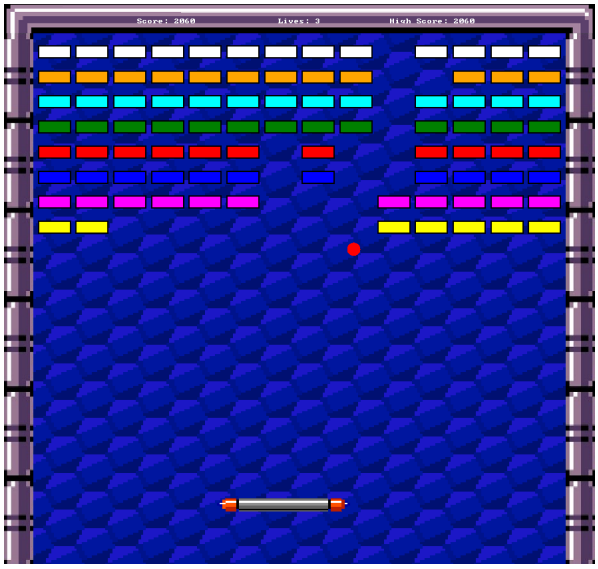
INFO—F-202

Langages de Programmation 2

Rapport Projet

Berthion Antoine - 566199 Omelyanyuk Oleksandra - 589880

2 janvier 2025



1 Introduction

Dans le cadre du cours de **Langages de Programmation (INFO—F-202)**, nous avons développé un projet prenant pour base le concept du jeu vidéo *Arkanoid*. Ce projet a été implémenté en **C++** en utilisant la bibliothèque graphique **Allegro**.
L'objectif principal du jeu est de détruire des lignes de briques en les frappant à l'aide d'une balle. Cette balle est contrôlée indirectement par une raquette manipulée par le joueur, ainsi que par les murs environnants qui influent sur sa trajectoire par rebonds. Si la balle quitte l'écran par le bas, une vie est décomptée. Lorsque trois vies sont perdues, la partie se termine par un écran de ***game-over*** pour le joueur.

De plus, le jeu contient des bonus qui peuvent faciliter l'expérience, ainsi que des niveaux avec une disposition de briques qui varie.

Dans ce rapport, nous détaillerons les tâches exactes implémentées, ainsi que le fonctionnement général du jeu et la structure du code.

2 Tâches accomplies

2.1 Tâches de base

Les tâches de base consistaient à concevoir un niveau fonctionnel d'un jeu inspiré d'*Arkanoid*. Pour ce faire, nous avons implémenté les éléments principaux, tels que la raquette, les briques, le score, les vies, et un message de fin de jeu.

La raquette permet de faire rebondir la balle en modifiant sa trajectoire. La formule utilisée pour déterminer la nouvelle trajectoire est inspirée des consignes initiales, mais a été légèrement ajustée [1] pour rendre les rebonds plus naturels. La direction de la balle après un rebond est donnée par l'angle α calculé selon l'équation suivante :

$$\alpha = 30 + 150 \left(1 - \frac{x}{L}\right),$$

où x représente la distance entre le point de contact de la balle et le bord gauche de la raquette, et L correspond à la longueur de la raquette.

Les briques sont affichées à l'écran en fonction des données lues depuis un fichier au format *.map*. Ce format de fichier spécifie leur position relative et pourra être amélioré ultérieurement pour inclure le type de chaque brique et les informations relatives aux bonus. Cette structure facilitera la création de différents niveaux et permettra une personnalisation aisée de la disposition des briques.
Le score et le nombre de vies restantes sont affichés en haut de l'écran pendant toute la durée de la partie. Lorsque la partie se termine, un message indiquant le score final s'affiche, offrant également la possibilité de redémarrer le jeu en appuyant sur la touche *Entrée*.

2.2 Tâches additionnelles

Après avoir implémenté la base du jeu, nous avons ajouté plusieurs fonctionnalités supplémentaires.

