Rapport pour le projet de INFOF202

Ransy Lenny Lejeune Lucas

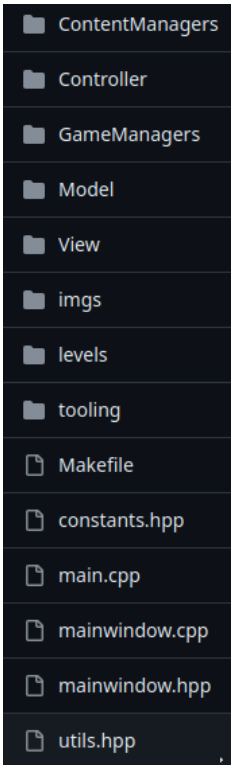
Janvier 2024

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.

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.

1 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 `imgs` contient les images utilisées dans le programme// - 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.

2 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. Les classes et méthodes utiliseés dans les classes suivantes seront bien-sûr abordées dans la suite.

```
int main(int argc, char *argv[]) {
    std::srand(static_cast<unsigned>(time(nullptr)));
    auto c = std::make_shared<ContentManager>(nullptr);
    auto ws = std::make_unique<WelcomeScreen>(c);
    c->changeContents(std::move(ws));
    MainWindow window(c);
    window.show(argc, argv);
    return Fl::run();
}
```

```
class MainWindow : public Fl_Window {
    private:
        std::shared_ptr<ContentManager> contents;
    public:
        MainWindow(std::shared_ptr<ContentManager> contents);
        void draw() override;
        int handle(int event) override;
        static void Timer_CB(void *userdata);
};
```

2.1 Modèle

Il nous faut maintenant un modèle fonctionnel. Le but est de d'abord faire un plateau de jeu. Tout ceci se passe dans le dossier `Model`.
Les rangées sont des objets de classe `Lane`, classe qui est définie dans le fichier `lanes.hpp`.

```
class Lane {
    private:
        const unsigned int id_num;
    public:
        Lane(const unsigned int id_num);
        unsigned int getId() const;
        virtual void diveUpdate() {}
        virtual ~Lane() {}
};
```

La classe `BoardModel` contient alors un vecteur de pointeurs vers des objets `Lane`.

```
class BoardModel {
private:
    std::shared_ptr<FinishLane> the_finish_lane;
    std::vector<std::shared_ptr<Lane>> lanes {};
    unsigned time = 0;
public:
    BoardModel(std::vector<std::shared_ptr<Lane>> lanes);
    void updateTurtles(std::shared_ptr<Lane> lane);
    void update(); // moves the objects on the board
    bool gameWon();
    bool isOutOfBoard(Frog& frog);
    bool frogOnLily(Frog& frog);
    void addLane(std::shared_ptr<Lane> lane);
    std::vector<std::shared_ptr<Lane>> getLanes();
    bool anyCollision(Frog& frog);
    void handleCollision(Frog& frog);
    ~BoardModel() {}
    unsigned getTime() const  { return time; }
};
```

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 3.2), `SafeLane` et `RoadLane`.

```
class SafeLane: public Lane {
    public:
        SafeLane(const unsigned int id);
        ~SafeLane() {}
};
```

```
class MovingObjectLane: public Lane {
    protected:
        std::vector<std::shared_ptr<MovingObject>> mv;
        int lane_speed;
    public:
        MovingObjectLane(const unsigned int id, int lane_speed=0);
        bool frogCollide(Frog& frog);
        std::vector<std::shared_ptr<MovingObject>> getMovingObjects();
        virtual void handleAfterCollision(Frog& frog) = 0;
        virtual bool waterLane() const = 0 ;
        virtual ~MovingObjectLane() {}
};
```

```
class RoadLane: public MovingObjectLane {
    public:
        RoadLane(const unsigned int id_num
                , const unsigned int& car_by_pack
                , const unsigned int& space_between_cars
                , const unsigned& space_between_packs
                , const int& first_car_placement
                , const unsigned int& size_car
                , const int& speed=1);
        bool waterLane() const override { return 0; }
        void handleAfterCollision(Frog& frog) override;
        std::vector<std::shared_ptr<Car>> getCars() const;
         ~RoadLane() {}
};
```

