Rapport pour le projet de INFOF202

Ransy Lenny Lejeune Lucas

Janvier 2024

Ce rapport vise a documenter le projet "Frogger" que nous avons réaliser. Nous traiterons les tâches que nous avons réalisées et comment nous les avons réalisées. Pour cela, nous parlerons des classes que nous avons codées pour faire ce projet, comment elles sont réparties dans le code et comment elles intéragissent entre elles. Nous justifierons aussi comment nous avons utilisé le modèle de conception MVC dans la partie jeu.

Table des matières

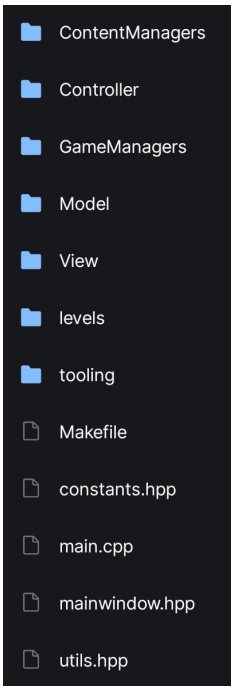
1 Résumé des tâches réalisées et fonctionnement du jeu

Nous avons réalisé les tâches principales et toutes les tâches additionnelles, sauf la tâches de l'éditeur de niveau. Ouvrons le jeu et voyons ce qui se passe.

2 Structure des fichiers: Utilisation du MVC

La structure des fichiers peut se diviser en plusieurs parties: les fichiers du jeu, ceux des menus, les mains et les images du jeu.

Le dossier `ContentManagers` contient tout les fichiers de code en rapport avec la gestion des menus du jeu.



Les dossiers `Controller`, `Model` et `View` contiennent le code parmettant de faire tourner le jeu (la partie plateau, qui ne compte pas le menu). Cette disposition met en évidence l'utilisation du MVC. Prenons la gestion de la grenouille comme exemple. Les fichiers gérant les contrôles de celle-ci se trouvent dans le dossier `Controller`, ceux qui gèrent ses propriétés dans le plateau se trouvent dans `Model` et enfin, ceux qui gèrent son affichage se trouvent dans `View`. Tout le code de ces fichiers est utilisé dans les fichiers de `GameManagers` pour avoir un jeu fonctionnel.

Le dossier `levels` contient les données des niveaux sauvegardés en fichiers `.csv`, et les scores obtenus sur ceux-ci. Plus de détails seront donnés dans les chapitres concernés.

Le dossier `tooling` contient tout les outils construits grâce aux outils venant la librairie FLTK qui sont surtout utilisés dans la partie `View` et `ContentManagers`.

Le fichiers `constants.hpp` contient toutes les constantes utilisées dans le projet, comme par exemple la taille des bouttons dans le menu.

Enfin, les fichiers `main.cpp` et `mainwindow.hpp` s'occupent d'assembler toutes les classes et fonctions ensemble, pour obtenir l'application Frogger au complet.

3 Réalisation de la base du jeu (Tâches de base)

Bien sûr, pour cette tâche, la première chose à faite a été de faire un support pour utiliser fltk. C'est là que les fichiers `main.cpp` et `MainWindow.hpp` rentrent en jeu.

3.1 Modèle

Il nous faut maintenant un modèle fonctionnel. Le but est de d'abord faire un plateau de jeu.

Les rangées sont des objets de classe `Lane`, classe qui est définie dans le fichier `lanes.hpp`. La classe `BoardModel` contient alors un vecteur de pointeurs vers des objets `Lane`. Pour avoir différents types de rangées, il suffit alors de définir ces types grâce à de l'héritage sur la classe `Lane`. Nous définissons alors dans le fichier `lane.hpp` les classes `FinnishLane` (nous en parlons dans le chapitre

4.2), `SafeLane` et `RoadLane`.

