DEAC Alexandra
ROGER Clément
EISE 4

PROJET C++

Livin' Tomorrow

2020 - 2021

Mme Braunstein

SOMMAIRE

Description de l'application

- Synopsis
- Diagramme UML

Implémentation

- Mini-jeux
  - Le jeu des bâtonnets
  - Le démineur
  - Le taquin
  - MiniJeuAvecSolution
    - Le Juste prix
    - Le nombre manquant
    - Le pendu
- Jeu

Guide d'utilisation

- Procédure d'installation de la librairie SFML
- Procédure d'exécution

Description de l’application

L’objectif de ce projet est de réaliser une application dont le thème principal est le monde d'après. Pour le développement de cette application, certaines contraintes ont été fixées par l’enseignant ce qui nous a amené à avoir l’idée du développement d’un jeu vidéo de réflexion.

Synopsis

Notre jeu s’intitule *Livin’ Tomorrow* et prend place en 2107, alors que le monde est surpeuplé. La solution trouvée par les gouvernements en place est donc de faire passer à chaque individu de plus de 16 ans un test, que seule 5% de la population parviendrait à réussir. En cas d’échec, la personne sera exécutée sur le champ.

Notre jeu se compose donc de six mini jeux apparaissant dans l’ordre suivant :

- Le Juste Prix
- Le pendu
- Le jeu des bâtonnets
- Le jeu du nombre manquant
- Le démineur
- Le taquin

Le jeu comporte également deux niveaux de difficulté (facile et difficile) permettant de rallonger sa durée de vie et de permettre à tout le monde d’y jouer.
En cas d’échec à l’un des mini-jeux, la partie est perdue et il faut recommencer du début.
Concernant les mini-jeux avec solution (Pendu, Juste Prix, Nombre Manquant) il est possible de décommenter les lignes affichant la solution dans le Terminal (présentes dans les constructeurs) afin de faciliter la correction.

Diagramme UML

Notre diagramme UML (présent dans *Rapport/UML.xmi*) se compose de dix classes (*Jeu*, *MiniJeu*, *MiniJeuAvecSolution*, *TextEntry* ainsi qu'une classe pour chaque mini jeu) et de, au plus, quatre niveaux de hiérarchie (par exemple la classe *Pendu* dérive de *MiniJeuAvecSolution* qui elle même dérive de *MiniJeu,* elle même dérivant de *Jeu*).

D'autres fichiers sources (n'étant pas des classes) sont aussi présents dans le dossier *Jeu*, c'est le cas de tous les fichiers *Vue[...].cc* (*VueDemineur.cc, VueTaquin.cc ...)*. Ces fichiers implémentent toutes les fonctions relatives à l'interface graphique des mini-jeux en question et sont utiles afin de séparer les parties graphiques et jouables.

On peut aussi voir sur le diagramme UML l'utilisation d'une classe abstraite ITransition qui joue le rôle d'interface. Cette interface a été ajoutée car tous les objets de notre jeu utilisent une méthode de transition permettant de réaliser la transition graphique entre chaque jeu. Cette méthode est virtuelle pure, ne renvoie rien et prend en argument un objet de la librairie SFML, un sf::Render& window. Ce sont l'objet Jeu et l'objet abstrait MiniJeu qui implémentent cette interface graphique.

Implémentation

Au niveau de l'implémentation, nous sommes partis du diagramme UML précédemment présenté que nous avons réalisé en respectant les différentes contraintes définies par l'enseignant.
Nous allons ici apporter une petite description du fonctionnement de chaque objet en mettant en évidence les contraintes que nous devions respecter.

Mini-jeux

Notre jeu est constitué de six mini-jeux différents qui partagent des caractéristiques en commun. En effet, on peut soit gagner soit perdre un mini-jeu. De plus, pour que le jeu ne soit pas trop facile, chaque mini-jeu possède un nombre d'essais avant de perdre. Ces observations nous ont conduit à créer une classe abstraite possédant donc deux attributs : un *boolean _victory* qui permet de savoir si on a gagné ou perdu le jeu et un entier *_nb_try* permettant de définir le nombre d'essais pour chaque mini-jeu.

