Rapport sur le projet de Langage de Programmation : Arkanoïd

5 janvier 2025

Prénom	Nom	Matricule
Lucas	Verbeiren	000591223
Ethan	Van Ruyskensvelde	000589640

Table des matières

1 Introduction

Ce projet consiste à implémenter une version fonctionnelle du célèbre jeu `Arkanoid` en utilisant les principes de la programmation orientée objet.

Pour ce faire, nous utilisons le langage de programmation `C++` et la bibliothèque `Allegro` pour l'interface graphique. Nous structurons le projet en suivant les principes de l'architecture `Modèle-Vue-Contrôleur` (MVC), afin de séparer clairement les différentes responsabilités du code.

L'objectif principal est d'implémenter un niveau fonctionnel avec les mécaniques de jeu suivantes :

— Déplacement de la raquette
— Rebonds de la balle
— Gestion des briques
— Gestion des vies
— Gestion du score
— Niveaux multiples
— Briques colorées
— Bonus

Enfin, ce projet vise également à montrer notre compréhension des concepts principaux de la programmation orientée objet et notre capacité à structurer un projet complet. À travers cette réalisation, nous cherchons à fournir un programme robuste, modulaire et bien documenté.

2 Tâches accomplies

Nous avons réalisé toutes les tâches de base, c'est-à-dire :

— Rebond correct de la balle sur les différentes surfaces comme les briques, les murs et la raquette
— Déplacement de la raquette
— Un niveau comportant 8 lignes de 14 briques
— Un affichage du score qui est mit à jour en gagnant 1 point par brique cassée. Si le joueur a cassé toutes les briques, un message de victoire est affiché
— Un système de vie où le joueur a 3 vies maximum par partie. S'il les perd toutes, un message de défaite est affiché

Nous avons également effectué les tâches additionnelles suivantes :

— Un système de niveau où chaque niveau est encodé dans un fichier '.txt', avec la possibilité d'encoder un bonus pour une brique,
— Déplacement de la raquette avec la souris,
— Des briques avec différentes couleurs et où chaque couleur fait remporté des points différents. De plus, nous sauvegardons après chaque partie le meilleur score dans un fichier 'score.txt' et le meilleur score est affiché en cours de partie. Nous avons également implémenté une touche qui permet de réinitialiser le meilleur score,

— L'ajout de briques argentées et dorées. Une brique dorée ne peut jamais être cassée et une brique argentée a besoin d'être touché deux fois par la balle pour pouvoir être cassée,

— Le bonus qui permet d'agrandir la raquette lorsque le joueur attrape la capsule bleue,

— Le bonus qui permet de ralentir la vitesse de la balle lorsque le joueur attrape la capsule orange,

— Le bonus qui permet de gagner une vie en attrapant la capsule grise,

— Le bonus qui permet de tirer un laser en attrapant la capsule magenta,

— Le bonus qui permet de diviser la balle en trois instances d'elle même si le joueur attrape la capsule cyan.

3 Interface des différentes classes, leurs rôles et liens avec les autres classes

3.1 Modèle

Toutes les classes utilisées pour le modèle se trouvent dans le dossier `src/model`.

3.1.1 GameBoard

La classe GameBoard est l'élément central du modèle. Elle coordonne et fait évoluer les autres composants du modèle au fil du temps. Le tableau 1 détaille ces composants ainsi que les classes associées à chacun :

Composant du modèle	**Classe**
Le score	ScoreManager
Le compteur de vie	LifeCounter
Le bonus actif	Bonus
Les capsules de bonus qui tombent	BonusPill
Les lazers	Lazer
La raquette	Racket
Les bordures du plateau de jeu	Border
Les briques	Brick
Les balles	Ball

TABLE 1 – Correspondance entre les composants du modèle et les classes.

Le contrôleur qui possède l'horloge du jeu doit pouvoir signaler à la GameBoard de mettre à jour les composants du modèle et combien de temps s'est écoulé entre cette mise à jour et la précédente. Nous implémentons cela via la méthode `update` de `GameBoard` : Chaque fois que celle-ci est appelée à un instant T, avec en paramètre une durée $deltaT$, `GameBoard` met à jour tous les composants du modèle à l'état correspondant à l'instant $T + deltaT$.

3.1.2 Vec2

La classe `Vec2` représente un vecteur à deux dimensions. Celle-ci implémente les opérateurs usuels des vecteurs, e.g. l'addition, la soustraction. Cette classe ne représente pas un élément du jeu directemment. Nous l'utilisons pour représenter des points et des vecteurs dans le plan.