Nous voulons d'abord définir ce qu'est une voiture, ceci se passe dans le fichier `movingobjects.hpp`.

```
class MovingObject {
    protected:
        const int speed;
        int x;
        const unsigned int size;
        const unsigned int lane_id;
    public:
        MovingObject(const int speed, int x, const unsigned int size, const unsigned lane_id);
        void move();
        unsigned getSize() const;
        unsigned getId() const;
        // returns the x coordinate of the center of the object
        int getCenterX() const;
        std::tuple<int, int> getBoundaries() const;
        int getX() const;
        // returns true if this element collides with the frog
        virtual bool collide(Frog& frog) = 0;
        int getSpeed() const;

        // Methods regarding diving turtles
        virtual void dive() {}
        virtual void undive() {}
        virtual bool isDiving() const;
        virtual ~MovingObject() {}
};
```

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

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`.

```
class Frog {
    private:
        // Frog Position and Direction (important for display)
        unsigned int lane_number;
        int x;
        FrogDirection direction = FrogDirection::North;
        int lives;
        std::shared_ptr<Score> score;
    public:
        Frog(unsigned int lane_number, int x, std::shared_ptr<Score> score);

        // Standard getters
        unsigned int getLane() const ;
        int getX() const;
        int getLives() const;
        FrogDirection getDirection() const;

        // Methods in charge of moving the frog (interacting with the controller)
        void goUp();
        void goDown();
        void goLeft();
        void goRight();

        // Moves the frog in a given direction (helpful when frog is sitting on logs)
        void go(int speed);
        // Returns false only if the frog is outside
        bool inBoard();

        // Methods in charge of life and death of the frog
        void resetPos();
        bool alive();
        void kill();
        void inWaterLilies();

        ~Frog(){}
};
```

2.2 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.

```
class Controller {
    private:
        std::shared_ptr<Frog> f;
        std::map<char, bool> is_pressed;
        unsigned int count = 0;
```

```
    public:
        Controller(std::shared_ptr<Frog> f);
        void decrement();
        void processKey(char& c);
        void updatePressedKeys(const char&& c);
        void resetPressedKeys();
        ~Controller() {}
};
```

2.3 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 des objets de cette classe. L'intérêt de ces classes est de pouvoir utiliser la méthode `draw` pour afficher l'objet sur la plateau.

Dans movingobjectview.hpp:

```
class MovingObjectView {
    protected:
        std::shared_ptr<MovingObject> mv;
        std::unique_ptr<RectangleDrawer> object_drawer;
    public:
        MovingObjectView(std::shared_ptr<MovingObject> mv);
        virtual void draw();
        std::shared_ptr<MovingObject> getMovin();
        virtual ~MovingObjectView() {}
};
```

```
class CarView: public MovingObjectView {
    public:
        CarView(std::shared_ptr<Car> c);
        ~CarView() {}
};
```

Dans laneview.hpp:

```
// An abstract class that helps model a lane
class Lane {
    private:
        const unsigned int id_num;
    public:
        Lane(const unsigned int id_num);
        unsigned int getId() const;
        virtual void diveUpdate() {}
        virtual ~Lane() {}
};
```

```
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() {}
};
```

```
class SafeLane: public Lane {
    public:
        SafeLane(const unsigned int id);
        ~SafeLane() {}
};
```

```
class MovingObjectLane: public Lane {
    protected:
        std::vector<std::shared_ptr<MovingObject>> mv;
        int lane_speed;
    public:
        MovingObjectLane(const unsigned int id, int lane_speed=0);
        bool frogCollide(Frog& frog);
        std::vector<std::shared_ptr<MovingObject>> getMovingObjects();
        virtual void handleAfterCollision(Frog& frog) = 0;
        virtual bool waterLane() const = 0 ;
        virtual ~MovingObjectLane() {}
};
```

```
class RoadLane: public MovingObjectLane {
    public:
        RoadLane(const unsigned int id_num
                , const unsigned int& car_by_pack
                , const unsigned int& space_between_cars
                , const unsigned& space_between_packs
                , const int& first_car_placement
                , const unsigned int& size_car
                , const int& speed=1);
        bool waterLane() const override { return 0; }
        void handleAfterCollision(Frog& frog) override;
        std::vector<std::shared_ptr<Car>> getCars() const;
        ~RoadLane() {}
};
```

