Spécifications du projet de printemps 2026

C++ PoP – Sections Électricité et Microtechnique

Brick Breaker

Interface du projet Brick Breaker

Table des matières

1 Introduction **4**

2 Problèmes liés à l'implémentation en C++ **4**
- 2.1 Format des données 4
- 2.2 Tolérance sur les tests en virgule flottante 4
- 2.3 Traitement des erreurs d'arrondi causées par le formatage des fichiers 4

3 Modélisation des composantes du jeu **5**
- 3.1 L'arène de jeu 5
- 3.2 Les briques 5
  - 3.2.1 Rainbow_brick 5
  - 3.2.2 Ball_brick 5
  - 3.2.3 Split_brick 6
- 3.3 Les balles (classe `Ball`) 6
- 3.4 La raquette (classe `Paddle`) 6

4 Dynamiques du jeu **6**
- 4.1 Déplacement d'une balle 7
  - 4.1.1 Impact d'une balle sur l'arène 8
  - 4.1.2 Impact d'une balle sur une autre balle 8
  - 4.1.3 Impact d'une balle sur une brique 9
  - 4.1.4 Impact d'une balle sur la raquette 9
- 4.2 Déplacement de la raquette 10
- 4.3 Destruction d'une brique 10
- 4.4 Déroulement du jeu 10

5 Sauvegarde et lecture de fichiers tests **10**
- 5.1 Caractéristiques des fichiers tests 10
- 5.2 Vérifications et gestion des erreurs 11

6 Affichage et interaction dans la fenêtre graphique **11**
- 6.1 Affichage 11
- 6.2 Actions réalisables à partir de l'interface 12
- 6.3 Interactions avec le clavier 12
- 6.4 Interactions avec la souris 12
- 6.5 État final 13

7 Architecture logicielle **13**
- 7.1 Décomposition en sous-systèmes 13
- 7.2 Décomposition du sous-système Modèle 13
- 7.3 Module générique indépendant `tools` 14
  - 7.3.1 Indépendance de `tools` 14
  - 7.3.2 Relation « Possède-un » 14
  - 7.3.3 Types à implémenter 14
- 7.4 Module graphique de bas niveau (`graphic`) 14

8 Répartition du projet en trois rendus notés **14**
- 8.1 Rendu 1 14
- 8.2 Rendu 2 14
- 8.3 Rendu 3 15

9 Constantes **16**
9.1 Constantes globales du Modèle (`constants.h`) 16
9.2 Constante générique (`tools.h`) 16
9.3 Constante du sous-système de Contrôle (`gui.cc`) 16

1 Introduction

Ce projet est indépendant de celui du semestre dernier. Le lien réside néanmoins dans la mise en œuvre des grands principes (*abstraction*, *réutilisation*), le respect des conventions de présentation du code et l'application des connaissances accumulées jusqu'à présent dans ce cours.

Le but du projet est avant tout de se familiariser avec deux autres concepts fondamentaux : la *séparation des fonctionnalités* (*separation of concerns*) et l'*encapsulation.* Ces principes deviennent indispensables pour structurer un projet d'envergure en *modules* indépendants.

L'accent sera mis sur l'articulation entre *module* et *structure de données*, ainsi que sur la robustesse des *modules* face aux erreurs. Par ailleurs, l'ordre de complexité des algorithmes sera évalué à l'aide de fichiers de test plus exigeants que les autres.

Il est possible d'aller au-delà des spécifications de la donnée, mais cela n'octroiera aucun bonus ; l'objectif est d'éviter que vous ne consacriez un temps excessif à ce projet au détriment de vos autres matières.

Dans tous les cas, le respect des consignes de la donnée et des documents de rendu est obligatoire. Vos éventuelles contributions personnelles ne doivent en aucun cas interférer avec les présentes instructions.

Le projet étant réalisé par groupes de deux, il peut donner lieu à un oral final individuel approfondi et noté. Chaque membre du groupe doit donc maîtriser le fonctionnement de l'ensemble du projet. Une performance insuffisante lors de cet oral entraînera une baisse individuelle des notes obtenues pour les rendus.

Principe : Le jeu reprend le concept d'un casse-briques : le joueur contrôle une raquette afin de faire rebondir une ou plusieurs balles dans le but de détruire toutes les briques du niveau.
Le niveau est délimité par une arène carrée. Il contient une raquette en forme d'arc, mobile sur le bord inférieur à l'aide de la souris, des briques carrées déclinées en trois types distincts, et des balles dont la trajectoire est influencée par la vitesse de la raquette au moment de l'impact.
Le score, initialement à zéro, augmente à chaque collision entre une balle et une brique ainsi qu'en fin de partie pour chaque vie restante.

