

# Rapport du TER

11 mai 2010

Romain MANESCHI  
Audrey NOVAK  
Jonathan FHAL  
Mélanie KÖNIG  
Laurent MAILLET

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Généralités . . . . .	4
1.2	Fonctionnalités et contraintes . . . . .	4
1.3	Quelques définitions . . . . .	4
1.3.1	Bibliothèque Logicielle . . . . .	4
1.3.2	Ligne de produits . . . . .	4
1.3.3	Jeux d'Arcades . . . . .	5
1.4	Outils utilisés . . . . .	5
1.4.1	Choix du langage . . . . .	5
1.4.2	IDE . . . . .	5
1.4.3	Google Code . . . . .	6
1.4.4	SVN . . . . .	6
1.5	Modélisation UML . . . . .	6
1.6	Organisation . . . . .	6
1.6.1	Répartition des tâches . . . . .	6
<b>2</b>	<b>Code métier</b>	<b>7</b>
2.1	Le pattern MVC . . . . .	7
2.2	Objets . . . . .	8
2.3	Eléments . . . . .	10
2.3.1	Patron de conception Etat . . . . .	10
2.3.2	Actions . . . . .	11
2.4	Formes . . . . .	12
2.5	Collision . . . . .	13
2.5.2	Le test de collision de proximité . . . . .	14
2.5.1	recherche de proximité . . . . .	14
2.6	Le jeu . . . . .	17
<b>3</b>	<b>Contrôleur</b>	<b>18</b>
3.1	Souris . . . . .	18
3.2	Clavier . . . . .	18
3.3	Effets . . . . .	19
<b>4</b>	<b>Graphique</b>	<b>20</b>
4.1	Réutilisation des composants de flex . . . . .	20
4.2	Dessiner en ActionScript . . . . .	21
4.3	Liaison avec le code métier . . . . .	23
4.4	Textures et le pattern "decorator" . . . . .	24
4.5	Les écouteurs du graphisme . . . . .	25
<b>5</b>	<b>Patron de conception Fabrique</b>	<b>26</b>
<b>6</b>	<b>Créateur de jeu</b>	<b>27</b>
6.1	Utilisation du framework . . . . .	27
6.1.1	Créations des formes . . . . .	27
6.1.2	L'onglet central . . . . .	28
6.1.3	Modifications des propriétés d'un objet . . . . .	28
6.1.4	Génération de code . . . . .	29

6.2	petite conclusion . . . . .	29
<b>7</b>	<b>Jeux</b>	<b>30</b>
7.1	Le Menu . . . . .	30
7.2	Les Commandes . . . . .	30
7.3	Le Jeu . . . . .	31
7.4	Les données . . . . .	32
<b>8</b>	<b>Conclusion</b>	<b>33</b>
8.0.1	Difficultés . . . . .	33
8.0.2	Connaissances acquises . . . . .	33
<b>9</b>	<b>Bibliographie</b>	<b>35</b>
9.1	Général . . . . .	35
9.2	Collision . . . . .	35
9.3	Contrôleur . . . . .	35
9.4	Créateur de jeux . . . . .	35
<b>10</b>	<b>Annexe</b>	<b>36</b>
10.1	Guide pour l'utilisation du créateur de jeux . . . . .	36
10.1.1	Création d'un premier jeu . . . . .	36
10.1.2	Ajout des éléments à la scène . . . . .	37
10.1.3	Modification des propriétés des objets . . . . .	37
10.1.4	Génération du code . . . . .	40
10.1.5	Utilisation du code généré . . . . .	40
10.1.6	Ajouter des éléments au menu-accordéon . . . . .	41

# 1 Introduction

## 1.1 Généralités

Ce projet est réalisé dans le cadre de l'unité d'enseignement TER, en première année de Master Informatique à l'université des Sciences de Montpellier. Il a été proposé et encadré par le professeur Christophe Dony. Notre groupe est composé de 5 étudiants.

Ce projet consiste à créer un ensemble de classes qui permettra de programmer plus facilement des jeux de type "arcade". Ces jeux pourront être utilisés sur Internet, quelque soit le navigateur et le système d'exploitation de l'utilisateur. En outre ils pourront également tourner directement dans le système de l'utilisateur.

## 1.2 Fonctionnalités et contraintes

Avec l'aide de notre framework, un utilisateur pourra facilement créer des jeux d'arcades et donner aux objets qu'ils contiennent des comportements courants dans ce type de jeux tels que la collision par exemple.

La principale contrainte que nous imposait le sujet était que l'utilisateur puisse programmer un petit jeu d'arcade en quelques lignes, ou tout du moins en peu de temps et ceci en étendant simplement les classes de notre framework. La seconde contrainte était que les jeux créés à l'aide du framework puisse être utilisable sur Internet.

De plus, afin de mettre en pratique nos cours d'UML, mais surtout pour bénéficier d'une architecture de classes claire et facilement maintenable, nous avons décidé d'implémenter nos classes sous la forme du modèle MVC.

## 1.3 Quelques définitions

Avant de commencer à présenter notre framework, il nous paraît important de rappeler quelques définitions :

### 1.3.1 Bibliothèque Logicielle

Egalement appelée librairie, une bibliothèque logicielle est constituée d'un ensemble de fonctions qui pourront être utilisées sans avoir à les réécrire. Ces fonctions sont la plupart du temps regroupées par thèmes.

### 1.3.2 Ligne de produits

Une ligne de produits est un ensemble de produits mis à disposition par une entreprise pour répondre à un même besoin, ou ayant des caractéristiques communes. Elle permet également d'augmenter la productivité tout en diminuant le temps et donc le coût de réalisation du produit.

#### **Ligne de produits logiciels**

Ensemble de logiciels appartenant à un domaine particulier, ici la création de jeux d'arcades. En informatique une ligne de produits est également appelée un framework. Généralement, un framework est codé dans un langage objet, par conséquent il est composé d'une classe mère de laquelle découle plusieurs classes filles. Ainsi un programmeur doit réimplémenter les classes qui l'intéressent pour créer son logiciel.

Une ligne de produits logiciels est une surcouche des bibliothèques et permet donc non

seulement de pouvoir réutiliser du code mais aussi et surtout de donner une architecture précise aux logiciels qui l'utilisent. De plus un framework doit être nécessairement extensible pour s'adapter à un plus grand nombre de logiciels.

### 1.3.3 Jeux d'Arcades

Les jeux d'arcades sont principalement des jeux à deux dimensions. La jouabilité est très simple ce qui rend ce type de jeux très populaires. Il n'y a généralement pas d'intelligence artificielle (ou très peu évoluée) et encore moins de parties réseaux (les joueurs s'affrontent sur la même machine).

A l'origine ce type de jeux à été créé pour les salles de jeux ou certains bars, c'est pour cela que le niveau du jeu est exponentiel, afin que le joueur remette en permanence de l'argent pour faire vivre son personnage. Ceci explique aussi le fait qu'il n'y ait pas de sauvegardes des parties.

Quelques exemples de jeux bien connus : Pac-Man, Casse-Briques, Ping-pong...

## 1.4 Outils utilisés

### 1.4.1 Choix du langage

Ce projet impose de pouvoir utiliser cette ligne de produits pour créer nos propres jeux d'arcades. Ces jeux doivent être jouables sur Internet. De nos jours il n'existe pas beaucoup de langages de programmation permettant cela. Les deux plus répandus sont JavaScript et Flex. Flex étant un tout nouveau langage développé par Adobe, il nous à paru intéressant de le découvrir et d'exploiter sa richesse plutôt que de réutiliser un langage connu de tous : JavaScript.

### ActionScript

L'ActionScript est le langage propriétaire d'Adobe, développé dans le but de contrer JavaScript. La syntaxe est très proche de JavaScript et permet de faire des applications orientées objet.

### Flex

Ce nouveau langage basé sur ActionScript permet de créer des clients internet riches avec une syntaxe à balises. Ainsi les scripts écrits en Flex sont transformés en ActionScript puis compilés en SWF (format propriétaire d'Adobe également appelé Flash). Ainsi tout navigateur équipé de Flash peut lire les applications Flex. Nous avons donc créé notre framework en ActionScript, car c'est un langage objet, actuellement en version 3 (donc stable et évolué). Cela laisse donc le choix aux utilisateurs de notre framework de réimplémenter nos classes en ActionScript ou de développer directement leurs applications en Flex. L'inconvénient de ce choix est que toute la technologie est propriétaire. Il faudra donc que l'utilisateur final ait tous les outils nécessaires à la programmation et surtout compilation de Flex.

### 1.4.2 IDE

Adobe a fait le choix de ne pas développer d'IDE spécialisé pour son langage, préférant utiliser Eclipse par le biais d'un plugin. Eclipse est un IDE très populaire car libre de droit et très performant puisqu'en version 3. Nous ne sommes donc pas perdus puisque nous avons l'habitude de développer avec cet IDE.

### 1.4.3 Google Code

Pour pouvoir travailler en groupe nous avons eut besoin d'un site internet pour regrouper nos codes et nos idées. Google Code est une plate-forme nous offrant un Wiki pour échanger nos idées, un serveur SVN pour partager nos sources, un espace de stockage de fichiers et enfin un contrôleur de bugs afin d'être prévenu d'éventuels problèmes. De plus il est entièrement gratuit et est hébergé chez Google, ce qui nous assure une plus grande sérénité quand à la sauvegarde de nos données.

### 1.4.4 SVN

Puisque Google nous offre un serveur SVN nous l'utilisons pour garder la même version pour chacun des développeurs. Sous Linux nous utilisons le client RapidSvn pour sa simplicité (un tutorial sur son utilisation est disponible sur le site du projet). Sous Windows nous utilisons Tortoise pour les mêmes raisons.

## 1.5 Modélisation UML

Nous utilisons le logiciel BOUML pour modéliser l'architecture du framework car il est simple d'utilisation, multi-plateforme et gratuit.

## 1.6 Organisation

### 1.6.1 Répartition des tâches

La répartition des tâches s'est effectuée selon les préférences de chacun.

