

**GROUPE AD1A**

DELPECH Maxime

FILAUDEAU Marius

LABBI Noaym

VERON Lucas

XU Nic

UNIVERSITÉ  
PARIS-CITÉ

PROJET DE

**PROGRAMMATION**

# Table des matières

[Table des matières 2](#_Toc167038748)

[Introduction 3](#_Toc167038749)

[Objectif 3](#_Toc167038750)

[L'équipe 3](#_Toc167038751)

[Comment jouer ? 3](#_Toc167038752)

[Outils utilisés 4](#_Toc167038753)

[Plateforme collaborative 4](#_Toc167038754)

[Choix de l’IDE + version JDK + Système d’exploitation 4](#_Toc167038755)

[Contraintes (demandées par le sujet) 4](#_Toc167038756)

[Librairies 4](#_Toc167038757)

[Contraintes Techniques 4](#_Toc167038758)

[Conception 4](#_Toc167038759)

[Choix de la structure 4](#_Toc167038760)

[Structure du projet (+ diagramme en annexe) 5](#_Toc167038761)

[Implémentations (dossier src) 5](#_Toc167038762)

[Gestion du sous-package display 5](#_Toc167038763)

[Gestion de la vue 5](#_Toc167038764)

[Gestion de l’engine 6](#_Toc167038765)

[Gestion du sous package game (et surtout breakout) 7](#_Toc167038766)

[Fonctionnement général 7](#_Toc167038767)

[Création des entités dans un jeu 8](#_Toc167038768)

[Gestion de la logique 8](#_Toc167038769)

[Gestion des briques 8](#_Toc167038770)

[Gestion des bonus 9](#_Toc167038771)

[Gestion des niveaux 10](#_Toc167038772)

[Améliorations possibles 10](#_Toc167038773)

[Conclusion 10](#_Toc167038774)

[Annexes 11](#_Toc167038775)

# Introduction

## Objectif

L’objectif de ce projet est de créer un jeu de casse-briques, en utilisant comme fondement les connaissances acquises lors du premier semestre de la deuxième année de licence informatique en Programmation Orientée Objet. Ce projet vise non seulement à consolider les acquis du premier semestre, mais également à améliorer notre capacité à travailler efficacement en groupe, comme enseigné lors de la période de préprofessionnalisation du semestre précédent. Le résultat final de ce projet démontrera ainsi la maîtrise des principes fondamentaux du travail en groupe (tels que l’utilisation de git, la communication, …) des membres du groupe.

## L'équipe

Le groupe fût bien équilibré car chacun avait un rôle bien défini, si nous devions attribuer un rôle à chaque membre du groupe comme dans un projet d’entreprise, cela donnerait : (\*)

* DELPECH Maxime : Responsable de l’expérience utilisateur, rôle essentiel pour la création des menus et l’aspect visuel du jeu.
* VERON Lucas et FILAUDAU Marius : Responsables de la partie physique du code, rajoutent continuellement de la physique et adaptent au fil du temps la partie physique à chaque refactorisation majeure dans la partie breakout.
* LABBI Noaym : L’intermédiaire entre la partie jeu et la partie physique, a joué un rôle majeur durant le merge de la physique grâce aux connaissances acquises sur la structure et la partie physique.
* XU Nic : Responsable du projet dans sa phase de conception et développement durant les premières semaines et a pris le rôle de pilote de projet et développeur. A majoritairement contribué dans la création de la structure et assuré la modularité du projet, également refactorisé de nombreuses fois le code pour conserver l’intégrité de la structure au fil des progressions.

\*La contribution des membres du groupes ne se limite seulement pas à ce qui a été écrit plus haut mais plutôt un résumé.

## Comment jouer ?

Dans ce projet, le joueur aura le choix entre une version classique du casse-briques et des niveaux avec des configurations de briques variées. De plus, une option de mode marathon sera disponible, offrant une expérience similaire à un jeu de type *Space Invaders*. Cette fonctionnalité a été rendue possible grâce à la modularité du code. (Cf Structure du Projet)

# Outils utilisés

## Plateforme collaborative

Gitlab a été utilisé comme plateforme de codage collaborative. Discord a été utilisé comme moyen de communication, les issues ont également été utilisées comme moyen de communication entre le groupe et le chargé de TD afin d’avoir un suivi.