2 Problèmes liés à l'implémentation en C++

2.1 Format des données

Les variables, les constantes (section 9) et les noms liés au programme à réaliser sont indiqués en `caractères à chasse fixe`. On utilisera la double précision pour les calculs en virgule flottante ; par conséquent, toutes les positions et grandeurs continues sont définies par le type `double`.

2.2 Tolérance sur les tests en virgule flottante

Les tests d'égalité (par exemple pour une norme nulle) doivent être remplacés par un test mettant en œuvre la tolérance **epsil_zero** sur l'écart entre la valeur théorique et la valeur calculée. Le test retourne `true` si la valeur absolue de cet écart est strictement inférieure à `epsil_zero`.

2.3 Traitement des erreurs d'arrondi causées par le formatage des fichiers

Les arrondis liés au formatage des fichiers imposent d'effectuer des tests moins stricts lors de la lecture des données comparativement aux tests effectués durant l'exécution du jeu.
C'est pourquoi la valeur `epsil_zero` doit être considérée comme nulle pour les tests d'inégalités suivants lorsqu'ils sont effectués à la lecture d'un fichier. En revanche, elle doit systématiquement être utilisée pendant le jeu, notamment pour l'ensemble de la section 4 :

- **Intersection :** la distance minimale entre deux figures est strictement inférieure à `epsil_zero`.
- **Inclusion :** minimum + epsil_zero <= valeur <= maximum − epsil_zero.

3 Modélisation des composantes du jeu

3.1 L'arène de jeu

L'arène est représentée par un carré gris dont le sommet inférieur gauche correspond à l'origine $(0, 0)$ et dont le côté a une dimension de `arena_size`. L'axe X est orienté positivement vers la droite et l'axe Y positivement vers le haut.

3.2 Les briques

Une brique est définie par un carré de centre (x, y) et d'une taille supérieure ou égale à `brick_size_min`. Elle doit être intégralement comprise dans l'arène. Il en existe plusieurs types possédant chacun des propriétés spécifiques.

3.2.1 Rainbow_brick

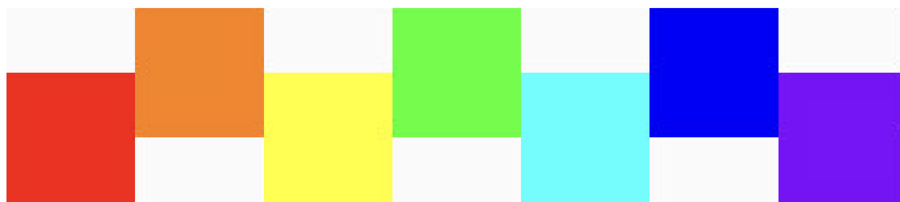

FIGURE 1 – Les sept `Rainbow_brick` possibles

Une `Rainbow_brick` possède un attribut `hit_points` compris entre 1 et 7, indiquant le nombre de coups restants avant sa destruction. Sa couleur dépend de cette valeur et change donc après chaque impact :

– 1 : rouge ; 2 : orange ; 3 : jaune ; 4 : vert ; 5 : cyan ; 6 : bleu ; 7 : violet.

3.2.2 Ball_brick

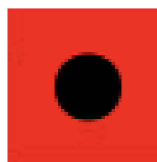

FIGURE 2 – Une `Ball_brick`

Une `Ball_brick` génère une nouvelle balle lorsqu'elle est percutée. Cette dernière hérite de la position du centre de la brique, du vecteur `delta` de la balle incidente et possède un rayon `new_ball_radius`. Graphiquement, cette brique est un carré rouge contenant un disque noir de rayon `new_ball_radius` en son centre représentant la balle.

3.2.3 Split_brick

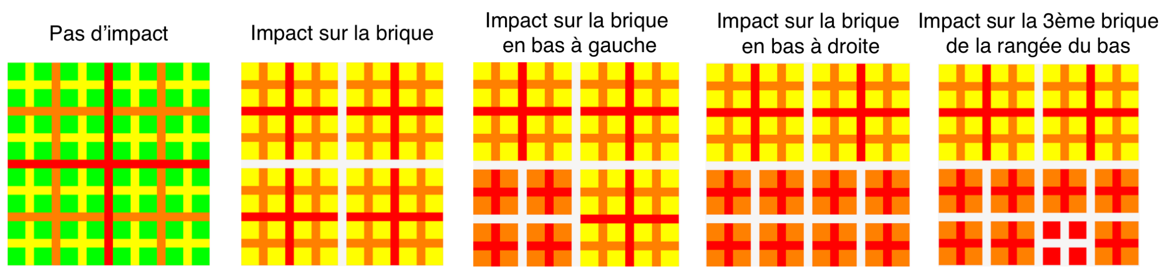

FIGURE 3 – Différents impacts et divisions succesifs à partir d'une seule `Split_brick` ayant initiallement un côté de taille 70. Après le dernier impact on peut voir 4 petits carrés complètement rouges, ceux-ci seront immédiatement détruit sans se diviser au prochain impact.

