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

Dans le cadre de ce projet, nous avons dû réaliser une version fonctionnelle avec une interface graphique du célèbre jeu `Arkanoid` en utilisant les principes de la programmation orientée objet.

Le développement a été réalisé en `C++` avec l'utilisation de la bibliothèque `Allegro` pour l'interface graphique. Le projet a été réalisé à l'aide du `Modèle-Vue-Contrôleur` (MVC) afin de séparer clairement les différentes responsabilités du code.

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

— 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

Pour ce projet, 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 ces différentes tâches additionnelles :

— 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

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. Cela est implémenté via la méthode `update`. Chaque fois que celle-ci est appelée à un instant T, avec un paramètre $deltaT$, `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.

3.1.3 RectangleShape

La classe `RectangleShape` est utilisée pour représenter n'importe quel objet rectangulaire, tel qu'une brique ou 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`. Afin de pouvoir profiter du polymorphisme, 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`.

Puisque dans le jeu, la raquette doit être déplacée uniquement latéralement, nous fournissons le setter `setCenterX`, spécifiquement dédié à cela.

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 par rapport au centre de la raquette.

3.1.8 Ball

La classe `Ball` représente une balle du jeu. Celle-ci possède essentiellement une position, un rayon, et une vitesse, 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 normalisé, c'est-à-dire que son module vaut 1. 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 (6) avec un `RectangleShape`.

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

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 encore plus ralentir. 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 qu'il soit cumulable contrairement aux autres bonus, 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 `deltaTime` secondes.

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 T.

3.1.13 LifeCounter

La classe `LifeCounter` gère le nombre de vies du joueur. Toutefois, la responsabilité de notifier les pertes et gains de vie au `LifeCounter` revient à la `GameBoard`. Par exemple, lorsque le joueur n'a plus de balle en jeu, `GameBoard` doit envoyer un message `LifeCounter` pour décrémenter 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`. La méthode `update` permet de faire monter le bonus de la distance qu'il aurait parcouru après `deltaTime` secondes.

3.2 Vue

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

Cette classe est une plateforme centrale pour dessiner toutes les pièces et éléments du jeu. Elle travaille avec `GameBoard` pour accéder aux entités comme la raquette, les balles, les briques, etc. Elle intègre des objets comme `RacketUi`, `BallUi`, `BrickUi`, et autres pour traduire l'élément modèle en élément vue qui pourra être dessiné par Allegro. Cette classe permet de centraliser les appels de dessin pour tous les éléments du jeu.

3.2.3 Formes Géométriques

— **Rectangle** : Classe de base pour les objets comme `WallUi`, `RacketUi`, et autres.
— **Circle** : Classe de base pour les objets comme `BallUi`.
— **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

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. Il dépend de `GameBoard` pour toute la logique du jeu, et interagit avec `DisplayGame` pour l'affichage. Il utilise également `LevelManager` pour gérer les niveaux et charger leurs données.

Il est responsable de gérer la boucle principale du jeu, de gérer les événements d'entrée comme le clavier, de charger les niveaux via le `LevelManager`, de contrôler les transitions de victoire et de défaite, et d'interagir avec Allegro pour la gestion des événements, du timer, et de l'affichage, tout en appelant `DisplayGame` pour dessiner l'état du jeu en cours.

3.4 Log

Dans l'optique de faciliter le débogage, essentiellement pour les collisions, nous ajoutons le singleton `Log`, permettant d'enregistrer des évènements classés par catégories lors de l'éxécution. Nous pouvons activer ou désactiver des catégories à la compilation en modifiant le code.

4 Logique du jeu

Nous allons maintenant décrire en détail ce qui se passe dans notre code à partir du moment où l'utilisateur lance le programme, et le moment où la balle touche une brique pour la première fois.

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 la grille du jeu et la vue. Lors de sa création, *ControllerGame* crée l'objet *GameBoard* représentant la grille de jeu, la partie "modèle". L'objet *LevelManager* sera également créé. Il 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 *LevelManager* lui passe une copie de ces

différents objets à l'aide des setters. Ensuite, il crée aussi l'objet *DisplayGame* qui gère la partie graphique du jeu. Il 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 de la grille 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 la grille du jeu ainsi que pour afficher l'état du jeu à l'écran. Nous séparons la vitesse de mise à jour de la grille du nombre d'images affichées par Allegro à chaque seconde pour des raisons de performance. La grille sera 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* et à chaque "tick" du timer, la *GameBoard* et tous les objets qu'elle possèdent sont mis à jour. A chaque mis à jour, la *GameBoard* vérifie si la balle est entrée en collision est un ou plusieurs objets. Si c'est le cas, elle résout les collisions (6), 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, celle-ci est supprimée. Pour les pillules de bonus et les lasers, il n'y pas de collision à résoudre, simplement faire disparaitre l'objet suffit.

A la fin de chaque mis-a-jour du *ControllerGame*, le programme vérifie s'il y a encore au moins une balle en jeu. Si ce n'est pas le cas, il retire une vie.

Enfin, le *ControllerGame* demande au `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.

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

5 Modèle-Vue-Contrôleur

Tout au long du développement du programme, nous avons essayé le plus possible de respecter le modèle de conception `'Modèle-Vue-Contrôleur'`. Cela implique que notre programme soit divisé en trois parties distinctes :

— `Modèle`

— `Vue`

— `Contrôleur`

5.1 Modèle

Cette partie contient toute la logique pour la grille du jeu. Elle permet d'avoir l'état de la grille et de la modifier. Elle ne possède donc rien qui est lié à l'affichage comme Allegro. Elle permet de ne devoir uniquement gérer la partie logique du jeu, sans devoir s'occuper d'afficher en plus l'état du jeu. La classe *GameBoard* représente l'état du jeu et possède une méthode pour pouvoir être mise à jour.

5.2 Vue

La partie vue permet seulement d'afficher l'état de la grille à un moment donné. La vue va demander au modèle son état pour l'afficher à l'écran à l'aide de la bibliothèque Allegro. Elle ne peut donc en aucun cas modifier l'état du jeu, donc le modèle. Toutes les méthodes de modification du modèle dans la classe *GameBoard* sont privées, ce qui garantit que la classe *DisplayGame* n'a aucun moyen de modifier *GameBoard*. Les seules méthodes publiques de *GameBoard* sont des getters constants qui assurent qu'aucune modification du modèle n'est possible.

5.3 Contrôleur

Cette troisième partie permet de contrôler le modèle en lui demandant de faire certaines choses, ou en lui demandant de lui donner quelque chose. Le contrôleur permet aussi de dire à la vue quand afficher quelque chose. Grâce à lui, nous pouvons gérer le modèle et la vue en les séparant et les faire fonctionner tous les deux, avec la garantie que la vue ne saura pas modifier le modèle. C'est la classe *ControllerGame* qui joue ce rôle. Elle possède un *shared pointer* vers *GameBoard* pour lui demander de se mettre à jour, et un *shared pointer* vers *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`. L'utilisation du modèle `Modèle-Vue-Contrôleur` a été essentielle pour nous répartir les tâches entre nous deux, 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.

Lexique

Résoudre les collisions Dû à notre approche pour la physique du jeu dite "discrète" (6), 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.