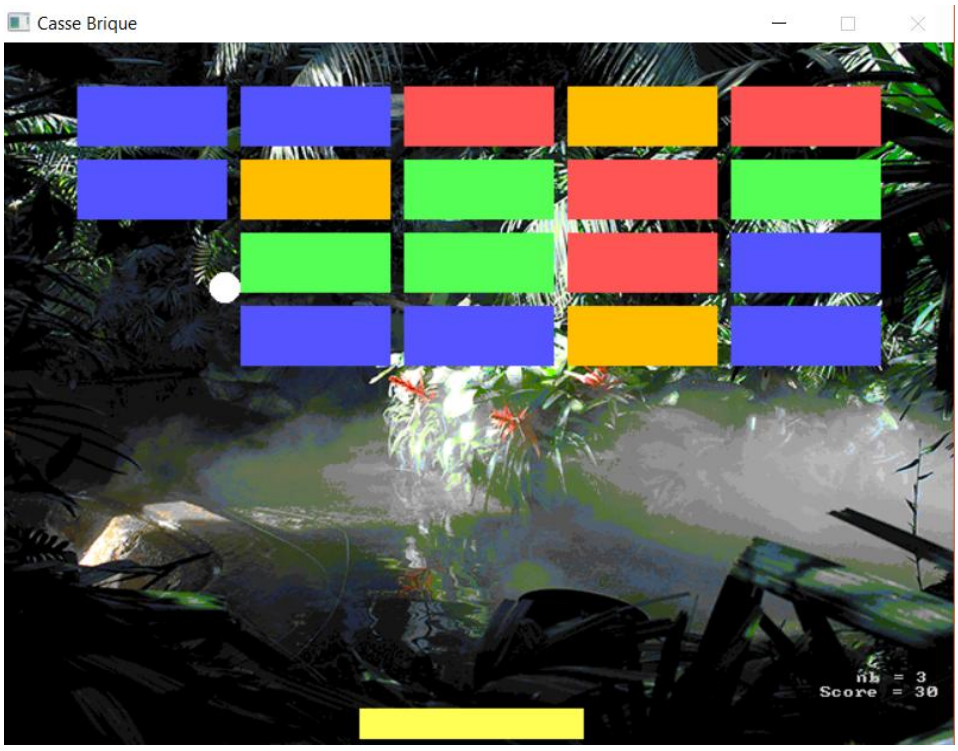
IUT AMIENS

Rapport du Projet de POO

Réalisation d'un Casse-Briques en langage C++

Amaury Gelin – Johanna Leng

07/07/2017



Capture d'écran d'une partie en cours

Dans le cadre de notre cours sur la Programmation Orientée Objet, il nous a été demandé de réaliser un jeu de Casse-Briques. L'objectif est de mettre en œuvre les différents concepts de ce paradigme de programmation pour pouvoir produire un programme générique et modulaire.

Table des matières

Introduction

La Programmation Orientée Objet est un paradigme de programmation fondamental en Informatique. Il s'agit de définir des objets qui représentent des concepts ou des entités du monde physique. Ces objets possèdent une structure interne composée d'attributs, ainsi qu'un comportement détaillé par des méthodes. L'objectif est de représenter ces objets et leurs relations les uns avec les autres, c'est-à-dire réaliser un modèle objet. L'interaction entre les objets permet de concevoir des fonctionnalités et de résoudre des problèmes de manière claire, modulaire et générique. Ces qualités sont fondamentales dans la réalisation d'un programme, car elles permettent de le rendre adaptable, réutilisable et maintenable.

Lors de notre cours sur la Programmation Orientée Objet orchestré par M. Delahoche, il nous a été demandé de réaliser un jeu de Casse-Briques. C'est un sujet intéressant et académique, puisqu'il nous fait manipuler les différents concepts de la POO. On imagine bien que sans l'utilisation de ces concepts, la réalisation du jeu deviendrait très fastidieuse, car la généricité nécessaire (notamment pour chaque brique, ou de manière générale chaque forme de notre jeu) serait difficilement réalisable. Nous avons respecté le cahier des charges formulé dans l'énoncé initial, et nous avons même apporté quelques extensions comme par exemple la gestion des bonus. Notons tout de même le fait que nous avons intégré le menu du jeu dans la modélisation, mais que nous ne l'avons pas implémenté graphiquement. Cette restriction a été décidée en cours, pour respecter des délais assez courts. Pour ce projet, le langage C++ nous a été imposé. Il fallait également intégrer un environnement graphique, et nous avons choisi d'utiliser la librairie Allegro qui a été introduite en cours.

Au fil de ce rapport, nous allons présenter de manière détaillée toutes les étapes qui ont permis de produire notre jeu présenté lors de la soutenance. Nous nous attarderons dans un premier temps sur la modélisation objet, qui est probablement le point le plus important dans la réalisation de notre programme puisque toutes les fonctionnalités développées en découlent. Nous expliquerons ensuite les différents concepts utilisés pour rendre les comportements du jeu génériques : l'héritage, le polymorphisme d'inclusion, les relations d'agrégation et de composition, ou encore la notion de classes abstraites. Ensuite, nous expliciterons les fonctionnalités importantes pour le déroulement d'une partie, parmi lesquelles on trouve les déplacements et les collisions. Nous toucherons quelques mots sur l'interface graphique ; les instructions utilisées sous Allegro ainsi que la stratégie de double-buffering, primordiale au bon fonctionnent du jeu. Par la suite, nous présenterons les quelques extensions de Game-Design que nous avons apportées au jeu, dans le but de le rendre plus agréable à jouer. Nous aborderons également les difficultés rencontrées lors de la réalisation de ce projet. Enfin, nous prendrons du recul sur notre programme et nous énoncerons les éventuelles améliorations possibles.

1) Modélisation Objet

a) Diagramme de classes

Dans le cadre de la POO, la première chose à faire lors de la réalisation d'un programme est de concevoir le modèle objet. C'est le point le plus important, car toutes les fonctionnalités de notre programme découleront de ce modèle. Le temps passé à la conception doit être au moins égal au temps passé à l'implémentation.

Pour réaliser notre diagramme de classes, nous avons utilisé un langage de modélisation bien connu : UML (Unified Modeling Language). Nous n'avons pas représenté les attributs/méthodes pour ne pas grandement alourdir le diagramme, nous détaillerons ces éléments au fil de nos explications. Ceci dit, nous avons représenté les méthodes virtuelles pures, qui sont très importantes dans la modélisation.