Premièrement, nous avons défini différents types de briques et intégré cette information dans les fichiers déterminant leur disposition. Cela inclut notamment :

- les **briques argentées**, qui nécessitent deux coups pour être détruites ;
- les **briques dorées**, qui sont indestructibles.

Chaque type de brique est associé à un score spécifique. Cette structure a également permis d'intégrer un système de meilleur score, sauvegardé dans le fichier *highscore.txt*. Ce meilleur score est conservé entre les sessions du jeu, affiché en haut de l'écran aux côtés des vies et du score actuel, et peut être réinitialisé en appuyant sur la touche **R**.

Grâce à ces options, il est possible de concevoir des niveaux variés comportant des briques de différents types. Dans le jeu, les niveaux se succèdent automatiquement lorsqu'un niveau est complété (toutes les briques destructibles ont été éliminées). Il est également possible de changer manuellement de niveau en appuyant sur les flèches gauche ou droite du clavier.

Par ailleurs, nous avons ajouté une fonctionnalité permettant de contrôler la raquette avec la souris, en plus des touches du clavier. Ces deux modes de contrôle sont disponibles simultanément. Le jeu vérifie d'abord l'état du clavier et, si aucune touche n'est activée, il prend en compte les mouvements de la souris vers la gauche ou la droite.

Finalement, des **bonus** ont été ajoutés au jeu. Ces bonus peuvent être contenus dans des briques, et l'information correspondante est également stockée dans les fichiers *.map*. Lorsqu'un bonus est présent, une lettre l'indique sur la brique associée. Les bonus implémentés sont les suivants :

- **Laser** : des lasers peuvent être tirés depuis la raquette vers le haut de l'écran, détruisant toutes les briques touchées, sauf les dorées.
- **Agrandir** : la raquette devient plus large.
- **Attraper** : la raquette attrape la balle, qui peut être relâchée en appuyant sur la touche **Espace**. Si aucune action n'est effectuée, la balle se relâche automatiquement après un certain temps.
- **Ralentissement** : la balle se déplace plus lentement pendant un temps limité.
- **Interruption** : la balle se divise en trois. Aucune vie n'est perdue si deux des trois balles tombent.
- **Joueur** : une vie supplémentaire est accordée au joueur.

Aucun bonus n'a d'effet cumulatif, à l'exception du bonus de ralentissement. Lorsqu'un nouveau bonus est attrapé, l'effet du précédent est annulé.

1. En effet, la formule de base ne permettait des angles qu'entre 30° et 120°. Nous l'avons modifiée à des fins de symétrie.

3 Classes et logique du jeu

Dans cette partie du rapport, nous développerons davantage l'implémentation du jeu ainsi que la logique générale.
Nous nous baserons sur le diagramme des classes ci-dessous dans la figure 1 pour présenter la structure générale du jeu et expliquer les classes implémentées plus en profondeur, ainsi que leurs interactions.

