Une image contenant texte, ciel, capture d’écran, plein air

Description générée automatiquement

Table des matières

[Introduction 3](#_Toc164208979)

[En quoi consiste le projet 3](#_Toc164208980)

[Comment se joue le Casse-Brique 3](#_Toc164208981)

[Brève présentation de l’équipe 3](#_Toc164208982)

[Outils utilisés 4](#_Toc164208983)

[Plateforme collaborative 4](#_Toc164208984)

[L’utilisation de l’IA 4](#_Toc164208985)

[Choix de l’IDE + version JDK + Système d’exploitation 4](#_Toc164208986)

[Contraintes (demandées par le sujet) 5](#_Toc164208987)

[Librairies 5](#_Toc164208988)

[Architecture 5](#_Toc164208989)

[Conception 5](#_Toc164208990)

[Choix de la structure 5](#_Toc164208991)

[Structure du projet (+ diagramme en annexe) 6](#_Toc164208992)

[Encapsulation 6](#_Toc164208993)

[Principe de Responsabilité Unique 6](#_Toc164208994)

[Explication sur le choix de la structure et son fonctionnement 6](#_Toc164208995)

[Implémentations (dossier src) 7](#_Toc164208996)

[Gestion du sous-package display 7](#_Toc164208997)

[Gestion de la vue 7](#_Toc164208998)

[Gestion de l’engine 7](#_Toc164208999)

[Gestion du sous package game (et surtout breakout) 9](#_Toc164209000)

[Fonctionnement général 9](#_Toc164209001)

[Création des entités dans un jeu 9](#_Toc164209002)

[Gestion de la logique 9](#_Toc164209003)

[Gestion des briques 10](#_Toc164209004)

[Gestion des bonus 10](#_Toc164209005)

[Gestion des niveaux 11](#_Toc164209006)

[Test 11](#_Toc164209007)

[Améliorations possibles 11](#_Toc164209008)

[Conclusion 11](#_Toc164209009)

[Annexe 11](#_Toc164209010)

# Introduction

## En quoi consiste le projet

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 pré-professionnalisation 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.

## Comment se joue le Casse-Brique

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-invader. Cette fonctionnalité a été rendue possible grâce à la modularité du code. (Cf Structure du Projet)

# Brève présentation de 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 : (\*)

* DELPECHE Maxime : Responsable de l’expérience utilisateur, rôle essentiel pour la création des menus et l’aspect visuel du jeu.
* VEYRON Lucas et LECOMPTE 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.
* Labby 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.

# Outils utilisés

## Plateforme collaborative

Gitlab a été utilisée comme plateforme de codage collaborative, discord a été utilisée comme moyen de communication.

## L’utilisation de l’IA

L’aide de l’Intelligence Artificielle a été sollicitée durant ce projet, notamment le plus connu actuellement, ChatGPT qui donne la plupart du temps de bonnes suggestions mais ses réponses deviennent génériques et inutiles lorsque les questions deviennent plus spécifiques, cela s’explique par son fonctionnement qui dépend de la quantité de données analysées pour fournir les réponses. C’est d’ailleurs pour cela qu’il a tendance à mieux nous satisfaire lorsqu’il s’agit de web design et surtout pour du css, car le principe reste plus ou moins la même peu importe le projet.

Une image contenant texte, capture d’écran

Description générée automatiquementGithub Copilot a également été utilisé, majoritairement pour sa fonctionnalité d’auto-complétions qui analyse le code / le projet et propose automatiquement une suite du code que le programmeur a entamé en fonction de ce qu’il vient de faire ou du commentaire/nom de la fonction qu’il vient de mettre. Permettant ainsi de gagner du temps.

Github Copilot générant la suite du code (Breakout.java)

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

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

Description générée automatiquement Le projet a été codé avec l’IDE Microsoft Visual Studio Code avec le pack d’extensions Java et Git lens/graph\* pour pouvoir commit/push, pull et merge directement depuis l’IDE, le rendant ainsi plus polyvalent.

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

Le projet a été fait en collaboration par des gens qui utilisent MacOs, Linux et Windows 11.

\*L’application Github Desktop a également été utilisée pour commit/push/pull pour son interface graphique le rendant simple et intuitif.

# 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 pré-professionnalisation quant à lui a pour objectif de nous préparer à la vie professionnelle et donc laisser les étudiants choisir les étudiants par eux-mêmes et faire preuve d’autonomie face à un cahier des charges.

## Architecture

Le projet fini doit avoir un engine modulaire afin de pouvoir réutiliser la physique et pouvoir éventuellement l’appliquer directement sur un autre jeu.

# Conception

## Choix de la structure

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Afin de répondre aux contraintes structurelles, nous nous sommes inspirés du MVC (\*) qui est un patron de conception enseigné au premier semestre de L2 en POO et également au second semestre de L1 à l’occasion de l’IO projet. Le MVC étant souvent utilisé pour le web car la vue ne pouvant pas directement interagir avec la base de données, elle se charge des formulaires remplis par l’utilisateur, qui sont récupérés par le contrôleur. Le contrôleur va ainsi interroger le modèle avec les données du formulaire qui va à son tour interroger la BDD (base de données). Les données renvoyées par la BDD suivent le chemin inverse.