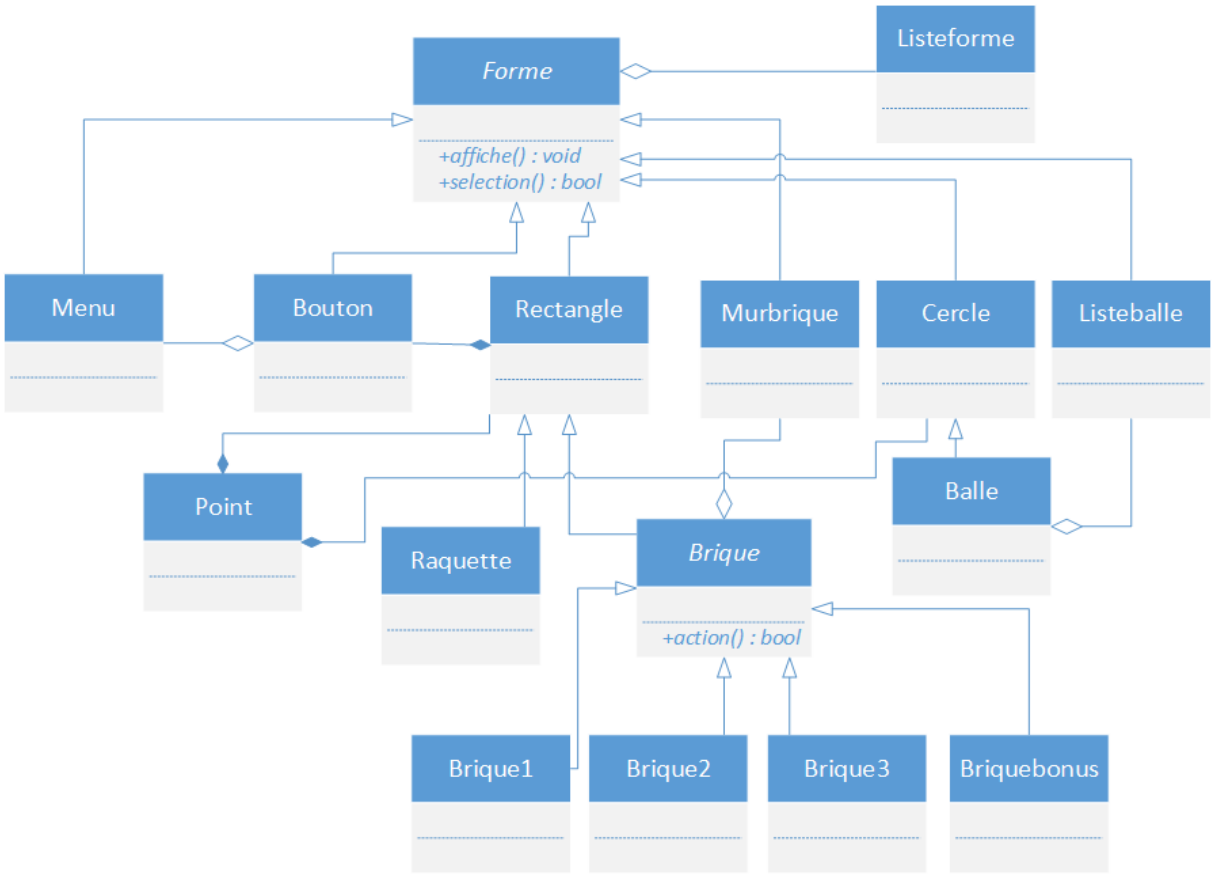


Diagramme de classes, en UML

b) Relations entre les classes

La classe Forme est la classe « fédératrice » de notre diagramme de classes. Elle est tout en haut de la hiérarchie, et 6 classes en héritent directement : Menu, Bouton, Rectangle, Cercle, Murbrique, Listeballe. Rappelons que l'héritage en UML se représente par une flèche allant de la classe fille à la classe mère. D'autres classes héritent indirectement de Forme, car elles héritent des classes dérivées de Forme. Forme est donc une classe essentielle de notre modèle, puisqu'elle va permettre de généraliser un certains nombre de comportements à tous types de formes. D'ailleurs, il s'agit ici d'une classe abstraite (nom en italique). En effet, elle possède des méthodes virtuelles pures (affiche et selection). Premièrement, cela signifie qu'affiche et sélection sont des comportements généraux que l'on aimerait pouvoir appeler pour n'importe quelle forme. Deuxièmement, cela signifie que Forme n'est pas instanciable. En effet, les méthodes virtuelles pures ne sont pas définies dans Forme, donc cette partie de code manquante ne permet pas d'instancier d'objet. La propriété 'virtuelle pure' se propageant par dérivation, il est nécessaire de définir ces méthodes dans les classes dérivées pour qu'elles ne soient pas également abstraites.

La classe Brique (héritant de Rectangle) est la deuxième classe abstraite de notre diagramme. En effet, elle possède une méthode virtuelle pure : action. Action sera un comportement un définir pour chaque type de brique différente. Ici nous avons 4 classes qui héritent de Brique : Brique1, Brique2, Brique2 et Briquebonus. Brique hérite de Rectangle car dans notre modèle, une brique est un rectangle avec de légères spécificités supplémentaires. De manière générale, une relation « **est un(e)** » doit être représentée par de l'héritage. Brique possède, en plus de Rectangle, un attribut compteur et un attribut score. Le compteur sert à déterminer le nombre de collisions nécessaires pour détruire une brique. Le score sert quant à lui à déterminer le score qu'apporte une brique lorsqu'elle est détruite.

Un autre point important du diagramme de classe est qu'il présente des relations d'agrégation et des relations de composition. Ces notions sont similaires, bien qu'il y ait une légère différence. Une relation de composition peut être vue comme une relation « **fait partie de** ». Une telle relation implique que les cycles de vie des deux classes en relation sont liés. L'objet, qui est composé d'autres objets, a besoin de ces objets pour exister ; ils font partie de sa définition. Si ces objets sont détruits, l'objet composé ne peut pas exister. Une relation de composition est représentée en UML par un losange plein, du côté de la classe qui compose l'autre. Dans notre diagramme, on a par exemple un rectangle qui est composé de deux points. Rectangle a donc deux attributs de type Point, et si on retire un de ces deux points, il parait logique que le rectangle n'existe plus. De même, Cercle est composé d'un centre de type Point ainsi que d'un rayon.

La relation d'agrégation peut être vue comme une relation « **a un/des** ». Les cycles de vie des objets en relation ne sont pas liés à priori. Par exemple, un livre a des pages, mais si on retire une page du livre, cela reste un livre. Cette différence avec la composition est cependant parfois sujette à l'interprétation du concepteur. En effet, un livre avec uniquement les couvertures et sans page est-il toujours un livre ? Dans notre diagramme de classes, nous avons choisi de représenter certaines relations par des relations d'agrégation. Par exemple, un menu a des boutons, un mur de brique a des briques. Si on enlève des briques au mur, nul doute que cela reste un mur de briques. D'ailleurs, d'un point de vue informatique, si on enlève toutes les briques au mur, le mur existera toujours en mémoire si on ne le détruit pas. Pour le mur de briques, le support d'agrégation utilisé est le type 'list', déjà présent en C++. Une liste peut exister, même si elle est vide. La tête de liste pointe tout simplement vers NULL. De même, un tableau sans élément existe bel et bien en mémoire. Dans notre modèle un menu possède d'ailleurs un tableau dynamique de boutons, donc le raisonnement est le même. La relation d'agrégation en UML est représentée de la même façon que la relation de composition, sauf que le losange est n'est pas plein.