La classe `SafeLane` représente une rangée basique sans aucun obstacle. La classe `RoadLane` est par contre un peu plus complexe. Nous voulons d'abord définir ce qu'est une voiture, ceci se passe dans le fichier `movingobjects.hpp`. Une voiture est un objet de classe `Car` qui contient quelques getters et une méthode `collide` qui servira à voir s'il y a une collision avec la grenouille. Cette classe hérite de la classe abstraite `MovingObject`. Cela nous permet de créer une classe abstraite appelée `MovingObjectLane` qui hérite de `Lane` et qui possède en instance un vecteur de pointeurs vers des objets de classe `MovingObject`. Cette classe possède une méthode qui permet de regarder s'il y a collision entre la grenouille et au moins un objet. La classe `RoadLane` est alors juste un successeur de la classe `MovingObjectLane`.

Toutes les classes et méthodes sont après assemblées dans la classe `BaordModel` pour faire un plateau fonctionnel.
Parlons maintenant de la grenouille. Elle sera définie par un objet de classe `Frog`, classe qui est définie dans le fichier `frog.hpp`. Les collisions entre celle-ci et les objets sont gérées par les objets qui possèdent des méthodes qui prennent un référence vers la grenouille en paramêtre. La classe `Frog` possède alors un getter vers la rangée à laquelle elle se trouve, et un autre vers sa position x (en pixels). Cette classe possède tout de même une méthode `inBaord` qui vérifie si la grenouille ne sort pas du plateau. C'est aussi dans cette classe que sont gérés la mort, mais nous parlerons de cela dans le chapitre 4.3. L'utilité des autres méthodes de cette classe sera abordée plus tard.

3.2 Controlleur

Les contrôles de la grenouille se passent dans la classe `Controller` qui se trouve quand le fichier `controller.hpp` qui est contenu dans le dossier `Controller`. La classe `Frog` possède déjà des méthodes permettant de changer sa position sur le plateau, il faut alors juste les utiliser quand nous appuyons sur une touche. La méthode `processKey` sert à gérer ces actions. Les contrôles se font avec les touches zqsd comme pour beaucoup de jeux.

3.3 Vue

Tout ce dont nous allons parler dans ce sous-chapitre se passe dans le dossier `View`. Pour résumer, pour presque chaque classe de la partie modèle, il y a une classe de la partie vue qui s'occupe de l'affichage de ce modèle.

D'abord, nous avons un fichier `movingobjectview.hpp` qui contient la classe `CarView` qui hérite de la classe `MovingObjectView`. L'intérêt de ces classes est de pouvoir utiliser la méthode `draw` pour afficher l'objet sur la plateau. Les couleurs sont choisies dans le constructeur et les tailles sont données par l'objet de classe `MovingObject` qu'elle prennent en paramêtre dans le constructeur.

Ensuite, nous avons un fichier `laneview.hpp` qui possède la classe abstraite `LaneView` et les classes héritantes `SafeLaneView`, `FinishLaneView` et `RoadLaneView`. Leur intérêt est aussi de pouvoir utiliser la méthode draw pour les afficher sur le plateau. La classe `RoadLaneView` possède en plus un vecteur de pointeurs vers des objets de type `CarView` pour pouvoir dessiner les voitures sur les rangées concernées. La méthode `draw` de `CarView` est alors appelée dans la méthode `draw` de `RoadLaneView`.

Nous parlerons de l'affichage de la grenouille dans les chapitres 4.3 et 4.5
Enfin, à l'aide de toutes ces méthodes `draw` (il y en a aussi une pour la grenouille, nous pouvons créer une classe `BoardView` dans le fichier `boardview.hpp` contenant une méthode `draw` elle même et qui va dessiner le plateau selon l'instance de classe `BoardModel` (sous forme de pointeur) qui lui sera donnée.