3.1.3 RectangleShape

La classe `RectangleShape` est utilisée pour représenter n'importe quel objet rectangulaire, e.g. une brique, la raquette.

Son interface permet de :

— Modifier le centre, la hauteur et la largeur du rectangle.

— Obtenir différentes informations sur le rectangle, e.g. la largeur, le centre.

— Vérifier si une superposition a lieu avec un autre rectangle.

3.1.4 Bounceable

La classe `Bounceable` représente un objet rectangulaire sur lequel la balle peut rebondir. Puisque toutes les instances de cette classes sont rectangulaires, celle-ci hérite de `RectangularShape`.

Par défaut, l'effet de rebond des `Bounceable`'s consiste simplement à inverser l'axe x et/ou y du vecteur directeur de la balle en fonction du type de rebond (rebond sur un coin, une surface verticale ou horizontale). Le type de rebond s'obtient via la méthode `getBounceType` renvoyant un enuméré `BounceType` représentant ces trois types de rebond. Cependant, chaque `Bounceable`'s peut modifier l'effet de rebond par défaut, qui est donné à tous les `Bounceable`'s, et ce en "overridant" la méthode `getDirVecAfterBounce`.

3.1.5 Briques

La classe abstraite `AbstractBrick` représente une brique. Puisque la balle peut rebondir sur les briques, `AbstractBrick` hérite de `Bounceable`. Une brique est définie comme un type de `Bounceable` ayant :

— Une couleur, directemment liée au nombre de points qu'elle rapporte au joueur, une fois que celui-ci la détruit.

— Une durabilité, représentant le nombre de coup que celle-ci peut encore subir avant d'être détruite.

— Le type type de bonus qu'elle contient.

Afin d'associer à sa couleur un nombre de points rapportés lors de la destruction, nous utilisons l'énuméré `AbstractBrick::Color`.

Lorsque la `GameBoard` détecte qu'une balle rebondit sur la brique, elle doit signaler à la brique qu'elle a été touchée. Ceci est implémenté via la méthode `hit` servant à décrémenter la durabilité et, si celle-ci est détruite, renvoyer le type de bonus que la brique contenait.

Les briques dorées ne pouvant pas être détruites contrairement aux autres, nous avons créé deux classes héritant de `brick` :

— `BasicBrick` Pour les briques implémentant les caractérstiques d'une brique simple, i.e. qui peut être détruite.

— `GoldBrick` Pour les briques dorées qui ne peuvent pas être détruite.

L'interface qu'offrent `BasicBrick` et `GoldBrick` est exactement la même interface que celle de `AbstractBrick`. Nous implémentons une "méthode-usine" ("factory") renvoyant un pointeur de brique pouvant pointer vers une `BasicBrick` ou une `GoldBrick`.

3.1.6 Border

La classe `Border` représente les trois bordures du plateau de jeu sur lesquelles les balles rebondissent (en haut, à gauche et à droite). `Border` hérite de `Bounceable` et n'implémente aucune méthode particulière.

3.1.7 Racket

La classe `Racket` représente la raquette. Puisque les balles peuvent rebondir dessus, celle-ci hérite de `Bounceable`.

Afin de déplacer la raquette lattéralement, la classe `Racket` possède le setter `setCenterX`.

La classe `Racket` modifie l'effet de rebond par défaut de `Bounceable` en surchargeant la méthode `getDirVecAfterBounce`. Cela permet d'ajouter l'effet de variation de l'angle entre la balle et l'horizontale en fonction de la distance entre le centre de la raquette et le point d'impact de la balle.

3.1.8 Ball

La classe `Ball` représente une balle du jeu. Celle-ci possède essentiellement une position, un rayon, une vitesse et un vecteur directeur. Nous nous assurons que la balle garde une vitesse de déplacement directement proportionnelle à son attribut vitesse, en gardant le vecteur directeur toujours normé, c'est-à-dire que son module vaut 1. Ceci est assuré par la méthode permettant de changer le vecteur directeur et le constructeur de `Ball`. Le scalaire vitesse est donc le seul pouvant influencer la vitesse de déplacement de la balle.

La méthode `checkCollision` permet de vérifier si la balle est rentrée en collision avec un `RectangleShape`. Cette méthode est utilisée par la `GameBoard` au moment de trouver les collisions qui ont eu lieu à un instant T. La méthode `collide` également appelée par `GameBoard` permet ensuite de resoudre la collision (7) avec un `RectangleShape`.