2) Généricité des comportements

La généricité des comportements est une des clefs pour produire un programme efficace. Elle est possible grâce aux différents concepts de la Programmation Orientée Objet. Bien-sûr, il est nécessaire de maitriser ces concepts, pour pouvoir les utiliser de la manière la plus efficiente possible. Nous allons expliquer dans cette partie comment nous avons pu rendre les comportements du notre jeu génériques.

a) Héritage et polymorphisme d'inclusion

L'héritage est une des notions les plus importantes de la POO. Comme expliqué précédemment, l'héritage est utilisé lorsque deux classes sont liées par une relation « **est un(e)** ». Par exemple, dans notre projet, une balle est un cercle. La relation d'héritage est très puissante, puisqu'elle nous permet d'accéder aux attributs et aux méthodes de Cercle qui sont en 'public' ou en 'protected' depuis Balle. Ainsi, une balle possède un centre, un rayon, mais également des attributs spécifiques qui ne sont pas dans la classe Cercle : une vitesse par exemple. On peut donc recycler des attributs/méthodes déjà présents dans Cercle, sans avoir à les redévelopper. Pour afficher une balle, on doit afficher le cercle. Qu'à cela ne tienne, nous développons une méthode affiche dans Cercle, et nous pouvons directement l'appeler avec une instance de la classe Balle ! C'est ce que l'on appelle le polymorphisme d'inclusion.

b) Classes abstraites et méthodes virtuelles pures

L'abstraction est une autre notion majeure de la POO. Déjà évoquée lors de la présentation du diagramme de classes, cette notion permet de rendre génériques certains comportements utilisés dans différentes classes qui dérivent d'une même classe mère, mais qui ne font pas exactement la même chose. Prenons un exemple simple pour illustrer l'abstraction. Nous savons que nous voulons pouvoir afficher n'importe quelle forme de notre jeu. Mais afficher un rectangle n'est pas la même chose que d'afficher un cercle, cela demandera des instructions graphiques différentes. La première chose à laquelle nous pourrions penser, c'est de développer deux fonctions distinctes : afficheCercle et afficheRectangle. Cependant, on constate assez aisément que cela ne permet pas de rendre l'affichage de formes générique, donc c'est une mauvaise solution. Il faut déclarer une méthode virtuelle pure affiche dans la classe mère fédératrice (qui devient donc abstraite et donc non instanciable) et définir le comportement de affiche dans Cercle et dans Rectangle. Une ligature dynamique est ainsi créée, quand une forme appellera la méthode affiche, la bonne méthode sera appelée en fonction du type de la forme. Le principe est similaire avec une méthode virtuelle (non pure), sauf que le comportement de la méthode est également défini dans la classe mère, ce qui la rend donc instanciable. Au final, on pourra faire du

‘.affiche()’ sur n’importe quelle forme pour pouvoir l’afficher, ce qui est beaucoup plus propre et générique.

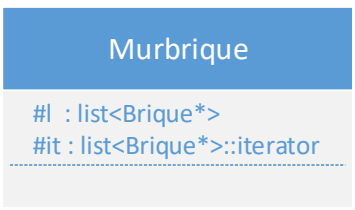
De même, on veut que chaque forme de notre jeu soit sélectionnable, pour éventuellement gérer des évènements à la souris (ce qui n’est pas le cas ici), ou pour pouvoir tout simplement gérer les collisions (aspect très important du jeu). Une telle méthode renvoie donc un booléen : TRUE si les coordonnées du point passé en paramètre sont dans la forme, FALSE sinon. Cette méthode sélection dépend de la forme qui appelle la méthode. De même que pour affiche, on crée une méthode virtuelle pure dans Forme, et on définit son comportement dans les différentes formes.

Enfin, nous pouvons appliquer le même raisonnement sur la méthode action de Brique. Lors de la collision d’une balle avec une brique, l’action à réaliser ne sera pas forcément la même, notamment si on tente de gérer un système de bonus/malus.

c) Relations d’agrégation

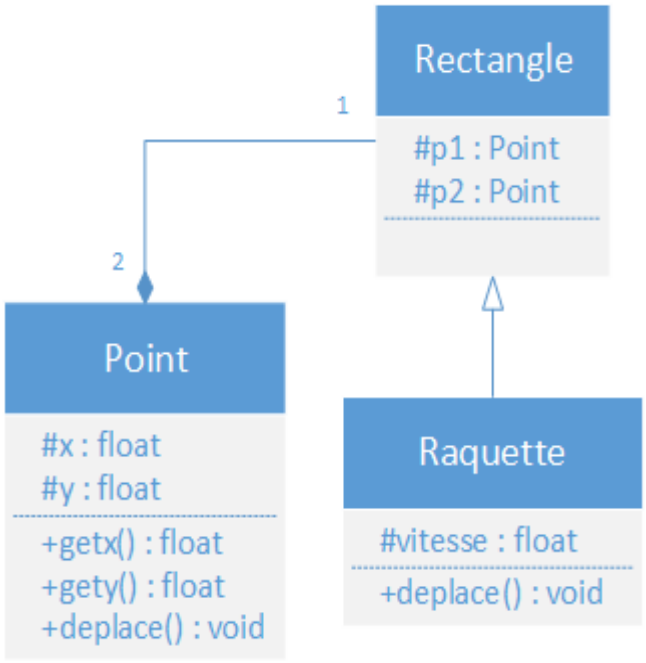
Le principe de la relation d’agrégation a déjà été expliqué précédemment dans le rapport. Nous allons nous contenter d’expliquer en quoi cette relation est grandement utile pour rendre certains comportements génériques.

Reprenons l’exemple de l’affichage. Maintenant que nous avons implémenté des méthodes virtuelles pures dans notre modèle, nous pouvons faire pour n’importe quelle forme du ‘.affiche()’. Le seul souci est lorsqu’on veut manipuler toutes les formes. On devrait à priori appeler la méthode affiche n fois, avec n égal au nombre de formes. C’est clairement beaucoup trop fastidieux. L’idée est donc de regrouper toutes les formes dans une classe agrégatrice : la classe Listeforme. On utilise dans la classe agrégatrice un support d’agrégation qui va nous permettre de parcourir facilement la liste de formes. Pour cette classe, le support d’agrégation choisi est le type ‘list’. On peut utiliser ce type, pour peu qu’on inclue <list.h> aux fichiers concernés. List est tout simplement ce qu’on appelle une classe Container. Une telle classe définit un support permettant de réunir des objets. Dans la théorie, il existe un Design Pattern permettant de réaliser proprement une classe Container. Pour la classe List, un autre Design Pattern est implémenté : Iterator. Il s’agit de définir un élément qui nous permette de parcourir notre Container. Voici un exemple de classe agrégatrice :



d) Relations de composition

La relation de composition permet aussi d'améliorer la généricité de certains comportements. Prenons l'exemple de la raquette, qui est un rectangle donc qui est composée de 2 points. Pour déplacer la raquette, il suffit de déplacer les deux points qui la constituent. Ainsi, nous pouvons développer une méthode deplace dans Point qui incrémente d'un pas dx la coordonnée sur l'axe des abscisses et d'un pas dy la coordonnée sur l'axe des ordonnées. On développe enfin une méthode deplace dans Raquette qui appelle la méthode deplace de Point sur les deux points de la raquette, et le tour est joué ! Nous reparlerons des déplacements dans la partie qui suit, donc nous nos explications ici restent assez succinctes. Voici ce que cela nous donne du point de vue de la modélisation en UML :



NB : seuls les attributs/méthodes pertinents pour notre exemple ont été représentés

