Documentation Technique : Jeu Casse-Brique

Groupe composé de :
- Clément ALEXANDRE
- Léo BONNEMENT
- Émile DAVID
- Simon ROGER

1. Introduction

Pour rappel, le projet consiste à développer un jeu de Casse-Brique, le principe est que le joueur manipule une barre pour faire rebondir une balle qui doit détruire des briques. Chaque brique cassée peut donner un bonus ou un malus. Le jeu propose plusieurs niveaux avec une difficulté croissante.

Pour rappel, les positions dans un jeu sont les mêmes qu'en traitement d'image, c'est-à-dire que le point (x = 0, y = 0) correspond au coin supérieur gauche de l'écran.

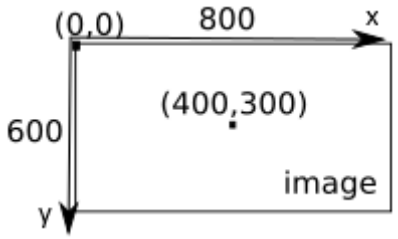

Figure 1 - Repère dans une fenêtre de jeu ou d'image

2. Choix de la technologie et normes

Nous avons choisi d'utiliser Python avec la bibliothèque PyGame pour développer notre jeu. Python est un langage que nous maîtrisons tous les quatre, ce qui nous offre une certaine aisance et nous permet de réaliser un jeu plus abouti. Malgré cela, PyGame est une découverte pour nous, nous la connaissons surtout de nom. Nous avons choisi cette bibliothèque car elle permet de créer une interface graphique dynamique, ce qui est idéal pour un jeu de casse-briques. Nous avons utilisé la norme de codage CamelCase pour ce projet.

3. Architecture du projet

L'architecture de projet se regroupe en 6 fichiers importants qui sont :

- **Assets** : Stocke les images, la police d'écriture et les sons du jeu.
- **Core** : L'élément central du jeu qui permet de tout mettre en relation et d'assurer la logique générale.
- **Entities** : Modélise les différentes entités du jeu à l'aide de classes.
- **Levels** : Génère les briques en fonction du niveau de la partie.
- **Ui** : Génère l'interface avec les rendus graphiques.
- **TestUnitaires** : Regroupe tous nos tests unitaires.

```
ALEXANDRE-BONNEMENT-DAVID-ROGER_PROJET/
├── main.py               # Point d'entrée du jeu
├── config/
│   └── settings.py       # Configuration globale (dimensions de l'écran, couleurs, vitesse, etc.)
├── assets/
│   ├── images/           # Images des briques, de la balle, du paddle, etc.
│   ├── sounds/           # Effets sonores pour collisions et musique de fond
│   └── fonts/            # Polices pour le texte
├── core/
│   ├── game.py           # Classe principale pour gérer la boucle de jeu et les événements
│   ├── utils.py          # Fonctions utilitaires
│   ├── collisions.py     # Classe pour la gestion des collisions
│   ├── lifeManager.py    # Classe pour la gestion des vies
├── entities/
│   ├── bonus.py          # Classe pour les bonus
│   ├── paddle.py         # Classe pour le paddle
│   ├── ball.py           # Classe pour la balle
│   └── brick.py          # Classe pour les briques
├── levels/
│   ├── levelGenerator.py # générateur de niveau
│   └── levelLoader.py    # Chargeur de niveau
├── testsUnitaires/
│   ├── test_Ball.py      # tests unitaires pour la classe Ball
│   ├── test_Brick.py     # tests unitaires pour la classe Brick
│   ├── test_Collisions.py # tests unitaires pour la classe Collisions
│   ├── test_LifeManager.py # tests unitaires pour la classe LifeManager
│   ├── test_Paddle.py    # tests unitaires pour la classe Paddle
│   ├── test_Utils.py     # tests unitaires pour la classe Utils
├── ui/
│   ├── renderer.py       # Gère le render des différents éléments du jeu
│   ├── menu.py           # Menu principal (start, quit)
│   ├── hud.py            # Interface utilisateur en jeu (score, vies restantes)
│   ├── breakMenu.py      # Affichage de l'écran de pause
│   └── gameOver.py       # Affichage de l'écran de fin de partie
└── README.md             # Documentation pour le projet
```

Figure 2 - Arborescence complet du projet

4. Détections des collisions et angles

Dans notre jeu de casse-briques, la gestion des collisions est un élément clé du gameplay. C'est grâce à elle que la balle peut casser les briques, rebondir sur les bords de l'écran et interagir avec la raquette.

Les collisions avec les murs sont vérifiées à chaque déplacement de la balle. Celle-ci est représentée par un point ayant des coordonnées en **X** et **Y**, ainsi qu'un **rayon (radius)** qui lui donne un volume visuel.

