EI-SE 4

Language Objet C++

COMPTE RENDU

Mini-Projet «Coupe du monde »

Laith ALKHAER - Alexandre VAUDELLE

27/01/2023

Introduction

Dans le cadre de la seconde année de notre formation centrée sur l'informatique et l'électronique, il nous est proposé un projet nous permettant de conclure ce module essentiel et de mettre en pratique nos compétences en programmation orientée objet en langage C++ acquises au cours de ce semestre, en réalisant pour la première fois une application simple mais complète.

Description de l'application

Le choix du type de l'application étant libre, nous avons décidé de faire un jeu simple de type «quiz» ou questions-réponses portant sur la coupe du monde de football 2022 et des sujets avoisinants avec une approche cynique et humoristique, du moins selon notre humour parfois discutable…

Le jeu comporte donc quatre types de questions: des questionnaires à choix multiple (QCM ou MCQ en anglais), des questions vrai-ou-faux (True-False en anglais) et des QCMs à photo et des questions ou il faut saisir une réponse à une question simple au clavier

Une série de questions des quatre types est proposée au joueur qui doit choisir une seule réponse à chaque question et à la fin de la partie un score sur 20 est affiché.

Installation

Pour installer le jeu, il suffit de cloner sur une machine Linux le répertoire Github du jeu avec la commande suivante:

git clone https://github.com/LaisA01/ALKHAER-VAUDELLE-Projet_CPP_EISE4

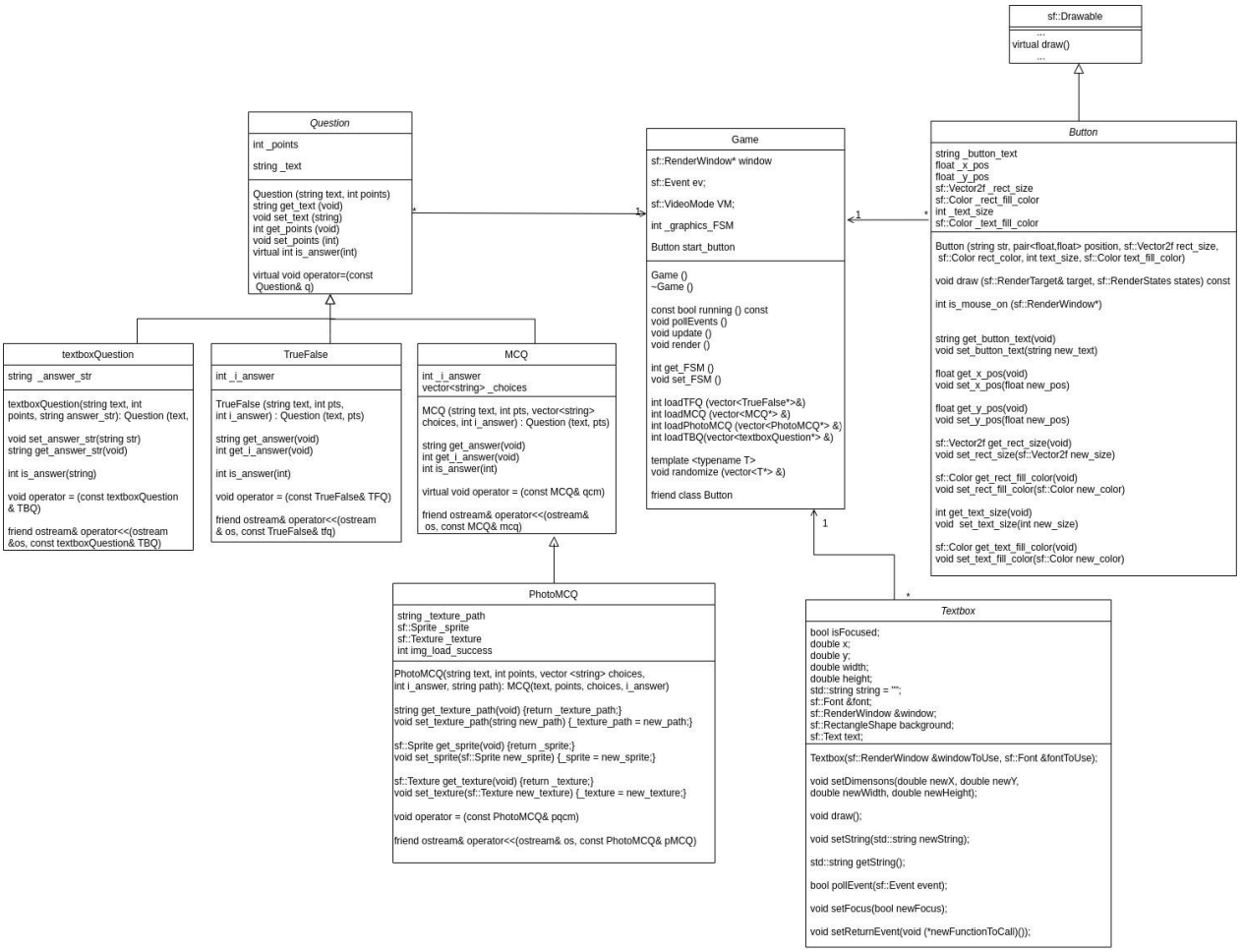
Le jeu étant dépendant de la librairie graphique SFML pour ses graphismes, il faut donc que cette librairie soit déjà installée sur la machine Linux avant de lancer le jeu. Si vous ne l'avez pas déjà fait, vous pouvez l'installer sur Linux avec la commande:

sudo apt-get install libsfml-dev

Enfin, il ne reste plus qu'à compiler le jeu en exécutant le fichier make fournit dans le répertoire avec la commande **make**. Le jeu est désormais installé et son exécutable porte le nom **quizz.out**.

Diagramme UML

Vous trouverez une version potentiellement plus lisible en consultant directement ce lien:
https://drive.google.com/file/d/1nPBEB1Krt7oDSQraXnuM632RiSvaMSZH/view?usp=sharing



Bilan de l'utilisation des contraintes:

Nous avons respecté le plus possible les contraintes imposées par le sujet. Voici résumé de cela:

- Contrainte: 8 classes:

Comme vous le voyez sur le diagramme de classes ci-dessus, nous avons bien implémenté 8 classes dans notre conception: une classe principale *Game*, 5 classes pour les questions sur 3 niveaux d'hiérarchie et 2 classes utilitaires pour les boutons et les boîtes de texte.

Nous tenons à préciser que le code de la classe *Textbox* qui gère - vous l'aurez deviné - les boîtes de saisie de texte a été récupéré sur un répertoire GitHub publique. Nous aurions pu le coder par nous-mêmes comme nous avions déjà fait pour les boutons mais, par souci de

temps, nous avons décidé de ne pas le faire puisque ce module C++ ne porte pas sur la programmation graphique.

- Contrainte 3 niveaux de hiérarchie:

Comme vous le voyez sur le diagramme de classes ci-dessus, nous avons bien trois niveaux hiérarchie: la classe *PhotoMCQ* hérite de la classe *MCQ* qui elle hérite de la classe abstraite *Question*.

- Contrainte: 2 fonctions virtuelles différentes et utilisées à bon escient:

➔ Dans notre classe *question*: On a fait d'une pierre deux coups en implémentant une **surcharge d'opérateur virtuelle,** implémenté aussi dans les classes qui héritent de *question* et qui sert donc à copier une question dans une autre:

```
virtual void operator=(const Question& q)
{
        _points = q._points;
        _text = q._text;
}
```

➔ Dans la classe *Question*, une fonction virtuelle qui prend l'indice d'une réponse et retourne 1 ou 0 si la réponse est correcte ou incorrecte:

```
virtual int is_answer(int i);
```

Nous estimons que ces fonctions virtuelles sont utilisées à bon escient car les différentes classes des différentes types de questions n'ont pas les mêmes membres donc il faut adapter ces méthodes pour les rendre compatibles.

- Contrainte: 2 surcharges d'opérateurs:

➔ Dans nos classes *MCQ*, *TrueFalse* et *textboxQuestion*: une surcharge d'opérateur qui retourne écrit dans le flux toutes les informations de la question. Cette surcharge d'opérateur nous a fait gagner du temps dans le débug:

```
friend ostream& operator<<(ostream& os, const textboxQuestion& tbq)
{
    os << tbq.get_text() << '/' << "points: "<< tbq.get_points() <<
    '/'<< "answer: " << tbq._answer_str << std::endl;
    return os;
}
```