3) Fonctionnalités importantes

Les spécificités du Casse-Briques nous ont amenés à développer un certain nombre de fonctionnalités plus ou moins complexes. Les fonctionnalités principales sont les déplacements et les collisions, et nous allons les détailler ci-dessous.

a) Déplacement de la balle

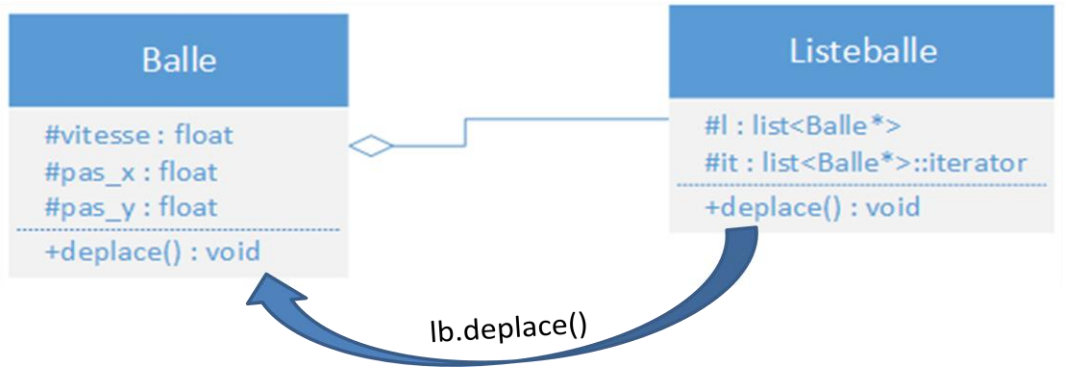
Code en annexe

Premièrement, le déplacement de la balle utilise la relation de composition avec la classe Point. Pour rappel, une balle est un cercle et est par conséquent constituée d'un point (son centre) et d'un rayon. Déplacer cette forme revient à déplacer son centre, donc à appeler la méthode deplace de Point. Voici un algorithme succinct qui présente le déplacement de la balle :

Appel de la méthode deplace() de Point
Si on touche une limite horizontale
    pas_x = -pax_x
Si on touche une limite verticale
    pas_y = -pax_y

Ceci admet que nous disposons d'attributs pas_x et pas_y dans Balle. Nous avons justement implémenté de tels attributs, ainsi qu'un attribut vitesse. L'attribut vitesse est la valeur absolue du pas, et doit être fournie en argument pour instancier une balle (via son constructeur). Pas_x et pas_y oscillent ainsi entre vitesse et –vitesse, selon la direction que la balle doit prendre. La balle se déplace ainsi à des angles de pi/4 [modulo pi/2].

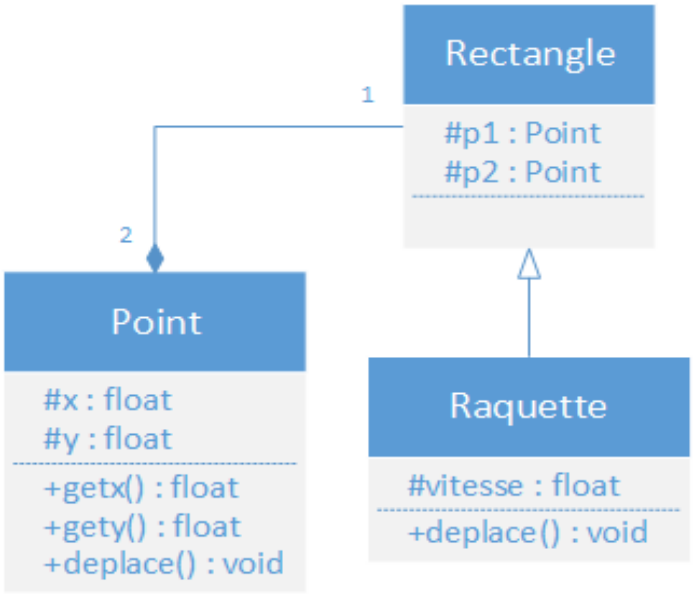
Nous avons décidé que notre jeu serait capable de gérer plusieurs balles en même temps sur le terrain. On a donc créé une classe agrégatrice (Listeballe) qui nous permet de parcourir la liste de balles à l'aide d'un itérateur. On développe donc une méthode deplace dans Listeballe qui boucle sur les balles pour appeler leur méthode deplace respective (qui appellent elles-mêmes la méthode deplace de Point, comme vu précédemment). On pourra donc instancier un objet lb de type Listeballe et faire un lb.deplace() pour que toutes les balles soient en mouvement !



b) Déplacement de la raquette

Code en annexe

Le déplacement de la raquette n'est pas compliqué, il utilise tout simplement les relations qui lient les classes Raquette, Rectangle et Point. En effet, la classe Raquette hérite de la classe Rectangle et cette dernière est composée de deux Points. Ainsi, Raquette est également composée de deux Points, puisque ces attributs sont protected. Comme pour le déplacement de la balle, il nous suffit d'appeler la méthode deplace développée dans Point. Le mouvement de la raquette ne s'effectuant que sur l'axe des abscisses, le pas de déplacement ne sera envoyé qu'au premier argument. Ce pas n'est d'autre que la vitesse de la raquette, initialisée à l'instanciation de celle-ci.



Nous nous heurtons cependant à un problème : la raquette peut pour le moment sortir de l'écran, or nous aimerions qu'elle soit bloquée sur les bords droit et gauche. Pour résoudre ce problème, nous avons pensé à faire un test supplémentaire sur la coordonnée x des deux points ; nous vérifions que celle-ci ne dépasse pas la fenêtre de jeu. Si ce n'est pas le cas, nous effectuons un léger déplacement des points vers le sens opposé. Ainsi, cela crée une sorte de tremblement de la raquette lorsqu'elle tente de sortir du terrain car elle est automatiquement reculée. L'algorithme est donc le suivant :

```
Si on appuie sur la flèche droite du clavier
    Si p2.getx() < bord droit
        p1.deplace(vitesse,0)
        p2.deplace(vitesse,0)
    Sinon
        p1.deplace(-3,0) //3 étant une valeur choisie arbitrairement
        p2.deplace(-3,0)
```

```
Si on appuie sur la flèche gauche du clavier
    Si p2.getx() > bord gauche
        p1.deplace(-vitesse,0)
        p2.deplace(-vitesse,0)
    Sinon
        p1.deplace(3,0)
        p2.deplace(3,0)
```

c) Collisions

Code en annexe

Le but de notre jeu est que la balle rebondisse sur la raquette et détruise les briques avec lesquelles elle rentre en collision. Ainsi, les collisions sont un point fondamental du jeu. Nous allons expliciter la stratégie que nous avons adoptée pour gérer celles-ci.