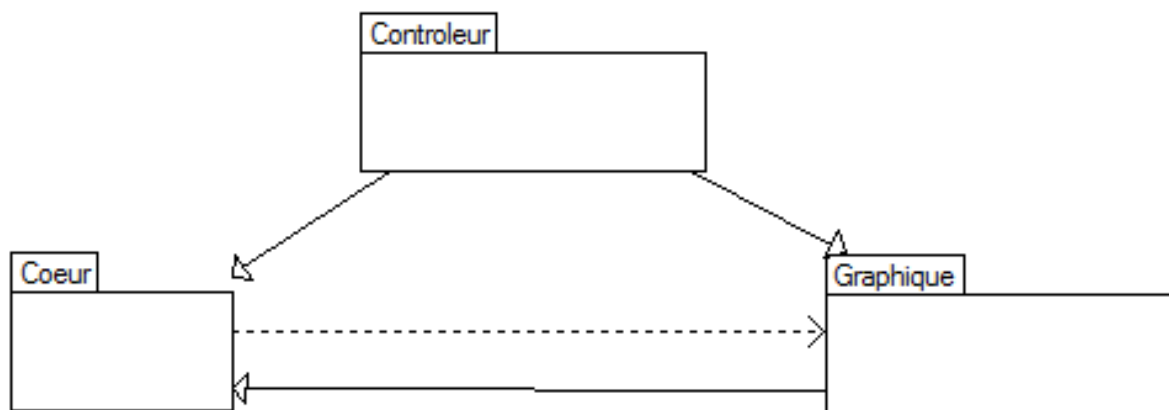
- Maneschi Romain (chef de groupe) : Partie graphique
- Maillet Laurent : Code métier
- König Mélanie : Contrôleurs, effets et collisions
- Novak Audrey : Créateur de jeux
- Fhal Jonathan : Création de jeux utilisant le framework

## 2 Code métier

La partie la plus importante de tout framework -et même de toute application fonctionnelle- réside dans la qualité d'implémentation du code métier. Plus le code métier d'un framework sera à la fois facile de compréhension et d'accès, réutilisable tout en restant léger et puissant, plus le framework sera rapide et offrira de nombreuses possibilités. C'est pourquoi nous avons décidé de consacrer la plus grosse partie de notre travail sur le coeur de notre framework.

Après une analyse attentive du problème que pose la réalisation d'un framework de jeu, nous avons rapidement pu construire une représentation interne du framework et ainsi décider des designs pattern qui seraient les mieux adaptés à sa réalisation.

### 2.1 Le pattern MVC



Modèle Vue Contrôleur

Avant toute chose, il nous a paru évident de bien différencier les représentations que l'on devait faire de chaque objet de notre framework. En effet dans un jeu, un objet possède une représentation graphique ainsi qu'une représentation interne (les données de l'objet en question). De plus il peut être nécessaire de vouloir qu'une action dirigée par un utilisateur permette d'avoir des effets sur notre objet, tant bien au niveau graphique, qu'au niveau des données de l'objet. Il nous a alors paru évident que nous devions nous inspirer du modèle MVC pour réaliser notre framework.

Pour tout objet du framework il a donc été nécessaire de représenter en deux parties différentes tout objet : d'un côté le graphisme (voir le package Graphique) et de l'autre les données (voir le package Coeur). Pour permettre de relier les deux parties d'un objet, il a fallu installer le graphisme en position d'écouteur du coeur. Ainsi lorsque les données du coeur d'un objet sont modifiées, le graphisme en est immédiatement informé et peut réagir en conséquence.

Les contrôleurs, qu'ils soient de type effet, souris ou clavier, agissent sur un objet du code métier en modifiant les propriétés de ceux-ci. Ces modifications seront répercutées sur le graphisme puisqu'il est écouteur de la partie métier.

## 2.2 Objets

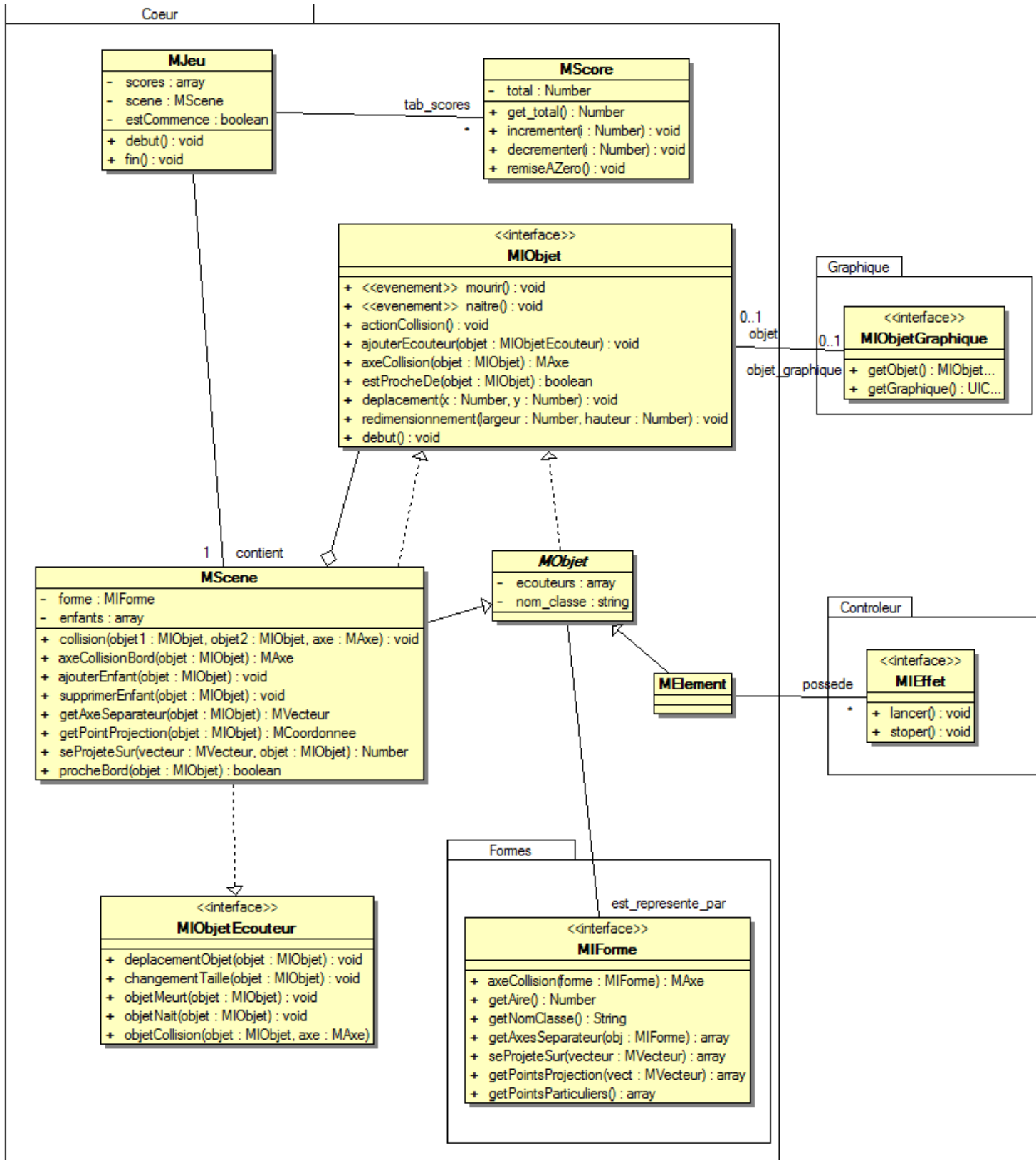


Diagramme de classe du coeur



L'architecture la plus logique pour les objets aurait été une interface avec toutes les fonction nécessaire dans nos objets et une classe abstraite implémentant cette interface en redéfinissant les fonctions communes à tous les objets. Malheureusement, nous nous sommes heurté à la première limitation de notre langage, les classes abstraites ne sont pas implémentables en Actionscript 3. Nous avons créé une interface MIObjet représentant l'ensemble des fonctionnalités nécessaires à la bonne implémentation d'un objet. A partir de cela nous avons créé une classe "abstraite" MObjet, implémentant la plupart des fonctions demandées par l'interface. Nous avons pensé que certaines fonctions sont génériques à tous les objets -par exemple changer les coordonnées de l'objet- alors que d'autres nécessitent d'être forcément spécifiées. A la construction d'un MObjet nous avons forcé le fait que les utilisateurs du framework doivent implémenter MIObjet. Ainsi grâce à cette implémentation, nous forçons les utilisateurs qui veulent étendre la classe MObjet à implémenter MIObjet, ce qui permet de ne pas nuire au bon fonctionnement de notre framework.

Deux types d'objets nous ont paru essentiels pour créer la base de n'importe quel jeu : les scènes et les éléments. A partir de la hiérarchie d'objets de base, nous avons donc créé ces deux nouveaux types d'objets.

Une scène est un objet sur lequel l'utilisateur n'est pas censé pouvoir effectuer d'action de base comme le déplacement (cela reste possible cependant mais pas forcément intéressant), mais c'est un objet pouvant contenir d'autres objets afin qu'en son intérieur se déroulent des actions (comme des déplacements ou des collisions). Nous avons mis en place un "composit pattern" : une scène est un objet contenant d'autres objets.

Un élément est un objet de base auquel on rajoute une liste d'objets qui l'écoutent et une liste d'effets. Les effets s'appliquent à tout moment sur un élément et permettent de lui donner vie (un déplacement perpétuel, changement de taille, etc.). Les éléments possèdent une liste d'effets afin de pouvoir combiner les différents effets possibles (agrandir l'élément tout en le déplaçant par exemple).

Les objets qui écoutent les éléments doivent forcément être des objets implémentant l'interface MIObjetEcouteur. C'est grâce aux méthodes de cette interface que sont faites les liaisons entre les éléments et les écouteurs. En effet lors de certaines actions spécifiques telles que le déplacement, un élément va prévenir les objets qui l'écoutent qu'il s'est déplacé afin que ceux-ci puissent agir en conséquence.