La méthode `update` prenant un paramètre représentant une durée $deltaT$ permet de mettre à jour la position de la balle à l'instant $T + deltaT$.

3.1.9 BonusType

L'énuméré contient les différents types de bonus implémentés, ainsi que la valeur `None`, représentant l'absence de bonus.

3.1.10 Bonus à durée limitée

Les bonus à durée limitée sont gérés par les classes `BasicTimedBonus` et `SlowDownBonus`, qui héritent de la classe abstraite `AbstractTimedBonus`. Cette dernière fournit une structure commune avec un attribut `BonusType` (indiquant le type de bonus) ainsi qu'une interface pour notifier le bonus de :

— L'écoulement d'une durée $deltaT$

— Réappliquer le bonus (Ceci est nécessaire uniquement pour les bonus "SlowDown" car les effets sont cumulables).

La classe `BasicTimedBonus` hérite de `AbstractTimedBonus` et gère le temps d'activité de celui-ci. Elle implémente la méthode `update` de l'interface de `AbstractTimedBonus`, permettant de diminuer le temps restant du bonus d'une durée donnée. Nous l'utilisons pour les bonus suivants : "WideRacket" et "Lazer". (Nous avions l'intention de l'utiliser également pour le bonus "Attraper" mais nous ne sommes pas parvenus à implémenter celui-ci dans les délais du projet.)

Le bonus "SlowDown" est cumulable, contrairement aux autres. Cela signifie que si le joueur attrape un "SlowDown" alors qu'un ou plusieurs "SlowDown" étaient déjà actifs, la balle doit ralentir davantage. La méthode `getSlowDownFactor` est spécifique à celui-ci et permet d'obtenir la valeur par laquelle la vitesse de base de la balle doit être divisée pour obtenir sa vitesse avec cette configuration de bonus "SlowDown". Cette valeur est proportionnelle à la somme des temps restants pour tous les "SlowDown" actifs.

Le fait que seul le bonus "SlowDown" soit cumulable contrairement aux autres, est la raison pour laquelle nous choisissons d'implémenter les bonus à l'aide de 3 classes dont une classe abstraite permettant d'unifier le comportement commun des deux autres.

3.1.11 BonusPill

La classe `BonusPill` représente une capsule/pillule de bonus contenant un `BonusType`. Celle-ci étant rectangulaire mais ne laissant pas la balle rebondir dessus, elle hérite seulement de `RectangleShape`. Elle possède également une vitesse de descente. La méthode isOverlapping, héritée de `RectangleShape` permet de vérifier si la pillule est rentrée en collision avec un autre `RectangleShape`. Nous utilisons cette méthode sur la raquette afin de savoir si la raquette a attrapé la `BonusPill`. La méthode `update` permet de faire descendre le bonus de la distance qu'il aurait parcouru après un nombre de secondes donné en paramètre.

3.1.12 ScoreManager

La classe `ScoreManager` gère et centralise les informations concernant le score, c'est-à-dire le meilleur score et le score du joueur à un instant donné.

3.1.13 LifeCounter

La classe `LifeCounter` s'occupe de gérer le nombre de vies du joueur. Cependant, c'est la `GameBoard` qui doit lui notifier les pertes et les gains de vie. Par exemple, lorsque le joueur n'a plus de balle en jeu, `GameBoard` doit envoyer un message au `LifeCounter` pour que celui-ci décrémente le nombre de vies.

3.1.14 Lazer

La classe `Lazer` représente un laser tiré vers le haut et provenant de la raquette. Celui-ci est de forme rectangulaire et ne fait pas rebondir la balle, il hérite donc uniquement de `RectangleShape`. Celui-ci possède une vitesse. Nous utilisons la méthode `isOlverapping` héritée de `RectangleShape` afin de vérifier si le laser a collisionné avec un autre `RectangleShape`, plus précisément : une brique. La méthode `update` permet de faire monter le bonus de la distance qu'il aurait parcouru après $deltaT$ secondes.

3.2 Vue

Toutes les classes utilisées pour la vue se trouvent dans le dossier `src/view`.

3.2.1 DisplayGame

La classe `DisplayGame` gère l'affichage global du jeu. Elle utilise la classe `Canvas` pour dessiner les éléments et dépend de `GameBoard` pour accéder aux données du jeu. Cette classe utilise des ressources graphiques Allegro (`ALLEGRO_DISPLAY`, `ALLEGRO_BITMAP`) et des polices (`ALLEGRO_FONT`). Mais d'abord, elle initialise Allegro et vérifie les erreurs d'initialisation.

