Projet Langage de programmation
Rapport

Gulzar Zia 000595624, Ando Rasamimanana 000570596

28-08-2025

Contents

1 Introduction

Dans le cadre du cours INFO-F-202 - Langages de programmation 2 de l'Université libre de Bruxelles nous avous été charger d'effectuer un projet, il a pour objectif de développer un jeu vidéo en utilisant la bibliothèque graphique Allegro. Le jeu à implémenter est une version personnalisée d'Arkanoid, un classique des jeux d'arcade sorti en 1986. Inspiré de Breakout (1976) et de Pong (1972), Arkanoid propose une expérience ludique où le joueur contrôle une raquette pour casser des briques à l'aide d'une balle.

Le développement de ce projet permet de mettre en pratique des concepts fondamentaux de la programmation orientée objet (POO), tels que la gestion des classes, l'encapsulation, et la modularité du code. Ce rapport détaille les différentes étapes de réalisation du jeu, les choix de conception adoptés, ainsi que les tâches accomplies dans le cadre du projet. Nous présenterons également les principales fonctionnalités du jeu et la logique sous-jacente au code, en mettant en avant l'utilisation du modèle de conception Modèle-Vue-Contrôleur (MVC).

Ce projet constitue une opportunité d'approfondir les compétences en développement logiciel tout en explorant les aspects ludiques et techniques d'un jeu d'arcade emblématique.

2 Tâches accomplies

2.1 Tâches de base

1. Raquette
2. Briques
3. Score
4. Vies
5. Victoire

2.2 Tâches additionnelles

1. Souris / trackpad
2. Briques colorées et highscore
3. Briques argentées et dorées
4. Bonus (power-up)
5. Laser
6. Agrandir
7. Attraper
8. Ralentissement
9. Interruption
10. Joueur

3 Contrôleurs et Bonus

GameContext. Conteneur de références centralisant l'état (`Paddle`, `Ball`, `Laser`, vies, `Level`, `Block`/`Capsule`). Partagé avec tous les contrôleurs et `Bonus`.

MovementController

```
class MovementController{
public:
    MovementController(GameContext& ctx);
    void update(float delta);
};
```

Résumé. Met à jour les positions des entités mobiles (balles, capsules, lasers).

PaddleController

```
class PaddleController {
public:
    PaddleController(Paddle& p, std::vector<Laser>& l,
                     float left, float right, BonusManager& bm);

    void onKeyDown(int keycode);
    void onKeyUp(int keycode);
    void onMouseMove(float mouseX);
    void update(float deltaTime);

    void activateLaserBonus();
    void launchBall();
};
```

Résumé. Gère entrées utilisateur (clavier/souris), déplacement, tir (laser), lancement de balle.

CollisionController

```
class CollisionController {
public:
    CollisionController(GameContext& ctx,
                        ScoreManager& sm, BonusManager& bm);

    void checkBallBlockCollisions(Ball& ball);
    void checkCapsulePaddleCollision();
    void checkLaserBlockCollisions();
    bool checkAllCollision();
};
```

Résumé. Détecte et résout les collisions (balles/briques, raquette, capsules, lasers). **Collaborations.** Met à jour `ScoreManager`, notifie `BonusManager`.

BonusManager

```
class BonusManager {
public:
    BonusManager(GameContext& ctx);
    void update();
    void onBlockDestroyed(const Block& block);
    void onCapsuleCollected(const Capsule& capsule);
    const std::shared_ptr<Bonus> getCurrentBonus() const;
    GameContext& getGameContext();
};
```

Résumé. Orchestration des bonus : génération, collecte, application, annulation. **Collaborations.** Collabore avec `CollisionController` et `PaddleController`.

Bonus — approche synthétique

Concept. Les bonus dérivent d'une classe abstraite `Bonus` (activation/annulation, visibilité, cycle de vie).

Types. `BonusType` catégorise : ENLARGE, LASER, CATCH, EXTRA_LIFE, SLOW_BALL, SPLIT.

Bonus simples — tableau récapitulatif

Nom	Effet	Notes d'implémentation
Enlarge	Augmente largeur raquette (temp.)	`Paddle::enlarge(amount)`; restaurer taille initiale.
ExtraLife	+1 vie	Incrémente `lives` dans `GameContext`.
Slow	Ralentit les balles (temp.)	Modifie `Speed` des `Ball`, restaure à expiration.

