Rapport de projet *langage C*

Jeu de Memory

Etudiants : Romain COCOGNE, Quentin COMBAL, Zayd EL HACHIMI

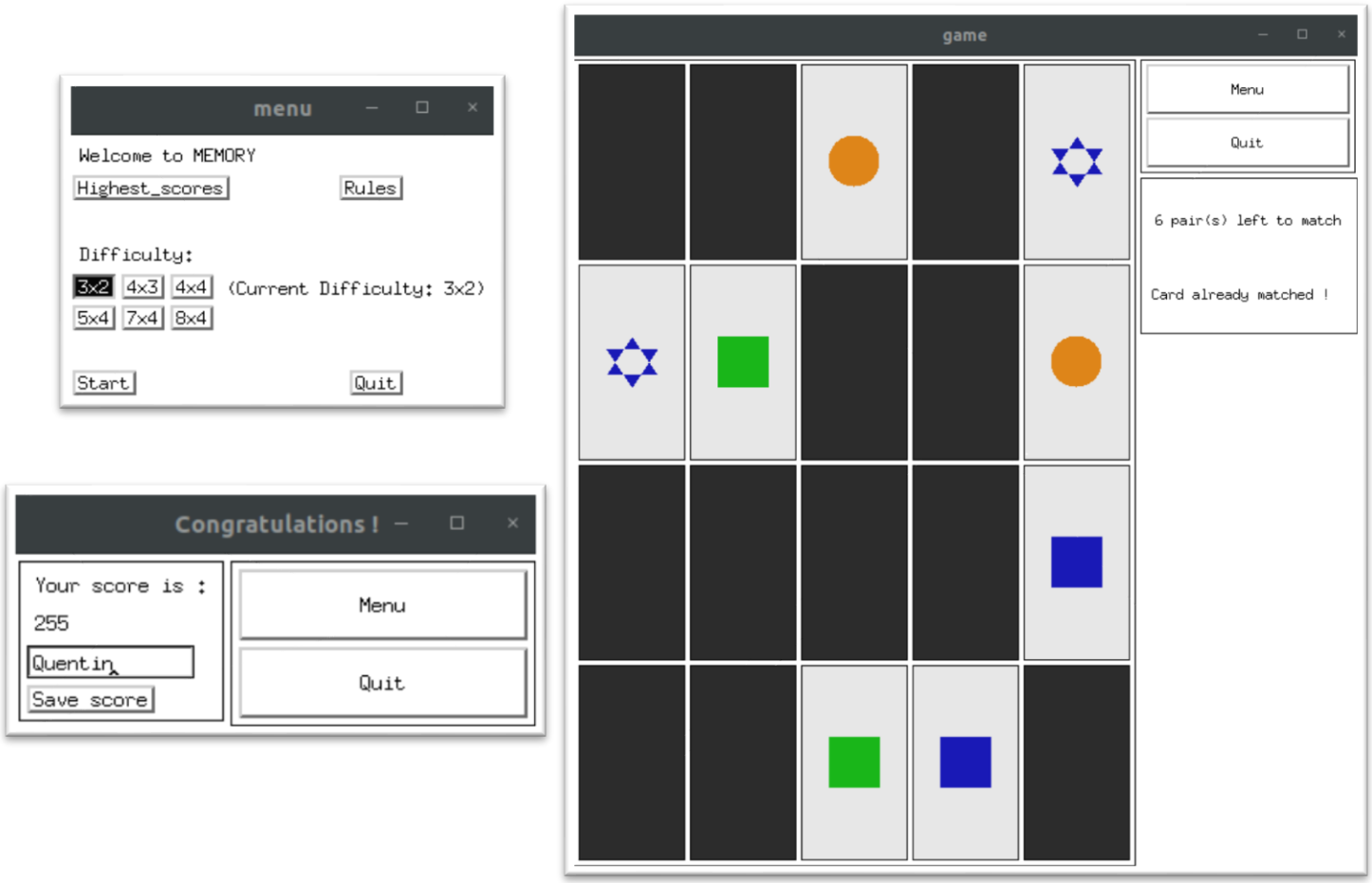
Promotion : ELEC 3 2018/2019

Introduction

Ce document présente notre application *Memory,* développée pour le projet de programmation C de l'année ELEC3 2018/2019. L'objectif est de créer un jeu de memory codé en C en implémentant les fonctions de la bibliothèque *libsx* pour créer l'interface graphique.