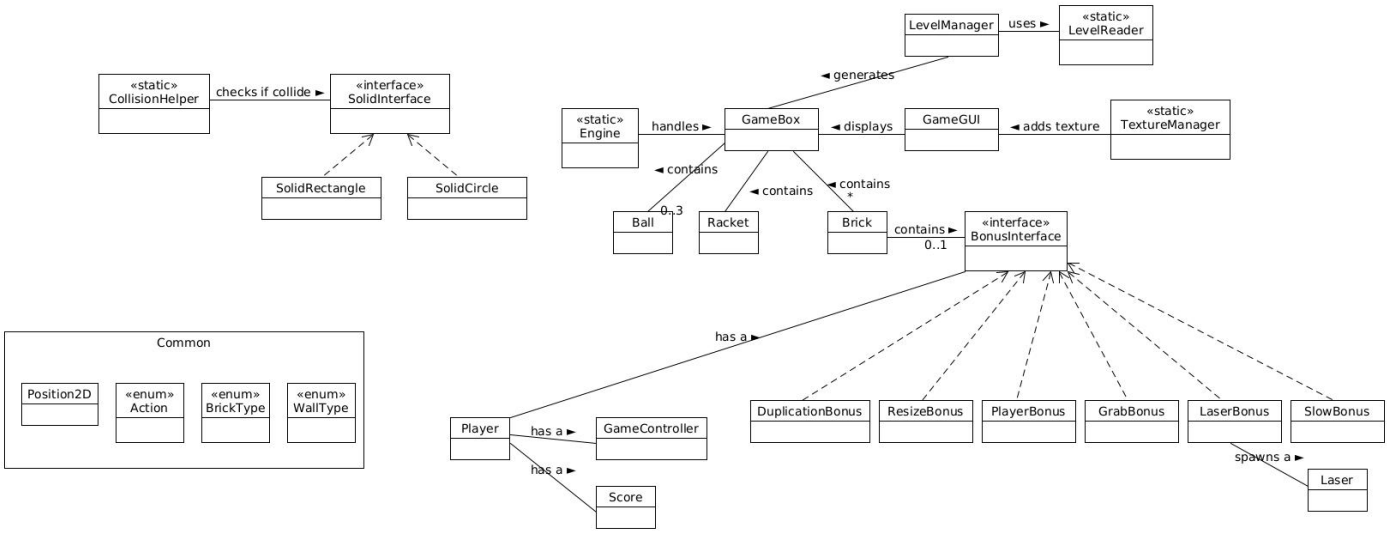


FIGURE 1 – Diagramme de classe du projet Arkanoïd

Nous allons décrire brièvement les différentes classes ainsi que leur fonctionnement.

3.1 SolidInterface

La classe **SolidInterface** est une interface qui regroupe les objets possédant des ***hitboxes***, c'est-à-dire des zones de collision. Bien que la classe ne gère pas directement ces interactions, elle facilite la détection des collisions entre objets solides. Cette interface est implémentée par la classe **SolidRectangle**, utilisée par les briques, la raquette et les bonus (décrits dans les sous-sections suivantes), ainsi que par **SolidCircle**, utilisée pour l'objet **Ball**. Les classes qui réalisent cette interface doivent implémenter les méthodes abstraites afin de compléter l'interface et être utilisées par le reste du code.

3.2 CollisionHelper

La classe statique **CollisionHelper** vient en aide à **SolidInterface** en facilitant la détection des collisions entre les objets qui implémentent cette interface. Ainsi, **CollisionHelper** est capable de gérer les collisions entre toutes les paires d'objets réalisant l'interface **SolidInterface**. Cette approche modulaire permet d'ajouter rapidement le support pour de nouveaux types d'objets solides, en les faisant simplement hériter de l'interface **SolidInterface**, tout en ajoutant les méthodes de détection de collisions avec les autres objets dans **CollisionHelper**.

3.3 Objets du jeu (Ball, Racket et Brick)

Les classes **Ball**, **Racket** et **Brick** représentent respectivement les objets de même nom dans le jeu. Chacun de ces objets dispose d'une hitbox, implémentée via l'interface **SolidInterface**, que nous avons abordée dans la sous-section 3.1. Ces objets possèdent également des attributs permettant de gérer leur vitesse, ainsi qu'une **Position2D** pour assurer leur déplacement dans l'espace de jeu.

3.4 Objets liés au Joueur (Player, GameController, Score)

Les classes **Player**, **GameController** et **Score** gèrent les informations relatives au joueur. La classe **GameController** permet à l'**Engine** du jeu de récupérer la configuration des touches et, grâce à elle, d'interpréter les appuis de touches de l'utilisateur pour les transformer en actions. Quant à la classe **Player**, elle agit comme un conteneur pour les données liées à l'utilisateur, telles que le score actuel. Elle stocke également le nombre de vies du joueur, l'objet actuellement dans son inventaire, l'état de sa raquette, ainsi que d'autres valeurs nécessaires au bon déroulement de la partie. L'accès au score et au **GameController** se fait via l'objet **Player**, facilitant ainsi l'accès à ces informations en passant en paramètre uniquement un pointeur vers l'objet **Player**.