Une scène est donc forcément un objet écouteur car elle doit savoir à tout moment ce qu'il se passe en elle : des objets se sont-ils rencontrés ? un objet a-t-il touché le bord de la scène ? Le joueur a-t-il fini de casser toutes les briques de son casse-brique ? Grâce à notre hiérarchie il devient facile de pouvoir répondre à toutes ces interrogations : la scène, lorsqu'elle reçoit un appel d'un des objets qu'elle contient, n'a qu'à agir en fonction.

## 2.3 Eléments

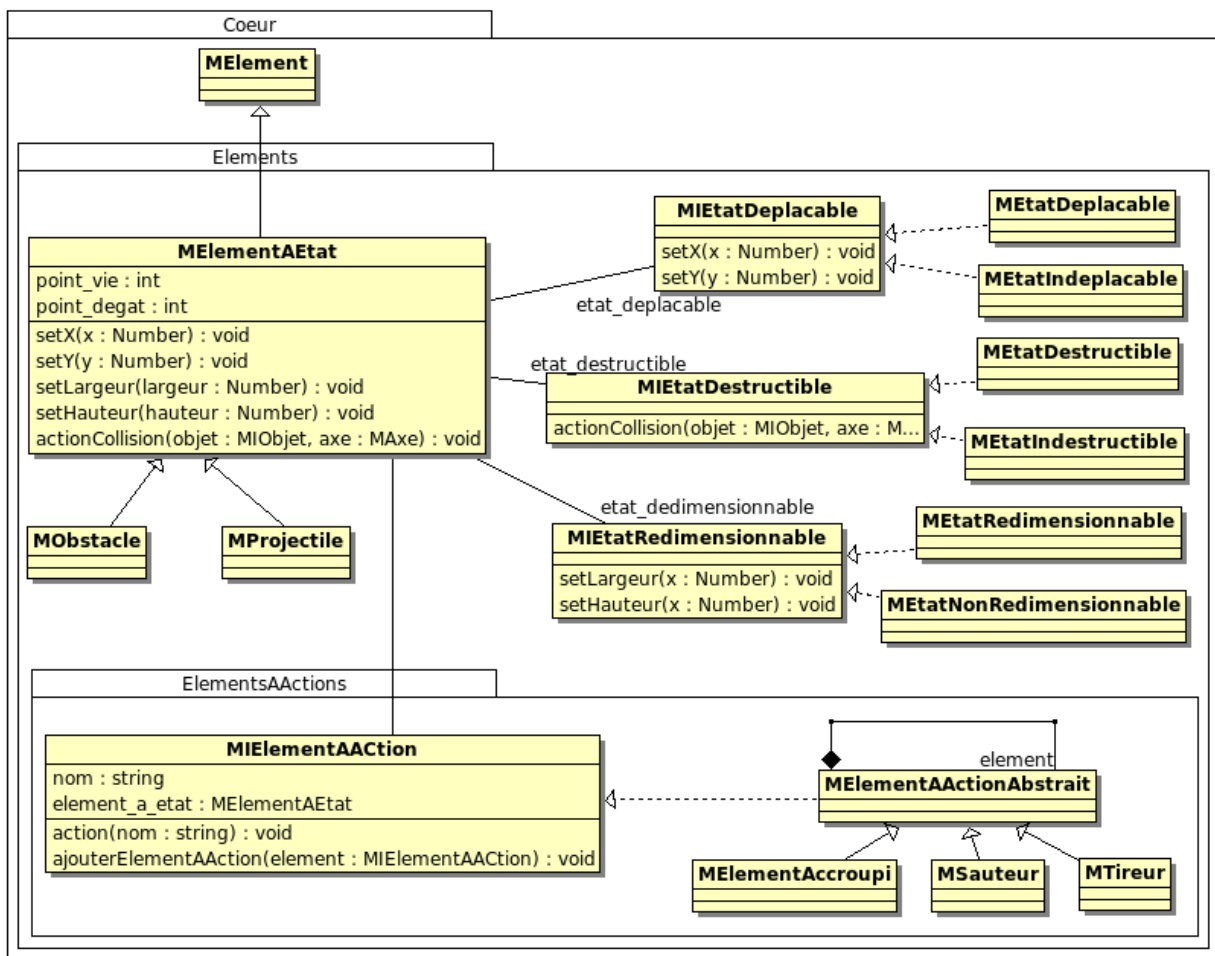
Après avoir réfléchi sur l'élément de base, un élément dont on peut positionner ses coordonnées, sa taille ou encore le fait qu'il puisse mourir, nous nous sommes intéressés à ce que pourrait vouloir faire un programmeur de jeux d'arcade. Avec notre élément de base et nos contrôleurs, il a déjà un large panel de possibilités et il peut quasiment tout faire mais notre but étant de lui faciliter la tâche au maximum, nous avons décidé d'ajouter plusieurs sous-éléments.

### 2.3.1 Patron de conception Etat

Un programmeur pourrait vouloir positionner ses objets en tant qu'objet déplaçable ou non, redimensionnable ou non et enfin et surtout destructible ou pas. Pour faire cela nous avons décidé de mettre en place le patron de conception Etat. Certes tous ces états n'ont que deux possibilités, mais il y a plusieurs raisons à ce choix :

- le programmeur peut vouloir changer d'état à tout moment
- un simple booléen aurait pu suffir mais les possibilités auraient alors été réduites
- cela permet de soulager la classe principale en déléguant du code à la classe représentant l'état

Pour toutes ces raisons nous avons choisi d'implémenter le patron de conception état.



La classe qui permettra tout ce processus est la classe **MElementAEtat** afin de ne pas toucher au code de la classe **MElement** qui est implémentée par un autre programmeur du

groupe. De plus elle contient un attribut de chaque type : `MIEtatDeplacable`, `MIEtatDestructible` et `MIEtatRedimensionnable` qui représentent : les états pour le déplacement, la destruction ou encore le redimensionnement. Ainsi le programmeur peut facilement changer les caractéristiques de son élément en positionnant ses états en fonction par exemple d'objet bonus trouvés dans la scène.

Ensuite nous nous sommes demandés quels types d'éléments nous pouvions construire à partir de ces états. Tout naturellement, nous avons immédiatement pensé aux obstacles implémentés par `MObstacle` qui est un élément indestructible et indéplaçable. Or, comme par définition il est indéplaçable, le programmeur ne pourrait plus le repositionner après l'avoir positionné une première fois sur la scène, afin d'éviter cela, nous le rendons indéplaçable seulement après avoir fait appel à la fonction `MJeu.getInstance.debut()`. Ainsi si le jeu a commencé alors l'obstacle n'est plus déplaçable.

Puisque nous avons à notre disposition différents types de contrôles nous avons prévu une implémentation de base de ceux-ci :

- par le clavier, c'est la première façon à laquelle on pense en tant que joueur. Ainsi nous avons implémenté la classe `MControleClavier` qui est un `MElementAEtat` implémentant `MIecouteurClavier` avec les touches de base que tout le monde à l'habitude d'utiliser pour jouer : les flèches pour se diriger.
- par la souris, qui sert donc à diriger l'objet.
- enfin un programmeur peut vouloir qu'un élément soit contrôlé par le PC, autrement dit que celui-ci soit un ennemi du joueur. Pour cela nous avons mis en place un `MMouvementPerpetuel` dans le `MElementAEtat`. Enfin nous avons pensé aux projectiles, qui sont des éléments déplaçables et destructibles. Mais dès lors il faut pouvoir "tirer" ces projectiles de façon la plus simple possible.

### 2.3.2 Actions

L'étape suivante a été ensuite plus directement ciblée vers les jeux d'arcade de type "shoot". En effet, nous nous sommes rendus compte que nous pouvions accélérer considérablement le développement d'une telle application en créant et en associant un `MProjectile` à un "MTireur". Le problème étant qu'un élément qui est à état, contrôlé par le clavier, la souris ou encore un mouvement doit pouvoir tirer. Nous nous sommes donc dit que le fait de tirer doit venir "décorer" nos éléments. Ainsi nous avons mis en place le patron de conception "decorateur". Pour cela, une simple interface pour définir une action est requise, ensuite chaque action prend en paramètre l'élément qu'elle décore. Ce pattern nous a ensuite permis d'imaginer ce qu'un développeur pourrait vouloir comme action, l'action de sauter est venue très rapidement ainsi que celle de s'accroupir. Nous aurions pu en implémenter plus mais le niveau actuel du graphique ne permettrait pas de les représenter. Mais notre framework étant extensible, elles seront facilement ajoutables ultérieurement.