3.2.2 Canvas

La classe `Canvas` permet d'afficher tous les éléments du jeu. Celle-ci utilise les getters de `GameBoard` pour accéder aux entités de la partie modèle, e.g. la raquette, les balles, les briques.

Afin de diminuer le couplage entre la vue et le modèle, notre affichage est découpé en 2 étapes se répétant pour chaque élément du jeu à afficher :

1. **Traduction** : Le `Canvas` construit l'élément servant à l'affichage correspondant à l'entité de la `GameBoard` à afficher. Par exemple, pour afficher la raquette, nous construisons l'objet `RacketUi` sur base de la `Racket` du `GameBoard`.
2. **Affichage** : Le `Canvas` appelle la méthode `draw` de l'élément à dessiner pour afficher celui-ci.

3.2.3 Formes Géométriques

— **Circle** : Classe de base pour les objets de forme circulaire, e.g. `BallUi`.

— **Rectangle** : Classe de base pour les objets rectangulaires, e.g. `WallUi`, `RacketUi`.

— **Responsabilités** :

  — Gérer les attributs géométriques (position, dimensions, couleurs).

  — Fournir une méthode `draw()` pour dessiner la forme.

3.2.4 Autres Classes

— **WallUi** : Représente les murs dans l'interface utilisateur, hérite de `Rectangle`.

— **RacketUi** : Représente la raquette du joueur, hérite de `Rectangle`.

— **LazerUi** : Représente les lasers, hérite de `Rectangle`.

— **BrickUi** : Gère l'affichage des briques, hérite de `Rectangle` et travaille avec `AbstractBrick`.

— **BonusPillUi** : Affiche les bonus tombants, hérite de `Rectangle`.

— **BallUi** : Représente les balles dans le jeu, hérite de `Circle`.

3.3 Contrôleur

Toutes les classes utilisées pour le contrôleur se trouvent dans le dossier `src/controller`.

La classe `ControllerGame` gère la logique principale du jeu, y compris les événements, la progression des niveaux et l'interaction entre les éléments du jeu. Celle-ci dépend de `GameBoard` pour toute la logique du jeu, et interagit avec `DisplayGame` pour l'affichage. Elle utilise également `LevelManager` pour gérer les niveaux et charger leurs données.

`ControllerGame` est responsable de :

— Gérer la boucle principale du jeu.

— Gérer les événements d'entrée comme le clavier.
— Charger les niveaux via le `LevelManager`.
— Contrôler les transitions de victoire et de défaite.
— Interagir avec Allegro pour la gestion des événements du timer et de l'affichage.
— Demander à `DisplayGame` d'afficher l'état du jeu en cours lorsque nécessaire.

3.4 Log

Les fichiers de la classe `Log` se trouvent dans le dossier `src/log`.

Dans l'optique de faciliter le débogage, essentiellement pour les collisions, nous ajoutons le singleton `Log`, qui permet d'enregistrer des évènements classés par catégories lors de l'éxécution. Nous pouvons activer/désactiver des catégories et ajouter des évènements dans des catégories. Toutefois, cela nécessite de modifier le code source et de recompiler.

4 Logique du jeu

Dans cette section, nous détaillons le fonctionnement du programme entre son lancement et la première fois que la balle collisionne avec une brique.

Au lancement du jeu, le programme commence par initialiser les différentes classes nécessaires : En premier lieu, nous instancions un objet `ControllerGame`. Celui-ci permet de contrôler le plateau de jeu et la vue. Lors de sa création, `ControllerGame` crée l'objet `GameBoard` dans un `shared_ptr` représentant le plateau de jeu, la partie "modèle". Il construit également un objet `LevelManager`. Ce dernier permet de charger tous les différents niveaux en une fois, avec les murs, la raquette et les briques.

Afin de préparer la `GameBoard` pour le niveau choisi, le `ControllerGame` passe une copie de tous les éléments du jeu obtenus par les getters du `LevelManager` à la `GameBoard` à l'aide des setters. Ensuite, il crée aussi l'objet `DisplayGame` en lui passant le `shared_ptr` pointant ver la `GameBoard` afin que celui-ci affiche le jeu. Lors de sa construction, `DisplayGame` initialise `Allegro` pour pouvoir afficher une fenêtre. À chaque fois que nous avons besoin d'afficher l'état du jeu, `DisplayGame` demande à `GameBoard` l'état du plateau de jeu pour l'afficher.

