Rapport Du Projet Arkanoid

Karbach Abdelmouhaïmin — Matricule: 593038

3 janvier 2025

Contents

1 Introduction

Dans ce rapport nous aborderons la réalisation du projet, réalisé dans dans le cadre du cours "Langage de programmation" en BA2 à l'ULB. Pour ce projet, il nous a été demandé de coder en C/C++, le jeu Arkanoid à l'aide de l'interface graphique allegro.

2 Tâches accomplie

Les tâches de base:

- Implémentation de la raquette
- Implémentation des briques
- Implémentation du score
- Implémentation des vies
- Implémentation de la victoire

Les tâches additionnelles:

- Implémentation des niveaux
- Implémentation des déplacements de la raquette avec la souris/trackpad
- Implémentation des briques colorées et le meilleur score
- Implémentation des briques dorées et argentées
- Bonus:
  - Agrandir
  - Attraper
  - Ralentissement
  - Joueur
  - Laser

3 Inputs

- Mouvements raquette: gauche - q ou a — droite - p ou d
- Mettre en vitesse une balle à l'arrêt: s ou espace
- Fermer le jeu lorsque plus de vie: x
- Reset le niveau: r
- Finir le niveau à la prochaine brique cassée: pavé numérique
- Passer au niveau suivant: n
- Passer au niveau précedent: b
- Réinitialiser le meilleur score: o

4 Classes

4.1 View

```
class View {

    private:

    ALLEGRO_TIMER* timer = nullptr;
    ALLEGRO_EVENT_QUEUE* queue = nullptr;
    ALLEGRO_DISPLAY* disp = nullptr;
    ALLEGRO_FONT* font = nullptr;
    ALLEGRO_FONT* fontLW = nullptr;
    ALLEGRO_BITMAP* background = nullptr;
    bool backgroundLoaded = false;
    bool redraw = true;
    bool playing = true;
    ALLEGRO_EVENT event;
    std::unique_ptr<Linker> linker;
    std::string scorestr = "SCORE: ";
    std::string lifestr = "LIFE: ";
    std::string pbstr = "HIGHSCORE: ";
    int posMouseX;
    int posMouseY;
    void must_init(bool, const char*);
    void must_init(void*, const char*);
    void drawBlocs();
    void drawBonus();

    public:

    View();
    virtual ~View();
    virtual int loadBackground();
    virtual void displaying();
};
```

La classe View permet l'affichage à l'aide d'allegro. Elle contient un objet Linker que nous verrons plus tard.

4.2 Cell

```
class Cell
{
private:

    int posx_;
    int posy_;
    bool broken_=false;
    int length=10;
    int wide=21;
    std::vector<int> color_map = {255,255,255};
    enum color color=NOCOLOR;
    unsigned short will_break = 1;
    enum bonus bonus = NOBONUS;

public:

    Cell(int, int);
    virtual ~Cell();
    virtual void cellBreak();
    virtual bool isBroken();
    virtual int getPosX();
    virtual int getPosY();
    virtual int getLen();
    virtual int getWide();
    virtual void reset();
    virtual void setColor(char);
    virtual std::vector<int> getColor();
    virtual int getPoints();
    virtual unsigned short getState();
    virtual char getBonus();
    virtual void setBonus(char);
};
```

La classe Cell représente les briques formant le bloc de brique du jeu.

4.3 Blocs

```
class Blocs
{
private:
    unsigned short sizes[2] = {0,0};
    Cell* cellMatrice[50][50];
    unsigned max_score = 0;
    unsigned short getLevelSize(unsigned short,unsigned short);
    void unloadLastLevel(unsigned short,short);

public:

    Blocs();
    virtual ~Blocs();
    virtual Cell* getCell(int,int);
    virtual void resetCells();
    virtual unsigned short* getMatriceSizes();
    virtual unsigned getMaxScore();
    virtual void loadLevel(unsigned short,short);
};
```

La classe Blocs regroupe plusieurs instances de la classe Cell.