## Choix de l’IDE + version JDK + Système d’exploitation

Une image contenant capture d’écran, printemps, nature, typographie

Description générée automatiquementLe projet a été codé avec l’IDE Microsoft Visual Studio Code avec le pack d’extensions Java et Git lens/graph pour pouvoir visualiser l’arbre du projet.

La rétrocompatibilité n’étant pas exigée dans le sujet ni primordiale en JAVA contrairement au web, les membres du groupe sont tous sur une version récente de jdk. (jdk +17)

Les membres du groupe sont équipés d’ordinateur tournant sur les 3 systèmes d’exploitation les plus répandues, c’est-à-dire Windows 10/11, MacOS et Linux.

# Contraintes (demandées par le sujet)

## Librairies

Contrairement au projet POO de S1 qui comportait de nombreuses contraintes car l’objectif était d’appliquer directement les acquis du semestre, le projet de programmation quant à lui a pour objectif de nous préparer à la vie professionnelle et donc laisser les étudiants choisir les outils nécessaires par eux-mêmes et faire preuve d’autonomie face à un cahier des charges.

## Contraintes Techniques

* Modularité
* Bonnes pratiques
  + Encapsulation (protéger les données et déterminer lesquelles bénéficient d’un accès)
  + Principe de responsabilité unique (découper les fichiers selon leur rôle afin de pouvoir plus tard identifier plus facilement où trouver le code à maintenir)
  + Modularité, exportabilité du code (pour être réutilisable)

# Conception

## Choix de la structure

Dans beaucoup de projets, on retrouve le MVC (Modèle-Vue-Contrôleur) comme grand standard dans les patrons d’architecture logicielle. Cependant, ce patron est plus adapté pour les projets qui manipulent des données et notamment les projets web étant donné que les modèles de données représentent le contenu des tables et sont affichées dans les vues par l’intermédiaire du contrôleur.

Ici, il n’est pas question de base de données mais de jeux. Mais il est tout à fait possible d’adapter le MVC dans le contexte d’un jeu vidéo : C’est pourquoi nous avons créé le GEV (Game, Engine, View) avec des principes similaires garantissant la modularité.

## Structure du projet (+ diagramme en annexe)

Vues (fenêtres, menus, …)

Moteur de jeu (construction d’objets, interactions physiques, déplacements, …)

Constructeur de jeux (règles, affichage parent etc…)

Règles du jeu 1

Construit un jeu à partir d’objets et crée des règles d’interactions

La vue affiche l’état du jeu

Modifie l’état des objets (position, vitesse, …)

Engine (E)

View (V)

Game (G)

Règles du jeu 2

# 

# Implémentations (dossier src)

## Gestion du sous-package display

Chaque sous-dossier et chaque classe a un rôle bien spécifique, souvent identifiable grâce à leur nom. En général, chaque sous-dossier contient un dossier rules qui abrite une classe abstraite définissant les modalités sur l'utilisation des objets.

### Gestion de la vue

Le dossier view comporte toutes les classes qui gèrent les interfaces graphiques en lien avec le projet. Chaque interface graphique a un rôle bien précis afin de rendre le dossier view plus explicite.

### Gestion de l’engine

#### Fonctionnement général

Le dossier engine contient le sous-dossier (rules) avec les classes PhysicalObject et GraphicalObject qui définissent ainsi l’ensemble des règles sur l’ensemble des entités de tous les jeux utilisant l’engine.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

PhysicalObject est ainsi chargé de la gestion de la physique sur les entités, la partie physique représentant représente l’une des principales difficultés de ce projet et comporte de nombreux éléments à prendre en compte, d’où son nombre important d’attributs.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

GraphicalObject gère l’aspect graphique et contient les outils nécessaires pour afficher les entités en jeu

#### Gestion des collisions

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquementLa collision entre objet est gérée par graphicalObject (la logique voudrait qu’elle soit dans physical, si ce n’est pas le cas, c’est probablement lié à un manque de temps dû à un retard sur la partie physique), chaque objet ayant des boundaries, les valeurs sont réutilisées pour détecter la collision entre objet.

Lorsque la balle est trop rapide ou bien que le PC soit trop lent, il arrive parfois que la balle ‘’traverse’’ les briques ou bien le paddle, la solution initiale était de séparer les ticks des fps afin de pouvoir augmenter le tickrate sans pour autant augmenter les FPS en dédiant des updates pour la logique et pour les FPS. Le but étant de proposer une bonne structure avec un temps imparti, il était bien plus raisonnable de crée une fonction pour palier à ce problème car la solution initiale demande une refactorisation conséquente et donc l’arrêt temporaire des autres branches pour éviter l’addition de conflits.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

#### Gestion des formes Une image contenant texte, capture d’écran, Police, logiciel Description générée automatiquement

L’engine comporte une liste (non exhaustive) de formes prédéfinies. Chaque forme contient logiquement une position, une taille et peut être représentée graphiquement par une forme colorée ou bien une image.