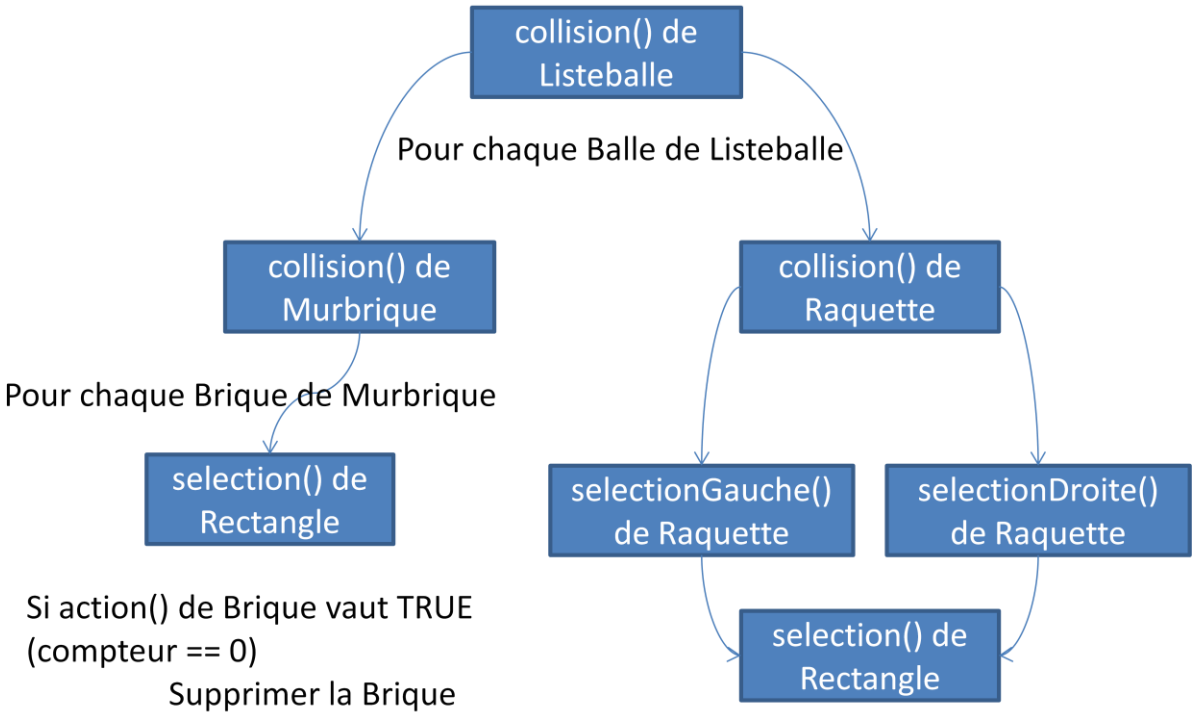
Une nouvelle fois, notre objectif est d'utiliser pleinement les relations que nous avons implémentées pour rendre nos comportements génériques. Nous avons développé une méthode collision() dans Listeballe, qui prend en paramètres les différents éléments du jeu pouvant être en collision avec une balle : la raquette et le mur de briques. C'est la seule et unique méthode concernant les collisions qui sera appelée depuis notre main. Cette méthode boucle sur chaque balle présente sur le terrain et appelle la méthode collision() de Murbrique ainsi que la méthode collision() de Raquette.

Attardons nous dans un premier temps sur la méthode collision() de Murbrique. Celle-ci, boucle sur les briques et appelle la méthode selection() de Rectangle. On rappelle que cette méthode est accessible depuis une brique puisque Brique hérite de Rectangle. Cette méthode renvoie VRAI si le point passé en paramètre est inclus dans le rectangle. On peut donc appeler cette méthode pour chaque point d'impact autour de la balle. Ici, nous avons choisi 4 points d'impact : les 4 points cardinaux principaux d'une boussole. On verra plus tard que ce choix peut poser problème dans certains cas et qu'il faudrait l'affiner. Quand une balle rentre en collision avec une brique, donc qu'une méthode selection() renvoie VRAI, alors le compteur associé à cette brique est décrémenté. Si celui-ci est égal à 0, la méthode action() de Brique est appelée. Elle supprime la brique, et active éventuellement un bonus dans les cas des briques prévues à cet effet.

Concernant la méthode collision() de Raquette, elle appelle deux méthodes : une méthode selectionDroite(), et une méthode selectionGauche(). Pourquoi ces deux méthodes ? Car nous nous sommes rendu compte que le rebond sur la raquette était beaucoup trop simplifié si on ne partitionnait pas la raquette. On a donc divisé la raquette en 2, et la balle possède un rebond différent en fonction du bord de la raquette qu'elle touche, et en fonction du sens dans lequel elle arrive. On aurait pu encore plus partitionner la

raquette, mais cela aurait complexifié le rebond et nous avons voulu rester sur des collisions relativement simples pour pouvoir nous attarder également sur d'autres parties du jeu.

Voici un schéma explicatif pour résumer le fonctionnement des collisions :



4) Interface graphique

Dans un jeu, nous devons nous confronter assez rapidement à la problématique de l'affichage. Nous verrons que ce n'est pas la partie la plus compliquée, il suffit tout simplement de connaitre les instructions graphiques qui permettent d'afficher tel ou tel élément. Pour rappel, nous avons choisi d'utiliser la bibliothèque graphique Allegro, car nos cours ont été réalisés avec celle-ci donc nous ne souhaitions pas perdre du temps à sortir des sentiers battus.

a) Instructions d'affichage

Pour notre jeu, nous devons afficher un mur de briques, une raquette et des balles. En clair, nous devons donc afficher deux types de formes : des rectangles et des cercles. Il existe des instructions sous Allegro pour tracer ces formes : rectfill() et circlefill(). Il nous suffit donc de développer une méthode affiche() dans la classe Rectangle et une dans la classe Cercle en y plaçant ces instructions. Ces méthodes seront accessibles pour toutes les classes héritant de Rectangle et de Cercle, donc on pourra afficher les éléments souhaités.

Un problème persiste, nous aimerions rendre l'affichage générique, puisque nous n'allons pas nous amuser à appeler dans le main ces méthodes affiche() pour chaque forme instanciée. Pour cela, nous utilisons l'historique principal : la classe Listeforme. Nous développons une méthode affiche() dans celle-ci, qui boucle sur les éléments présents dans l'historique (à savoir les trois éléments principaux du jeu : le mur de briques, la raquette et la liste de balles), et qui appelle respectivement leur méthode affiche(). Le principe est toujours le même, la méthode affiche() de Murbrique boucle sur les briques et appelle la méthode affiche() de Rectangle qui est accessible à partir d'une brique. De même, la méthode affiche() de Listeballe boucle sur les balles et appelle la méthode affiche() de cercle. Enfin, la raquette peut s'afficher directement en appelant la méthode affiche() de Rectangle.

Ainsi, si on instancie un objet Listeforme dans le main nommé ll, il nous suffira de faire un ll.affiche() dans la boucle principale de jeu pour que tous les objets soient affichés en permanence.

b) Stratégie du double-buffering

Code en annexe

Une autre problématique se pose : les éléments de notre jeu subissent un « clignotement » à l'affichage. Pour pallier à cela, nous avons choisi de mettre en place une stratégie de double-buffering. L'idée est simple, nous utilisons un buffer intermédiaire (nommé page ici) en plus du buffer vidéo. A chaque tour de boucle du jeu, on nettoie le buffer intermédiaire et on y place les éléments à afficher. Ensuite, on réalise une copie du buffer intermédiaire vers le buffer vidéo (à l'aide de la fonction blit d'Allegro). Cela permet de ne pas vider directement le buffer vidéo, et donc d'éviter le fameux clignotement.