- Pour détecter une collision avec le **mur gauche**, on vérifie si **X - radius** est inférieur à **0**. Si c'est le cas, la balle est hors de l'écran du joueur, et nous la renvoyons en inversant sa vitesse horizontale (**dx**).

- La même logique s'applique au **mur droit**, où nous vérifions si **X + radius** dépasse la largeur de l'écran définie dans *settings.py*.

- Pour le **mur supérieur**, nous contrôlons si **Y - radius** est inférieur à la hauteur de l'HUD. Si c'est le cas, la balle est renvoyée vers le bas en inversant sa vitesse verticale (**dy**).

Pour la raquettes et les briques, PyGame ne propose pas de méthode intégrée pour détecter la collision entre un cercle (la balle) et un rectangle (le raquettes ou une brique). Nous avons donc dû créer notre propre fonction. Pour ce faire, nous avons déterminé le **point du rectangle le plus proche** du centre de la balle. Ensuite, nous avons calculé la distance entre le centre de la balle et ce point en utilisant la formule suivante :

$$distance = \sqrt[2]{(coordBallX - PointProchX)^2 + (coordBallY - PointProchY)^2}$$

Si cette distance est inférieure au rayon de la balle, alors une collision est détectée. Une fois la collision confirmée, la balle est renvoyée de la même manière que pour les murs. Par exemple, si elle touche une brique par la droite, alors sa vitesse en **X est inversée (dx = -dx)** tandis que sa vitesse en **Y reste inchangée (dy = dy)** pour conserver la direction verticale du mouvement.

Pour ce qui est de la raquette, celui-ci a un angle de rebond très différent ! Sans cela, si aucune brique n'était présente dans le jeu, la balle suivrait toujours la même trajectoire, car **X** et **Y** ne seraient modifiés qu'en inversant leur signe lors des collisions avec les murs. Pour éviter ce problème, la raquette suit une **norme courante** dans les jeux de casse-briques, elle permet au joueur de contrôler la direction du rebond. Comme décrit dans le cahier des charges :

- Si la balle frappe le **centre** de la raquette, elle est renvoyée **verticalement vers le haut**.
- Si elle touche le **côté droit**, elle repart **vers la droite**.
- Si elle touche le **côté gauche**, elle repart **vers la gauche**.

Cela permet au joueur d'influencer la trajectoire de la balle pour viser les briques.

Figure 3 - Angle du rebond de la balle sur la raquette

Pour ce faire, nous allons calculer l'impact relatif de la balle sur la raquette par rapport au centre. Si cet impact donne une valeur de -1, la balle part à gauche ; s'il donne 1, la balle part à droite ; et si l'impact est de 0, la balle part tout droit. Ensuite, nous calculons le nouvel angle en radians, que nous appliquons à la vitesse de la balle pour simuler l'angle de rebond. Cela permet au joueur de contrôler la direction de la balle dans le jeu.

5. Génération des niveaux et disposition des briques

Dans notre jeu de casse-briques, la disposition des briques et la difficulté des niveaux sont générées dynamiquement en fonction du niveau actuel du joueur. Tout d'abord, la difficulté est adapté en ajustant la taille de la grille selon le niveau :

- Pour les niveaux **1 à 3**, la grille est de **10x10**.
- Pour les niveaux **4 et 5**, elle passe à **12x15**.
- À partir du niveau **6**, la grille s'agrandit à **15x20**.

Ensuite, la génération des briques se fait de manière aléatoire tout en respectant certains critères pour garantir une disposition équilibrée. Nous appliquons une **symétrie** et regroupons systématiquement les briques par **groupes de 2 à 5** afin d'éviter des niveaux avec des briques isolées aux extrémités de la grille par exemple. Chaque brique possède un nombre de vies attribuées aléatoirement en fonction du niveau. Par exemple, au **niveau 1**, une brique peut avoir **au maximum 2 vies**, tandis que dans les niveaux plus avancés, ce nombre peut **aller jusqu'à 6 vies**.

Enfin, la densité de briques augmente progressivement avec la difficulté : à partir du **niveau 10**, la grille est **presque entièrement remplie** !

Les briques sont stockées dans une **liste** et quand cette liste est vide alors on passe au niveau suivant.

6. Bonus

Pour ce qui est des bonus, ils sont une **utilisation et une amélioration** de fonctions déjà existantes. Par exemple, le bonus double bar repose sur l'attribut **width** de la classe **Paddle**. En augmentant cette valeur, la raquette s'élargit. Il en va de même pour les autres bonus : ils ne font qu'exploiter et/ou modifier des fonctions et classes déjà présentes dans le jeu. Pour activer ou désactiver les bonus.