## 2.4 Formes

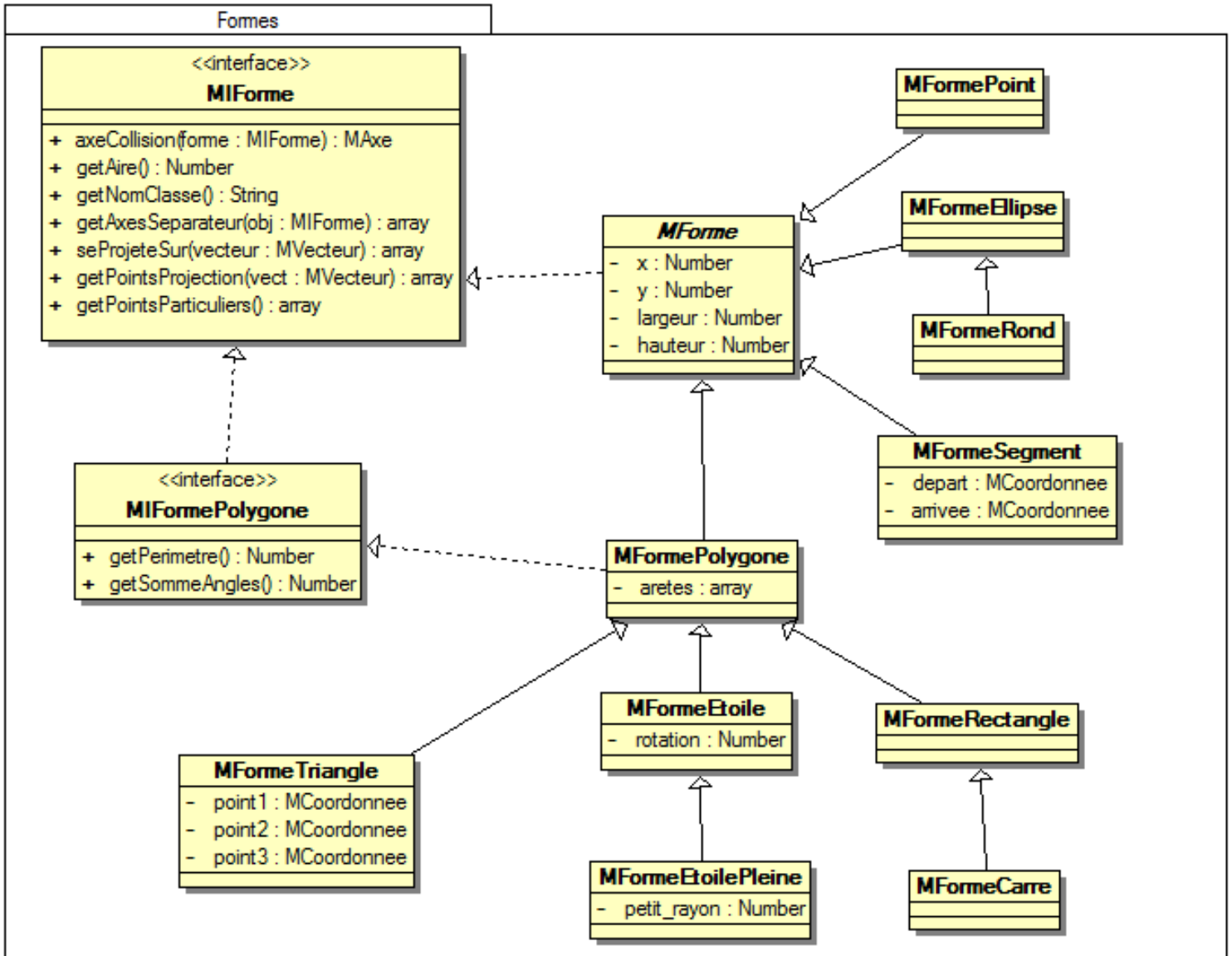


Diagramme de classe des formes

La réalisation des formes a été une partie importante de la réalisation des objets. Il est évident de penser que tout objet possède sa propre forme et que l'on voudra visuellement obtenir des rendus différents selon que l'on décide de créer des murs ou des balles. Dans le cadre de la réalisation de notre framework en suivant le pattern MVC, il a été obligatoire de procéder à deux types de représentations pour les formes : une graphique (présenté plus loin) et une dans le code métier. La partie graphique s'occupera uniquement du rendu à l'écran de la forme, alors que le coeur lui s'emploiera à conserver toutes les données de la forme.

Lors de la réalisation des formes, nous avons rencontré le même problème que pour les objets : nous voulions créer une classe abstraite **MForme** abstraite reprenant toutes les fonctions et tous les attributs de base nécessaire à la création cohérente d'une forme. Nous avons du finalement opter pour la même solution que précédemment : une interface **MIForme** et une classe **MForme** reprenant toutes les fonctions de bases génériques à toutes les formes.

On a distingué deux types de formes : les formes simples et les polygones. Toutes les formes sont caractérisées par des coordonnées  $x$  et  $y$ , ainsi que par une largeur et une hauteur. Ces quatre variables forment un rectangle dans lequel sera contenu la totalité de la forme, que ce soit une forme simple ou un polygone.

La différence entre les formes simples et les polygones réside dans le fait que les formes polygones ont besoin également d'un ensemble d'arêtes pour pouvoir être représentées. S'il est possible de représenter un triangle uniquement par des points, et de représenter un carré comme une forme simple, il devient beaucoup plus dur de représenter un polygone à  $n$  côtés uniquement par des points, ainsi que de représenter un carré pouvant tourner sur lui même uniquement à l'aide des quatre variables de base. Nous avons donc introduit des arêtes pour tout polygone ce qui permet de les dessiner et les représenter plus facilement.

Nous avons décidé de créer directement la plupart des formes de base ceci pour permettre à l'utilisateur de pouvoir créer des éléments directement sans avoir à tout implémenter. Ainsi s'il veut créer une balle il n'aura qu'à choisir une forme ronde représentée par la classe `MFormeRonde`.

Enfin c'est dans les formes qu'une des plus dures parties du coeur a été géré : les collisions. En effet la collision s'effectue selon la forme d'un élément, comme nous allons vous l'expliquer maintenant.

## 2.5 Collision

La détection de collision est l'un des plus importants aspects des jeux d'arcade, malheureusement elle est aussi souvent l'un de ses points faibles.

Quand il y a  $n$  objets distincts dans la scène, les tests doivent se faire entre  $O(n^2)$  différentes paires d'objets, c'est-à-dire que si nous avons 10 objets, il faudra faire  $10 * 10 = 100$  tests. Nous avons donc utilisé une approche en deux phases pour réduire le coût de ces tests :

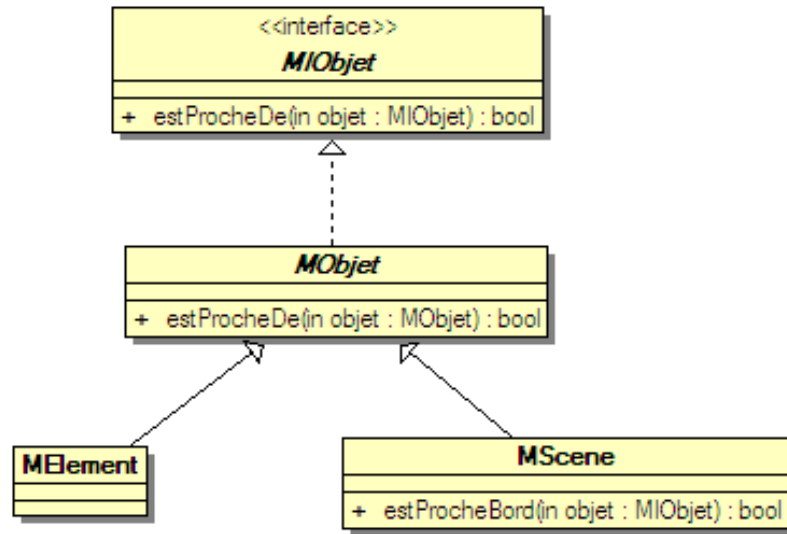
- La recherche de proximité (broad phase) permet d'éliminer la plupart des paires en utilisant un simple test sur les enveloppes rectangulaires (bounding boxes) des éléments de notre scène. Les paires d'éléments qui sont déclarées "proches" par cette sélection passent alors à la phase suivante.
- Le test de collision de proximité (narrow phase) utilise ensuite un algorithme plus fin pour détecter les collisions et donner les informations nécessaires à la gestion de la collision.

Ce système permet donc de limiter les tests de collision les plus coûteux aux paires d'éléments qui sont réellement proches l'un de l'autre.

### 2.5.1 recherche de proximité

La recherche de proximité entre deux objets contenus dans une scène se fait dans la fonction `estProcheDe(objet : MIObjet)` par un simple test sur les coordonnées et la largeur/hauteur des éléments pour savoir si leurs enveloppes rectangulaires se touchent.

La recherche de proximité entre la scène et un des éléments qu'elle contient se fait dans la fonction `estProcheBord(objet : MIObjet)` où il est vérifié pour chaque côté de la scène si l'enveloppe rectangulaire de l'objet le touche.



Recherche de Proximité

### 2.5.2 Le test de collision de proximité

Nous avons choisi de réaliser le test de collision de proximité avec la méthode des axes séparateurs. Cette méthode s'appuie sur le théorème de l'axe séparateur :

**théorème 1.** *Pour deux polyèdres convexes  $P$  et  $P'$ , disjoints, il existe un axe  $A$  sur lequel la projection des polyèdres  $P$  et  $P'$  forme deux intervalles disjoints.*

*Un tel axe est appelé axe séparateur.*

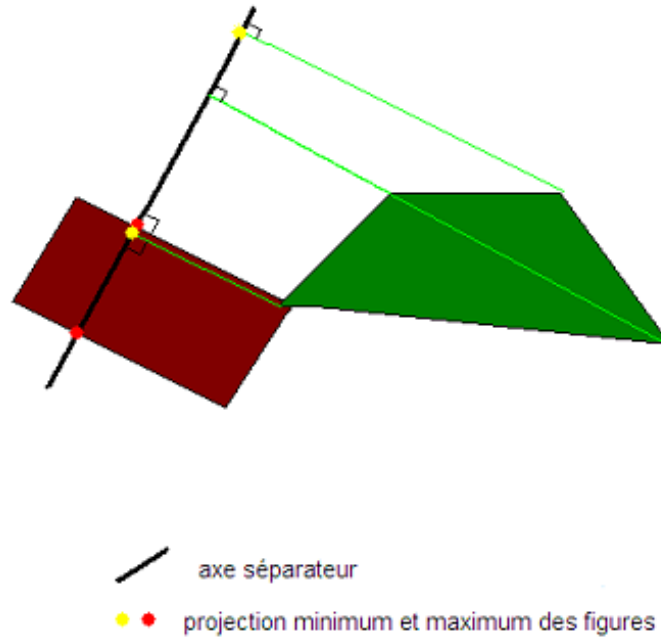
*Si les polyèdres sont disjoints, il existe un axe séparateur orthogonal à :*

- une face de  $P$ ,
- une face de  $P'$ ,
- une arête de chacun des deux polyèdres.