3.4 Assemblage pour faire un jeu fonctionnel

Il nous faut maintenant assembler le tout pour faire un plateau fonctionnel. Ceci se fait dans le dossier `GameManagers`, et plus particulièrement dans le fichier `gameloop.hpp`. Dans celui-ci se trouve la classe `GameLoop` prennant tout ce que nous avons défini (et d'autres éléments dont nous parlerons plus tard) en instance. Le plateau s'initialise alors avec le constructeur, et s'anime en exécutant à chaque frame la méthode `update`. Nous verrons plus tard où cette méthode est exécutée.

```
class GameLoop {
    private:
        std::shared_ptr<BoardModel> bm;
        std::shared_ptr<BoardView> bv;
        std::shared_ptr<FrogView> fv;
        std::shared_ptr<Frog> frog;
        std::shared_ptr<Score> score;
        std::shared_ptr<ScoreView> sv;
        std::shared_ptr<Controller> c;
        std::unique_ptr<ScoreSaver> ssv;
        std::shared_ptr<Score> best_score;
        std::unique_ptr<ScoreView> bs_show;
    public:
        GameLoop(unsigned int lvl);
        void update();
        std::shared_ptr<BoardModel> getModel();
        std::shared_ptr<BoardView> getView();
        std::shared_ptr<FrogView> getFrog();
        ~GameLoop() {}
};
```

4 Réalisation des tâches additionnelles

Maintenant que les tâches principales sont réalisées et que nous avons une bonne base, nous pouvons implémenter les fonctionnalités supplémentaires.

4.1 Rangées d'eau, buches et tortues

Modèle:
Dans le fichier `movingobjects.hpp`, nous implémentons les tortues et les buches. Les collisions ne seront pas gérée de la même manière que pour les voitures, donc nous faisons un override. Nous parlerons plus en détail des méthodes de la classe `Turtle` dans le chapitre 4.4.

```
class Turtle: public MovingObject {
    private:
        bool diving;
    public:
        Turtle(int speed, const unsigned int head, const unsigned int size
                , const unsigned lane_id);
        void dive() final override;
        void undive() final override;
        bool isDiving() const final override;
        bool collide(Frog& frog) final override;
        ~Turtle() {}
};
```

```
class Log: public MovingObject {
    public:
        Log(int speed, unsigned int head, const unsigned int size,
            const unsigned lane_id);
        bool collide(Frog& frog) final override;
        ~Log() {}
};
```

Dans le fichier `lane.hpp`, nous implémentons deux nouvelles classes héritantes de `MovingObjectLane`. Nous parlerons aussi plus tard des méthodes de la classe `TurtleLane`.

```
class LogLane: public MovingObjectLane {
    public:
        LogLane(const unsigned int id_num, const unsigned int& log_by_pack,
                const unsigned int& space_between_logs,
```

```
                const unsigned& space_between_packs,
                const int& first_log_placement,
                const unsigned int& size_log, const int& speed=0);
        bool water_lane() const override { return 1; }
        void handle_after_collision(Frog& frog) override;
        std::vector<std::shared_ptr<Log>> getLogs() const;
        ~LogLane() {}
};
```

```
class TurtleLane: public MovingObjectLane {
    unsigned int turtle_by_pack;
    unsigned int diving_pack_id;
    bool is_diving = true;
    unsigned int diving_time;   // In frames
    unsigned int undiving_time;
    unsigned int diving_count = 0;
    public:
        TurtleLane(const unsigned int id_num
                   , const unsigned int& turtle_by_pack
                   , const unsigned int& space_between_turtles
                   , const unsigned& space_between_packs
                   , const int& first_turtle_placement
                   , const unsigned int& size_turtle
                   , const int& speed=1
                   , const unsigned int diving_pack_id = 0
                   , const unsigned int diving_time = 180
                   , const unsigned int undiving_time = 180);
        std::vector<std::shared_ptr<Turtle>> getTurtles() const;
        void handle_after_collision(Frog& frog) override;
        bool water_lane() const override { return 1; }
        void pack_dive();
        void pack_undive();
        void dive_update() final override;
        ~TurtleLane() {}
};
```

Vue:
Les implémentations gérant la vue de ces deux nouvelles rangées est similaire que pour celle des rangées de voitures.

```
class LogView: public MovingObjectView {
    public:
        LogView(std::shared_ptr<Log> l);
        ~LogView() {}
};
```

```
class TurtleView: public MovingObjectView {
    public:
        TurtleView(std::shared_ptr<Turtle> t);
        void draw() final override;
        ~TurtleView() {}
};
```