Une fois que ces différents objets sont créés, le programme installe le nécessaire pour pouvoir utiliser le clavier et la souris dans le jeu. Puis, `ControllerGame` initialise le `ALLEGRO_TIMER` qui permet d'avoir un "tick" à une fréquence donnée. Nous utilisons celui-ci afin de mettre à jour `GameBoard` ainsi que pour afficher l'état du jeu à l'écran. Nous séparons la vitesse de mise à jour de `GameBoard` du nombre d'images affichées par Allegro à chaque seconde afin d'obtenir de meilleurs performances. Le plateau de jeu est mise à jour 500 fois par seconde tandis qu'Allegro affichera une image à l'écran 125 fois par seconde.

Ensuite, le `ControllerGame` démarre le `ALLEGRO_TIMER`. A chaque "tick" du timer, le `ControllerGame` récupère la position de la souris, notifie la `GameBoard` de la nouvelle position de la raquette (en fonction de la position de la souris), et demande à la `GameBoard` de se mettre à jour. A chaque mis à jour, la `GameBoard` met à jour tous ses composants et vérifie si la balle est entrée en collision avec un ou plusieurs objets. Si c'est le cas, elle résout les collisions (7), ajuste le score et fait apparaitre les pillules de bonus si nécessaire en les ajouttant au vecteur prévu à cet effet.

Pour les briques, elle signale également à la brique que celle-ci est touchée. Si la durabilité de la brique est 0, `GameBoard` supprime celle-ci de son vecteur de briques. Pour les pillules de bonus et les lasers, il n'y pas de collision à résoudre : `GameBoard` applique l'effet correspondant (i.e. destruction de brique, enregistrer le bonus comme bonus actif) et supprime simplement l'objet du vecteur correspondant.

A chaque mise-a-jour, le `GameBoard` vérifie s'il y a encore au moins une balle en jeu. Si ce n'est pas le cas, il retire une vie. Le `ControllerGame` demande à `GameBoard` le nombre de vies et de briques non-dorées restantes. S'il n'y a plus de briques non-dorées, le joueur a gagné. S'il ne reste aucune vie, le joueur a perdu. Ensuite si un de ces deux évènements survient, le controleur demande l'affichage de l'écran de fin approprié (victoire ou défaite).

Si des événements arrivent dans la *queue* d'Allegro, ils seront traités avant d'afficher le jeu à l'écran.

5 Modèle-Vue-Contrôleur

Notre imlémentation suivant les principes de l'architecture `'Modèle-Vue-Contrôleur'` implique que notre programme soit divisé en trois parties distinctes :

— `Modèle`

— `Vue`

— `Contrôleur`

5.1 Modèle

Cette partie contient toute la logique pour le plateau du jeu. Elle permet d'avoir l'état du plateau de jeu et de le modifier. Celle-ci est donc indépendante des composants servant pour la partie Vue et Contrôleur. La classe `GameBoard` représente l'état du jeu et possède une méthode pour pouvoir être mise à jour. C'est à l'aide de cette méthode que le contrôleur intéragit avec `GameBoard`.

5.2 Vue

La partie vue permet seulement d'afficher l'état du plateau de jeu à un moment donné. La vue demande au modèle son état pour l'afficher à l'écran, à l'aide de la bibliothèque Allegro.

5.3 Contrôleur

Cette troisième partie permet de contrôler le modèle et de notifier la vue lorsqu'il est nécessaire d'afficher l'état du jeu. Grâce à ce celui-ci, nous pouvons gérer le modèle et la vue indépendamment. Le contrôleur possède un `shared_ptr` vers l'objet *GameBoard* pour lui demander de se mettre à jour, et un *DisplayGame* pour lui demander d'afficher l'état du jeu.

6 Conclusion

Pour conclure ce rapport, nous dirons que ce projet nous a permis de développer nos compétences en programmation orientée objet grâce au développement d'une version fonctionnelle du

jeu `Arkanoid`. Le respect de l'architecture `Modèle-Vue-Contrôleur` a été essentielle pour nous répartir les tâches entre nous, et pour structurer notre code et garantir une séparation claire des responsabilités entre la logique du jeu, l'affichage du jeu, et le contrôle des événements.

Le fait que ce projet soit un jeu avec une interface graphique nous a également permis de prendre du plaisir à le réaliser tout en approfondissant notre connaissance du langage `C++`.

Cette expérience nous a non seulement permis de développer un programme abouti, mais également d'affiner nos compétences en conception, en programmation et en travail d'équipe.