La méthode des axes séparateur consiste donc à :

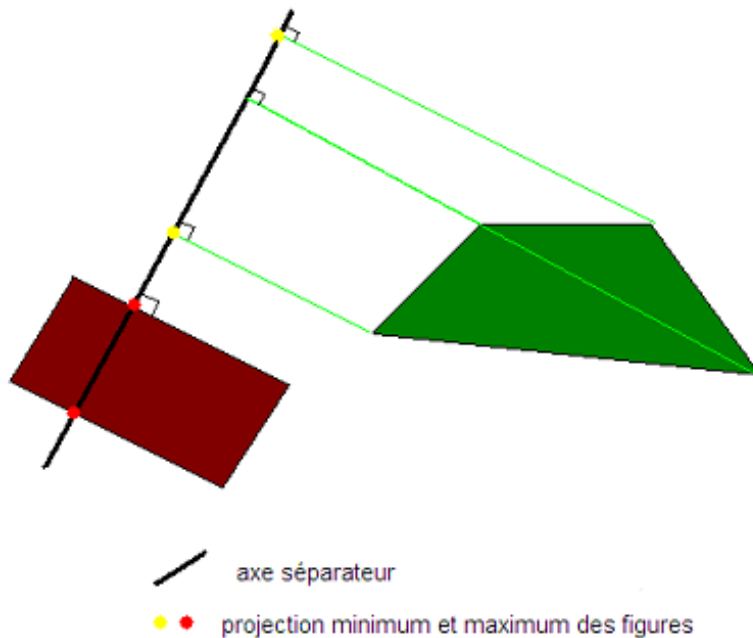
- déterminer tous les axes séparateurs potentiels entre les deux figures
- vérifier s'il en existe un sur lequel les projections des deux figures sont disjointes
- si un tel axe n'existe pas signaler la collision et renvoyer l'axe selon lequel la collision est arrivée

Voici une illustration de deux formes convexes en collision, on remarque bien que les projections des deux formes se chevauchent :



Exemple de polygones en collision

Si en revanche les deux formes ne sont pas en collision, les deux projections sont disjointes :



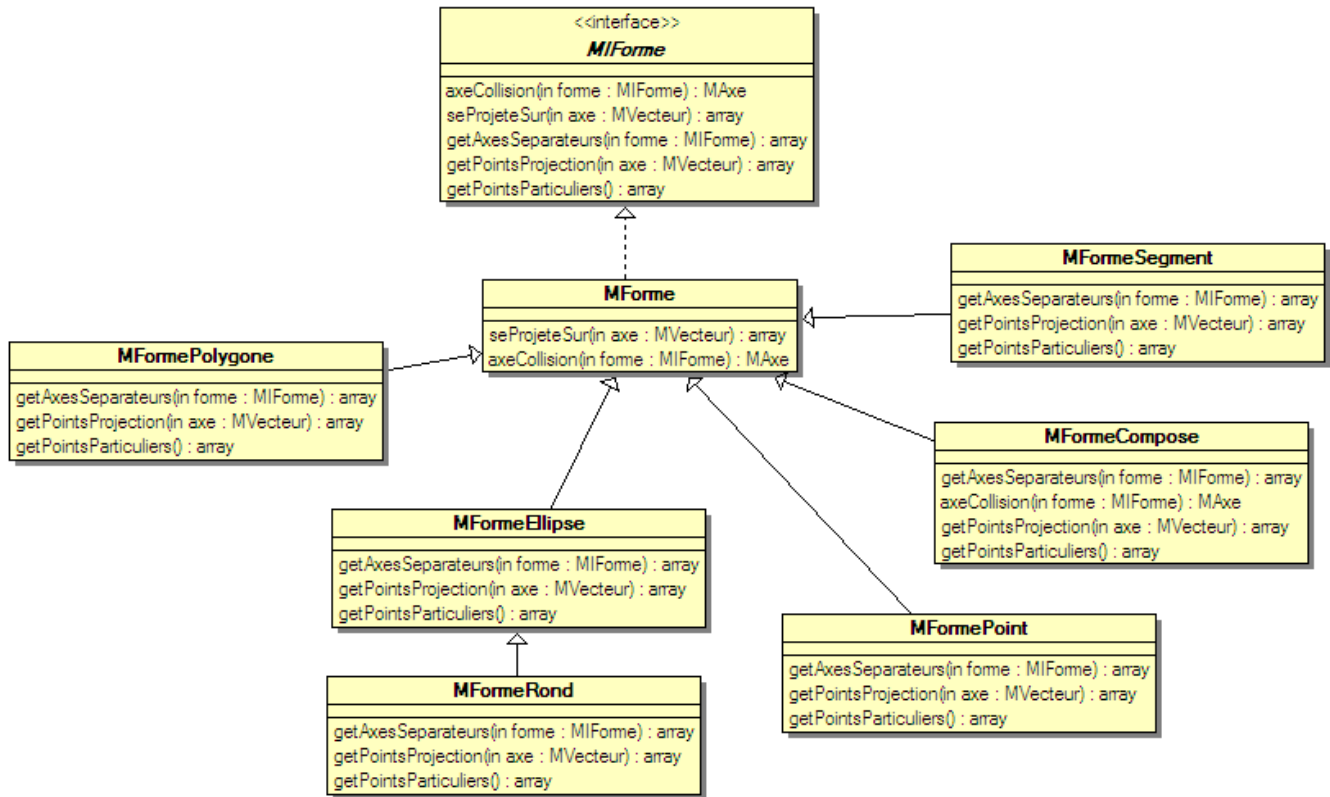
Exemple de polygones sans collision

Pour notre projet, c'est la fonction *axeCollision(forme : MForme)* implémentée dans *MForme* qui fait cela. Elle récupère les axes séparateurs des deux formes grâce à la méthode *getAxesSeparateurs(forme : MForme)* et demande à chaque forme de se projeter dessus grâce à la fonction *seProjeteSur(vecteur : MVecteur)*. Cette fonction récupère les points à projeter avec

la fonction *getPointsProjection(axe :MVecteur)* et retourne les valeurs minimales et maximales des deux projections.

Si les deux projections sont disjointes, l'axe est séparateur et donc la fonction s'arrête là et renvoie null.

S'il n'y a pas d'axe séparateur alors la fonction retourne l'axe sur lequel le chevauchement des projections était minimal, c'est selon cet axe que les deux formes se sont percutées.



Test de collision de proximité



A cette méthode générale s'ajoute un certain nombre de cas particuliers :

1. fonction *getPointsProjection(axe :MVecteur)* :

**Les polygones :** Les points à projeter sont évidemment les sommets qui composent le polygone.

**Les formes ronds :** Les points à projeter pour les formes rondes sont les points d'intersection entre le rond et l'axe séparateur.

**Les formes composées :** L'ensemble des points à projeter pour une forme composée est l'union des points à projeter pour chacune des formes qui la compose.

2. fonction *getAxesSeparateurs(forme :MIForme)* :

**Les formes polygones et les formes Segments :** Les axes séparateurs potentiels pour les polygones sont comme l'indique le théorème les axes qui sont perpendiculaires à ses arêtes. Pour le segment, c'est aussi l'axe qui lui est perpendiculaire.

**Les formes ronds et les formes points :** Les formes ronds et les formes points n'ayant pas d'arêtes, les axes séparateurs ne peuvent être les perpendiculaires aux arêtes. Heureusement, les axes potentiels pour les formes ronds et les points sont les droites qui relient les points retournés par sa méthode *getPointsParticuliers()* à ceux de l'autre forme. Pour les ronds les points particuliers sont le centre, pour un point, c'est lui même, pour les polygones, ce sont ses sommets et pour une forme composée l'ensemble de ses points particuliers est l'union des points particuliers des formes qui la compose. Donc par exemple, les axes séparateurs potentiels entre un rond et un triangle sont les axes qui relient le centre du cercle à chacun des sommets du triangle.

**Les formes composées :** L'ensemble des axes séparateurs pour une forme composée est l'union des axes séparateurs de chacune des formes qui la compose.

3. fonction *axeCollision(forme :MIForme)* :

**Les formes composées :** Pour détecter la collision dans une forme composée, il faut détecter la collision dans chacune des formes qui la compose. Si aucune d'elles n'est en collision alors la forme composée n'est pas en collision

## 2.6 Le jeu

Une fois les éléments du jeu mis en place il peut être intéressant de pouvoir les lancer et les stopper à bon escient. Notre framework possède une classe *MJeu* -qui est un singleton- permettant de lancer et de stopper les mouvements d'une scène quand celle ci le décide.

En effet à tout moment une scène peut récupérer l'instance du jeu et se placer en tant que scène principale de celui-ci. Elle peut donc demander quand elle le souhaite de lancer le jeu ou de le stopper ce qui aura pour effet de lancer les mouvements des éléments dans la scène ou de les arrêter. Par exemple si l'on souhaite un jeu avec un compte à rebours, il suffit de lancer un timer et au bout de ce timer demander au jeu de s'arrêter et de stopper tout. Ou encore si l'on fait un casse briques, on peut vérifier à tout moment la présence de brique dans la scène et continuer ou arrêter le jeu selon.

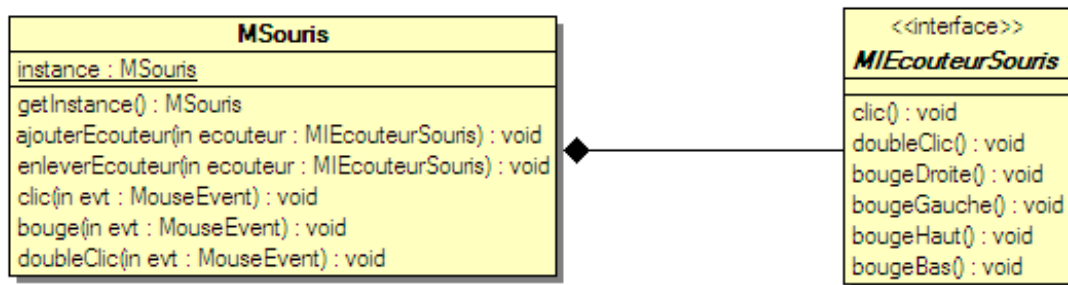
Pour finir le jeu permet en plus de compter un ou plusieurs scores, pratique pour les jeux multijoueurs.