Nous parlerons de l'affichage de la grenouille dans les chapitres 3.3 et 3.5.
Dans boardview.hpp:

```
class BoardView {
    private:
        std::vector<std::shared_ptr<LaneView>> lanes;
        std::shared_ptr<BoardModel> b;
    public:
        BoardView(std::vector<std::shared_ptr<LaneView>> lanes,
                  std::shared_ptr<BoardModel> b);
        void draw();
        std::vector<std::shared_ptr<LaneView>> getLaneList();
        ~BoardView() {}
};
```

2.4 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`.

```
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() {}
};
```

3 Réalisation des tâches additionnelles

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

3.1 Rangées d'eau, buches et tortues

Modèle: Dans le fichier `movingobjects.hpp`, nous implémentons les tortues et les buches.

```
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`.

```
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 waterLane() const override { return 1; }
        void handleAfterCollision(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 handleAfterCollision(Frog& frog) override;
        bool waterLane() const override { return 1; }
        void packDive();
        void packUndive();
        void diveUpdate() 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() {}
};
```

3.2 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 montrer 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() {}
};
```

3.3 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 (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 3.5.

```
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();
};
```

3.4 Tortues plongeantes

Dans le modèle, une tortue plonge si son instance `diving` est à `True` (voir 3.1). 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 3.1). Comme dans le jeu orginal, nous faisons plonger les tortues par paquets. 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.

3.5 Directions de la grenouille

Les directions sont définies avec l'enum class suivante: (dans le fichier `frog.hpp`)

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

La grenouille est alors affichée selon la direction dans laquelle elle se trouve.

3.6 Score

Le score est un objet de classe `Score`, classe qui est définie dans le fichier `score.hpp`.

```
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;
};
```

Le score est calculé de la manière suivante: Le score commence à 0. A chaque fois que la grenouille avance d'une rangée vers le nord, le score monte de 100 points. Si la grenouille revient en arrière, alors elle ne recommencera à gagner des points seulement quand elle aura réatteint la rangée la plus haute qu'elle avait avant atteint. Si la grenouille meurt ou atteint la ligne d'arrivée, le score est conservé et le même système recommence (c'est à dire que si la grenouille est téléportée à la case départ, et qu'elle avance d'une rangée vers le nord, alors elle regagne 100 points).

L'affichage du score sur le plateau se fait grâce à la classe `ScoreView` qui se trouve dans le fichier `scoreview.hpp`.

```
class ScoreView {
        std::shared_ptr<Score> score;
        Text score_text;
        bool is_best;
    public:
        ScoreView(std::shared_ptr<Score> score, int const& x, int const& y, bool is_best=false);
        std::string stringForScore();
        void draw();
```

```
        ~ScoreView() {}
};
```

3.7 Meilleur score

Le meilleur score et l'interaction avec les fichiers est gérée par la classe ScoreSaver. Les meilleurs score sont alors stockés dans le fichier `scores.csv` du dossier `levels`.

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

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

Le meilleur score est alors aussi affiché grâce à la classe `ScoreView` sur le plateau, mais aussi dans le menu de sélection des niveaux.

3.8 Gestion des menus et écran d'accueil

La majorité des fichiers chargés de la gestion des menus 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() {}

};
```

3.9 Niveaux et sélection de niveau

De la même manière que pour l'écran d'accueil, nous faisons un menu s'occupant de la sélection des niveaux.

```
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 Logique du jeu