Une `Split_brick` se divise en quatre briques plus petites du même type lorsqu'elle est percutée, à condition que la taille des nouvelles briques soit supérieure ou égale à `brick_size_min`. Ces briques correspondent aux quatre coins, séparés par une croix de largeur `split_brick_gap`. Le nombre de fois qu'une split brick se divise dépend donc directement de sa taille initiale.

Elle est représentée par un carré rouge complété par quatre petits carrés oranges dans les coins. Le processus est récursif pour chaque subdivision, en changeant de couleur à chaque étape selon l'ordre défini pour les `Rainbow_brick`.

3.3 Les balles (classe `Ball`)

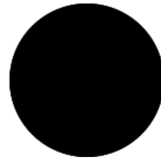

FIGURE 4 – Une balle

Une balle est définie par un cercle de centre (x, y), un rayon et un vecteur de déplacement `delta` (dont la norme est inférieure ou égale à `delta_norm_max`). La balle doit être maintenue dans l'arène, sauf pour le bord inférieur où son rayon est négligé. Elle est représentée par un disque noir.

3.4 La raquette (classe `Paddle`)

FIGURE 5 – La raquette

La raquette est définie par un cercle de centre (x, y) et d'un rayon donné. Le centre doit se situer sur ou sous l'axe des abscisses ($y \leq 0$). Pour qu'une portion du cercle soit visible dans l'arène, la condition $y + \text{rayon} > 0$ doit être satisfaite. Les deux points d'intersection avec l'axe x (extrémités de la raquette) doivent être compris dans $[0, \text{arena_size}]$. Seule la partie supérieure du cercle est affichée (arc de cercle noir), mais le cercle complet est pris en compte pour les calculs de collision.

4 Dynamiques du jeu

Le programme gère deux activités complémentaires et quasi simultanées grâce à la programmation par événements :

1. **L'interaction souris** : permet de définir le déplacement de la raquette ou de générer une balle s'il n'y en a plus (voir section 6.4).
2. **Le timer** : active des mises à jour à intervalle régulier (`dt` milisecondes) pour le déplacement des entités et la gestion des collisions.

La mise à jour doit être structurée selon les étapes suivantes :

Listing 1 – Pseudocode de la mise à jour (tant que status == `ONGOING`)

```
// Entrées : l'ensemble des entités du jeu

 Pour chaque balle:
    Deplacement de la balle
    Si centre en dessous du bord inferieur:
        Destruction de la balle
    Tant que collision balle, brique, raquette, bord:
        Annulation du déplacement
        Si nb_rebonds < nb_rebonds_max
            Deplacement avec rebond
Deplacement de la raquette si pas de collision brique, bord
Pour chaque balle:
    Si collision avec la raquette:
        Deplacement avec rebond sur la raquette  // pas compté dans nb_rebonds
        Tant que collision balle, brique, raquette, bord:
            Annulation du déplacement
            Si nb_rebonds < nb_rebonds_max:
                Deplacement avec rebond
Sinon si nb_balls == 0 ET lives == 0:
    status = LOST
Sinon si nb_bricks == 0:
    Ajustement_du_score en fonction de lives
    status = WON
```

4.1 Déplacement d'une balle

Le déplacement d'une balle est dicté par son attribut `delta`. À chaque mise à jour, ce vecteur est ajouté à sa position actuelle. Si la nouvelle position provoque une collision, le déplacement est annulé et le `delta` est modifié selon la nature de l'obstacle. Le déplacement est ensuite retenté avec le nouveau vecteur.

Pour éviter les boucles infinies (cas d'une balle coincée entre deux entités), le nombre de rebonds par mise à jour est limité par `nb_bounce_max`. Une fois ce seuil atteint, la balle conserve son nouveau `delta` mais son déplacement est suspendu jusqu'à la mise à jour suivante.

4.1.1 Impact d'une balle sur l'arène

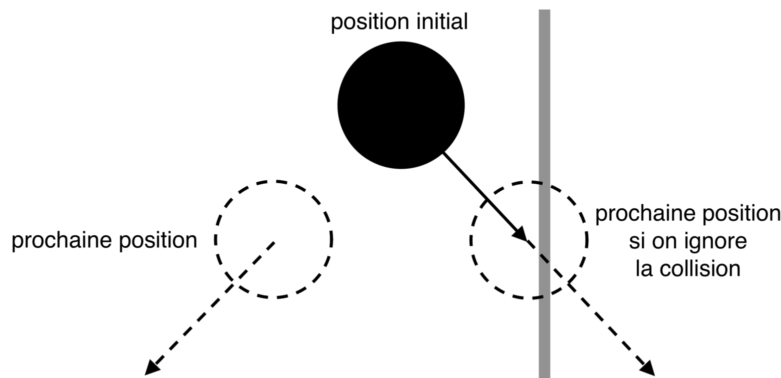

Figure 6 – Collision d'un balle sur l'arène

La collision ne concerne que les bords gauche, droit et supérieur. Si le centre de la balle descend sous le bord inférieur, elle est définitivement perdue.