## 3 Contrôleur

Le rôle du contrôleur dans le modèle MVC est de recevoir les événements de l'utilisateur et de lancer les actions qui en découlent. Pour les jeux d'arcades, les événements qui nous intéressent principalement sont les événements de la souris et les événements du clavier sur l'ensemble de l'application. Nous avons donc défini deux classes chargées de prévenir les objets écouteurs inscrits dans leur liste des événements de l'application.

### 3.1 Souris

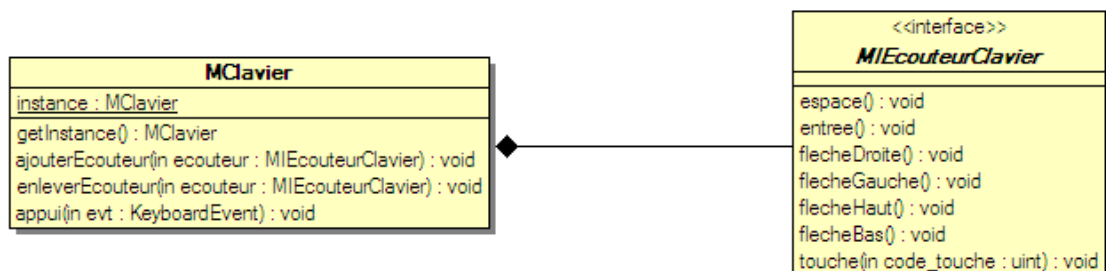
La classe MSouris est un singleton, elle s'enregistre auprès de l'application pour les événements de clic, de double-clic et de déplacement de la souris. Elle retransmet ensuite à ses écouteurs les actions de clic, de double-clic et de déplacement vers le haut, bas, gauche, droite de la souris grâce à un calcul sur les coordonnées de l'événement de déplacement. Les MIEcouteurSouris ont donc pour chaque événements une fonction qui contient le code à effectuer lors de l'événement et que MSouris appelle quand il faut.



La gestion des événements souris de l'application

### 3.2 Clavier

La classe MClavier est aussi un singleton, elle s'enregistre auprès de l'application pour les événements d'appui de touche. Dans la fonction appui elle regarde quelle touche est appuyée pour pouvoir appeler la fonction adéquate de ses écouteurs. Les touches flèche haut, bas, gauche, droite, espace et entrée ont une fonction particulière dans les écouteurs car ce sont les touches les plus utilisées pour les jeux, pour les autres touches, une fonction commune est appelée avec en paramètre le code numérique de la touche et c'est au programmeur de différencier les touches appuyées s'il en a besoin.



La gestion des événements claviers de l'application

### 3.3 Effets

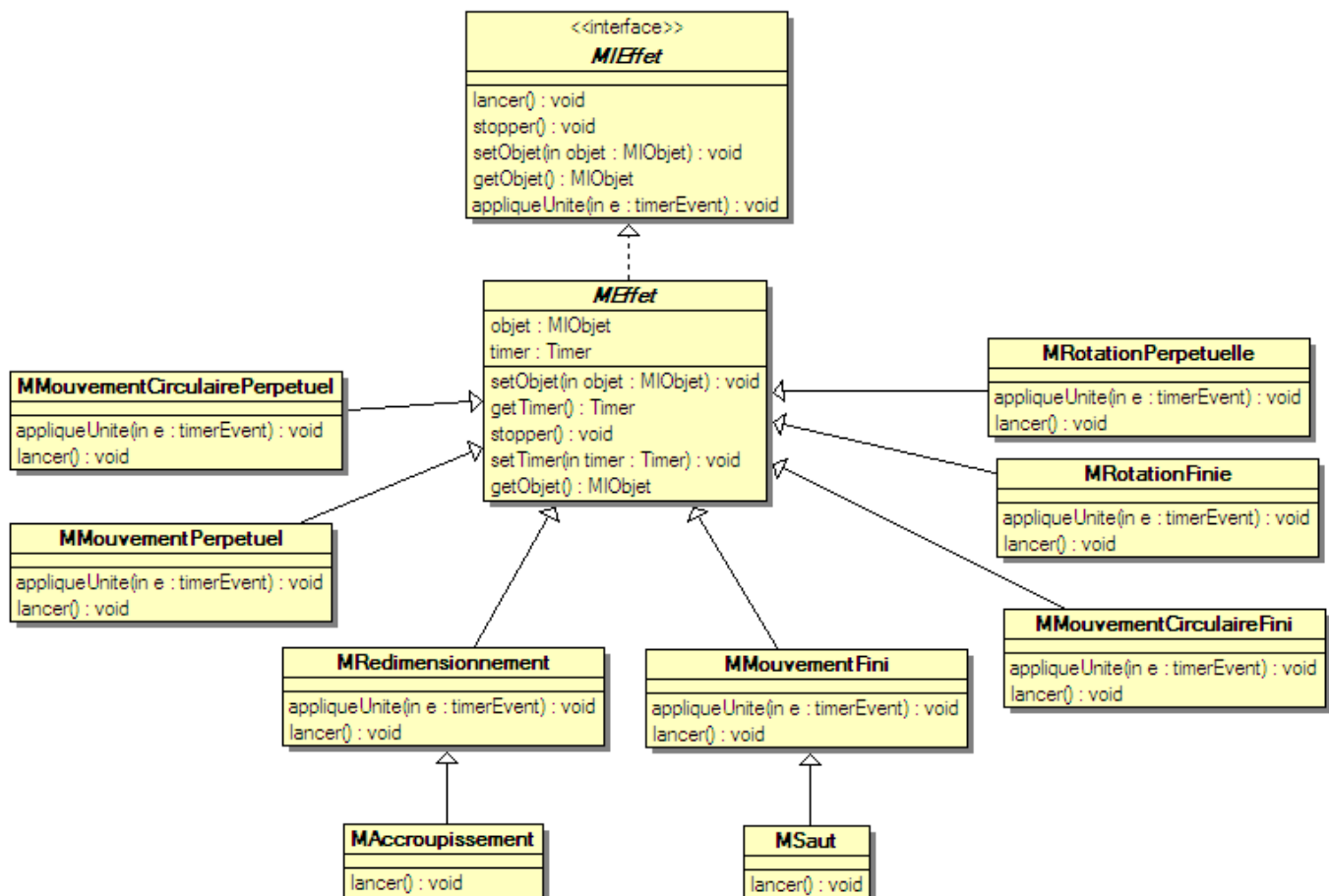
Les effets sont l'ensemble des traitements nécessaires pour créer l'illusion d'actions comme le mouvement, le changement de taille... Un effet s'applique sur un objet MIObjet, il utilise un timer pour fournir 50 changements par seconde et donner ainsi l'impression de continuité de l'effet.

Il y a deux types d'effets, les effets à effectuer sur un temps fini ou les effets à effectuer de manière infinie. Par exemple, un mouvement peut être un déplacement jusqu'à un point en un temps donné ou un mouvement selon une direction et une vitesse avec rebondissement lorsqu'un obstacle est rencontré.

Pour les effets finis, au lancement de l'effet, on calcule les modifications à faire sur l'objet toutes les unités de temps ( pour obtenir 50 images/seconde, l'unité de temps est de 20 ms) puis on lance le timer le nombre de fois qu'il faut pour finir l'effet.

Pour les effets infinis le principe est le même sauf que l'on lance le timer un nombre de fois infinis.

Voici notre hiérarchie d'effets :



Les effets

## 4 Graphique

La première chose que voit un joueur lorsqu’il joue à un jeu vidéo est le graphisme. Malgré que le plus gros travail ait été apporté au coeur de notre framework, nous avons voulu qu’un programmeur puisse créer simplement des objets qui sont graphiquement complexes. En effet, un jeu qui a un très bon gameplay et/ou un très bon concept ne sera pas utilisé, si celui-ci à un graphisme peu attrayant.

Puisque nous utilisons l’API flex d’Adobe pour ce framework, nous avons dû nous limiter à ce qu’il peut nous offrir. Et comme dans toute API graphique, celle d’adobe est très complète mais son utilisation et implémentation reste bien souvent assez mystérieuse.

### 4.1 Réutilisation des composants de flex

**Class**      public class Container  
**Inheritance**    Container → UIComponent → FlexSprite → Sprite → DisplayObjectContainer → InteractiveObject → DisplayObject → EventDispatcher → Object

