API — Projet Dobble

1 Introduction

L'objectif de ce projet est de développer un programme de plus grande envergure que les TP habituels, sous la forme d'une application graphique interactive. Le but du projet est d'implémenter le jeu Dobble, décrit dans les slides en ligne : Lien vers la présentation du projet

2 Présentation du projet

2.1 Installation des dépendances

Contrairement aux TP que vous avez déjà fait, ce projet utilise des bibliothèques logicielles externes. Il s'agit de la `SDL`, une bibliothèque de rendu graphique et de gestion des entrées au clavier et à la souris, utilisée par de nombreux jeux vidéo.

Afin de développer une application utilisant la `SDL`, vous devrez installer les fichiers de développement (en-têtes et binaires) de cette bibliothèque. Vous devrez aussi utiliser `CMake` qui permet de simplifier la compilation de programmes C utilisant des bibliothèques externes, ainsi que `ImageMagick` pour manipuler les images présentes dans le jeu.

2.2 `CMake` et compilation

Une fois ces dépendances installées, il faut exécuter la commande `CMake` pour générer le fichier `Makefile`. Il est de plus préférable de le faire dans un dossier dédié pour ne pas mélanger fichiers de compilation et sources.

```
# Vous devez être placé dans le dossier du projet
$ ls
cmake data header src CMakeLists.txt
# Création du dossier
$ mkdir build
# On se place dans le dossier pour la compilation
$ cd build/
# Génération des fichiers Makefile
$ cmake ..
# Compilation du projet
$ make
# Exécution du projet
$ ./dobble
```

2.3 Fichiers fournis

Les fichiers `graphics.h` et `graphics.c` contiennent le code de gestion graphique du programme. Ces fichiers manipulent la bibliothèque SDL et vous n'avez normalement pas à les modifier. Leur intérêt est de vous abstraire les aspects graphiques bas niveau.

Le code du jeu est contenu dans les fichiers `dobble.h` et `dobble.c`, vous devez modifier ces fichiers pour implémenter le jeu. Le code du jeu (et la bibliothèque SDL) se basent sur **la programmation événementielle**. En d'autres termes, le programme consiste en une boucle principale (située

dans `graphics.c`) qui appelle d'autres fonctions lorsque des événements surviennent (situées dans `dobble.c`), par exemple un clic de souris.

La liste suivante décrit les événements à traiter et les fonctions correspondantes appelées dans `dobble.c`. A noter que certains événements sont indépendants de l'utilisateur, par exemple les événements de timer ou les demandes de raffraîchissement.

— Mouvement de la souris : `onMouveMove(x, y)`
— Clic d'un bouton de souris : `onMouseClick()`
— Seconde écoulée (si le timer est activé) : `onTimerTick()`
— Demande de raffraîchissement : `renderScene()`

3 Objectifs

Les points suivants doivent être implémentés pour le rendu final. Pour implémenter ces fonctionnalités, vous devrez utiliser des structures de données que vous définirez. **Dans un projet, il faut toujours commencer par définir les structures de données nécessaires avant de commencer l'implémentation**. Certaines étapes nécessitent de se référer à des compléments présentés dans la section "compléments techniques".

3.1 Fonctionnalités obligatoires

Les fonctionnalités suivantes doivent obligatoirement être implémentées au cours du projet. Ces fonctionnalités sont la base du jeu Dobble.

Etapes de jeu 3 étapes de déroulement du jeu doivent se distinguer : *Début jeu*, *Jeu* et *Jeu terminé*. Avant le début du jeu, le compteur n'est pas modifié. Le jeu démarre lorsque le joueur clique sur une carte, et se termine lorsque le compteur arrive à 0. La fin du jeu doit être signalée au joueur, qui doit alors pouvoir relancer une partie.

Lecture des fichiers de carte Le contenu des cartes (numéros d'icônes) doit être lu à partir d'un fichier dont le nom est passé en argument au programme. Ces fichiers texte contiennent une carte par ligne, les icônes étant représentées par des numéros. La première ligne du fichier contient le nombre de cartes ainsi que le nombre d'icônes par carte.