Cependant, en JAVA, où tout est sous forme d'objets accessibles partout, le rôle du contrôleur devient moins essentiel. Ainsi, pour répondre aux critères de modularité, d'évolutivité et d'implémentation de la physique de notre cahier des charges, nous avons décidé d'adopter une structure adaptée. Au cours de la phase de conception et de développement, nous avons opté pour une séparation du jeu en sous-dossiers selon le schéma suivant :

* Le dossier display qui contient l’engine du jeu et l’affichage
* Le dossier Game qui contient les dossiers où sont stockés les informations du jeu

Cette structuration nous permet ainsi de :

* Distinctement séparer le traitement des données de leur présentation visuelle.
* Anticiper les évolutions futures et favoriser la modularité.
* Diviser le code de manière à faciliter sa compréhension et son édition en mode projet.

En outre, cette structuration rend le code plus lisible, plus compréhensible, plus évolutif, mais quant à son implémentation, elle n’en est pas moins compliquée (Cf Difficultés Rencontrées).

(\*) Si nous devions comparer la structure à un modèle MVC classique :

* Le dossier display se charge notamment d’être la vue (car il stocke le fichier view) et le modèle (car il stocke les données concernant l’engine)
* Le dossier game quant à lui se chargera d’être le modèle (car il stocke les données du jeu) et également le contrôlleur car il contient le fichier avec toutes la partie logique et intéraction du jeu (comme Breakout.java qui se situe donc dans le dossier breakout)

## Structure du projet (+ diagramme en annexe)

### Encapsulation

Chaque champ dans notre projet est soit privé, soit protégé (sauf pour les champs primitifs statiques finaux). Leur accès par les classes clientes est contrôlé à l'aide de getters, setters, itérateurs, etc. afin de s’assurer qu’il n’y a aucune violation de l’encapsulation.

### Principe de Responsabilité Unique

Chaque classe a une responsabilité clairement identifiable et une nomenclature explicite, par exemple :

- GraphicalObject et PhysicalObject : constituent l’engine et sont respectivement responsables de la gestion des objets sur la partie graphique et physique

- Breakout.java : Comporte la logique du casse-brique.

- GamePanel : Interface graphique dédiée au jeu.

### Explication sur le choix de la structure et son fonctionnement

Durant la phase de conception et de développement d'un projet, il est essentiel de prendre en compte les contraintes imposées par le client, ainsi que les contraintes de temps. Dans notre cas, le client, représenté par le sujet du projet et le chargé de travaux dirigés (TD), a exprimé le souhait d'avoir une structure modulaire et de pouvoir facilement réutiliser le code pour d'autres jeux tout en séparant la logique du jeu de sa physique. Pour répondre à ces attentes, nous avons décidé d'organiser notre projet dans le dossier source (src) comme suit :

Le dossier display qui contient ainsi :

- Le sous-dossier engine qui contient ainsi les classes essentielles pour la création des entités dans les jeux avec leurs propriétés physique et des formes prédéfinies.

- Le sous-dossier view qui s’occupe de la partie vue comme dans un MVC

Le dossier game qui contient :

- Le sous-dossier Breakout (ou autres jeux), qui quant à lui se charge de créer les entités dont il a besoin dans le dossier entity pour le jeu en utilisant les constructeurs déjà existants dans l’engine, stock les images nécessaires aux jeux dans le dossier assets et stocke la logique du jeu et des niveaux dans Breakout.java et Level.java

- Peux contenir d’autres dossiers qui correspondent à d’autres jeux (avec idéalement la même structure)

La séparation de l’engine et du jeu permet ainsi d’avoir la physique séparé totalement du jeu et également pouvoir modifier les entités facilement (cela permet par exemple de remplacer la balle ronde par une balle ovale en utilisant juste oval.java au lieu de circle.java du dossier shape) et également changer ou bien même créer et rajouter des jeux facilement sans tout réécrire car il suffira de créer dans le dossier game, un sous-dossier (ex : flipper) qui initialisera toutes les entités nécessaires (ex : balles, murs, etc) et réécrire les règles (ex : Flipper.java)

# Implémentations (dossier src)

## Gestion du sous-package display

Chaque sous-dossier et chaque classe ont 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.

TODO : Maxime (rajouter des précisions SI nécessaires et ne pas hésiter à mettre une image d’illustration)

### Gestion de l’engine

#### Fonctionnement générale

Le dossier engine contient le sous-dossier (rules) avec les classes PhysicalObject et GraphicalObject qui érigent 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 chargée de la gestion de la 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 automatiquement La 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 (cf fig), 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 forme prédéfinie. 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 abstraire 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

TODO : Marius/Lucas (ptet Noaym mais il a beaucoup de travail rn)

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

### Fonctionnement général

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

Description générée automatiquement Le 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.

### Création des entités dans un jeu

Une image contenant texte, capture d’écran

Description générée automatiquement Les 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 automatiquement

La 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 automatiquement

C’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 un 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 contrôlleurs comme ça 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, un 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

TODO : Maxime (explications de Level.java devrait suffire j’pense)

# Test

On n’a pas encore fait de tests :x

# Améliorations possibles

On le saura quand on aura fini

# Conclusion

Mentir x)

# Annexe

Faudra update le graph uml du coup et c’tout.