Bonus spécifiques

Laser. Active/désactive le mode laser de la raquette. **Catch.** Permet de capturer puis relancer la balle. **Interruption/Split.** Duplique les balles et/ou annule des effets en cours.

4 Modèle

Types de base & objet commun

```
class Point{ float x,y; /* get/set,+,+= */ };
class Size { float w,h; /* get/set */ };
class Speed{ float vx,vy; /* get/set,+,+= */ };

class Object{
```

```
  Point pos; Size size; Speed spd; bool visible=true;
public:
  virtual void updatePosition(float dt=1.0f){
    pos = Point{ pos.getX()+spd.getSpeedX()*dt,
                 pos.getY()+spd.getSpeedY()*dt };
  }
  virtual ~Object()=default;
};
```

Résumé. `Point`, `Size`, `Speed` = primitives; `Object` = base commune.

Ball

```
class Ball: public Object{
  float radius; bool caught=false;
public:
  void updatePosition() override;
  void catchBall(const Paddle&); void launchBall();
  std::vector<Ball> split();
  bool IsBallMissed() const; void resetBallPosition();
};
```

Résumé. Balle mobile (capture/relance, split, sortie).

Block (incl. Gold/Silver)

```
class Block: public Object{
  int hp; unsigned score;
public:
  int getHp() const; void incrementHits(); virtual void onHit();
  unsigned getScoreValue() const;
  void setCapsule(std::shared_ptr<Capsule> c);
  bool hasCapsule() const;
};
```

Résumé. Brique destructible (PV, score, capsule). **GoldBlock** : indestructible. **SilverBlock** : robuste (HP élevés).

Capsule

```
class Capsule: public Object{
  std::shared_ptr<Bonus> bonus;
public:
  void updatePosition() override;
  bool checkCollision(Paddle& p) const;
};
```

Résumé. Transporte un bonus; collectée par la raquette.

Laser

```
class Laser: public Object{
  bool active=true;
public:
  void updatePosition(float dt) override;
  bool isActive() const; void setInactive();
};
```

Résumé. Projectile vertical tiré en mode laser.

Paddle

```
class Paddle: public Object{
  bool laser_mode=false;
public:
  void enlarge(float amount);
  bool isLaserModeEnabled() const;
  void setLaserMode(bool on);
};
```

Résumé. Raquette contrôlée par le joueur; peut s'agrandir, capturer balle, tirer laser.

Level

```
class Level {
public:
    Level(Size screenSize, size_t rows, size_t cols,
          Size blockSize, float spacing_x, float spacing_y);
    Level(Size blockSize, Point startPos, Point spacing);

    void generateBlocks(const std::vector<std::string>& layout);
    std::vector<std::shared_ptr<Block>>& getBlocks();
    void resetBlocks();
    std::vector<std::shared_ptr<Capsule>> getCapsules() const;
    const std::vector<std::shared_ptr<Block>>& get_blocks() const;
};
```

Résumé. Représente un niveau de jeu (grille de briques, espacement, capsules). **Méthodes clés.** `generateBlocks()` crée les briques selon un layout, `resetBlocks()` réinitialise, getters exposent briques et capsules. **Collaborations.** Utilisé par `LevelManager` pour charger/contrôler la progression.

LevelManager

```
class LevelManager {
public:
    LevelManager(Size blockSize, Point startPos, Point spacing);

    bool loadLevelFromFile(const std::string& filename);
    bool loadNextLevel();
    bool loadLevel(size_t levelIndex);
```

```
    bool hasNextLevel() const;

    std::shared_ptr<Level> getCurrentLevel() const;
    size_t getCurrentLevelIndex() const;
    size_t getTotalLevels() const;

    void resetToFirstLevel();
    void loadLevelFilesFromDirectory(const std::string& directory);
};
```

Résumé. Gère la progression entre niveaux. **Méthodes clés.** `loadLevelFromFile()`, `loadNextLevel()`, `hasNextLevel()` pour navigation ; getters exposent l'état courant et la liste ; `resetToFirstLevel()` pour recommencer. **Collaborations.** Interagit avec les entités du jeu via le `Level` courant.