— Bord horizontal : le signe de `delta_y` est inversé.

— Bord vertical : le signe de `delta_x` est inversé.

Si la balle est en collision avec deux bords (dans un coin) on considère le bord correpondant à sa coordonnée la plus éloignée du centre (x ou y de la future position qui produit une collision).

4.1.2 Impact d'une balle sur une autre balle

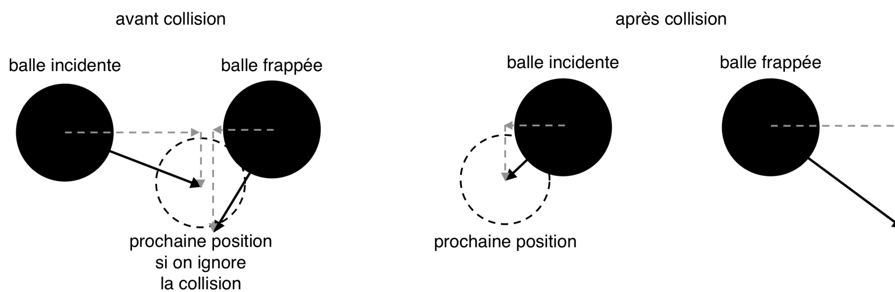

Figure 7 – Collision d'un balle avec une autre balle de même taille

Le choc est considéré comme parfaitement élastique. On utilise le rayon au carré (r^2) comme approximation de la masse en 2D. Soit v_n la vitesse nominale (projection du `delta` sur la droite reliant les centres). L'impulsion corrigée par les masses est définie par :

$$impulsion = (-v_n + v_autre_n) \cdot \frac{2 \cdot r_autre^2}{r^2 + r_autre^2}$$

Le nouveau `delta` est la somme du `delta` et de cette impulsion. La norme du `delta` est ensuite bridée à `delta_norm_max` pour éviter que la balle ne traverse des obstacles par effet tunnel.

4.1.3 Impact d'une balle sur une brique

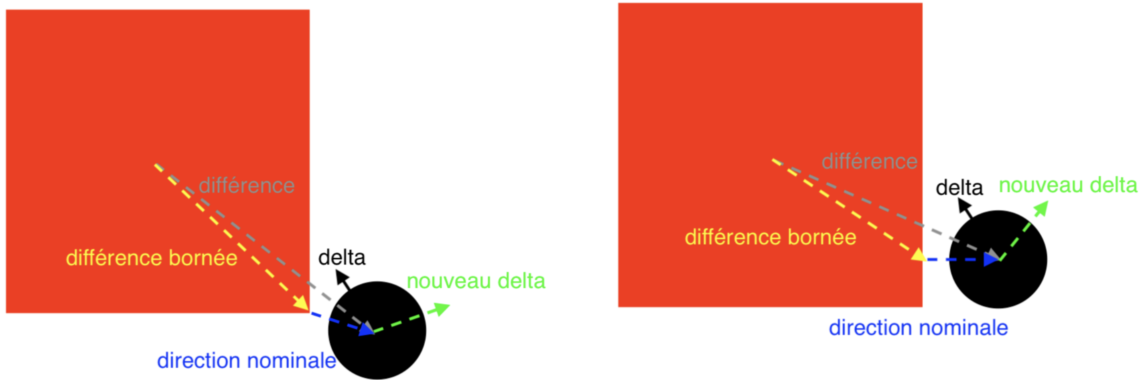

FIGURE 8 – Collision d'un balle avec une brique (sans destruction de la brique par simplicité)

Afin de déterminer la direction nominale (axe reliant le entre de la balle et le point d'impact), on calcule la différence entre les centres de la balle et de la brique. On calcule aussi la différence bornée à la moitié de la taille du carré. La soustraction entre la différence réelle et la différence bornée donne la direction nominale de la collision. Le nouveau delta se calcul en inversant la composante nominale : $v_{new} = v - 2v_n$.