4.2 Nénuphars

Dans le fichier `Model/waterlilies.hpp`, nous définissons une nouvelle classe `WaterLilies` qui va représenter les nénuphars.

```
class WaterLilies {
    private:
        int x;
        bool visited=false;
    public:
        WaterLilies(int x);
        int getX();
        bool collide(Frog& frog);
        bool hasBeenVisited();
        void visit();
        ~WaterLilies() {}
};
```

Nous pouvons maintenant parler de l'implémentation de la classe `FinishLane`.

```
class FinishLane: public Lane {
    private:
```

```
        std::vector<std::shared_ptr<WaterLilies>> lilies;
    public:
        FinishLane(const unsigned int id);
        std::vector<std::shared_ptr<WaterLilies>> getLilies();
        ~FinishLane() {}
};
```

La rangée de fin contient des nénuphars. Si la grenouille se trouve sur un nénuphar, elle y reste et une nouvelle grenouille apparait, mais ceci n'est que ce que le joueur voir. En réalité, nous affichons juste une grenouille sur le nénuphar, mais nous gardons la même grenouille tout le long de la partie. Nous la téléportons juste à a case départ à chaque fois qu'elle touche un nénuphar. Si tout les nénuphars sont remplis, alors un écran de victoire s'affiche. Ceci se fait en utilisant la classe `FullScreenJPEGImage` du fichier `tooling/image_classes.hpp` et en utilisant l'image `imgs/won.jpeg`. Ces actions se passent dans la méthode `GameLoop::update()`.

4.3 Vies de la grenouille

Les vies de la grenouille sont représentées par l'instance `lives` de la classe `Frog`. A chaque exécution de `GameLoop::update()`, on regarde si la grenouille respecte une des conditions pour perdre une vie. Si oui, alors on décrémente `lives` avec la méthode `kill()`. Si `lives` est à 0, alors on affiche un écran de défaite de la même manière que nous l'avons fait avec la victoire (avec le fichier `imgs/lose.jpeg`.

La classe `FrogView` s'occupe de dessiner tout ce qui est relié à la grenouille. Nous parlons ici seulement des vies, le reste sera abordé dans le chapitre 4.5. La méthode `showLives()` est alors appelée dans la méthode `draw()`.

```
class FrogView {
    std::shared_ptr<Frog> frog;
    JPEGDrawer current_image{paths::frog_north_jpeg, frog->getX(),
                                static_cast<int>(frog->getLane()), HEIGHT, WIDTH};
public:
    FrogView(std::shared_ptr<Frog> f);
    void showLives();
    void draw();
};
```

4.4 Tortues plongeantes

Dans le modèle, une tortue plonge si son instance `diving` est à `True` (voir 4.1). Ce statut change surtout le comportement des collisions avec les tortues dans le modèle. Nous ordonnons à la tortue de plonger avec les méthodes `dive()` et `undive()`. Ces ordres sont alors données dans la classe `TurtleLane` (voir 4.1). Comme dans le jeu orginal, nous faisons plonger les tortues par paquets. Nous pouvons choisir les intervalles de temps entre la plongée et la remontée des tortues d'un paquet d'une rangée dans les paramêtres du constructeur de la classe. Nous pouvons aussi choisir quel paquet va plonger dans les paramêtres. La méthode `dive_update()` va alors se charger de faire plonger et remonter les tortues en boucle avec les paramêtres choisis. Cette méthode est exécutée à chaque frame dans la méthode `GameLoop::update()`.

Nous changons forcément commment est affichée la tortue quand elle plonge. Pour cela, nous faisons en sorte que la méthode `TurtleView::draw()` ne marche que quand la tortue est à la surface.