Vues

Toutes les vues héritent de `Drawable`, conservent une référence sur un objet du modèle, et implémentent `draw()` via Allegro. `GameView` orchestre ensuite l'ordre de dessin en appelant `draw()` sur chaque vue.

Base : Drawable

```
class Drawable {
public:
    virtual void draw() = 0;

    ALLEGRO_COLOR getFrameColor() const;
    ALLEGRO_COLOR getFillColor() const;
    void setFrameColor(const ALLEGRO_COLOR& new_color);
    void setFillColor(const ALLEGRO_COLOR& new_color);

    virtual ~Drawable() = default;
};
```

Rôle. Contrat d'affichage minimal avec gestion des couleurs cadre/remplissage et méthode `draw()`.

Exemple concret : BlockView

```
class BlockView : public Drawable {
public:
    BlockView(const Block& b, ALLEGRO_COLOR frame, ALLEGRO_COLOR fill);
    void draw() override;
};
```

Rôle. Dessine une brique si elle est visible (rectangle + couleurs). **Collaborations.** Lit l'état du modèle `Block` (position, taille, visibilité).

Autres variantes (même motif)

```
class PaddleView : public Drawable {
public:
    PaddleView(const Paddle& p, ALLEGRO_COLOR frame, ALLEGRO_COLOR fill);
    void draw() override;
};
```

```
class CapsuleView : public Drawable {
public:
    CapsuleView(const Capsule& c, ALLEGRO_COLOR frame, ALLEGRO_COLOR fill);
    void draw() override;
};

class LaserView : public Drawable {
public:
    LaserView(const std::vector<Laser>& l, ALLEGRO_COLOR frame, ALLEGRO_COLOR
        fill);
    void draw() override;
};
```

Résumé. Même principe : chaque vue lit son modèle associé et le dessine (rectangle/forme) avec couleurs.

Orchestrateur : GameView

```
class GameView {
public:
    void addRenderable(std::unique_ptr<Drawable> d);
    void renderAll();
    void clearRenderables();
};
```

Rôle. Conteneur de vues; appelle `draw()` sur chaque `Drawable` dans l'ordre d'insertion.

5 Logique du jeu

Lors du lancement du programme, la bibliothèque Allegro est initialisée afin de gérer les fonctionnalités graphiques, les entrées utilisateur et les timers. Les ressources nécessaires (couleurs, polices, images, sons) sont chargées, puis une fenêtre de jeu est créée avec des dimensions prédéfinies. Les différents objets du jeu sont ensuite instanciés : la raquette (`Paddle`), la balle (`Ball`), les briques (`Block`) et les capsules bonus (`Capsule`). Chaque brique peut avoir une couleur différente et, dans certains cas, contenir une capsule bonus qui sera libérée lorsqu'elle sera détruite.

Boucle principale. Après l'initialisation, la boucle du jeu démarre. Elle gère :

- le rendu graphique,
- les entrées utilisateur,
- la logique de jeu (déplacements, collisions, effets).

Entrées utilisateur. Le programme attend les événements : touches du clavier ou mouvements de la souris.

- Flèches gauche/droite : déplacement de la raquette.

- Barre d'espace (si laser activé) : tir de lasers vers le haut depuis la position de la raquette.

Déplacements et collisions.
- La balle est mise à jour à chaque itération selon sa vitesse et sa direction.
- Si elle touche le bord supérieur ou les côtés : rebond.
- Si elle dépasse le bord inférieur : perte d'une vie et repositionnement au centre.
- Si elle touche la raquette : rebond avec direction dépendant du point d'impact.
- Si elle touche une brique : la brique devient invisible, la balle rebondit et, si une capsule était contenue, celle-ci tombe.

Capsules. Les capsules tombent verticalement après la destruction d'une brique. Si le joueur les attrape avec la raquette, elles déclenchent différents effets :
- agrandir la raquette,
- activer le mode laser,
- ralentir les balles,
- donner une vie supplémentaire, etc.