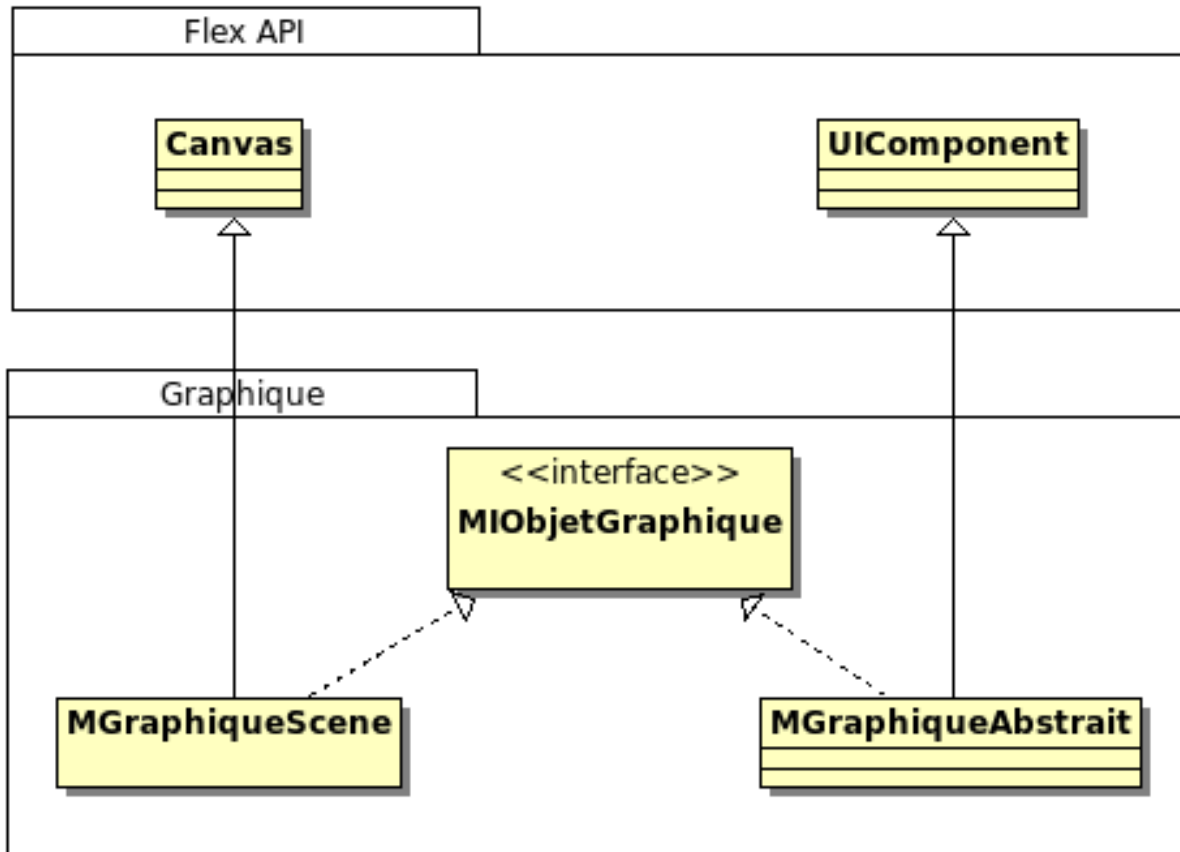
Héritage dans l’API 3.5 de flex

Lorsque nous avons fini de mettre en place notre code métier, il nous a fallu pouvoir afficher quelque chose. Pour cela nous avons essayé plusieurs composants qu’ils soient conteneur ou contenu. Puisque nous avons créé un framework entier et que notre code métier prend en compte tous les calculs, il nous a paru évident d’utiliser des composants de plus haut niveau possible, afin de ne pas surcharger inutilement notre framework avec des fonctions hérités de l’API qui deviendraient inutiles. Ainsi nous avons trouvé des composants de type DisplayObject qui sont les premiers composants affichables dans la hiérarchie de l’API de flex.

En outre nous voulons que nos utilisateurs puissent écrire leur code, non seulement en ActionScript3, mais aussi en mxml. Pour cela nous avons dû descendre un peu dans la hiérarchie pour aller chercher UIComponent. Cette classe est donc notre point d’entrée dans l’API d’adobe. Ce composant permet d’ajouter des UIComponent dans un UIComponent, ce qui est pratique pour notre Pattern Composite du code métier entre les MScene et les MElements. Malheureusement nous perdrons la programmation mxml car on ne peut pas ajouter d’enfants dans un UIComponent en mxml, nous sommes donc encore descendu dans la hiérarchie pour représenter nos MScene, et le conteneur Canvas est celui de plus haut niveau qui permet l’ajout d’un enfant en mxml. De plus, il nous permet de positionner nos objets en absolu.

Ensuite, le plus compliqué a été de comprendre comment fonctionne les UIComponent. Après avoir effectué de multiples recherches et encore plus de tests nous sommes finalement arrivés à cerner le fonctionnement interne de cette classe. En effet, à chaque fois qu’un UI-Componant doit être affiché à l’écran, la fonction updateDisplayList est appelée. Nous l’avons donc tout naturellement réimplémentée.

## 4.2 Dessiner en ActionScript



Hiérarchie entre notre graphisme et celui de Flex

Par le biais de l'héritage dans nos classes graphiques, et l'attribut `graphics`, qui est le point d'entrée pour tout ce qui concerne le dessin en ActionScript, nous pouvons dessiner à l'écran des rectangles, des ellipses mais aussi des lignes. De plus flash nous permet de colorier nos formes avec de la couleur et des dégradés. Enfin il nous a paru intéressant de pouvoir ajouter des images et des textes dans nos formes, nous devons utiliser d'autres classes mais le tout doit rester invisible pour le programmeur qui utilise notre framework.

Une hiérarchie peut donc être mise en place, nos différentes formes seront dessinées par les classes graphiques les représentants alors que les textures devront pouvoir être appliquées à toutes ces formes.

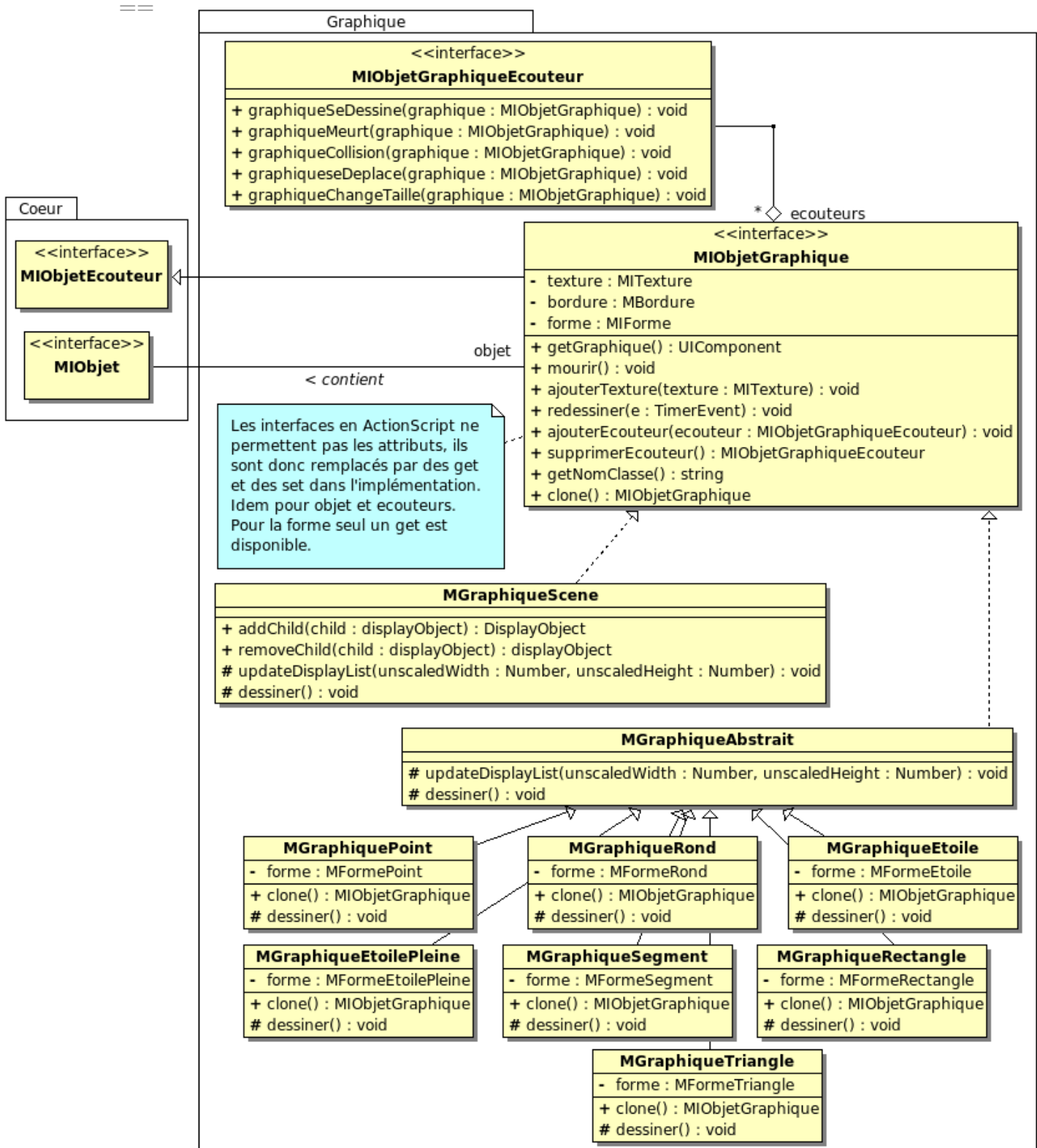


Diagramme UML de la partie graphique

### 4.3 Liaison avec le code métier

Comme nous pouvons le voir la classe `MGraphiqueAbstrait` est la classe abstraite qui s'occupe de la relation entre l'API flex, le code métier et le programmeur. En effet celle-ci permet de positionner l'objet au travers des fonctions `setX` et `setY`, mais aussi de positionner la taille de l'objet avec les fonctions `setWidth` et `setHeight`. Ces fonctions sont des réimplémentations des fonctions de la classe `UIComponent`, elles doivent donc se mettre en accord avec l'API flex, pour qu'il continue d'afficher ces objets mais aussi et surtout ces fonctions doivent mettre à jour notre modèle afin que la détection de la collision puisse continuer.

Ayant fait le choix d'utiliser le patron de conception Model-View-Contrôleur (plus communément appelé MVC) pour notre framework, il a tout naturellement fallu positionner la Vue (donc le graphisme) comme écouteur du modèle. Ainsi lorsque le modèle se déplace ou se redimensionne, le graphisme en est prévenu et l'image vue par l'utilisateur correspond au modèle de notre framework. Les fonctions citées plus haut ont eu juste à faire passer l'information au modèle, ainsi le graphisme se remet à jour puisqu'il est écouteur du modèle.