Les règles du jeu sont les suivantes : Les cartes sont présentées par paires et sont disposées face cachée en début de jeu. Lors d'un coup, le joueur sélectionne deux cartes et les retourne. Si les cartes sont identiques, elles restent visibles, sinon, elles sont retournées face cachée. Le jeu s'arrête lorsque toutes les paires sont visibles.

Présentation de l'application



A gauche : menus de début de partie et de fin de partie
A droite : plateau de jeu

Caractéristiques de l'application :

- 6 difficultés prédéfinies
- Jusqu'à 16 paires de cartes différentes
- 4 formes et 4 couleurs possibles pour chaque carte
- Enregistrement et affichage des 10 meilleurs scores
- Affichage des règles
- Possibilité de revenir au menu ou de quitter l'application à tout moment de la partie

Menu de début de jeu :

Au lancement du programme, le menu de début de jeu s'affiche. Depuis cette fenêtre le joueur peut sélectionner un des six niveaux de difficulté proposés. Cette sélection est affichée par un enfoncement du bouton ainsi que par un message indiquant : *« Current difficulty : largeur x hauteur ».* Par défaut, la difficulté est 3x2.

L'utilisateur peut également consulter les 10 meilleurs scores actuels et afficher les règles du jeu.

Pour lancer la partie, l'utilisateur doit cliquer sur le bouton « start ». Il peut également quitter le programme en cliquant sur le bouton « quitter ».

Fenêtre de jeu :

La fenêtre de jeu est composée du plateau de jeu à gauche, ainsi que de boutons de navigation et une *infobox* à droite. Le plateau de jeu présente les cartes faces cachées, que l'utilisateur peut retourner avec un clic gauche. Les cartes sont associées par paires de formes coloriées. Lorsque le joueur retourne deux cartes, elles restent face visible si les cartes forment une paire, sinon elles sont retournées face cachée au prochain clic.

Le nombre de paires restantes à découvrir est affiché dans l'*infobox* à droite. Le joueur peut également y lire des informations indicatives, lui précisant par exemple de ne pas choisir deux fois la même carte.

La partie se termine lorsque toutes les cartes sont face visible. Le joueur peut alors voir le plateau de jeu au complet, et est invité à cliquer une fois de plus pour afficher l'écran de fin.

Fenêtre de fin :

Sur cette fenêtre s'affiche le score du joueur à gauche ainsi qu'une fenêtre de navigation à droite. Si le score du joueur est suffisamment élevé, ce dernier à la possibilité d'enregistrer son nom pour qu'il soit affiché parmi les 10 meilleurs scores. Dans ce cas, le tableau actualisé des meilleurs scores s'affiche.

Le joueur peut ensuite quitter l'application ou revenir sur le menu pour relancer une nouvelle partie.

Structure du programme

Le programme est composé de plusieurs modules regroupés en trois catégories

- **Logique de jeu :** carte.h, jeu.h
- **Gestion des scores :** liste.h**,** player.h, score.h
- **Implémentation graphique :** forme.h, couleur.h, display.h, callbacks.h

Cette organisation nous permet d'avoir une séparation entre la logique du jeu et sa représentation graphique. Si nous décidions de changer la librairie utilisée pour l'interface graphique, nous pourrions réutiliser les modules de logique et de gestions des scores.

Voici une description des différents modules. Une description plus détaillée des algorithmes est disponible avec les commentaires du code.

Carte :

Ce module nous permet de représenter une carte du jeu, on définit une structure « *Card »* composé des objets « *int id »* et « *int mode »* où « id » sera l'identifiant de la carte tandis que « mode » représentera l'état de la carte (cachée, retournée, découverte). Nous décidons de n'implémenter aucune fonction relative à la carte car la structure choisie ne nécessite aucune opération complexe et nous pouvons nous passer de fonctions d'accès à l'*id* et au *mode* qui peuvent être modifiés directement.

Jeu :

Ce module utilise le module *Carte*. On modélise une partie du jeu *Memory* en définissant une structure « *Jeu* » caractérisée par le nombre de cartes dans la partie (nbCartes), le nombre de cartes non découvertes par le joueur (nbCartesRestantes), le nombre de coups joués (nbCoups), et l'étape du jeu (etape).