5) Game design

Pour rendre notre jeu plus agréable à jouer, nous avons pensé à ajouter des fonctionnalités qui n'ont pas été abordées en cours. Cela a donc demandé un peu de créativité. Nous allons les présenter ci-dessous.

a) Génération aléatoire du mur de briques

Code en annexe

Premièrement, nous nous sommes rendu compte que le jeu était beaucoup trop répétitif si le mur de briques était toujours le même. Ainsi, nous avons voulu rendre la génération du mur de briques aléatoire.

De base, le constructeur de Murbrique parcourait deux boucles for (avec pour compteurs i et j, paramétrés de manière judicieuse pour avoir un espacement régulier entre chaque brique) pour remplir le mur. Etant donné que le support d'agrégation est de type list, nous avons utilisé la méthode push_back pour ajouter nos briques à l'historique mur (instanciation dynamique avec le mot clef new pour une meilleure gestion de la mémoire). Il nous a suffit d'ajouter à cela une variable entière prenant une valeur aléatoire entre deux bornes définies (entre 1 et 3 par exemple). A chaque tour de boucle, on instancie une brique de type 1, 2 ou 3 en fonction de la valeur de cette variable.

b) Classe Briquebonus

Code en annexe

Pour rendre les parties encore plus originales, nous avons choisi d'implémenter une classe Briquebonus, en plus des 3 types de briques existants déjà. Le principe est qu'en cassant une brique de ce type, un bonus s'active. Il nous a donc fallu imaginer des bonus.

Nous avons mis en place 4 bonus différents :

- Une augmentation de la taille de la balle : le rayon de la balle est augmenté à chaque fois que l'on obtient ce bonus.
- Une augmentation de la vitesse du jeu : la vitesse de la balle ainsi que la vitesse de la raquette augmentent à chaque fois que l'on obtient ce bonus. Les deux éléments changent également de couleur pour marquer le fait que le bonus est actif.
- Une inversion des mouvements de la raquette : lorsqu'on appuie sur la flèche droite, la raquette se déplace vers la gauche et vice-versa. La raquette change de couleur pour marquer le fait qu'elle est inversée (il s'agit d'un attribut booléen nommé « inverse » dans Raquette qui est mis à VRAI). Si on obtient deux fois ce bonus, la raquette reprend un comportement normal.
- Une augmentation du nombre de balles : une balle est ajoutée à l'historique Listeballe à chaque fois que l'on obtient ce bonus. Cette nouvelle balle réagit comme la balle principale au niveau du déplacement, des collions…

Nous voulions que ces bonus soient choisis aléatoirement. Nous avons donc ajouté un attribut bonus dans Briquebonus, qui n'est d'autre qu'un entier prenant une valeur aléatoire entre 1 et le nombre de bonus disponibles (cette valeur est affectée dans le constructeur de Briquebonus). Nous avons également spécialisé le comportement de la méthode action(), qui pour rappel est une méthode virtuelle pure de Brique. Dans cette méthode, on teste la valeur de l'attribut « bonus » et on effectue les traitements voulus.

Pour que les briques bonus soient plus rares que les autres briques, nous avons pensé à revoir légèrement la construction du mur. La valeur aléatoire qui prenait des valeurs entre 1 et 3 prend dorénavant des valeurs entre 1 et 10. Si elle vaut 1, 4 ou 7 c'est une brique de type 1 qui est instanciée, si elle vaut 2, 5 ou 8 c'est une brique de type 2, si elle vaut 3, 6 ou 9 c'est une brique de type 3 et enfin si elle vaut 10 c'est une brique de type bonus. Ainsi, il y a 1 chance sur 10 d'instancier une brique bonus dans le mur.

c) Interface « user-friendly »

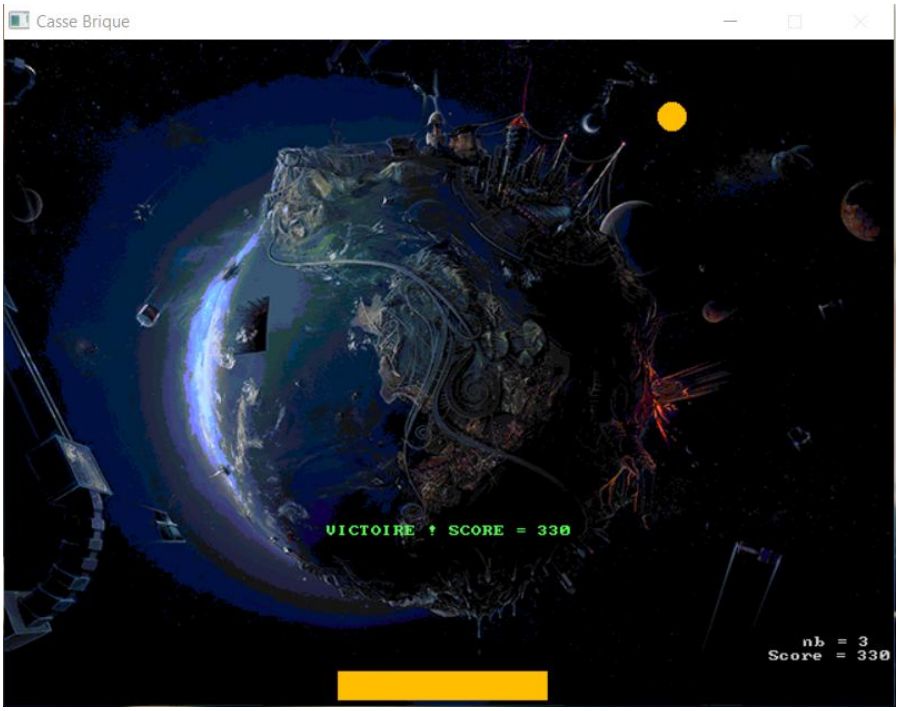
Code en annexe

Jusqu'alors, notre jeu n'est pas très attrayant. Nous avons pensé à améliorer l'expérience de jeu en implémentant des thèmes pour chaque partie. Un thème est composé d'une musique ainsi que d'une image de fond. Pour représenter cela, nous avons utilisé une structure avec 2 champs. Cette structure est remplie aléatoire en début de chaque partie, à l'aide d'une fonction theme_init(). Nous avons ajouté deux dossiers à notre projet : un dossier contenant des images et un dossier contenant des musiques. Pour le moment, 4 thèmes ont été ajoutés : un thème James Bond, un thème Plage, un thème Jungle et un thème Espace.

Exemple du thème Plage (+ bonus nombre de balles)

Nous avons également mis en place un score de jeu, que vous pouvez voir en bas à droite de l'écran évoluer tout au long de la partie. Dans la classe abstraite Brique, nous avons mis un attribut « score », dont la valeur varie en fonction de la brique instanciée. Une brique cassable en n coups rapportera n*10 points quand elle sera détruite (donc quand sa méthode action() renverra VRAI).

Lorsqu'on gagne ou perd une partie, le message est affiché (en vert si on a gagné, en rouge sinon), nous indiquant notre score final. Pour représenter l'état gagné/perdu, nous avons tout simplement utilisé deux variables globales booléennes nommées LOSE et WIN. On peut se permettre de mettre ces variables en global, bien que cela soit souvent fortement déconseillé, car elles ont réellement un statu global pour notre jeu. De plus, on aimerait pouvoir y accéder depuis plusieurs endroits. Par exemple, on sait que l'on a perdu lorsque la balle franchit la limite basse de l'écran. Ceci est accessible directement dans le déplacement de la balle. Au contraire, on a gagné lorsqu'il n'y a plus de briques sur le terrain. Ceci est accessible dans la classe Murbrique, à l'aide de l'attribut nbBriques.