3.5 LevelManager et LevelReader

Ces classes simples sont essentielles pour la lecture et la gestion des niveaux du jeu. La classe **LevelReader** permet de lire les fichiers au format `.map` selon l'encodage décrit dans la section 2.2. Elle crée ainsi des objets **GameBox**, dont les détails sont présentés dans la section 3.7. La classe **LevelManager** gère l'avancée de l'utilisateur à travers les niveaux. Elle permet de changer facilement la **GameBox** actuelle en modifiant le pointeur correspondant, en passant en paramètre une référence vers celui-ci. Cette opération de changement de niveau est effectuée par le **GameEngine**. Les chemins vers les fichiers des niveaux sont stockés dans un vecteur, à travers lequel l'utilisateur va boucler. Lorsque le dernier niveau est terminé, le jeu revient automatiquement au premier niveau.

3.6 Interface visuelle (GameGUI et TextureManager)

L'interface visuelle constitue une part importante de notre projet. Toutefois, elle est séparée de la logique du jeu et n'a aucune connaissance de ce qui se passe en son sein. En effet, elle ne connaît pas les règles du jeu et reçoit uniquement des instructions concernant les éléments à afficher, à partir des informations sur l'état de la partie. L'interface visuelle utilise un **TextureManager** pour récupérer efficacement les textures du jeu et les afficher. Ces textures sont chargées à partir d'images bitmap via la bibliothèque graphique **Allegro**. Une fois chargées, elles ne sont plus rechargées, ce qui permet de réduire l'usage des ressources et d'optimiser les performances.

3.7 GameBox

La **GameBox** est le conteneur principal de notre projet. Elle a en effet accès à la raquette, aux balles, ainsi qu'aux briques, qu'elles soient équipées ou non de bonus. La GameBox conserve également des informations sur les éventuels lasers ou bonus présents à l'écran. En plus de ces objets, la GameBox dispose de plusieurs méthodes permettant d'interagir avec eux. Toutefois, elle ne possède aucune connaissance sur le fonctionnement global du jeu. En effet, elle est limitée à des méthodes d'interaction avec les objets mentionnés précédemment. Toute la logique du jeu est centralisée et gérée par le cœur du jeu, le **GameEngine**, décrit dans la sous-section suivante.

3.8 GameEngine

Le **GameEngine** constitue le moteur de notre jeu. Il est le seul à connaître les règles du jeu. Ainsi, à chaque frame, la méthode dédiée au "tour de logique" est appelée. L'engine se charge de plusieurs tâches essentielles : il vérifie les collisions, applique les réactions appropriées (comme retirer des vies aux briques, faire rebondir la balle, déplacer un laser, etc.), et gère les entrées du joueur via le **GameController**. Il évalue également l'état de la partie (victoire ou défaite), applique la logique des bonus actifs, et met à jour les variables du joueur, telles que le score et le nombre de vies.

La centralisation de la logique dans le **GameEngine** permet de faciliter l'ajout de nouveaux objets interagissant avec les autres, sans avoir à modifier les autres classes. En effet, toutes les interactions se font via le moteur de jeu, ce qui simplifie grandement l'implémentation et la maintenance du code.

3.9 BonusInterface

Dans cette sous-section, nous aborderons brièvement la gestion des bonus. À l'instar de l'interface *SolidInterface*, nous avons souhaité regrouper les différents types de bonus sous une interface commune. Toutefois, le contrat de l'interface **BonusInterface** est moins contraignant : il garantit simplement que les bonus pourront appliquer et annuler leur logique à l'aide de deux méthodes, `applyLogic` et `revertLogic`. Cette approche simplifie grandement l'implémentation des différents types de bonus.
En plus de ces méthodes, **BonusInterface** contient certains attributs utiles à la gestion de la logique des bonus, tels que le flag `isActive` ou le `TTL` (Time To Live), qui détermine la durée de vie du bonus. Bien que ces attributs soient utiles, leur utilisation n'a pas toujours été optimale, faute de temps.