➔ Dans notre classe *question*: On a fait d'une pierre deux coups en implémentant une **surcharge d'opérateur virtuelle,** implémenté aussi dans les classes qui héritent de *question* et qui sert donc à copier une question dans une autre:

```
virtual void operator=(const Question& q) //surcharge d'operateur
d'affectation
   {
       _points = q._points;
       _text = q._text;
   }
```

- Contrainte: 2 conteneurs différents de la STL:

➔ std::vector:

Évidemment, nous avons utilisé le conteneur de la STL le plus utile sans modération. Par exemple, dans le fichier *game.cpp* le stockage des questions du quizz dépend sur une collection de vecteurs:

```
vector<Button> current_buttons_list;
vector<Button> current_buttons_list_PhotoMCQ;


vector<MCQ*> MCQ_vector;
vector<TrueFalse*> TF_vector;
vector<PhotoMCQ*> PhotoMCQ_vector;
```

➔ std::pair:

Nous avons utilisé le conteneur *std::pair* dans un cas d'utilisation très classique: la manipulation de coordonnées et de dimensions. Vous pouvez en trouver dans le fichier *Button.h* qui contient le code de notre classe bouton:

```
class Button : public sf::Drawable
{
private:

    pair <float, float> _button_pos; //abscisce, ordonnée du coin en haut à droite
    pair <float, float> _rect_size;  //dimension du cadre du bouton
```

- Contrainte: diagramme de classe UML complet: cf diagramme ci-dessus.

- Contrainte: commentaire du code: cf. code.

- Contrainte: pas de méthodes/fonctions de plus de 30 lignes: la méthode la plus longue *Game::pollEvents()* fait 29 lignes sans compter les *case* et les *break;* (on a fait un *switch* imbriqué donc il y en avait trop pour les inclure dans le compte).

- Contrainte: utilisation d'un Makefile avec une règle "all" et une règle "clean": cf makefile fourni.

Description du code et fiertés

Comme vous pouvez le constater en consultant le diagramme UML , le jeu est construit autour d'une classe *Game* dont les fonctions qui mettent à jour la logique du jeu et l'affichage *SFML* sont membres publiques. Pour faire tourner le jeu, il suffit donc d'appeler ces fonctions en boucle jusqu'à la fin de la partie dans la fonction *main*

D'un point de vue POO, cette classe Game est composée de *Questions* répartis sur des sous-classes respectives en fonction du type, ainsi que des *Buttons* pour les réponses de ces *Questions*.

Nous sommes particulièrement fiers de l'organisation de notre code qui s'inspire de celle des jeux beaucoup plus complexes. Cette organisation permet une grande modularité, quelque chose que nous avons beaucoup apprécié lors du développement de ce jeu car nous avons changé d'avis sur nombre d'aspects en cours de route. Par exemple, lorsque nous rencontrions des bug nous pouvions très rapidement discerner les bugs dans la logique du jeu des bugs graphiques puisque ces deux aspects sont gérées par deux fonctions indépendantes (*game::update()* et *game::render()* respectivement) qui communiquent entre elles via des flag globaux et des membres de l'instance de *Game*.

Autre exemple: lorsque nous cherchions à tester de nouveaux types de questions, il suffisait de bloquer un seul flag pour contourner les fonctionnalités déjà testées et ne tester que les nouvelles.

Toujours sur le même sujet de modularité, nous avons créé, au tout début du projet, une classe *Button* qui gère tous les boutons du jeu, ce qui nous a fait économiser un temps considérable vu le nombre de boutons différents que nécessite un jeu de type quizz.

Enfin, pour conclure sur les fiertés, nous apprécions beaucoup les sons et la musique que nous avons ajouté au jeu qui améliorent considérablement l'expérience utilisateur:

```
this->_click_sound_buffer.loadFromFile("audio/click.wav");
this->_click_sound.setBuffer(_click_sound_buffer);

this->_back_ground_music.openFromFile("audio/background_music.wav");
this->_back_ground_music.setVolume(20);
this->_back_ground_music.setLoop(true);
this->_back_ground_music.play();
```

Conclusion

Ce projet fut une expérience de programmation essentielle pour notre formation, notamment en nous permettant de résumer tout une matière qui nous a été enseignée sur plusieurs mois en un seul projet.

«Je suis fier d'avoir renouer avec le C++, genre le projet est pas dingue, mais d'un pt de vu introspection j'ai compris beaucoup de chose, ce que j'aime réellement dans l'informatique c'est le debug en réalité» - Alexandre VAUDELLE (2023)