Exemple de partie gagnée (+ bonus vitesse)

Enfin, nous avons inséré des bruitages dans notre jeu. Si une balle rentre en collision avec une brique, ou une raquette, un bruitage s'active. De même il y a un bruitage différent par bonus récupéré, en concordance avec les effets de celui-ci. Nous n'allons pas détailler plus que cela la mise en place des bruitages puisqu'il s'agit uniquement d'instructions Allegro à utiliser aux bons endroits.

6) Problèmes rencontrés

Un problème a été rencontré au niveau des inclusions de fichiers entre eux. En effet, notre projet a été organisé en plusieurs fichiers pour en améliorer grandement la lisibilité générale. Chaque déclaration de classe est contenue dans un fichier d'entête .h du nom de la classe, tandis que les définitions des méthodes sont regroupées dans un fichier .cpp du nom de la classe également. Enfin, le fichier main.cpp contient le programme principal. Or, selon notre architecture de classes, la classe Listeforme a besoin de la classe Forme puisque par définition c'est une liste de formes, et la classe Forme a également besoin de la classe Listeforme puisque nous avons mis un attribut static de type Listeforme. Une double inclusion de la sorte créerait une boucle infinie pour le préprocesseur. De ce fait, nous avons dû mettre uniquement le prototype de la classe Forme dans Listeforme.h, sans faire d'inclusion du fichier Forme.h.

Un autre léger problème rencontré est que nous avions besoin d'utiliser le buffer intermédiaire dans plusieurs parties du programme (donc dans plusieurs fichiers). Ainsi, nous l'avons logiquement mis en variable globale. Ceci dit, pour qu'une même variable globale soit effective dans différent fichiers, il faut utiliser un mot clef que nous ne connaissions pas encore : extern. Ce mot clef permet de dire qu'on fait référence à une variable globale définie ailleurs dans le programme.

7) Améliorations possibles

Une première amélioration possible est qu'on aurait pu penser différemment quelques relations du diagramme de classes. En effet, la classe Bouton aurait par exemple pu hériter de Rectangle, car il s'agit bien d'un rectangle avec un texte à l'intérieur. Or nous avons utilisé une relation de composition avec Rectangle. Rien ne justifie vraiment cela, si ce n'est le fait que nous avons fait ce choix au moment où nous avons étudié le chapitre sur la composition.

En parlant de la classe Bouton, une autre amélioration possible est qu'on aurait pu implémenter graphiquement le menu dans notre jeu. Pour l'instant, le menu fait uniquement partie du diagramme de classes mais n'est pas utilisé graphiquement. Nous avons fait ce choix avec M. Delahoche car les délais impartis pour réaliser ce projet étaient courts.

Pour l'instant, les collisions entre différentes balles ne sont pas gérées. Une méthode selection() est implémentée dans la classe Cercle, donc cela devrait être assez simple à réaliser. Dans collision() de Listeballe, lorsqu'on boucle sur les balles avec un itérateur, on a accès à la balle courante. Il suffit donc de tester si la balle courante entre en contact avec une autre balle à l'aide de la méthode selection() de Cercle.

Au sujet des collisions, nous avons choisi de ne prendre en compte que 4 points d'impact autour de la balle (qui sont les points cardinaux principaux d'une boussole). Ceci n'est pas judicieux dans tous les cas, et nous nous en sommes rendu compte à l'aide du bonus qui augmente le rayon de la balle. En effet, lorsque la balle devient plus imposante, il faudrait insérer un nombre plus important de points d'impact autour d'elle pour que les collisions soient optimales. Par contre, mettre 8 points d'impact alors que la balle est assez petite fait buguer les collisions puisque 2 méthodes selection() peuvent renvoyer VRAI en même temps donc il y a une double collision à un endroit où il devrait n'y en avoir qu'une seule. Une solution potentielle pourrait être de mettre à jour le nombre de points d'impact autour de la balle lorsque son rayon change. On pourrait donc réaliser une fonction interne à Balle (privée) nommée maj_points_impact qui est appelée lorsque que le rayon de la balle est modifié. Cette méthode testerait si le rayon atteint une certaine valeur, et mettrait à jour les points d'impact de la balle si nécessaire.

Enfin, le déplacement de la balle aurait pu être complexifié. Nous avons volontairement choisi de le garder relativement simple car nous devions rendre une version jouable dans des délais assez courts. Or, cette amélioration aurait demandé de beaucoup plus réfléchir sur le plan mathématique.

Conclusion

En conclusion, ce projet a été très formateur puisqu'il nous a permis de mettre en pratique toutes les connaissances concernant la Programmation Orientée Objet que nous avons étudiées durant le semestre, à travers la réalisation d'un jeu très académique : le Casse-Briques. Nous avons dû réfléchir à une modélisation objet qui nous permette de réaliser un programme efficace, extensible et générique. Pour cela, nous avons réalisé un diagramme de classes à l'aide du langage UML, très utilisé en modélisation. La réalisation de ce diagramme est un point primordial du projet, puisque nos choix de conception faits à cette étape orienteront tout notre programme. Il est important de faire des choix judicieux pour gagner un temps précieux lors de l'implémentation des différentes fonctionnalités.

L'objectif principal de ce projet était, selon nous, d'utiliser les concepts de la POO pour rendre les fonctionnalités de notre jeu modulaires et génériques. C'est dans ce cadre que nous avons mis en place diverses relations entre classes (héritage, composition, agrégation), ou encore que nous avons implémenté des classes abstraites contenant des méthodes virtuelles pures judicieusement choisies. En développant les fonctionnalités principales du jeu (l'affichage, les déplacements, les collisions…), nous avons pris conscience de la force de la POO. Nous avons fait énormément de réutilisation de code, si bien que celui-ci n'était pas si lourd, malgré le nombre de classes assez conséquent. Très souvent dans ce rapport, nous avons rappelé les relations qui liaient nos classes pour justifier l'utilisation directe de tel attribut ou de telle méthode. Les seules fonctionnalités redéveloppées concernaient des spécialisations nécessaires. Notre main final est très succinct, ce qui prouve que nous avons réussi à bien appréhender la POO.

A travers ce projet, nous avons également pu laisser libre cours à notre créativité. Quelles classes développer ? Quels attributs et quelles méthodes y mettre ? Quelles classes devraient jouer le rôle de classes fédératrices ? Quelles méthodes devraient être virtuelles ? Autant de réponses auxquelles nous avons dû répondre pour développer de nouvelles fonctionnalités pour notre jeu. Nous avons également dû nous mettre à la place de l'utilisateur, pour pouvoir imaginer un Game-design original qui donne envie de rejouer à notre jeu. C'est dans ce contexte que nous avons mis en place un système de bonus, de thèmes, ou encore de bruitages.