4.4 Racket

```
class Racket
{
private:

    float posx_=310;
    float posy_=650;
    float lenSide_[2]={50,10};
    int baseLen_=50;

public:

    Racket();
    virtual ~Racket();
    virtual float getPosX();
    virtual float getPosY();
    virtual float* getlenSide();
    virtual void moveLeft();
    virtual void moveRight();
    virtual void moveTo(int);
    virtual void setLenSide(int);
    virtual int getBaseLen();

};
```

La classe Racket représente le joueur et donc la raquette.

4.5 Ball

```
class Ball
{
private:

    float posx_=310;
    float posy_=600;
    float radius_=7;
    float speed_=-6;
    float base_speed_=-6;
    float decompSpeed_[2] = {0,0};

public:

    Ball();
    virtual ~Ball();
    virtual float getPosY();
    virtual float getPosX();
    virtual void setPosX(float);
    virtual void setPosY(float);
    virtual float getRadius();
    virtual void move();
    virtual float* getDecompSpeed();
    virtual void setDecompSpeed(float,int);
    virtual float getSpeed();
    virtual void ballReset();
    virtual void setSpeed(float);
    virtual float getBaseSpeed();

};
```

La classe Ball représente la balle avec laquelle le joueur joue.

4.6 Bonus

```
class Bonus : public Ball
{
private:
    enum bonus bonus;
    std::vector<int> color = {255,255,255};
public:
    Bonus(int,int);
    virtual char getBonus();
    virtual void setBonus(char);
    virtual std::vector<int> getColor();
    virtual void setColor(int,int,int);
};
```

La classe Bonus représente les bonus que le joueur pourra attraper au cours de la partie. La classe Bonus hérite de la classe Ball.

5 Logique

Au lancement du programme, un objet View est créé, initialisant toute l'interface graphique Allegro. Cet objet View crée lui-même un objet Linker, qui lui-même crée les principaux éléments du jeu tels que la balle, la raquette et les blocs. Dès le lancement l'objet Blocs génère le premier niveau et le Linker charge le meilleur score de ce niveau et le stocke dans une variable. Un vecteur de Bonus vide est aussi initialiser pour pouvoir gérer les bonus plus tard. Après l'initialisation du jeu, l'interface attends que le joueur lance le jeu en appuyant sur espace, ce qui donne une vitesse à la balle, qui de base était nulle. A une fréquence de 60 fois par seconde les methodes checkInteractionRB(), checkInteractionBB() et checkInteractionRBonus() de la classe Linker sont appelé pour vérifier s'il y a des interaction entre les différents objets. La méthode checkInteractionBB() appel la méthode checkInteractionCB(int,int) pour chaque brique du niveau. Pour savoir si un élément intéragit avec un autre élément il suffit de comparer les positions des deux éléments en ajoutant leur épaisseur et largeur. Pour ensuite déterminer la direction que la balle prendra lors d'une interaction avec la raquette, est donné dans les directives du projet, tandis que pour la direction que prend

la balle en interagissant avec un bloc, elle ne l'est pas. Pour cette interaction nous avons dû choisir une manière optimale de gérer les collisions.