En plus de ces attributs, les mini-jeux possèdent des actions en commun ce qui nous permet de créer 6 différentes méthodes virtuelles :

- virtual void play() = 0 : permet de jouer au niveau fonctionnel
- virtual void display(sf::RenderWindow& window) = 0 : permet de réaliser l'affichage du mini-jeu
- virtual void init_background(sf::RenderWindow& window)const = 0 : permet d'initialiser l'affichage graphique du fond de chaque jeu
- virtual void print_end(const bool win,sf::RenderWindow& window) = 0 : permet de faire l'affichage graphique de la fin de chaque jeu
- virtual void set_final_text(const bool win)const = 0 : permet d'initialiser le texte affiché lors de la transition (en cas de victoire/défaite)
- virtual void init_transition(sf::RenderWindow& window) = 0 : permet d'initialiser le fond de la transition

Ces méthodes virtuelles seront toutes implémentées dans les mini-jeux. On n'oublie pas aussi de redéfinir la fonction venant de l'interface *Itransition*. Comme cette fonction est la même pour chaque mini-jeu, nous l'implémentons directement dans cette classe.

Afin de tester tous les mini-jeux et s'assurer de leur bon fonctionnement, nous avons également implémenté des *testcase* pour chacun. Ces derniers testent toutes les fonctionnalités (non graphiques) des mini-jeux et sont présents dans le dossier *TestCase* où l'on peut les lancer en faisant la commande *make* puis *./testcase* .

Le jeu des bâtonnets

Ce mini jeu est le seul des six à se jouer face à un adversaire. Le principe est le suivant : nous avons devant nous vingt bâtonnets, chaque joueur prend tour à tour un, deux ou trois bâtonnets, le joueur qui prend le dernier bâtonnet perd la partie.
Ce jeu ne comporte pas de surcharge d'opérateurs ni de conteneur de la STL, néanmoins il comporte une IA afin de simuler le joueur adverse (appelé Eddy dans l'histoire). Le principe est plutôt simple : lorsque la difficulté est facile, Eddy prend un nombre aléatoire entre 1 et 3 à chaque fois (sauf lorsqu'il peut gagner il prend le nombre de bâtonnets adapté). Lorsque la difficulté est difficile, Eddy prend (4 - n) bâtonnets avec n le nombre de bâtonnets que le joueur adverse (l'utilisateur) a pris (cette stratégie est l'une des stratégies optimales pour gagner selon certaines conditions).

Le démineur

Le principe du jeu du démineur est relativement simple. On a une grille de 8x8 où sont cachées 8 à 10 bombes. A chaque fois que l'on clique sur une case, si ce n'est pas une bombe, on affiche combien de bombes sont adjacentes à celle-ci. Si on parvient à découvrir toutes les cases sans toucher les bombes, le jeu est gagné.

Au niveau de l'implémentation, on utilise un *vector* de *vectors* d'entiers, *_plateau*, pour représenter le plateau de jeu. En effet, on a décidé de représenter le plateau avec le code suivant : -1 représente une bombe et les chiffres de 0 à 8 représentent le nombre de bombes adjacentes à la case.
Chaque plateau de jeu est généré aléatoirement. En effet, à chaque création du jeu, on tire de manière aléatoire la position des bombes dans la grille.
De plus, pour mémoriser quelles cases ont été ouvertes, on décide d'utiliser un *vector* de *vectors* de *booléens*, *_open_tiles* , où on a : *true* pour une case ouverte et *false* pour une case pas encore ouverte.

Comme nous avons réalisé une interface graphique, dans la partie fonctionnelle nous récupérons, après avoir testé leur validité, les coordonnées du clic de l'utilisateur. Celles-ci sont stockées dans un

conteneur de la STL nommé *Pair* ou le membre *first* correspond à la position du click en longueur et le membre *second* correspond à la position du click en hauteur.
On teste ensuite si la case à déjà été ouverte ou non à l'aide de la variable _open_tiles, si elle a déjà été ouverte on ne fait rien, sinon on l'ouvre.

Lorsque l'on tombe sur une bombe, on perd une vie. En fonction du niveau de difficulté choisi, on a une ou trois vies. Si on ouvre toutes les cases du plateau sans perdre toutes ses vies, on gagne et on peut passer au mini-jeu suivant.

Nous avons ajouté la surcharge de l'opérateur << pour simplifier l'affichage du plateau de jeu dans le terminal. Cela peut-être utile pour avoir la solution du jeu si jamais on est bloqué, car on pourra voir ou se trouve les -1 (bombes).

Le taquin

Le jeu du taquin se compose de huit tuiles composant une image et d'une case vide. Le but du jeu est de reconstituer l'image en déplaçant une des cases adjacentes à la case vide dans celle-ci, le tout en moins de *nb_try* tentatives (valant *150* en mode facile et *100* en mode difficile). Plusieurs grilles sont disponibles et sont présentes dans le fichier *grilles.txt* d'où une est chargée aléatoirement à chaque début du mini jeu.
Ce mini jeu comporte notamment trois *vectors* contenant respectivement les *paths* vers les huit images, les positions des huits cases ainsi que le vecteur solution.
Nous avons également surchargé l'opérateur == afin de pouvoir comparer deux vecteurs et donc d'écrire facilement la comparaison entre le vecteur position et le vecteur solution afin de tester la victoire du joueur.
Le principe de l'implémentation est plutôt simple : à chaque clic de l'utilisateur non récupérons le numéro de l'image sur laquelle il a cliqué puis nous vérifions qu'elle est adjacente à la case vide. Si oui, nous échangeons les coordonnées de la case vide et de la case déplacée puis nous testons si le vecteur position est égal au vecteur solution.