Condition de fin. Le jeu continue en mettant à jour les objets, en détectant les collisions et en appliquant les effets jusqu'à ce que :
- toutes les briques soient détruites (victoire), ou
- que le joueur perde toutes ses vies (défaite).

6 Modèle-Vue-Contrôleur (MVC)

L'architecture logicielle du projet suit le modèle de conception *Modèle-Vue-Contrôleur* (MVC), un paradigme classique en génie logiciel qui permet de séparer les responsabilités entre gestion des données, affichage et logique de contrôle. Cette séparation améliore la maintenabilité, la réutilisabilité et les possibilités d'extension du code.

Modèle

Le **Modèle** regroupe toutes les classes représentant l'état et la logique interne du jeu indépendamment de l'affichage. Cela inclut :
- les entités de base (`Ball`, `Paddle`, `Block`, `Capsule`, `Laser`), héritant de `Object`, qui encapsulent position, vitesse, dimensions et comportements ;
- les gestionnaires d'état global comme `Level`, `LevelManager` et `ScoreManager`, qui structurent la progression et le suivi des scores.

Le modèle n'a aucune dépendance vers Allegro ou les entrées utilisateur : il se limite à la logique métier du jeu.

Vue

La **Vue** assure l'affichage graphique des éléments du modèle à l'écran, via la bibliothèque Allegro. Chaque vue (`BlockView`, `PaddleView`, `CapsuleView`, `LaserView`) hérite de `Drawable` et contient une référence vers l'objet du modèle correspondant. La méthode `draw()` rend visuellement l'état courant de l'objet. Le rôle central de la vue est purement graphique : elle observe le modèle sans jamais le modifier.

Contrôleur

Le **Contrôleur** coordonne les interactions entre le joueur, le modèle et la vue. Il regroupe :

- `PaddleController`, qui traduit les entrées clavier/souris en déplacements ou actions (tir laser, lancement de balle) ;
- `MovementController`, qui met à jour les positions des objets mobiles à chaque itération de la boucle ;
- `CollisionController`, qui détecte les collisions et met à jour le modèle en conséquence (destruction de briques, rebonds, capsules collectées) ;
- `BonusManager`, qui orchestre la génération et l'application des effets de bonus.

Ainsi, le contrôleur agit comme médiateur : il interprète les événements, manipule le modèle et déclenche le rafraîchissement des vues.

Resume MVC

L'application de l'architecture MVC dans ce projet illustre plusieurs bénéfices identifiés en ingénierie logicielle :

- **Séparation des préoccupations** : la logique métier (Modèle), l'affichage (Vue) et les entrées/utilisateurs (Contrôleur) sont clairement isolés ;
- **Testabilité accrue** : le modèle peut être testé indépendamment de l'interface graphique ;
- **Réutilisabilité et extensibilité** : de nouvelles vues (par exemple un affichage 3D) ou de nouveaux contrôleurs (IA, mode multijoueur) peuvent être intégrés sans réécrire le modèle ;
- **Clarté conceptuelle** : les étudiants peuvent ainsi expérimenter un design pattern fondamental appliqué à un cas concret de jeu vidéo.

En résumé, les classes du projet incarnent fidèlement le patron MVC : le **Modèle** gère l'état, les **Vues** en fournissent la représentation graphique, et les **Contrôleurs** coordonnent les actions du joueur et l'évolution du jeu.

7 Conclusion

Ce projet de jeu inspiré d'Arkanoid a permis d'appliquer les concepts de la programmation orientée objet, notamment la modularité, la gestion des classes,

et les interactions entre objets. L'intégration de la bibliothèque Allegro nous a offert une meilleure compréhension des bibliothèques graphiques et de la gestion des événements en temps réel.

La structure du code s'appuie partiellement sur le modèle MVC, avec une séparation entre la logique du jeu, l'affichage, et les interactions utilisateur. Cette organisation a facilité la gestion des collisions, des mouvements des objets, et des bonus. L'ajout de fonctionnalités supplémentaires, telles que les capsules et le mode laser, a enrichi le gameplay tout en consolidant nos compétences en développement.

En résumé, ce projet a été une expérience formatrice qui a combiné la théorie et la pratique, tout en nous permettant de créer un jeu fonctionnel et personnalisable.