4.1.4 Impact d'une balle sur la raquette

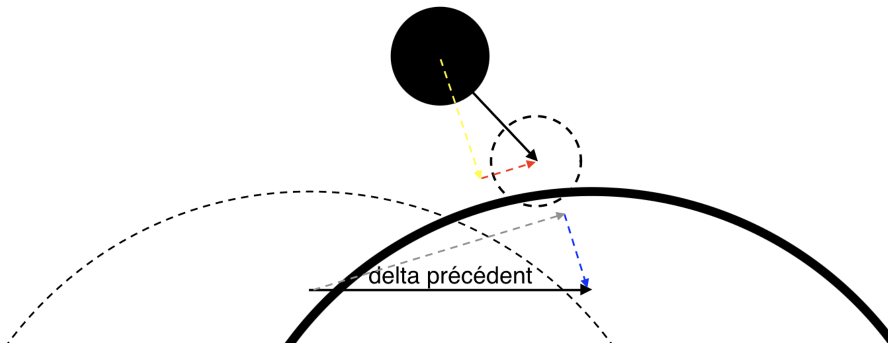

FIGURE 9 – Situation avant la collision d'une balle sur la raquette

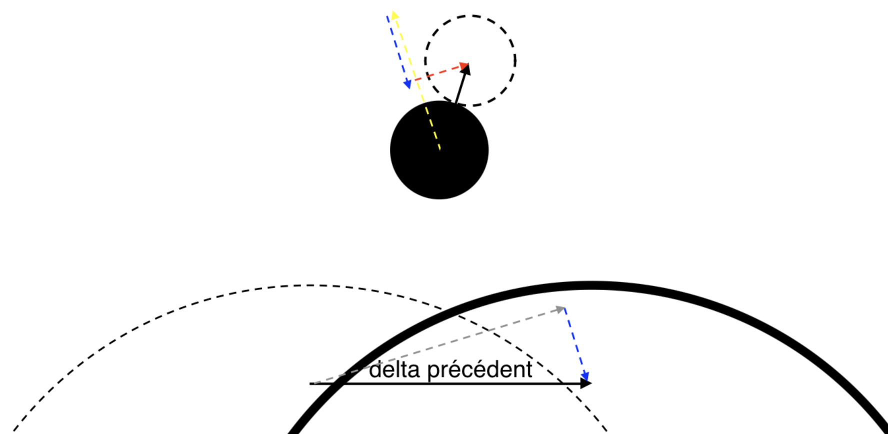

FIGURE 10 – Situation après la collision d'une balle sur la raquette

Le calcul est identique à celui d'une collision entre deux balles (4.1.2), en considérant le rayon de la raquette comme infini et son `delta` comme son dernier déplacement enregistré.

4.2 Déplacement de la raquette

La raquette suit la position x de la souris. Si l'écart est faible, elle se positionne sur la souris. Sinon, sa vitesse est plafonnée à `delta_norm_max`. Toute collision avec une brique ou un bord annule le déplacement. Une collision avec une balle entraine immédiatement un déplacement de la balle selon la section 4.1.

4.3 Destruction d'une brique

Une brique est affectée dès qu'un impact est détecté :

— **Rainbow_brick** : perd 1 `hit_point` ; détruite à 0 (voir section 3.2.1).

— **Ball_brick** : détruite immédiatement, génère une nouvelle balle (voir section 3.2.2).

— **Split_brick** : détruite immédiatement, se divise en quatre `Split_brick` plus petites (voir section 3.2.3).

4.4 Déroulement du jeu

Après l'initialisation, chaque rebond sur une brique augmente le score de `score_per_hit`. Si toutes les balles sont perdues, une nouvelle peut être générée au sommet de la raquette contre une vie (voir section 6.4). La partie s'achève par une victoire s'il n'y a plus de briques et le score augmente de `score_per_life` par vie restante, par une défaite s'il n'y a plus de balle ni de vie.

5 Sauvegarde et lecture de fichiers tests

Votre programme doit être capable d'initialiser l'état du jeu à partir d'un fichier texte et de mémoriser la configuration actuelle dans un fichier de même format. Cela permet de créer des scénarios de tests personnalisés via un simple éditeur de texte.

5.1 Caractéristiques des fichiers tests

L'opération de lecture doit être robuste et indépendante des éléments suivants :

- Présence de lignes vides ou de sauts de ligne (`\n`, `\r`).
- Commentaires commençant par le caractère # en début de ligne.
- Espaces superflus (indentations, espaces avant ou après les données).

Le fichier suit un ordre strict : le score, le nombre de vies, les données de la raquette, les briques, puis les balles. Dans le format ci-dessous, x et y désignent la position (réels), r le rayon (réel), c la taille d'un côté (réel), t le type de brique (0 : `Rainbow_brick`, 1 : `Ball_brick` ou 2 : `Split_brick`), h les `hit_points` (1, 2, 3, 4, 5, 6 ou 7) qui est présente seulement pour les `Rainbow_brick`, dx et dy les composantes du vecteur de déplacement (réels), et $nb_entites$ le nombre total d'objets d'une catégorie (entier).

```
# Nom du scenario de test

# score
score

# lives
lives

# paddle
x y r

# bricks
nb_bricks
    t0 x0 y0 c0 [h0]
    t1 x1 y1 c1 [h1]
    ...

# balls
nb_balls
    x0 y0 r0 dx0 dy0
    x1 y1 r1 dx1 dy1
    ...
```

Listing 2 – Format général du fichier test

5.2 Vérifications et gestion des erreurs

Lors de la lecture, le programme doit valider les points suivants (détails en sections correspondantes) :

- Score et nombre de vies positifs ou nuls.
- Inclusion non-stricte dans l'arène des briques, des balles (en ignorant le rayon pour le bord inférieur) et de la raquette (arc de cercle).
- Taille de chaque brique $\geq$ `brick_size_min`.
- Validité des types : $t \in \{0, 1, 2\}$ et `hit_points` $\in \{1, \ldots, 7\}$.
- Dynamique : norme du `delta` $\leq$ `delta_norm_max`.
- **Absence de collisions initiales :** aucune intersection entre deux briques, entre la raquette et une brique, entre deux balles, entre une balle et une brique, ou entre la raquette et une balle.