La classe Shape.java contenu dans le dossier shapes/rules est une classe abstraite dont toutes les formes prédéfinies étendent.

Cette structuration des formes permet de modifier individuellement chaque forme en pleine progression sans impacter tout le code ou à devoir modifier tout le code et également pouvoir changer la forme des entités facilement en changeant juste l’appel constructeur.

#### Gestion de la physique

Pour le côté physique, il y a trois classes principales qui permettent d’appliquer les lois essentielles au jeu.

Premièrement, la classe Vector2D, qui va nous permettre de définir des positions, vitesses et accélérations des objets : on peut accéder à la norme d’un vecteur, on peut le normaliser, l’additionner ou soustraire à un autre vecteur, et on peut également le multiplier par un scalaire ou même calculer l’angle entre 2 vecteurs.

Ensuite, nous avons la classe abstraite PhysicalObject, qui va représenter un objet physique : balle, mur, paddle, tout ce qui constitue le « plateau de jeu ». Toutes les classes des dits objets physiques vont étendre cette classe. Un PhysicalObject possède les attributs :

public static final long serialVersionUID = 2L;

public boolean active;

protected double mass;

protected Vector2D position ;

protected Vector2D speed = new Vector2D(0, 0);

protected Vector2D acceleration = new Vector2D(0, 0);

protected boolean movable; //indicates if the object can move, if its position is influenced by collisions or frictions

protected GraphicalObject representation;

protected double rotationCoeff=1; //coeff of rotation after the paddle hit the ball with speed; has an impact on the next collision

protected Vector2D normalVectorVT = new Vector2D(0, 1);

protected Vector2D normalVectorVB = new Vector2D(0, -1);

protected Vector2D normalVectorHR = new Vector2D(1, 0);

protected Vector2D normalVectorHL = new Vector2D(-1, 0);

protected Slope slope;

protected Vector2D topRightPosition;

protected Vector2D bottomLeftPosition;

protected Vector2D bottomRightPosition;

protected double MAX\_SPEED = 0.8;

Les méthodes principales de la classe sont celles d’update de position, ou de vitesse, mais encore celles d’application de force, ou celles participant à la gestion de collision. Nous y reviendrons plus en détail après.

Enfin, la classe PhysicsEngine est celle qui va permettre aux lois physiques d’être appliquées. Elle possède en attribut une liste de PhysicalObjects qui va recueillir tous les objets du jeu et le but sera de les faire interagir correctement. Pour ce faire, elle contient la fonction update que voici :

 public void update(double deltaTime) {

        applyGravity(deltaTime);

        applyGravitationalForces(deltaTime);

        handleCollisions(deltaTime);

        applyFriction(FRICTION\_COEFFICIENT);

            // updating objects position relatively to the time spent

         for (PhysicalObject object : physicalObjects) {

            object.updateVelocity(deltaTime);

            if (object.isActive() && object.isMovable()){

                //System.out.println("vitesse: "+object.getSpeed());

                //System.out.println("acceleration: "+object.getAcceleration());

                //System.out.println("DeltaTime: "+deltaTime);

                //System.out.println(object.getPosition());

                object.updatePosition(deltaTime);

                object.getRepresentation().setPosX((int)object.getPosition().getX());

                object.getRepresentation().setPosY((int)object.getPosition().getY());

            }

        }

    }

Le but d’update est donc d’actualiser la position des PhysicalObjects « movable ». Elle va appliquer les différentes forces physiques sur tous les objets : gravité, force de gravitation s’il y a un champ gravitationnel en jeu, les collisions entre objets, « movable » ou non, et les frottements de l’air.

Ces différentes fonctions vont actualiser l’accélération des objets liée aux forces appliquées à ceux-ci par la première loi de Newton : somme des forces vaut la masse multipliée par l’accélération.