MiniJeuAvecSolution

Plusieurs mini-jeu possèdent une solution et implémentent l'objet TextEntry, qui est un objet SFML créé par nos soins et qui permet de faire une boîte de saisie de texte pour l'utilisateur.
On a alors décidé de créer cette classe abstraite qui possède deux arguments qui sont des strings : _solution et _user_entry et qui définit une fonction virtuelle pure "void check_entry(const string entry)const = 0" qui permet de vérifier que l'entrée utilisateur est correcte.

Le Juste prix

Ce mini jeu (aussi appelé jeu du plus ou moins) consiste tout simplement à trouver un nombre aléatoire entre N_{min} et N_{max} (ici valant respectivement 1 et 100). Pour cela, l'utilisateur propose un nombre auquel l'IA ne répondra que par "C'est plus", "C'est moins" ou "C'est gagné". La difficulté du jeu repose dans le nombre de propositions autorisées : 8 lors du mode facile et 6 lors du mode difficile.
Concernant l'implémentation, elle n'a rien de compliquée : nous testons tout d'abord l'entrée de l'utilisateur (si c'est bien un entier entre 1 et 100) puis nous traitons la réponse avec un *if* tout simplement.

En cas de difficulté pour passer le jeu, il est possible de décommenter la ligne 15 du fichier *Juste_Prix.cc* pour que la solution du mini-jeu s'affiche dans le terminal.

Le nombre manquant

Ce mini-jeu stimule la réflexion et la logique. Le principe est simple, une grille 3x3 va s'afficher à l'écran. Dans chaque case on pourra trouver un nombre. Cette matrice constitue une suite logique ou il manque un chiffre, représenté par un "?" dans l'affichage graphique. En fonction de la difficulté choisie, vous avez une ou trois chances pour trouver le nombre manquant pour résoudre cette suite.

Au niveau de l'implémentation, nous utilisons un *vector* de *vectors* de *strings* qui va contenir la suite logique. Pour diversifier les suites logiques, on a à notre disposition un fichier contenant différentes matrices et leur solution. A la création de l'objet on parcourt ce fichier et on choisit une matrice de manière aléatoire.
Au niveau de la fonction *play()*, l'implémentation est simple, on commence par tester si l'entrée utilisateur est bien un nombre. Si c'est le cas, on compare l'entrée utilisateur avec la solution. La victoire intervient si la solution et l'entrée utilisateur sont égales et que le nombre d'essais n'est pas nul.

En cas de difficulté pour passer le jeu, il est possible de décommenter la ligne 20 du fichier *missingnumber.cc* pour que la solution du mini-jeu s'affiche dans le terminal.

Le pendu

Lors du début d'une partie, on voit affiché à l'écran un mot contenant un certain nombre de lettres et dont seule la première lettre est dévoilée. Le but du jeu est donc de retrouver le mot caché. Pour cela, vous pouvez proposer des lettres ou un mot directement si vous pensez connaître la réponse. Mais attention, vous avez un nombre limité de chances, déterminé par le niveau choisi. Ce serait trop simple sinon.

Au niveau de l'implémentation, on initialise l'attribut *_solution*, hérité de la classe *MiniJeuAvecSolution*, en parcourant un fichier et en sélectionnant un mot de manière aléatoire. On a à notre disposition deux fichiers : un avec des mots simples pour le niveau facile et un avec des mots (beaucoup) plus compliqués pour le niveau difficile.
De plus, on utilise un *vector* de *strings* : *_used_letters*, qui permet de contenir les lettres déjà essayées par l'utilisateur pour les afficher ensuite à l'écran. De la même manière, pour sauvegarder les lettres trouvées par l'utilisateur, nous conservons un attribut *_found_letters* qui est un *vector* de *booléens* de la taille du mot à trouver.