Si une erreur est détectée, un message (du module `message` fourni) doit être affiché et le programme doit adopter le comportement suivant :

- **Rendu 1 :** interruption immédiate du programme.
- **Rendus 2 et 3 :** affichage de l'interface graphique avec un canevas vierge (blanc) et des compteurs numériques initialisés à zéro.

6 Affichage et interaction dans la fenêtre graphique

À partir du rendu 2, l'exécution du programme ouvre une fenêtre `gtkmm` contenant l'interface graphique utilisateur. Celle-ci présente les boutons et les informations du jeu d'un côté, et le rendu visuel complet dans un canevas (*canvas*) de l'autre.

6.1 Affichage

Le canevas dédié au dessin a une taille initiale et minimale définie par la constante `drawing_size`, exprimée en pixels. La taille de la fenêtre peut être modifiée durant l'exécution. Un changement de dimensions ne

doit cependant introduire aucune distorsion dans le dessin : un cercle doit rester un cercle, quelles que soient les proportions de la fenêtre.

Le module graphique de bas niveau (`graphic`) met à disposition une table de couleurs (`enum Color` et fonction `set_color()`) pour les fonctions de dessin. Vous devez le compléter pour dessiner les différents éléments du jeu (carré, cercle, arc). Les entités suivantes utilisent des formes et couleurs spécifiques :

- **Fond :** blanc.
- **Bordure de l'arène :** grise.
- **Rainbow_brick :** carré dont la couleur dépend de ses `hit_points` :
  - 1 : rouge ; 2 : orange ; 3 : jaune ; 4 : vert ; 5 : cyan ; 6 : bleu ; 7 : violet.
- **Ball_brick :** carré rouge contenant un disque noir de rayon `new_ball_radius` en son centre.
- **Split_brick :** carré rouge sur lequel s'ajoutent quatre carrés oranges dans les coins (si leur taille $\geq$ `brick_size_min`) séparés par `split_brick_gap`. Le processus est récursif, changeant de couleur à chaque étape selon l'ordre défini pour la `Rainbow_brick`.
- **Balle :** disque noir correspondant aux dimensions de son cercle.
- **Raquette :** arc de cercle noir.

6.2 Actions réalisables à partir de l'interface

L'interface utilisateur doit comporter les éléments suivants :

- **Exit** : quitte l'application.
- **Open** : charge un fichier de test. Les structures de données précédentes sont supprimées ; l'écran reste blanc en cas d'erreur de lecture.
- **Save** : sauvegarde l'état actuel du jeu dans un fichier.
- **Restart** : réinitialise la partie à partir du dernier fichier chargé avec succès.
- **Start/Stop** : lance ou suspend l'exécution en continu.
- **Step** : calcule une unique mise à jour (pas à pas).

Lorsque l'exécution en continu est active, seul le bouton **Start/Stop** doit rester interactif. Après chaque lecture ou mise à jour, un rafraîchissement de l'affichage (`drawing.queue_draw`) et des compteurs (`update_infos`) est requis.

Les compteurs affichées sont : `score`, `lives`, `bricks` et `balls`.

6.3 Interactions avec le clavier

Les raccourcis clavier suivants doivent être implémentés :

- Touche **'s'** : identique au bouton **Start/Stop**.
- Touche **'1'** : identique au bouton **Step**.
- Touche **'r'** : identique au bouton **Restart**.

6.4 Interactions avec la souris

Le déplacement du curseur sur le canevas (`on_drawing_move`) actualise la position x vers laquelle la raquette va se diriger. Indépendamment, s'il n'y a plus de balles en jeu mais qu'il reste des vies, un clic gauche (`on_drawing_left_click`) génère une nouvelle balle au sommet de la raquette (en tenant compte de `epsil_zero`). Cette balle possède un rayon `new_ball_radius` et un vecteur `delta` (0, `new_ball_delta_norm`). Un clic doit déclencher un rafraîchissement du dessin (`drawing.queue_draw`) pour visualiser la balle, même si le jeu est en pause.

