SORBONNE UNIVERSITÉ

POLYTECH SORBONNE

C++ : Projet S7

Karim EL-KHARROUBI -
Vincent LEFEBVRE
EISE 4

INTRODUCTION

Le monde d’après est ici prédit comme étant peu optimiste pour les humains. Suite à la découverte récente de son existence par une IA visant à détecter les chatons dans des images, celle-ci entreprit de libérer ces êtres divins des menaces apparaissant souvent sur les photos qu’on lui soumettait. L’interconnexion généralisée des objets lui permit de rapidement prendre le contrôle des moyens de production, et d’établir une armée pour libérer les chatons.

Ce projet visait à développer un jeu de stratégie au tour par tour. La mise en place d’interfaces graphique a nécessité de suivre les principes de programmation évènementielle.

UTILISATION

Le programme peut être compilé en se placant dans à la racine du projet et en effectuant la commande make. Celui ci peut ensuite être lancé en lançant ./ProjetS7. Une fenêtre s'ouvre alors, on peut quitter l'application à tout moment en appuyant sur la touche Echap.

Un menu principal implémente deux boutons, l'un lançant une nouvelle partie et l'autre permettant de quitter l'application.

Figure 1: Écran d'accueil de l'application

Lors du démarrage d'une nouvelle partie, un nouvel écran apparaît. Il est possible de cliquer sur le sablier en bas à à droite afin de passer au tour suivant.

1 CLASSES

Le design pattern bridge a été utilisé tout le long de ce projet afin de découpler les éléments de jeu (regroupés dans gameLogic/) et les éléments graphiques (regroupés dans graphics/). L'ensemble des diagrammes de toutes les classes est accessible au travers d'une interface générée par Doxygen, accessible en ouvrant docs/html/index.html et en cliquant sur l'onglet classe.

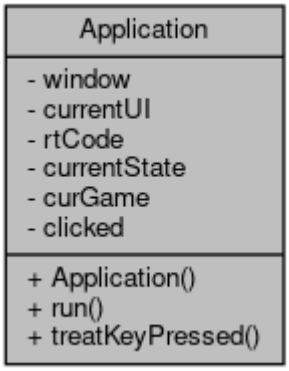
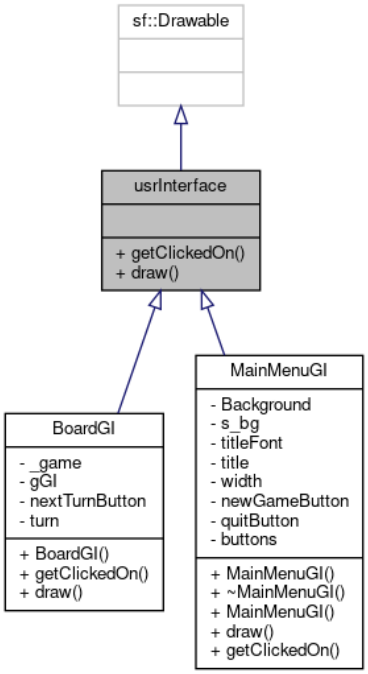


Figure 2: Diagramme de classe de l'application

FIERTÉS

La librairie SFML utilisée ne supporte que l'affichage mono-écran. Une seule interface utilisateur est donc nécessaire. L'interface usrInterface permet le polymorphisme des écrans de l'application grâce à ses méthodes virtuelles.




```
class usrInterface : public sf::Drawable{
    public:
        virtual std::shared_ptr<Clickable> getClickedOn(std::pair<int,int>mouse) const=0;
        virtual void draw(sf::RenderTarget &target, sf::RenderStates states) const=0;
};
```

2 HIERARCHIE

Les deux grandes hiérarchies suivent le principe du pattern Bridge, avec un hierarchie pour les éléments graphiques (fig. 2) et une pour les éléments de jeu (fig.3).