5.1 Collisions balle-brique

Pour déterminer la prochaine position de la balle nous avions dans un premier temps adopter une implémentation avec une vitesse et un angle, il suffisait de calculer la position avec: ***vitesse*sin(angle)*** pour les y et la même en remplaçant sin par cos pour les x. Mais cette façon de faire s'est trouvée problématique pour les collisions avec les briques. Nous avons donc implémenter la vitesse décomposée, avec donc une vitesse en x et une vitesse en y, ce qui simplifie les rebonds sur les côtés des briques et sur le haut. Il suffit de changer le signe d'une des deux composantes de vitesse en fonction de si la balle viens de par le haut ou bas, ou bien par la droite ou la gauche. Et pour savoir de quel côté la balle est il suffit de voir la plus petite distance entre les composante de la position de la balle et la composante correspondante des points avec les plus petites coordonnées et les plus grandes coordonnées. Mais avec cette méthode, il peut y avoir des comportements non souhaité dans certains cas lorsque la balle arrive sur un angle de la brique avec un certain angle, pour éviter un maximum ces comportement, la direction de la balle est aussi prise en compte. Nous avons aussi constater un autre comportement non souhaité, dans certains cas la balle pouvait se retrouvé coincé dans une brique, pour régler ce problème il a suffit de déplacer directement la balle en dehors de de la brique après la collision. La méthode cellBreak() de la classe Cell est ensuite appelée pour cassé la brique et on vérifie si la brique contenait un bonus. Si c'est le cas, un nouvel objet Bonus est créer et ajouté au vecteur contenant les bonus.

5.2 Collisions balle-raquette

Les collisions entre la balle et la raquette sont simple, lorsque les coordonnées de la balles sont entre les coordonnées de la raquette en x et correspondent à la coordonnée de la raquette en y, l'équation donné dans les instructions est appliquée.

5.3 Collisions bonus-raquette

Les collisions entre les bonus et la raquette sont tout aussi simple que les collisions entre la balle et la raquette. Il suffit de faire la même chose pour chaque bonus dans le vecteur contenant tout les bonus en jeu. Lorsqu'une collision est détecté le bonus est appliqué et la capsule du bonus est supprimer. Pour que le bonus soit appliqué, le précédent doit être désactivé, si le bonus attrapé est différent du précédent. Ce qui permet de pouvoir cumuler certains bonus.

5.4 Fin de niveau

Pour savoir si le niveau est terminé, nous avons opté pour comparer le score actuel et le score maximal, qui est encodé dans le fichier du niveau. Lorsque la fin d'un niveau est détectée on vérifie si le prochain niveau existe, et si c'est le cas il est chargé par la méthode de Linker, ***nextLevel()***, qui elle appel la méthode ***loadlevel(level,next)***, qui elle même appel la méthode getPBFromFile(level) permettant la récupération du meilleur score à partir du fichier, avec elle est appelée aussi la methode ***loadLeve(level,next)*** de la classe Blocs, cette méthode décharge le niveau terminé et charge le nouveau, elle donne donc les couleurs et les positions des cell.

5.5 Perte de balle

Pour savoir si la balle est perdue il suffit de vérifier si la position de la balle en y est plus grande que la taille de l'espace de jeu, dans ce cas 800. Cette vérification est faite lors de l'appel la méthode ***moveBall()*** de la classe Linker. Et lorsque la balle est perdu, il suffit de vérifier si il reste de la vie au joueur et si c'est le cas, ***resetBall()*** de la Linker est appelé pour remettre la balle en jeu.

6 Modèle-Vue-Contrôleur

Le modèle de conception Modèle-Vue-Contrôleur a bien été utilisé pour l'implémentation de ce projet. Pour mieux appuyer l'utilisation de ce modèle, les fichier on été limité à 3, un pour le modèle, un pour la vue et un pour les contrôles. La classe dans le fichier View s'occupe exclusivement de l'affichage à l'aide de la librairie allegro, le fichier elements ne contient que les classes des objets utilisé dans le jeu, et dans le fichier linkLogGame toute la logique du jeu y est gérée.

7 Difficultés rencontrées

La principal difficulté rencontrée lors de ce projet fut l'initialisation de l'affichage à l'aide d'allegro, étant donné l'absence de documentation officiel. D'autres difficultés moins importante on été rencontrées, tel que la gestion des collisions balle-brique. Les difficulté créer lors de la conception du code on aussi été rencontré, tel que l'oublie de l'utilisation des unique_ptr, il a donc fallut réadapté au mieux le code pour les utiliser.