Sélection d'une carte Les deux cartes (haut et bas) sont sélectionnées aléatoirement parmi le jeu de cartes. Les cartes du haut et du bas doivent être différentes. Lors d'un changement de cartes, les nouvelles cartes ne doivent pas être égales aux cartes précédentes.

Placement des icônes Les icônes doivent être placées de façon aléatoire sur les cartes de jeu.

Clic sur une icône (*picking*) Lorsque le joueur clique dans la zone d'une carte, le programme doit déterminer si une icône a été cliquée. Si l'icône cliquée est la bonne (icône en commun des deux cartes), le joueur voit son score augmenter et 3 secondes supplémentaires lui sont attribuées. Sinon le joueur perd 3 secondes.

Dans un premier temps, vous pouvez vous contenter de parcourir les icônes des cartes pour vérifier si le joueur a cliqué sur l'une d'entre elles. Dans un second temps, un autre algorithme plus élaboré pourra être implémenté (voir Section "Fonctionnalités additionnelles").

Compte à rebours Le compteur de temps doit être affiché et cet affichage doit être mis à jour lorsque le compteur change. Le compteur est décrémenté à chaque seconde, et est modifié lorsque le joueur clique sur une icône.

Score Le score du joueur doit être affiché et mis à jour lorsque le joueur clique sur une icône.

3.2 Fonctionnalités additionnelles

Si vous avez implémenté toutes les fonctionnalités obligatoires, vous êtes libres d'ajouter des fonctionnalités supplémentaires. Voici quelques idées :

Positionnement complexe d'icônes Vous pouvez modifier le positionnement des icônes afin de rendre le jeu plus complexe. Vous pouvez par exemple varier les échelles des icônes, leur espace etc... Attention, les icônes ne doivent pas se recouvrir.

Personnalisation des icônes Ajouter une gestion des cartes d'icônes, de sorte que votre programme puisse prendre en argument une carte d'icônes à utiliser. Vous pouvez créer des cartes d'icônes vous-memes à partir des conseils données dans la Section "Préparation des images". Attention toutefois à la taille de l'archive de rendu.

Statistiques Collecter et afficher des statistiques sur le jeu, par exemple sur le score ou le temps moyen pour localiser les icônes. Plus intéressant, afficher des statistiques et des courbes de temps moyen de localisation d'icônes en fonction de leur placement et de leurs caractértiques (angle, position...).

Version "old school" Proposer une version en noir et blanc de Dobble, et tester sa difficulté comparativement à la version couleur.

Picking élaboré Au lieu de parcourir les icônes afin de trouver celle sur laquelle a cliqué le joueur, une idée serait de trier ces icônes selon leurs coordonnées, puis de trouver celle sur laquelle l'utilisateur a cliqué en effectuant une recherche dichotomique. L'idée serait d'utiliser deux listes, une contenant les références aux icônes triée par coordonnée x et l'autre par coordonnée y.

4 Rendu

4.1 Date

Le projet devra être rendu avant **le 21 Décembre à 23h**.

4.2 Contenu

Votre rendu devra être constitué de :
- Tout le code source fourni et complété, dont l'archive a été générée avec `make package_source` **impérativement** (voir ci-après).
- Un rapport de 3 pages maximum décrivant :
  - Les solutions choisies pour les différentes parties du projet
  - Les difficultés rencontrées et les solutions retenues
  - Le détail du temps alloué à chacune des parties du projet

4.3 Création de l'archive de rendu

`CMake` peut aussi être utilisé pour générer des archives contenant le code source d'un projet pour le partager. Nous utiliserons cette fonctionnalité pour le rendu. Pour générer votre archive de rendu, utilisez les commandes suivantes :

```
# Votre compte-rendu doit être au format PDF, dans le dossier du projet
# Par exemple, "rapport.pdf"
$ ls
build  cmake  data  header  src  CMakeLists.txt  rapport.pdf
# La création d'archive se fait avec CMake + make
$ cd build/
$ make package_source
[...]
# L'archive a été créée
$ ls
...  2017.12-dobble-rendu.tar.gz  ...
```

5 Compléments techniques

5.1 Installation des outils

Selon la distribution que vous avez choisie, une des commandes suivantes devrait vous permettre d'installer SDL 2, cmake et ImageMagick :

Debian — Ubuntu

```
$ sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-ttf-dev \
        cmake imagemagick
```

CentOS — Fedora

```
$ sudo yum install SDL2-devel SDL2_image-devel SDL2_ttf-devel \
        cmake ImageMagick
```

Cygwin

Paquets à installer avec l'installeur, `cyg-get` ou `Chocolatey`. Notez qu'un serveur X11 doit aussi être installé et en cours d'exécution pour faire fonctionner le projet (paquet `xinit`).

```
libSDL2-devel libSDL2_image-devel libSDL2_ttf-devel cmake ImageMagick
```

5.2 Structures et pointeurs

Voir le TP 6.