6.5 État final

Lorsque la partie prend fin, un message (du module `message`) doit signaler la victoire ou la défaite dans le terminal. Seules les actions **Exit**, **Open**, **Save** et **Reset** restent utilisables et toutes les entités du jeu doivent être immobiles. Si le jeu est initialisé avec un fichier test correspondant à une victoire ou à une défaite, alors la fenêtre est initialisée directement dans cet état.

7 Architecture logicielle

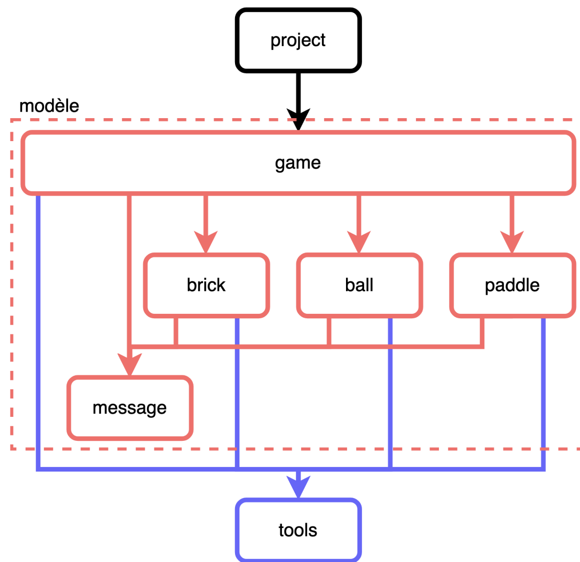

FIGURE 11 – Architecture sans interface graphique (rendu 1)

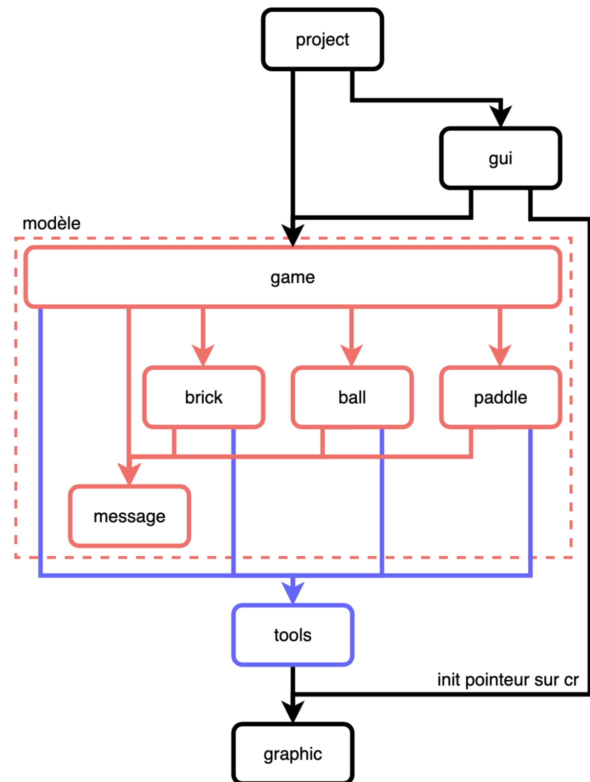

FIGURE 12 – Architecture complète avec GUI

7.1 Décomposition en sous-systèmes

L'architecture logicielle définit l'organisation minimale du projet en sous-systèmes, chacun ayant une responsabilité propre (*séparation des sonctionnalités*) :

- **Sous-système de Contrôle** : gère le dialogue avec l'utilisateur. Si une action utilisateur impose un changement d'état, ce sous-système sollicite le Modèle, seul responsable de la gestion des données. Il comprend :
  - Le module `project` : contient la fonction `main`.
  - Le module `gui` : (dès le rendu 2) gère l'interface graphique mise en œuvre avec `gtkmm`.
- **Sous-système du Modèle** : responsable de la gestion des structures de données du jeu. Il est structuré sur plusieurs niveaux d'abstraction selon les principes d'*abstraction* et de *réutilisation*.
- **Utilitaire générique (`tools`)** : module indépendant du Modèle gérant les points, cercles et carrés, ainsi que les tests d'intersection et d'inclusion. Il fait office de bibliothèque mathématique.
- **Sous-système de Visualisation (`graphic`)** : dessine l'état du jeu via les formes élémentaires de `tools`. Il centralise toutes les dépendances vis-à-vis de `gtkmm` pour le rendu graphique.

7.2 Décomposition du sous-système Modèle

Le Modèle orchestre la simulation et se divise en plusieurs modules pour en maîtriser la complexité :

- **Niveau supérieur (`game`)** : gère le déroulement global du jeu, la cohérence du Modèle et les entrées/-sorties (fichiers). C'est l'unique point d'entrée pour les fonctions appelées hors du Modèle.
- **Niveau intermédiaire (`brick`, `ball`, `paddle`)** : définit les entités du jeu. Une hiérarchie de classes est requise pour les briques et optionnelle pour les entités mobiles.
- **Niveau inférieur (`message`)** : rassemble les fonctions d'affichage des messages liées au Modèle (erreurs détectées lors de la lecture).