7 Interfaces des classes

```
1     ControllerGame();
2
3     virtual ~ControllerGame();
4
5     void process();
```

FIGURE 1 – Interface de ControllerGame

```
1     LevelManager();
2
3     virtual ~LevelManager();
4
5     void nextLevel();
6
7     void previousLevel();
8
9     const vector<shared_ptr<AbstractBrick>> &getBricks();
10
11    const shared_ptr<Racket> getRacket() const;
12
13    const vector<shared_ptr<Border>> &getBorders() const;
```

FIGURE 2 – Interface de LevelManager

```
    Ball(const Vec2 &center, Vec2 directionVec, double radius = BALL_RADIUS,
         double speed = BALL_SPEED);

    Ball(const Ball &other);

    Ball(Ball &&);

    Ball &operator=(const Ball &);

    Ball &operator=(Ball &&);

    virtual ~Ball();

    double getRadius() const noexcept;

    const Vec2 &getCenter() const noexcept;

    const Vec2 &getDirvec() const noexcept;

    void setSpeed(unsigned speed);

    void setDirVec(const Vec2 &vec);

    Vec2 getSimplePenetrationVec(const RectangleShape &rectangleShape) const;

    bool checkCollision(const RectangleShape &rectangleShape) const;

    void collide(const Bounceable &bounceable);

    void update(double deltaTime);
```

FIGURE 3 – Interface de Ball

```
    virtual ~AbstractTimedBonus();

    BonusType getBonusType() const;

    virtual bool update(double deltaT) = 0;

    virtual void reapply(){};
```

FIGURE 4 – Interface de AbstractTimedBonus

```
    BasicTimedBonus(BonusType bonusType);

    BasicTimedBonus(const BasicTimedBonus &);

    BasicTimedBonus(BasicTimedBonus &&);

    BasicTimedBonus &operator=(const BasicTimedBonus &);

    BasicTimedBonus &operator=(BasicTimedBonus &&);

    virtual ~BasicTimedBonus();

    virtual bool update(double deltaT);
```

FIGURE 5 – Interface de BasicTimedBonus

```
    SlowDownBonus();

    SlowDownBonus(const SlowDownBonus &);

    SlowDownBonus(SlowDownBonus &&);

    SlowDownBonus &operator=(const SlowDownBonus &);

    SlowDownBonus &operator=(SlowDownBonus &&);

    virtual ~SlowDownBonus();

    virtual void reapply() override;

    virtual bool update(double deltaT) override;

    virtual double getSlowDownFactor() const;
```

FIGURE 6 – Interface de SlowDownBonus

```
    BonusPill(const Vec2 &center, BonusType bonusType);

    BonusPill(const BonusPill &);

    BonusPill(BonusPill &&);

    BonusPill &operator=(const BonusPill &);

    BonusPill &operator=(BonusPill &&);

    BonusType getBonusType() const;

    void update(double deltaTime);
```

FIGURE 7 – Interface de BonusPill

```
enum class BonusType {
    None,
    SlowDown,
    ExtraLife,
    WideRacket,
    SplitBall,
    Lazer
};
```

FIGURE 8 – Enuméré BonusType

```
1     Border(const Vec2 &center, double width, double height);
2
3     Border(const Vec2 &topLeft, const Vec2 &bottomRight);
4
5     Border(const Border &other);
6
7     Border(Border &&);
8
9     Border &operator=(const Border &);
10
11    Border &operator=(Border &&);
12
13    virtual ~Border();
```

FIGURE 9 – Interface de Border

```
1     enum class BounceType { Vertical, Horizontal, Corner };
2
3     virtual ~Bounceable();
4
5     virtual Vec2 getDirVecAfterBounce(const Vec2 &closestPoint,
6                                       const Vec2 &dirVec) const;
7
8     virtual BounceType getBounceType(const Vec2 &point) const final;
9
10    static std::string bounceTypeToString(BounceType bounceType);
```

FIGURE 10 – Interface de Bounceable

```
    enum class Color : size_t {
        defaultBrick = 1,
        white = 50,
        orange = 60,
        cyan = 70,
        green = 80,
        red = 90,
        blue = 100,
        magenta = 110,
        yellow = 120,
        silver = 200,
        gold = 0,
    };

    virtual ~AbstractBrick();

    static std::unique_ptr<AbstractBrick>
    makeBrick(const Vec2 &center, double width, double height, Color color,
              BonusType bonusType = BonusType::None);

    virtual BonusType hit();

    virtual Color getColor() const;

    virtual size_t getScore() const;

    virtual uint8_t getDurability() const;

    virtual bool isDestroyed() const;

    virtual BonusType getBonusType() const;

    virtual bool hasBonus() const;

    virtual std::shared_ptr<AbstractBrick> clone() = 0;
```