La variable étape correspond aux états possibles d'une partie qui sont au nombre de 4 : l'état où aucune carte n'est retournée, l'état où une première carte est retournée, l'état où deux cartes sont retournées et attendent vérification, et enfin l'état où la partie est finie. On donne à chaque état un entier le représentant dans la structure de « *Jeu* ».

Cette structure contient également les pointeurs vers les deux cartes retournées, afin de pouvoir comparer les cartes et de modifier leurs états. Enfin, la structure possède un tableau qui regroupe toutes les cartes du jeu.

Les fonctions de ce module permettent les actions suivantes :

- **Initialiser une partie :** Initialisation des paramètres d'une structure Jeu déclarée au préalable. Un tableau d'objets *Card* est instancié, initialisé, puis mélangé. La partie peut ensuite démarrer.
- **Jouer un coup :** Si le jeu est encore dans les étapes de découverte de cartes, enregistrement de l'adresse des cartes retournées. Le jeu est avancé d'une étape.
- **Vérifier un coup** : Vérification d'un coup en comparant deux cartes retournées. Faire avancer la partie à l'état de partie finie si nbCartesRestantes est nul. Sinon, fait revenir le jeu à l'état initial de découverte des cartes.

Liste :

On définit dans ce module la structure de *« Liste »* qui nous permettra de représenter une liste de joueurs (*Player*). Une *« Liste »* est composée de *« nœuds »* chacun caractérisé par sa valeur et par son suivant, cette structure suit bien le concept de structure chainée.

On construit aussi les fonctions de base nous permettant de gérer une *Liste*.

Ce module est générique et peut être utilisé par n'importe quel type. Nous avons fait le choix de le rendre modulable afin de pouvoir réutiliser les Listes pour de possibles améliorations du jeu.

Player :

On définit ici la structure « *Player* » pour représenter les joueurs et leurs scores, ainsi chaque Player est caractérisé par son nom dans la variable « *char *name* » et par son score dans la variable « *int score* ».

On construit alors les fonctions qui permettront d'organiser l'emplacement des *Player*s dans une liste et cela en couplant les fonctions du module « liste.h » avec celles de « player.h » dans le module « score.h ».

Score :

Ce module contient les fonctions qui organiseront les « *Player* » dans une liste, c'est-à-dire faire en sorte que l'ajout d'un nouveau nom de joueur et son score soit possible dans une liste de joueurs contenu dans un fichier, mais aussi d'ordonner les joueurs selon leurs scores.

Nous avons décidé d'utiliser les *Player* dans le module score et pas l'inverse (gérer les scores depuis le module Player), car la structure *Player* est particulièrement utile dans la gestion des scores et facilite grandement le tri et la sauvegarde dans le fichier. De plus, en utilisant les joueurs de cette façon, le reste du programme n'a pas besoin d'utiliser la structure *Player* et l'ajout des scores se fait en toute transparence avec la fonction *saveScore*.

Le fichier de scores se nomme *.score* pour rendre inaccessible les données de score à l'utilisateur. Ce fichier contient au maximum 10 scores et ces derniers sont calculés de façon à prendre en compte la difficulté et le nombre de coups utilisés pour finir la partie. La formule utilisée est la suivante :

*(1/10+1/nbCoups)*difficulté*100*. Cette formule permet de prendre en compte le ratio difficulté/nbCoups tout en ajoutant un facteur difficulté/10 qui va valoriser les joueurs prenant une grille avec plus de cartes.

Forme :

Le concept du jeu *Memory* est très visuel, ce qui nous a amené à définir pour chaque carte une forme graphique que l'on définit grâce à la bibliothèque *libsx.* Nous avons choisi de créer 4 formes : CARRE, ROND, TRIANGLE, ETOILE, dans un type enum «*FORME* » et une structure « *Forme* » caractérisée par la largeur et hauteur de la forme (*int w,h*), le nombre de points utilisé pour dessiner la forme, le nom désignant cette forme ainsi que la chaine de points constituant la forme graphiquement.

On construit les fonctions utiles de copie des points, de création des chaines de points composant les formes et d'initialisation d'objets « *Formes* » en assignant à chaque *Forme* la chaîne de points qui la construit.