7.3 Module générique indépendant `tools`

7.3.1 Indépendance de `tools`

Ce module est une bibliothèque mathématique pure. Aucun terme lié au jeu (brique, balle, raquette, constantes du jeu) ne doit apparaître dans `tools`. Il contient les fonctions de collision essentielles aux niveaux supérieurs.

7.3.2 Relation « Possède-un »

Les classes du Modèle doivent utiliser les types de `tools` via la composition (relation « possède-un ») plutôt que par l'héritage (relation « est-un »).

7.3.3 Types à implémenter

Nous demandons d'implémenter les types `Point` (position ou vecteur 2D), `Circle` et `Square`, ainsi que les fonctions utilitaires (ex : `intersects`). Étant des entités de bas niveau, l'utilisation de `struct` est autorisée.

7.4 Module graphique de bas niveau (`graphic`)

Dès le rendu 2, le module `tools` offrira des capacités de dessin pour ses types (sauf `Point`). Pour maintenir l'indépendance de `tools` vis-à-vis d'une bibliothèque spécifique, les appels à `gtkmm` sont encapsulés dans `graphic`. Ce dernier définit la table de couleurs et les fonctions de tracé effectives.

8 Répartition du projet en trois rendus notés

Chaque rendu est précisément détaillé dans un document indépendant. L'exécutable produit doit s'appeler `project`.

8.1 Rendu 1

Son architecture est précisée par la Figure 11. Ce rendu est testé en indiquant un nom de fichier sur la ligne de commande selon la syntaxe suivante : `./project t01.txt`.

Le programme doit initialiser l'état du jeu en construisant une première version des structures de données. Il s'arrête dès la première erreur détectée dans le fichier : il faut alors afficher dans le terminal le message d'erreur correspondant fourni par le module `message`, puis quitter le programme.

Le programme s'arrête également après la lecture du fichier s'il n'y a aucune erreur ; dans ce cas, un message indiquant le succès de la lecture est affiché.

Le rendu 1 ne doit pas utiliser GTKmm.

8.2 Rendu 2

Son architecture est précisée par la Figure 12. L'objectif de ce rendu est de construire les structures de données, de les dessiner et de finaliser l'interface graphique (à partir des fichiers fournis).

Ce rendu, utilisant GTKmm, est exécuté comme suit :

— Si un nom de fichier est indiqué sur la ligne de commande, il doit être ouvert afin d'initialiser l'interface graphique et le dessin.

— Si aucun nom n'est fourni, le programme initialise l'interface graphique avec un canvas blanc et attend que l'utilisateur demande l'ouverture d'un fichier.

Ce rendu est testé en effectuant plusieurs opérations de lecture, écriture et relecture via l'interface graphique.

Attention : à partir du rendu 2, le programme ne doit plus se fermer en cas de détection d'erreur. Dès la première erreur détectée lors de la lecture, il faut :

— afficher dans le terminal le message d'erreur fourni ;
— interrompre la lecture ;
— détruire toutes les entitées initialisées
— afficher un canvas vide avec les compteurs à 0 ;
— attendre une nouvelle commande de l'utilisateur.

Pour le rendu 2, seule la gestion du déplacement de la raquette à la souris est demandée. Les collisions avec les balles sont ignorées, mais celles avec les briques doivent être prises en compte pour le blocage du déplacement. Aucune action dynamique sur les briques ou les balles n'est requise.

Un rapport doit décrire les choix de structures de données.

8.3 Rendu 3

Ce rendu utilise toujours GTKmm et correspond à la version complète du projet. Le jeu doit s'initialiser de la même manière qu'au rendu 2.

L'objectif principal est d'implémenter la dynamique complète du jeu : déplacement des balles, gestion des rebonds, destruction des briques et fin du jeu.

Un rapport doit également être rédigé.

9 Constantes

9.1 Constantes globales du Modèle (constants.h)

```
#include "tools.h" // epsil_zero et enum Color de graphic.h

constexpr double arena_size = 100.0;
constexpr double new_ball_radius = 1.0;
constexpr double new_ball_delta_norm = 0.8;
constexpr double delta_norm_max = 3.0;
constexpr double split_brick_gap = 3.0;
constexpr double brick_size_min = 3.0;
constexpr unsigned nb_bounce_max = 5; // limite pour eviter les boucles infinies
constexpr unsigned score_per_hit = 10;
constexpr unsigned score_per_life = 300;
constexpr unsigned dt = 25; // intervalle de temps entre mise à jour en millisecondes
```

9.2 Constante générique (tools.h)

```
constexpr double epsil_zero = 0.125;
```

9.3 Constante du sous-système de Contrôle (gui.cc)

```
constexpr unsigned drawing_size = 500; // taille du dessin initiale en pixels
```