* La force de gravité va simplement appliquer aux objets une force verticale orientée vers le bas dont la norme est la constante de gravité, ici adaptée car on travaille sur des pixels et non des mètres…
* Pour la force de gravitation, on va venir chercher le vecteur allant de la position de l’objet this (objet attiré) à la position de la planète (objet attractif), puis on applique une force de même direction que ce vecteur et de norme :

double forceMagnitude = G \* (this.getMass() \* planete.getMass()) / (distance \* distance);

Avec G une constante de gravitation

* La gestion des collisions est sans doute la plus grande difficulté rencontrée dans la partie physique du projet, car elle a nécessité l’implémentation de méthodes annexes compliquées.

Le principe est le suivant : on prend chaque objet, et on regarde si cet objet va toucher un autre objet grâce à leurs représentations graphiques.

Si l’objet s’apprête à en toucher un autre, on appelle une fonction située dans la classe Entity qui va résoudre la collision : on récupère le point d’impact, la normale de la collision. On va ensuite récupérer un angle d’incidence et un angle de rotation (permet de créer un effet sur la balle en fonction de sa vitesse et celle de l’objet en collision) qui nous permettront de trouver l’angle de réflexion en fonction de l’orientation de l’objet en collision (horizontal, vertical ou autre). Une fois cet angle de réflexion trouvé, il ne reste plus qu’à appliquer une force suivant son cosinus et son sinus et de norme inchangée (la balle va à la même vitesse et accélère de la même manière).

* Enfin, la force de frottement est appliquée à chaque objet et a pour norme un coefficient de frottement adapté au jeu qui va aller en sens inverse de l’objet.

Une fois l’accélération actualisée, on peut actualiser les vitesses des objets, puis leurs positions, grâce aux relations :

* La vitesse est la primitive de l’accélération
* La position est la primitive de la vitesse

On va ainsi remonter de l’accélération au vecteur à ajouter à la position afin de créer le mouvement des différents objets.

## Gestion du sous package game (et surtout breakout)

### Fonctionnement général

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquementLe sous-dossier game est censé contenir les dossiers des jeux, dans notre cas, game/breakout et un dossier rules qui contient une classe Game.java qui sera adapté en fonction de la catalogue de jeux présentés.

\* Le fonctionnement reste similaire pour les autres sous-dossier présent dans game.

### Création des entités dans un jeu

Une image contenant texte, capture d’écran

Description générée automatiquementLes entités sont stockées dans sous-dossier entity, chaque entitée est représentée par une classe qui porte son nom (ex : Player.java) et hérite la classe abstraite Entity contenu dans entity/rules.

Cela permet de faciliter la gestion des entités en passant par l’intermédiaire d’une classe abstraite qui appelle directement PhysicalObject et GraphicalObject dans l’engine.

### Gestion de la logique

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquementLa logique se fait directement par [NomDuJeu].java, Level.java comme son nom l’indique s’occupe des niveaux

Une image contenant texte, capture d’écran, Police

Description générée automatiquementC’est par exemple dans Breakout.java qu’on retrouvera les fonctions pour qui contrôlent les entités présentes dans le jeu.

L’update se fait également dans Breakout et est unifié.

### Gestion des briques

Les briques fonctionnent avec une HashMap où chaque Integer correspond à une image (ou couleur) correspondant à l’usure de la brique. Une image contenant texte, capture d’écran

Description générée automatiquement

La création des briques et gérée dans Breakout.java ou bien dans Level.java qui font office de contrôleurs comme cela a été stipulé dans la partie conception. Par exemple la version générique createBrick se situe dans Breakout.java. Un update sur les briques se fait continuellement dans Breakout.java et nous avons décidé d’utiliser Iterator<Brick> car cela permettait parcourir la liste des briques encore présentes et de les retirer facilement en cas de collision une fois la vie à 0.

Une image contenant texte, capture d’écran

Description générée automatiquement

### Gestion des bonus

Au même titre que les briques, une HashMap permet de lier chaque type de bonus à l’image qu’il correspond. Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

C’est également dans Breakout.java que se fait l’update des bonus, la fonction applyBonus contient un switch qui parcourt l’ensemble des bonus existants énumérés dans l’enum BonusType.

### Gestion des niveaux

Le mode campagne est accessible depuis le menu "Level" et comprend 6 niveaux. Pour débloquer chaque niveau, il est nécessaire de remporter des victoires. Chaque niveau est créé à partir d'un fichier "level" qui contient une fonction de création dédiée pour chaque niveau.

# Améliorations possibles

En attente de la fin du projet

# Conclusion

À remplir

# Annexes

Update le Graph UML