Nous évoquerons aussi le cas particulier du **Bonus Laser**, qui permet l'apparition de lasers à l'écran. Ce bonus nécessite un traitement spécifique dans le **GameEngine** pour générer et gérer les lasers. Cependant, il aurait été possible de gérer cette logique directement dans le bonus, mais par manque de temps, nous avons préféré centraliser cette logique dans le **GameEngine**. Cette incohérence résulte d'un choix structurel qui aurait pu être optimisé si le projet en avait permis une gestion plus souple des bonus.

4 Modèle-Vue-Contrôleur

Dans cette section, nous discuterons de l'application du modèle **Modèle-Vue-Contrôleur** (MVC) dans notre implémentation du projet.
L'architecture Modèle-Vue-Contrôleur se compose de trois parties distinctes : la vue, le contrôleur et le modèle. Nous allons expliquer comment chaque composant est représenté dans notre projet.

4.1 Modèle

Le **modèle** contient les données principales du jeu, telles que les objets du jeu (balles, briques, raquette, etc.), les scores, et les vies restantes (par l'intermédiaire de l'objet **Player**). Dans notre projet, le modèle est représenté par la classe `GameBox`. Cette classe agit comme un conteneur pour ces données et inclut des méthodes pratiques permettant de manipuler et de modifier ces informations. La logique du jeu (telles que les règles concernant les collisions et les bonus) ainsi que l'affichage (via la vue) sont séparés de la `GameBox`, ce qui permet à celle-ci de se concentrer uniquement sur la gestion des données, tout en étant utilisée par le contrôleur et la vue.

4.2 Vue

La **vue** dans notre architecture correspond à la présentation graphique de l'application. Elle est responsable de l'affichage des informations à l'utilisateur, sans connaissance des règles du jeu ou de la logique interne. Dans notre projet, cette fonction est assurée par la classe `GameGUI`, munie de son `TextureManager`. La vue ne gère que l'affichage des données fournies par le modèle et reste totalement découplée de la logique du jeu.

4.3 Contrôleur

Le **contrôleur** est chargé de la gestion de la logique du jeu et des actions de l'utilisateur. Dans notre architecture, la classe `GameEngine` joue le rôle principal du contrôleur. Elle est responsable de la mise à jour de l'état du jeu en fonction des actions de l'utilisateur (comme les entrées clavier ou souris) et de la logique de jeu (gestion des collisions, calcul des scores, etc.). En d'autres termes, `GameEngine` modifie l'état du modèle, coordonne l'interaction entre la vue et le modèle et assure le bon déroulement du jeu.

5 Utilisation des LLM

Dans cette courte partie, nous discuterons de l'utilisation des LLM dans le cadre du projet. Aucun LLM n'a été utilisé pour l'aspect implémentation ainsi que la compréhension du projet. Cependant, ce

rapport à été corrigé du point de vue de la syntaxe et de la grammaire par DeepL ainsi qu'un modèle GPT. Notons tout de même qu'aucune des informations du rapport n'a été produite par une autre personne que l'auteur.

6 Conclusion

Ce projet a permis de mettre en œuvre une version du jeu *Arkanoid* en utilisant le langage C++ et la bibliothèque graphique Allegro. À travers l'implémentation de diverses fonctionnalités telles que la gestion des collisions, des bonus, et des niveaux, nous avons pu développer un jeu complet et fonctionnel.

L'architecture Modèle-Vue-Contrôleur a joué un rôle crucial dans la séparation des responsabilités, facilitant ainsi l'extension du jeu et la maintenance du code. En intégrant différentes mécaniques de jeu et en offrant une interface graphique intuitive, nous avons abouti à un projet à la fois technique et ludique. Ce travail met en lumière l'importance de la modularité dans la conception de jeux vidéo et permet de mieux comprendre les interactions entre les différents composants d'une application complexe.