4.5 Directions de la grenouille

Les directions sont gérées avec l'enum class suivant:

```
enum class FrogDirection {
    North, South, East, West
};
```

4.6 Score

```
class Score {
    private:
        unsigned the_score = 0;
        int max_lane = 0;
    public:
        Score() = default;
        Score(unsigned score): the_score(score) {}

        void reachedWaterlily();
        void update(int new_lane_id);
        unsigned getScore() const;
        void resetBestLane();

        friend bool operator<(Score const& s1, Score const& s2);
        friend bool operator>(Score const& s1, Score const& s2);
        friend bool operator==(Score const& s1, Score const& s2);
        ~Score() = default;
};
```

4.7 Meilleur score

Le meilleur score et l'interaction avec les fichiers est gérée par la classe ScoreSaver

```
class ScoreSaver {
    private:
        unsigned lvl;
        const std::string file_name { paths::scores };
        std::map<unsigned, unsigned> scores;
    public:
        ScoreSaver(unsigned level): lvl(level) {}
        void writeToFile();
        void getFromFile();

        Score getHighScore();
        void setNewScore(Score const& score);
        void resetHighScore();
        void setLevel(unsigned& level);
        ~ScoreSaver() {}
};
```

4.8 Gestion des menus et écran d'accueil

La majorité des fichiers chargés de la gestion de smenus se trouvent dans ContentManagers.

```
class WindowContents;

class ContentManager {
    private:
        std::unique_ptr<WindowContents> contents;
        std::unique_ptr<GameLoop> gl;
    public:
        ContentManager(std::unique_ptr<WindowContents> first_contents):
            contents(std::move(first_contents)), gl(nullptr) {}

        void changeContents(std::unique_ptr<WindowContents> new_contents);

        void manageButtonPush(int x, int y);

        void contentManageAction(actions& action);

        void startGame(std::unique_ptr<GameLoop> g);
        void show();

        static void updateWithAction(std::shared_ptr<ContentManager> cm, actions& action);

        ~ContentManager() { }
};

class WindowContents {
   protected:
        std::weak_ptr<ContentManager> cm; // Observer
    public:
        WindowContents(std::shared_ptr<ContentManager> cm): cm(cm) {}
        WindowContents(std::weak_ptr<ContentManager> cm): cm(cm) {}
        virtual void draw() = 0;
        virtual void manageButtonPush(int x, int y) = 0;
        virtual void manageAction(actions& action) = 0;
        std::weak_ptr<ContentManager> getCM() {
            return cm;
        }

        virtual ~WindowContents() {}
};
```

En clair, ces classes fonctionnent de la manière suivante: Ce qui est affiché dans la fenêtre hérite de WindowContents, ainsi, nos menus, et la sélection de niveau héritera de WindowContents, ContentManager est la classe se chargeant d'alterner les différents contenus que l'on affiche à l'écran, par exemple, si je décide d'appuyer sur le bouton pour aller au menu de sélection de niveau, le menu de sélection de niveau et le menu de base sont tous les deux des WindowContents, et ce qui permet de changer entre les deux est l'instance de ContentManager.
L'écran d'accueil est représenté par la classe suivante

```
class WelcomeScreen: public WindowContents {
    private:
        ActionButton start_game_button;
        ActionButton go_to_levels;
        Text welcome;
   public:
        WelcomeScreen(std::shared_ptr<ContentManager> cm): WindowContents(cm) {}
        void manageButtonPush(int x, int y) override;
        void manageAction(actions& action) override;
        void draw() override;
        ~WelcomeScreen() {}

};
```

4.9 Niveaux et sélection de niveau

```
class LevelSelect: public WindowContents {
    public:
        LevelSelect(std::shared_ptr<ContentManager> cm): WindowContents(cm) {
            ss.getFromFile();
            best_score_show.setString("Highest␣Score:␣" + std::to_string(ss.getHighScore().getScore()));
        }
        LevelSelect(std::weak_ptr<ContentManager> cm): WindowContents(cm) {
            ss.getFromFile();
            best_score_show.setString("Highest␣Score:␣" + std::to_string(ss.getHighScore().getScore()));
        }
        void draw() override;
        void manageButtonPush(int x, int y) override;
        unsigned getLevel();
        void manageAction(actions& action) override;
        ~LevelSelect() {}

};
```

4.10 Vies