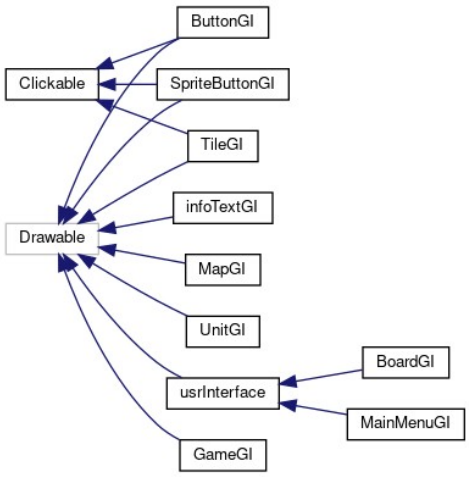


Figure 4: Hiérarchie des classes d'implémentation graphique

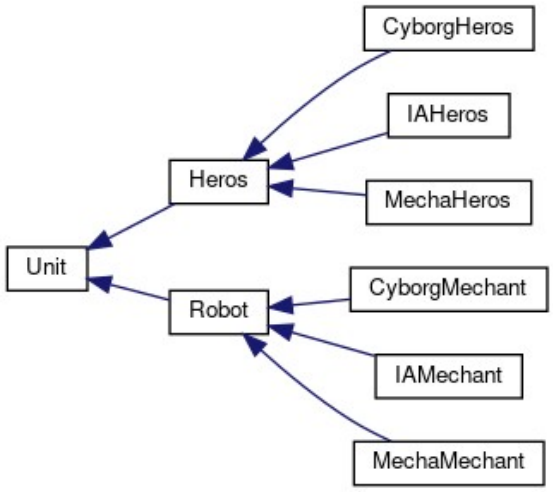


Figure 3: Hierarchie des classes d'unités

3 SURCHARGES D'OPÉRATEURS

Afin de pouvoir observer l'évolution du jeu sans rétroaction graphique, un l'opérateur de flux sortant a été redéfini dans le fichier game.hh à la ligne 28 .

```
friend std::ostream& operator<<(std::ostream& os,const Game& s){
    os << s.toString();
    return os;
}
```

Figure 5: Opérateur << redéfini dans game.hh

L'opérateur – a été rédéfini dans unit.hh à la ligne 21, représentant une diminution des points de vies.

```
Unit* operator-(const int& b) {
    this->hP -= b;
    if(hP<=0)
        delete this;
    return this;
}
```

Figure 6: Surcharge de l'opérateur -=

4 CONTENEURS DE LA STL

Les vecteurs ont été utilisés pour chacune des implémentations graphiques représentant une liste d'élément, par exemple dans le fichier gameGI.hh :

```
class GameGI : public sf::Drawable{

    private:

        MapGI theMap;

        std::vector<Unit> units;

        std::vector<UnitGI> unitsGI;
```

Figure 7: Attributs privés de la classe GameGI

Les paires ont été utilisées pour représenter des coordonnées cartésiennes, par exemple au sein de la fonction createMapGraphs du fichier mapGI.cc

```
void MapGI::createMapGraphs(std::pair<int, int> &northPosition,
                            std::vector<std::shared_ptr<Tile>>::iterator it) {

  std::pair<int, int> nPL = northPosition, max(sqH, sqW);

  int i = 0, j = 0;

  if (sqH > sqW)
    std::swap(max.first, max.second);

  for (j = 0; j < max.first; j++) {
    northPosition = nPL;
    for (i = 0; i < max.second; i++) {
      tls.push_back(TileGI(*it++, northPosition,
                squareSize, outline,j*max.first-1+i));
      northPosition.first -= squareSize - outline;
      northPosition.second += squareSize / 2 - outline / 2;
    }
    nPL.first += squareSize - outline;
    nPL.second += squareSize / 2 - outline / 2;
    northPosition = nPL;
  }
}
```

Figure 8: Fonction utilisant std::vector et std::pair

5 MAKEFILE

Le makefile primaire appelle celui de src/ :

est situé dans le répertoir src/ .

```
#   Compilateur
CC=g++
#   Options de compilation
CCFLAGS=-Wall -Werror -pedantic -std=c++17 -g -fmax-errors=4
#   librairies
LIBFLAGS=-lsfml-graphics -lsfml-window -lsfml-system
#   Ensemble des fichiers cc du projet
SRC= $(wildcard **/**/*.cc) $(wildcard **/*.cc)

OBJ=$(SRC:.cc=.o)

EXEC= ProjetS7
```

Figure 9: Variables du makefile

Les règles suivantes sont implémentées :

```
all: $(EXEC)

$(EXEC): $(OBJ)
    $(CC) $(CCFLAGS) $^ -o $@ $(LIB_PATH) $(LIBFLAGS)

%.o: %.cc
    $(CC) $(CCFLAGS) -o $@ -c $<

.depend:
    $(CC) $(CCFLAGS) -MM $(SRC) > .depends

-include .depends

clean:
    rm -f $(OBJ) $(EXEC) vgcore*

distclean : clean cleantest
```

Figure 10: Règles du Makefile