FIGURE 11 – Interface de AbstractBrick

```
    BasicBrick(const Vec2 &center, double width, double height, Color color,
               uint8_t durability, BonusType bonusType = BonusType::None);

    virtual ~BasicBrick();

    virtual std::shared_ptr<AbstractBrick> clone() override;
```

FIGURE 12 – Interface de BasicBrick

```
    GoldBrick(const Vec2 &center, double width, double height);

    virtual ~GoldBrick();

    virtual BonusType hit() override;

    virtual std::shared_ptr<AbstractBrick> clone() override;
```

FIGURE 13 – Interface de GoldBrick

```
    GameBoard();

    virtual ~GameBoard();

    void update(double deltaTime);

    void shootLazer();

    unsigned long getScore() const;

    const LifeCounter &getLife() const;

    unsigned long getNumBricks() const;

    const std::vector<std::shared_ptr<Ball>> &getBalls() const;

    const std::vector<std::shared_ptr<AbstractBrick>> &getBricks() const;

    const std::vector<std::shared_ptr<BonusPill>> &getDescendingBonuses() const;

    const Racket &getRacket() const;

    const std::vector<std::shared_ptr<Border>> &getBorders() const;

    const std::vector<std::shared_ptr<Lazer>> &getLazers() const;

    void setRacketAtX(double centerX);

    void setBricks(const std::vector<std::shared_ptr<AbstractBrick>> &bricks);

    void setRacket(const std::shared_ptr<Racket> racket);

    void setBorders(const std::vector<std::shared_ptr<Border>> &borders);

    void resetLifeCounter();

    void resetScore();

    void saveBestScore();

    unsigned long getBestScore() const;

    void resetBestScore();

    void clear();
```

FIGURE 14 – Interface de GameBoard

```
    Lazer(const Vec2 &center);

    Lazer(const Lazer &);

    Lazer(Lazer &&);

    Lazer &operator=(const Lazer &);

    Lazer &operator=(Lazer &&);

    void update(double deltaTime);
```

FIGURE 15 – Interface de Lazer

```

    LifeCounter();

    LifeCounter(const LifeCounter &);

    LifeCounter(LifeCounter &&);

    LifeCounter &operator=(const LifeCounter &);

    LifeCounter &operator=(LifeCounter &&);

    LifeCounter(unsigned numLives);

    virtual ~LifeCounter();

    void reset();

    const LifeCounter &operator--();

    const LifeCounter &operator++();

    const LifeCounter &operator+=(unsigned numLife);

    void setNumLives(uint8_t numLives);

    operator unsigned() const;
```

FIGURE 16 – Interface de LifeCounter

```
1     Racket(const Vec2 &center, double width, double height);
2
3     Racket(const Racket &other);
4
5     Racket(Racket &&);
6
7     Racket &operator=(const Racket &);
8
9     Racket &operator=(Racket &&);
10
11    virtual ~Racket();
12
13    void setCenterX(double centerX);
14
15    Vec2 getDirVecAfterBounce(const Vec2 &closestPoint,
16                              const Vec2 &dirVec) const override;
```

FIGURE 17 – Interface de Racket

```
    RectangleShape(const Vec2 &center, double width, double height);

    RectangleShape(const Vec2 &topLeft, const Vec2 &bottomRight);

    RectangleShape(const RectangleShape &other);

    RectangleShape(RectangleShape &&);

    RectangleShape &operator=(const RectangleShape &);

    RectangleShape &operator=(RectangleShape &&);

    virtual ~RectangleShape();

    void setCenter(const Vec2 &centerPos);

    void setWidth(double newWidth);

    void setHeight(double newWidth);

    const Vec2 &getCenter() const noexcept;

    double getWidth() const noexcept;

    double getHeight() const noexcept;

    double getLeft() const noexcept;

    double getRight() const noexcept;

    double getBottom() const noexcept;

    double getTop() const noexcept;

    Vec2 getTopLeft() const noexcept;

    Vec2 getTopRight() const noexcept;

    Vec2 getBottomLeft() const noexcept;

    Vec2 getBottomRight() const noexcept;

    bool isOverlapping(const RectangleShape &other);
```