5.3 Lecture d'un fichier en langage C

La gestion de fichiers en langage C se fait par l'intermédiaire de la structure `FILE`, située dans `stdio.h`. Cette structure contient entre autres la position en cours du "curseur" dans le fichier. Le contenu de cettre structure ne doit pas être manipulé directement, il faut passer par des fonctions dédiées présentées ci-après.

`FILE* fopen(const char* filename, const char* mode)` permet d'ouvrir un fichier. Cette fonction renvoie un pointeur vers une structure `FILE` ou `NULL` si l'ouverture a échoué. **Il faut toujours vérifier que le fichier est ouvert avec succès en testant la valeur de retour.** La fonction prend en paramètre le nom du fichier, ainsi qu'une chaine de caractères indiquant le mode d'ouverture du fichier. Ce mode d'ouverture peut par exemple être `''r''` pour lecture (read), ou `''w''` pour écriture (write). D'autres modes d'ouverture sont possibles, toutefois ils ne devraient pas être utiles pour ce projet. Référez-vous à la documentation de la fonction pour plus d'information.

Exemple d'utilisation :

```
FILE* f = fopen(''fichier.txt'', ''r''); // ouvre le fichier ''fichier.txt'' en lecture
if (f == NULL) {
   // erreur d'ouverture
   printf(''Echec d'ouverture du fichier.\n'');
}
```

`int fscanf(FILE* stream, const char* format, ...)` permet de lire dans un fichier. Cette fonction est similaire à `scanf`, elle prend en paramètre un lien vers un fichier ouvert, un format et une liste d'arguments (noté par `...`). Elle renvoie le nombre d'élements lus.

Exemple d'utilisation :

```
int i, j, nbRead;
/* lit deux entiers dans le fichier et les place dans les variables i et j. f doit être
ouvert en lecture. */
nbRead = fscanf(f, ''%d %d'', &i, &j);
if (nbRead != 2) {
   // erreur si deux entiers n'ont pas été lus
   printf(''Erreur de lecture.\n'');
}
```

`void fclose(FILE* file)` permet de fermer un fichier préalablement ouvert. **Un fichier doit toujours être refermé lorsqu'il n'est plus utilisé.**

Exemple d'utilisation :

```
fclose(f); // ferme le fichier f préalablement ouvert.
```

5.4 Préparation des images

Ce projet utilise de nombreuses icônes à afficher sur les cartes. Afin de stocker les images de manière efficace, il est préférable de combiner toutes les icônes en une matrice d'images, qui sera la seule image chargée par le programme (voir figure 1). Le programme n'a ensuite qu'à dessiner la portion de la matrice d'image qui correspond à l'icône devant être dessiné.

La commande `montage` du paquet `ImageMagick` permet de créer ce type de matrices à partir d'un

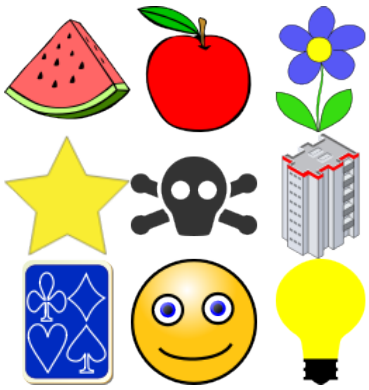

FIGURE 1 – Une matrice d'icônes

dossier contenant des images. Dans les sources du projet, un script shell est fourni pour effectuer le montage correspondant.

```
cd data/
# Assembler en matrice d'icônes de 90x90 pixels les fichiers
# dans OPEN_CLIP_ART
./Assembler90x90.sh OPEN_CLIP_ART/*
```