Ce module est construit de façon à faciliter l'ajout de nouvelles formes. En effet, il suffit juste de changer la valeur de la macro NB_FORMES, ajouter le nom de la forme dans l'*enum* FORME et définir une combinaison de points, et tout le programme s'adapte pour afficher la nouvelle forme. Il faut toutefois faire attention à ajouter une couleur supplémentaire ou changer la façon dont la couleur de la carte est calculée, car sinon on risque de se retrouver avec deux paires de cartes qui ont la même forme et même couleur.

Couleur :

Afin de diversifier les cartes pour induire de la vivacité au jeu, nous avons ajouté des couleurs aux formes. Cela ajoute de la difficulté car le joueur va confondre deux cartes qui ont la même forme mais pas la même couleur, ou deux cartes qui ont la même couleur mais pas la même forme.

Ce module est plutôt spécifique à notre utilisation, il ne verra donc une utilité que dans des cas particuliers en dehors du *memory*. En effet, nous avons séparé la couleur d'arrière-plan et celle des cartes en deux et chacune à une méthode d'accès différente.

Couleur d'arrière-plan

L'arrière-plan possède un accès direct par indice. Il faut donc stocker toutes les couleurs relatives au décor dans la variable globale *couleurBg* et accéder à chaque couleur par l'indice qui correspond à la couleur.

Couleur des cartes

La couleur des cartes possède une méthode d'accès particulière. En effet il faut que dans la partie il n'y ait que des pairs uniques de formes et de couleurs, il est donc impératif de développer une formule d'accès aux couleurs qui évite la redondance. Pour décrire cette formule, nous allons prendre l'exemple de 4 couleurs. Il faut générer des permutations de sorte à obtenir la liste suivante :

c1, c2, c3, c4, c2, c3, c4, c1, c3, c4, c1, c2, c4, c1, c2, c3, avec *cx* une couleur, pour un total de 16 couleurs.

L'*id* de chaque carte va permettre d'afficher la couleur correspondante. La formule développée pour obtenir cette liste est la suivante :

(i%size+i/size)%size avec *i* l'*id* de la carte et *size* le nombre de couleurs.

Pour faciliter l'utilisation des couleurs, nous avons défini une structure de tableau comprenant un tableau et sa taille. Cette structure pourrait avoir son propre module avec ses propres fonctions, et ainsi être plus générique, mais nous n'y avons pas vu l'utilité au regard de la simplicité de la structure et de son utilisation uniquement dans le module couleur.

Nous avons aussi décider de créer deux variables globales pour l'arrière-plan et les cartes afin de n'accéder aux couleurs que depuis les fonctions d'accès définies.

Display :

Le module di*splay* regroupe toutes les fonctions d'initialisation de l'affichage, ainsi que toute les fenêtres.

On définit une structure *display* qui regroupe tous les éléments relatifs à l'affichage du jeu. Nous avons la taille de l'écran de jeu (en nombre de cartes), les Widgets relatifs à l'affichage comme les cartes cliquées ou

les panneaux latéraux d'information et de navigation, un autre Widget utile pour la sauvegarde des scores, et un pointeur sur un objet de type Jeu qui consiste en la partie actuelle. Ce pointeur est l'élément qui relie la partie affichage à la partie logique de notre programme.

Pour pouvoir accéder à tout moment aux informations relatives à l'affichage dans notre programme, nous définissons la variable globale *screen.*

Le module nous permet d'initialiser la variable globale et les couleurs utilisées dans l'affichage, de créer et d'afficher le menu, la fenêtre des règles du jeu, la fenêtre des scores, la fenêtre de fin de jeu ainsi que la fenêtre de jeu.

Toutes les interactions vont se faire via le module *callback*.

Callbacks :

Ce module gère toutes les interactions avec l'utilisateur et vérifie que les actions demandées par celui-ci sont valides.