FIGURE 18 – Interface de RectangleShape

```
    ScoreManager();

    ScoreManager(const ScoreManager &);

    ScoreManager(ScoreManager &&);

    ScoreManager &operator=(const ScoreManager &);

    ScoreManager &operator=(ScoreManager &&);

    ScoreManager(unsigned long currentScore, unsigned long bestScore)
        : currentScore_(currentScore), bestScore_(bestScore) {}

    virtual ~ScoreManager();

    void increaseScore(unsigned long value);

    void resetScore();

    void saveScore();

    unsigned long getCurrentScore() const;

    unsigned long getBestScore() const;

    void resetBestScore();
```

FIGURE 19 – Interface de ScoreManager

```
    double x;
    double y;

    Vec2();

    Vec2(double xComponent, double yComponent);

    Vec2(const Vec2 &);

    Vec2(Vec2 &&);

    Vec2 &operator=(const Vec2 &);

    Vec2 &operator=(Vec2 &&);

    virtual ~Vec2();

    double getModule() const;

    const Vec2 &normalize();

    Vec2 normalized() const;

    Vec2 clamped(const Vec2 &min, const Vec2 &max) const;

    bool operator==(const Vec2 &other) const;

    Vec2 operator+(const Vec2 &vec) const;

    Vec2 &operator+=(const Vec2 &vec);

    Vec2 operator-(const Vec2 &vec) const;

    Vec2 &operator-=(const Vec2 &vec);

    Vec2 operator-() const;

    Vec2 operator*(double scalar) const;

    Vec2 &operator*=(double scalar);

    operator Point() const;

    operator std::string() const;
```

FIGURE 20 – Interface de Vec2

```
    Canvas(shared_ptr<GameBoard> gameBoard, ALLEGRO_FONT *fontBrick);

    virtual ~Canvas();

    void draw();
```

FIGURE 21 – Interface de Canvas

```
    DisplayGame(shared_ptr<GameBoard> gameBoard);

    virtual ~DisplayGame();

    void draw();

    void gameOver();

    void gameWin();

    ALLEGRO_DISPLAY *getDisplay() const;
```

FIGURE 22 – Interface de DisplayGame

```
    Rectangle(Point center, float width, float height,
              ALLEGRO_COLOR fillColor = COLOR_BLACK,
              ALLEGRO_COLOR frameColor = COLOR_WHITE);

    virtual ~Rectangle();

    virtual void draw();
```

FIGURE 23 – Interface de Rectangel

```
    Circle(Point center, float radius, ALLEGRO_COLOR fillColor = COLOR_WHITE,
           ALLEGRO_COLOR frameColor = COLOR_BLACK);

    virtual ~Circle();

    virtual void draw();
```

FIGURE 24 – Interface de Circle

```
    float x = 0, y = 0; // Coordinates of the point

    Point(float valX, float valY);

    virtual ~Point();
```

FIGURE 25 – Interface de Point

```
    BallUi(Point center, float radius, ALLEGRO_COLOR color = COLOR_BLUE);

    virtual ~BallUi();

    virtual void draw() override;
```

FIGURE 26 – Interface de BallUi

```
    BonusPillUi(Point center, float width, float height, BonusType bonusType);

    virtual ~BonusPillUi();

    virtual void draw() override;
```

FIGURE 27 – Interface de BonusPillUi

```
    BrickUi(Point center, float width, float height, ALLEGRO_COLOR color,
            size_t durability, BonusType bonusType,
            const ALLEGRO_FONT *fontBrick);

    virtual ~BrickUi();

    virtual void draw() override;
```

FIGURE 28 – Interface de BrickUi

```
    LazerUi(Point center, float width, float height, ALLEGRO_COLOR = COLOR_RED);

    virtual ~LazerUi();

    virtual void draw() override;
```

FIGURE 29 – Interface de LazerUi

```
    RacketUi(Point center, float width, float height,
             ALLEGRO_COLOR = COLOR_WHITE);

    virtual ~RacketUi();

    virtual void draw() override;
```

FIGURE 30 – Interface de RacketUi

```
    WallUi(Point center, float width, float height,
           ALLEGRO_COLOR = COLOR_WHITE);

    virtual ~WallUi();

    virtual void draw() override;
```

FIGURE 31 – Interface de WallUi

Lexique

Résoudre les collisions Dû à notre approche pour la physique du jeu dite "discrète" (7), certains objets se retrouvent en superposition au lieu de collisionner (et rebondir). Résoudre les collisions consiste à replacer les objets à l'endroit où ils étaient juste avant la collision, et appliquer l'effet de rebond si nécessaire.

Physique discrète Simulation où le temps avance par étapes, ce qui peut parfois entraîner des imprécisions ou des chevauchements entre objets.