un exécutable nommé "step2" (ou tout nom équivalent) prenant en argument la largeur et la hauteur du labyrinthe à générer. Par exemple :

```
1        >./step2 10 10
```

- En cas d'arguments invalides ou en nombre incorrect, vous DEVEZ afficher un usage et quitter proprement.

Vous devez impérativement utiliser OCamlSDL pour la partie graphique et la gestion des évènements de ce projet.

Vous trouverez `OCamlSDL` et sa documentation à cette adresse : http://ocamlsdl.sourceforge.net/home.html.

- Une fois votre labyrinthe généré vous DEVEZ l'afficher avec `OCamlSDL`.
- Les murs doivent apparaitre CLAIREMENT.

Il pourrait être une bonne idée de prévoir un mécanisme adaptant la largeur d'une case selon la taille du labyrinthe affiché.

II.3 Etape 3

Voir s'afficher un labyrinthe est toujours accompagné d'une furieuse envie de le résoudre.

Utilisation :

- Vous DEVEZ compiler un exécutable nommé "step3" (ou tout nom équivalent) prenant en argument la largeur et la hauteur du labyrinthe à générer. Par exemple :

```
1          >./step3 10 10
```

- En cas d'arguments invalides ou en nombre incorrect, vous DEVEZ afficher un usage et quitter proprement.

- Vous DEVEZ proposer une fonction `solve` dont le type est à votre discrétion pour résoudre un chemin dans le labyrinthe généré.
- Votre fonction `solve` DOIT au moins prendre en paramètre les coordonnées de la case de départ et de la case d'arrivée.
- Vous DEVEZ afficher CLAIREMENT le chemin trouvé.
- Le choix de l'algorithme utilisé pour résoudre le labyrinthe est à votre discrétion, toutefois, vous DEVEZ être capables de l'expliquer en détails pendant la soutenance.

II.4 Etape 4

Générer, afficher et résoudre des labyrinthes parfaits à cases carrées n'est pas très difficile non ? Vous n'aurez donc aucun problème à générer et afficher des labyrinthes parfaits à cases hexagonales (en nid d'abeilles quoi) !

- Vous DEVEZ compiler un exécutable nommé "step4" (ou tout nom équivalent) prenant en argument la largeur et la hauteur du labyrinthe à générer ainsi que sa forme. Par exemple :

```
>./step4 10 10 --square
```

Ou encore :

```
>./step4 10 10 --hexagonal
```

- En cas d'arguments invalides ou en nombre incorrect, vous DEVEZ afficher un usage et quitter proprement.

Pour vous éviter de douloureuses duplications de code, une approche générique par functors parait appropriée, non ?

II.5 Etape 5

Enfin les choses sérieuses ! Avec votre labyrinthe résolvable à cases carrées ou hexagonales, il est temps de montrer votre créativité et votre maîtrise d'OCaml !

Vous allez réaliser un petit jeu vidéo au gameplay de votre choix en vous basant sur votre labyrinthe. Vous êtes entièrement libres et la limite est votre imagination ! Vous pouvez même adapter un jeu qui vous plait au contexte du projet !

Voici quelques exemples de jeux si vous êtes en mal d'inspiration :

- Pacman
- Space Hulk
- Diablo
- ...

`Le net regorge de sprites....`

- Le nom de l'exécutable et ses arguments sont à votre discrétion.
- Le labyrinthe DOIT être aléatoire.
- Le calcul de chemin de l'étape 3 DOIT être utilisé.
- Votre boucle de jeu DOIT clairement distinguer 3 étapes qui DOIVENT être dans cet ordre :
  - Gestion des évènements.
  - Logique du jeu et update du rendu.
  - Affichage du rendu.
- Votre jeu DOIT être bien :).

`L'utilisation d'un timer est toute indiquée si l'on souhaite s'abstraire de la vitesse du CPU...`

Chapitre III

Bonus

Une fois la partie obligatoire **terminée** et **fonctionnelle**, vous pouvez ajouter des bonus à votre projet, par exemple :

- Des animations
- Des menus
- Des sons / de la musique (bibliothèque libre, pas forcement OCamlSDL)
- Sauvegarde et chargement
- Configuration des touches
- ...

Chapitre IV

Consignes

Vous **DEVEZ** impérativement respecter les consignes suivantes :

- Tous les traits impératifs d'`OCaml` sont interdits, sauf concernant les exceptions, les séquences d'instructions quand nécessaire et les tableaux. Ceci est bien évidement limité au strict nécessaire et devra fait l'objet d'une justification en soutenance.
- Il est interdit que votre programme se termine à cause d'une exception non gérée.
- Vous avez le droit d'utiliser TOUT ce qu'OCaml met à votre disposition. La seule exception concerne la bibliothèque graphique qui **DOIT** être `OCamlSDL`.
- Votre rendu **DOIT** compiler avec `ocamlc` **ET** `ocamlopt`. Aucun script ne sera accepté en soutenance.
- Chaque étape de la partie obligatoire DOIT pouvoir être compilée et exécutée de manière indépendante des autres. L'organisation de votre code en conséquence est à votre discrétion.
- L'utilisation d'un Makefile pour compiler votre projet est obligatoire.
- Les étapes seront corrigées dans l'ordre du sujet. Il n'est donc pas utile de présenter une étape 2 sans étape 1 et ainsi de suite.
- Toutes les erreurs doivent être gérées, de façon appropriée, à tous les niveaux, sans exception à cette règle.
- Il n'existe pas de norme pour `OCaml` à `Epitech`. Toutefois, tout code illisible pourra être sanctionné arbitrairement. Une indentation claire, de l'espace entre les fonctions et le respect des 80 colonnes vous éviteront très probablement des ennuis.
- Une conception modulaire et générique de projet sera très appréciée.
- La soutenance mettra un accent tout particulier sur le contrôle de vos connaissances personnelles.
- Gardez un œil sur ce sujet car il est susceptible d'être modifié.
- Les évolutions du sujet seront notifiées sur le forum du module `B4 - Programmation Fonctionnelle` de l'intranet `Epitech`. Vous devez lire ce forum régulièrement.
- Si vous constatez des fautes ou des incohérences dans ce sujet, merci d'en notifier les auteurs afin d'apporter les corrections.

- Vous devez rendre votre projet sur le systême de rendu mis à votre disposition par `Epitech`. Le nom de repertoire est `ocaml_a-maze-ing`.
- Vos dépôts seront ramassés à l'heure exacte de la fin du projet, intranet `Epitech` faisant foi.
- Seul le code présent sur votre dépôt sera évalué lors de la soutenance. La documentation relative aux dépôts fournis par le `Koalab` est fournie avec ce sujet.

Bon courage !