Enfin, nous avons pris du recul vis-à-vis de notre programme. En effet, nous avons réussi à dégager un certain nombre d'améliorations possibles pour rendre notre jeu encore plus complet. Nous avons remis en question certains choix de conception et nous avons énoncé des alternatives. En POO, il n'y a jamais une seule manière de penser les choses. Il est important de remettre en perspective nos choix pour pouvoir faire évoluer notre projet dans le bon sens.

Annexes

Déplacement de la balle :

```
void balle::deplace()
{
    p.deplace(pas_x,pas_y);
    if(p.getx()+rayon>=640)
    {
        set_pas_x(-pas_x);
    }
    else if(p.gety()-rayon<=0)
    {
        set_pas_y(-pas_y);
    }
    else if(p.getx()-rayon<=0)
    {
        set_pas_x(-pas_x);
    }
    else if(p.gety()+rayon>=530)
    {
        LOSE = true;
    }
}

void balle::set_pas_x(float dx)
{
    pas_x=dx;
}
void balle::set_pas_y(float dy)
{
    pas_y=dy;
}
```

Déplacement de la raquette :

```
if(key[KEY_RIGHT])
{
    if(p2.getx()<SCREEN_W)
    {
        p1.deplace(vitesse,0);
        p2.deplace(vitesse,0);
    }
    else
    {
        p1.deplace(-3,0);
        p2.deplace(-3,0);
    }
}
if(key[KEY_LEFT])
{
    if(p1.getx()>0)
    {
        p1.deplace(-vitesse,0);
        p2.deplace(-vitesse,0);
    }
    else
    {
        p1.deplace(3,0);
        p2.deplace(3,0);
    }
}
```

Collisions :

```
void listeballe::collision(murbrique* m, raquette* r)
{
    for(it=l.begin();it!=l.end();it++)
    {
        r->collision(*it);
        m->collision(*it,r,this);
    }
}
```

```
void raquette::collision(balle* b)
{
    float x_centre_balle;
    float y_centre_balle;
    b->get_p(&x_centre_balle,&y_centre_balle);
    float rayon_balle = b->get_rayon();
    float dx = b->get_pas_x();
    float dy = b->get_pas_y();
    if((*this).selectionDroite(x_centre_balle,y_centre_balle+rayon_balle))
    {
        play_sample(beep_raquette,255,128,800,false);
        if(dx>0)
            b->set_pas_y(-dy);
        else if(dx<0)
        {
            b->set_pas_y(-dy);
            b->set_pas_x(-dx);
        }
    }
    else if((*this).selectionGauche(x_centre_balle,y_centre_balle+rayon_balle))
    {
        play_sample(beep_raquette,255,128,800,false);
        if(dx>0)
        {
            b->set_pas_y(-dy);
            b->set_pas_x(-dx);
        }
        else if(dx<0)
            b->set_pas_y(-dy);
    }
}
```

```
void murbrique::collision(balle* b,raquette* r,listeballe* lb)
{
    for(it=l.begin();it!=l.end();it++)
    {
        float x_centre_balle;
        float y_centre_balle;
        b->get_p(&x_centre_balle,&y_centre_balle);
        float rayon_balle = b->get_rayon();
        float dx = b->get_pas_x();
        float dy = b->get_pas_y();
        if((*it)->selection(x_centre_balle,y_centre_balle-rayon_balle) || (*it)->selection(x_centre_balle,y_centre_balle+rayon_balle))
        {
            play_sample(beep_brique,255,128,800,false);
            b->set_pas_y(-dy);
            if((*it)->action(b,r,lb))
            {
                suppression_brique(*it);
                return;
            }
        }
        if((*it)->selection(x_centre_balle+rayon_balle,y_centre_balle) || (*it)->selection(x_centre_balle-rayon_balle,y_centre_balle))
        {
            play_sample(beep_brique,255,128,800,false);
            b->set_pas_x(-dx);
            if((*it)->action(b,r,lb))
            {
                suppression_brique(*it);
                return;
            }
        }
    }
    if(nbBriques==0)
    {
        WIN = true;
    }
}
```

Boucle principale de jeu et double-buffering :

```
do{
    clear_bitmap(page);
    blit(image_fond,page,0,0,0,0,page->w,page->h);
    textprintf(image_fond,font,575,430,makecol(200,200,200),"nb = %d",listeforme::getnb());
    textprintf(page,font,550,440,makecol(200,200,200),"Score = %d",m.getScore());
    ll.affiche();
    r.deplace();
    lb.deplace();
    lb.collision(&m,&r);
    blit(page,screen,0,0,0,0,page->w,page->h);
} while(!key[KEY_ESC] && LOSE==false && WIN==false);
```

Génération aléatoire du mur de briques :

```
murbrique::murbrique()
{
    score=0;
    int r;
    nbBriques = 20;
    bloque();
    srand(time(NULL));
    for(int i=50;i<=490;i=i+110)
    {
        for(int j=30;j<=180;j=j+50)
        {
            r=rand()%10+1;
            if(r==1 || r==4 || r==7)
            {
                l.push_back(new brique1(i,j));
            }
            else if(r==2 || r==5 || r==8)
            {
                l.push_back(new brique2(i,j));
            }
            else if(r==3 || r==6 || r==9)
            {
                l.push_back(new brique3(i,j));
            }
            else if(r==10)
            {
                l.push_back(new briquebonus(i,j));
            }
        }
    }
    debloque();
}
```

Gestion des thèmes :

```
typedef struct Theme {
    char* image_background;
    char* son_background;
} Theme;

Theme theme_init()
{
    Theme* themes;
    themes = new Theme[4];

    themes[0].image_background="images/background/donkey_kong.pcx";
    themes[1].image_background="images/background/james_bond.pcx";
    themes[2].image_background="images/background/star_wars.pcx";
    themes[3].image_background="images/background/tropique.pcx";

    themes[0].son_background="sounds/background/donkey_kong.mid";
    themes[1].son_background="sounds/background/james_bond2.mid";
    themes[2].son_background="sounds/background/espace.mid";
    themes[3].son_background="sounds/background/yoshi.mid";

    srand(time(NULL));
    int tirage_theme=rand()%4;
    Theme theme_choisi = themes[tirage_theme];

    return theme_choisi;
}
```

Gestion des bonus :

```
bool briquebonus::action(balle* b,raquette* r,listeballe* lb)
{
    compteur--;
    if(compteur==0)
    {
        if(bonus==1)
        {
            play_sample(taillePlus,255,128,800,false);
            b->set_rayon(b->get_rayon()+5);
        }
        else if(bonus==2)
        {
            play_sample(vitessePlus,255,128,800,false);
            b->set_vitesse(b->get_vitesse()+0.2);
            b->set_couleur(makecol(255,215,0));
            r->set_couleur(makecol(255,215,0));
            r->set_vitesse(r->get_vitesse()+0.2);
        }
        else if(bonus==3)
        {
            play_sample(inversionRaquette,255,128,800,false);
            if(r->get_inverse()==false)
            {
                r->set_inverse(true);
                r->set_couleur(makecol(100,100,100));
            }
            else
            {
                r->set_inverse(false);
                r->set_couleur(makecol(255,255,100));
            }
        }
        else if(bonus==4)
        {
            play_sample(balleSup,255,128,800,false);
            lb->ajoute_balle(315,435,10,0.4,makecol(255,255,255));
        }
        return true;
    }
    else
        return false;
}
```