Comme pour les autres mini-jeux, nous avons réalisé une interface graphique. Au niveau du jeu, nous commençons par récupérer l'entrée de l'utilisateur dans l'argument *_user_entry* héritée de la classe abstraite *MiniJeuAvecSolution*. On vérifie que l'entrée est bien une lettre ou un mot.
Si on se retrouve face à un mot, on compare ce mot avec la solution, s'ils sont égaux, le joueur à gagné. Sinon, on diminue son nombre d'essais.
Au niveau d'une lettre, on vérifie si cette lettre appartient à la solution. Si elle appartient à la solution, et qu'elle n'a pas déjà été découverte, on l'affiche à l'utilisateur. Sinon on ne fait rien. Si elle n'appartient pas à la solution, on décrémente le nombre d'essais du joueur.

A chaque nouvelle entrée utilisateur, on tient à jour les attributs *_found_letters* et *_used_letters* qui sont notamment utilisés pour réaliser l'affichage graphique.

En cas de difficulté pour passer le jeu, il est possible de décommenter la ligne 25 du fichier *pendu.cc* pour avoir la solution.

Jeu

Au niveau de l'implémentation, la classe Jeu est composée de trois attributs qui sont tous des *std::size_t*. On a *_difficulty* qui permet de savoir le niveau de difficulté choisie par le joueur, *_passedGames* qui permet de savoir le nombre de mini jeux réussis par le joueur et *_playagain* qui permet de savoir si le joueur souhaite rejouer une partie ou non.

Le jeu commence par la sélection du niveau de difficulté puis nous lançons les mini-jeux dans l'ordre évoqué plus haut.

En cas de victoire à un mini-jeu, on incrémente l'attribut *_passedGames* pour qu'à l'aide d'une instruction *switch* on puisse savoir quel jeu lancer ensuite.

En cas de défaite ou de réussite de toutes les épreuves, on se voit proposer la possibilité de rejouer une partie. Si on clique sur “Rejouer”, une nouvelle partie se lance depuis le début (choix de la difficulté). Si on clique sur “Terminer”, la fenêtre se ferme et la partie est terminée.

Comme selon nous un jeu n’est rien sans musique, nous avons ajouté des sons en fond pour que l’ambiance dans laquelle se trouve le joueur soit cohérente avec le synopsis du jeu. Tout d’abord nous avons un son de cloches apocalyptique lors du lancement du jeu (issu de la chanson *Hells Bells* d’*AC/DC*), puis une musique de fond angoissante lorsque le jeu débute. Bien sûr, lorsque nous rejouons la partie, les musiques se lancent à nouveau correctement.
Enfin, nous avons également ajouté une *voix off* lors des affichages des textes (expliquant les règles et lors des victoires et défaites du joueurs) afin de rendre le jeu le plus agréable à jouer possible.

Guide d’utilisation

Pour réaliser l’affichage de ce projet, nous souhaitions offrir au joueur une meilleure expérience utilisateur qu’un affichage sur le terminal, nous avons donc décidé d’utiliser l’interface de programmation écrite en C++, *SFML* (*Simple and Fast Multimedia Library*). Cet ensemble d’outils permet de construire des projets tels que des jeux vidéo ou des programmes interactifs en donnant accès aux modules de fenêtrage, graphique, audio et réseau.

Procédure d’installation de la librairie SFML

L’installation de la librairie SFML est très simple sur un système LINUX et peut se faire de différentes manières. Cependant, comme la version de SFML qu’on a décidé d’utiliser est la plus récente, elle se trouve sur le repository officiel de la SMFL (https://github.com/SFML/SFML) et nous pouvons donc réaliser l’installation en une seule ligne de commande.
Il suffit seulement d’ouvrir un terminal de commande et d’écrire la ligne suivante :

```
sudo apt-get install libsfml-dev
```

Lorsque la commande est lancée, les différents packages s'installent. Une fois que c'est terminé, le job est fini. On peut utiliser la librairie SFML.

Procédure d'exécution

Comme nous utilisons une librairie, la compilation d'un programme ne se fait pas exactement comme un programme classique. En effet, nous allons devoir ajouter les librairies à lier à notre programme pour que celui-ci fonctionne.

Prenons l'exemple d'un fichier "*main.cc*" et détaillons les étapes de la compilation. Tout d'abord on commence par compiler :

```
g++ -c main.cc
```

Puis, on lie notre fichier compilé avec les modules SFML qu'on utilise dans le programme, par exemple les modules audio, graphiques et fenêtres :

```
g++ main.o -o executable -lsfml-audio -lsfml-graphics -lsfml-window -lsfml-system
```

Par la suite, il suffira d'exécuter le programme avec la commande ./executable.

Pour la compilation de notre projet, nous avons mis dans notre repository Github un Makefile permettant de compiler tous le projet à l'aide d'une simple commande "make". L'exécutable du programme s'appelle "Jeu", il faut donc lancer la commande ./Jeu pour exécuter le programme.