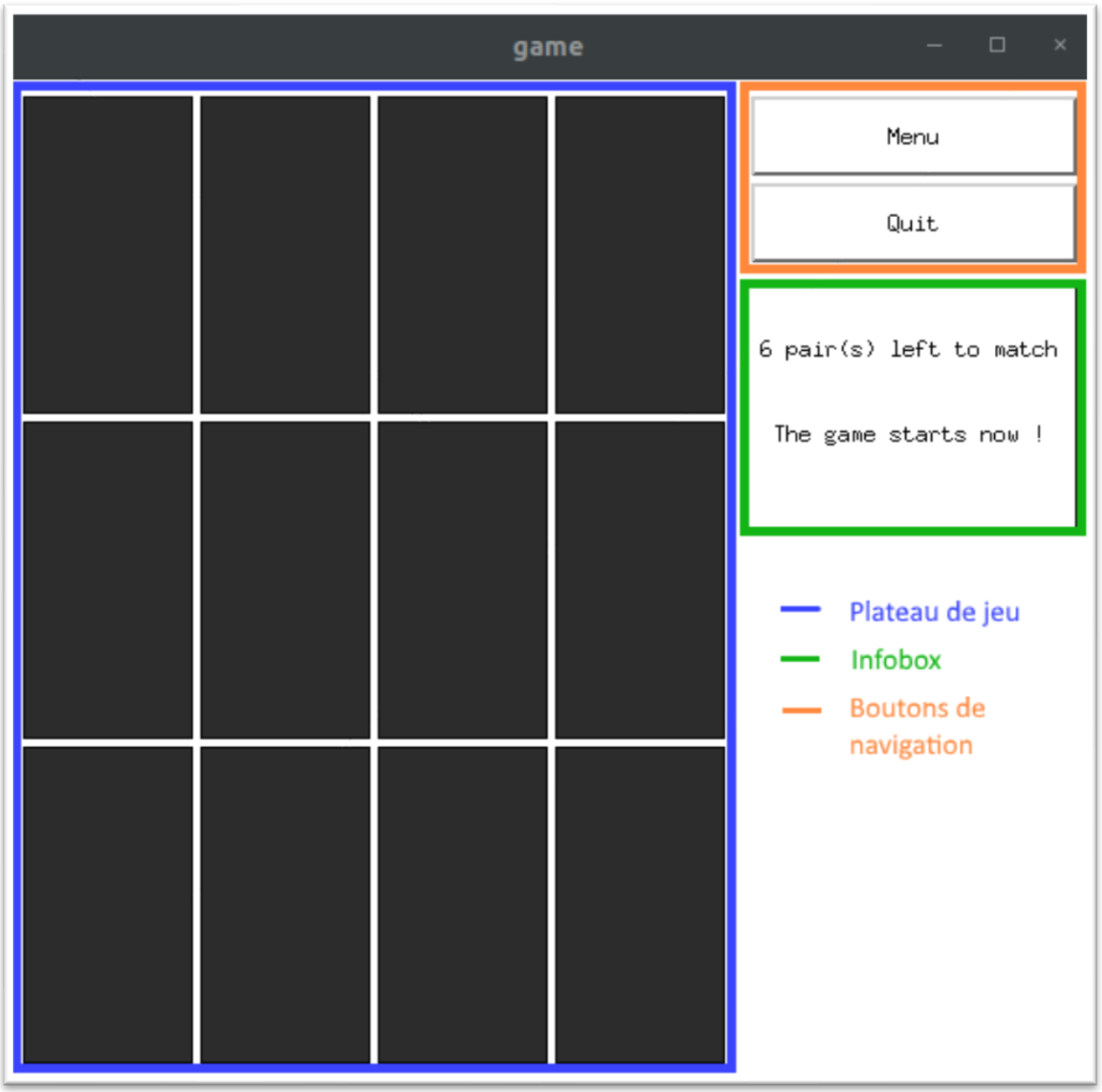
Callbacks va aussi gérer les animations à l'écran.

Depuis ce module on peut :

- Changer la taille de la grille de jeu
- Sauvegarder un score
- Quitter le programme
- Retourner une carte
- Redimensionner la fenêtre sans perdre les cartes retournées.

La fonction *retournerCarte* est centrale dans le fonctionnement correct de l'affichage. Cette fonction utilise la logique de jeu définie dans le module *Jeu* afin de vérifier à chaque clic de l'utilisateur si l'action est valide, pour ensuite jouer un coup et actualiser l'état du jeu en fonction du résultat. Par exemple, si l'utilisateur a déjà retourné une carte et en retourne une seconde, si les cartes sont différentes leur état va repasser en mode CACHE et au prochain coup du joueur leur affichage va se réinitialiser.

Annexe :


game
Menu
Quit
6 pair(s) left to match
The game starts now !
Plateau de jeu
Infobox
Boutons de navigation