## 4.4 Textures et le pattern "decorator"

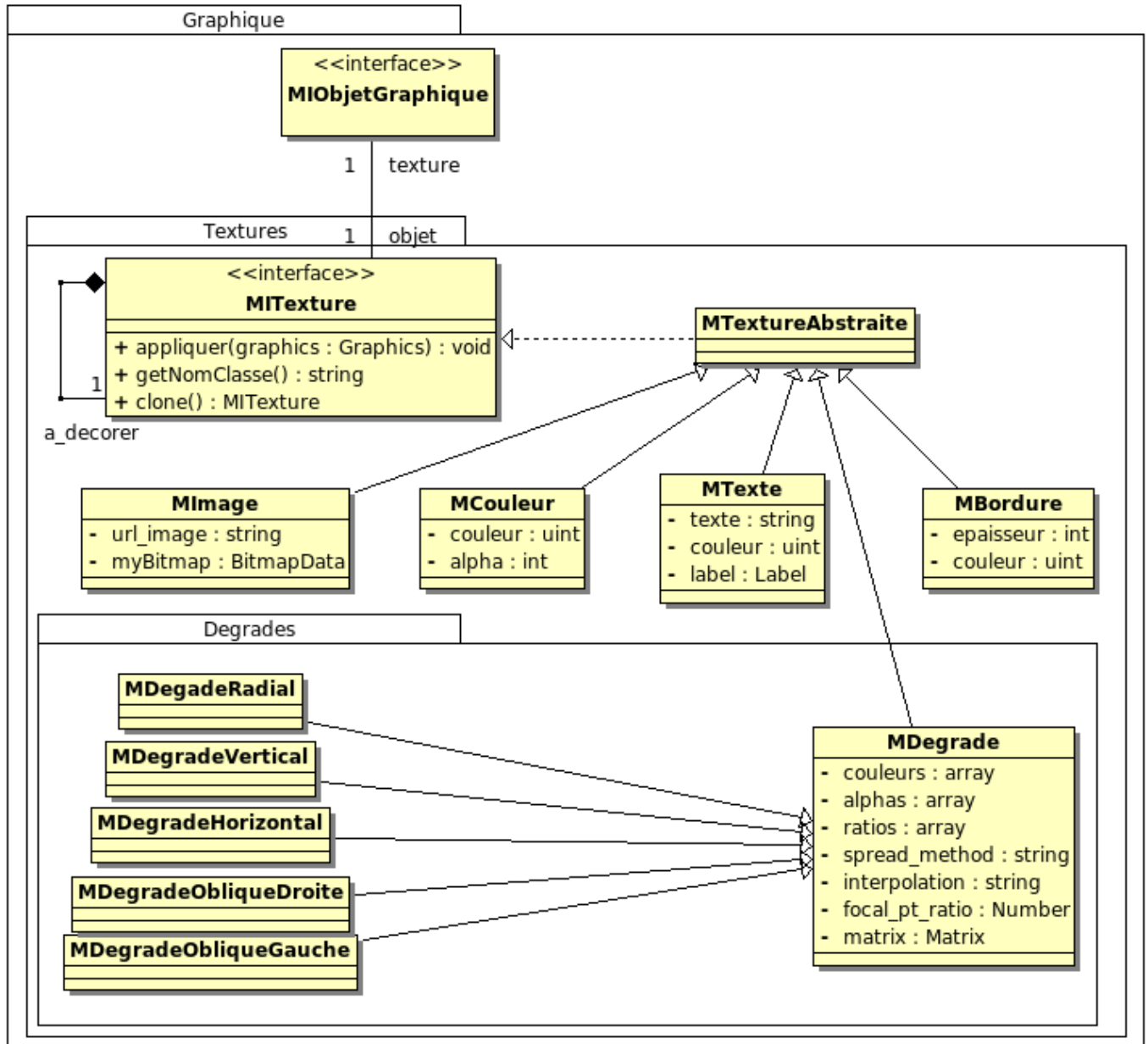


Diagramme UML des textures

Le dernier point de la partie graphique de notre framework est la texture d'un objet. En effet, une fois que nous avons réglé le problème de la forme il nous a fallu réfléchir au moyen de lui donner une texture. De base Flex nous permet de colorier une forme avec une couleur ou un dégradé. Pour plus de facilité nous permettons à l'utilisateur de créer des dégradés horizontaux, verticaux, radiaux mais aussi obliques. De plus nous souhaitons pouvoir mettre une image en tant que texture d'un objet. Pour cela nous avons dû charger l'image dans une matrix (pixel par pixel) avant de pouvoir dessiner cette matrix sur l'objet en question. Enfin la technique est à peu près similaire pour pouvoir poser un texte comme texture, mais cette fois, nous utilisons le fait qu'un **UIComponent** puisse contenir des enfants. Pour cela nous stockons le texte dans un **Label** avant de l'ajouter à l'objet. Malheureusement en agissant ainsi nous ne pouvons pas cacher le texte dépassant de la forme.



Enfin, pour que l'on puisse ajouter un texte ou une image par dessus une couleur ou un dégradé nous avons mis en place un patron de conception Décorateur. Ainsi nous pouvons ajouter à une forme autant de textures que l'utilisateur le souhaite. Puisque le flash gère très bien la transparence des couleurs, toutes les possibilités sont permises non seulement dans le code mais aussi et surtout à l'affichage.

## 4.5 Les écouteurs du graphisme

Après avoir commencé à programmer quelques jeux nous nous sommes rendus compte que le fait de devoir étendre systématiquement les classes graphiques pour pouvoir faire interagir les objets était trop contreignant. Partant de cela, nous avons décidé de placer des écouteurs également dans la partie graphique. Ainsi le programmeur qui souhaite pouvoir faire rebondir une balle, sans forcément réimplémenter la classe `MGraphiqueRond`, peut simplement l'écouter. Il sera alors prévenu lorsque la balle se déplace, se redimensionne ou encore lorsqu'elle collisionne. Comme vous pouvez le constater il s'agit d'un exemple et tous nos objets graphiques fonctionnent sur le même principe.

Ainsi, le programmeur a deux choix pour pouvoir agir sur ses objets graphiques. Le premier étant d'étendre l'une des classes graphiques, il lui faudra alors penser à appeler `super()` dans les fonctions de `MObjetGraphiqueEcouteur` pour appeler les `fire`. Le deuxième étant de simplement écouter un objet en ajoutant la classe qui implémente `MObjetGraphiqueEcouteur` dans les écouteurs de l'objet à surveiller.

## 5 Patron de conception Fabrique

Puisque le programmeur a à sa disposition tous ces éléments et qu'il va vouloir les représenter graphiquement par toutes nos formes, nous avons naturellement pensé à utiliser le patron de conception Fabrique.

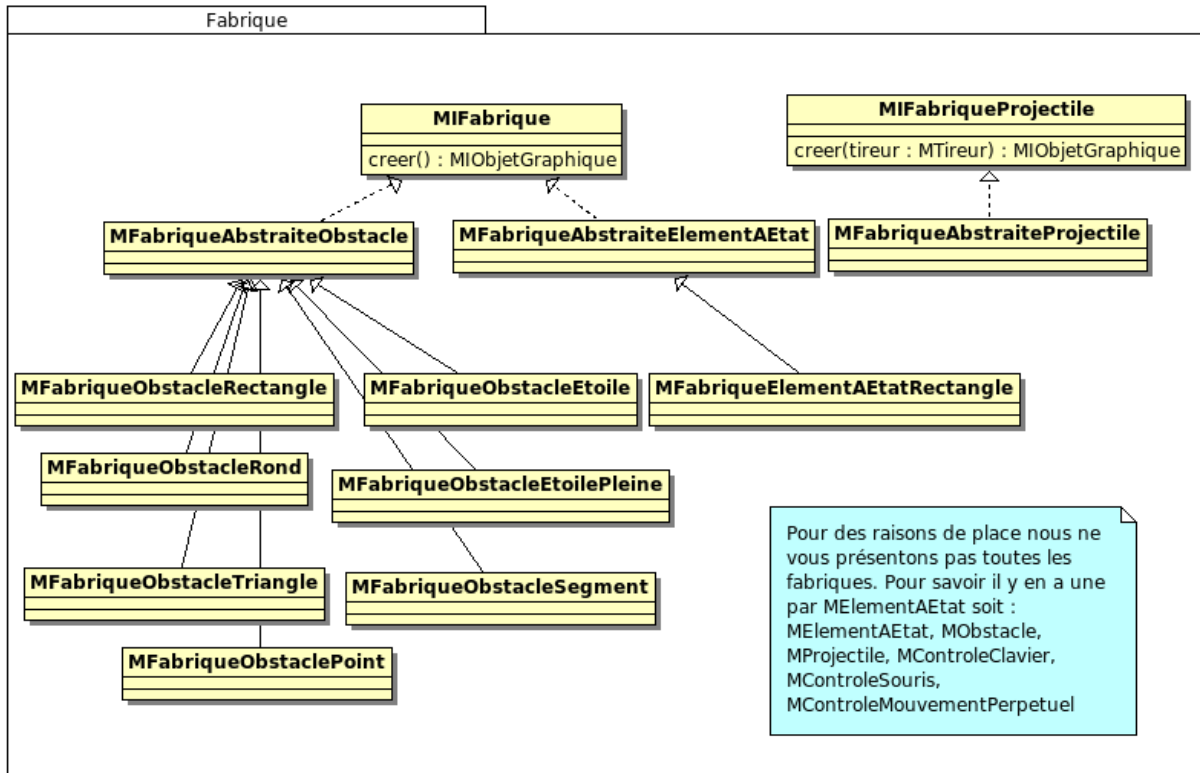


Diagramme UML de la fabrique

Comme vous pouvez le voir nous avons mis en place plusieurs fabriques pour que l'utilisateur puisse de façon transparente conjuguer les éléments avec les différentes formes. Grâce aux fabriques l'utilisateur dispose de plusieurs choix pour remplir ses scènes. En effet il peut le faire de manière "traditionnelle" en instanciant un `MIObjetGraphique` qui, de base crée un `MElement` (l'élément de plus haut niveau), puis en lui positionnant l'objet modèle qu'il souhaite (en appelant `setObjet(objet)`). Ou alors il peut faire appel à une de nos fabriques pour créer un `MProjectile` (qui est un `MElementAEtat`) rond par exemple. L'interface `MIFabrique` permet de rendre les classes `MFabriqueAbstraite[TypeElement]` abstraite, puisque action script 3 ne le permet pas.

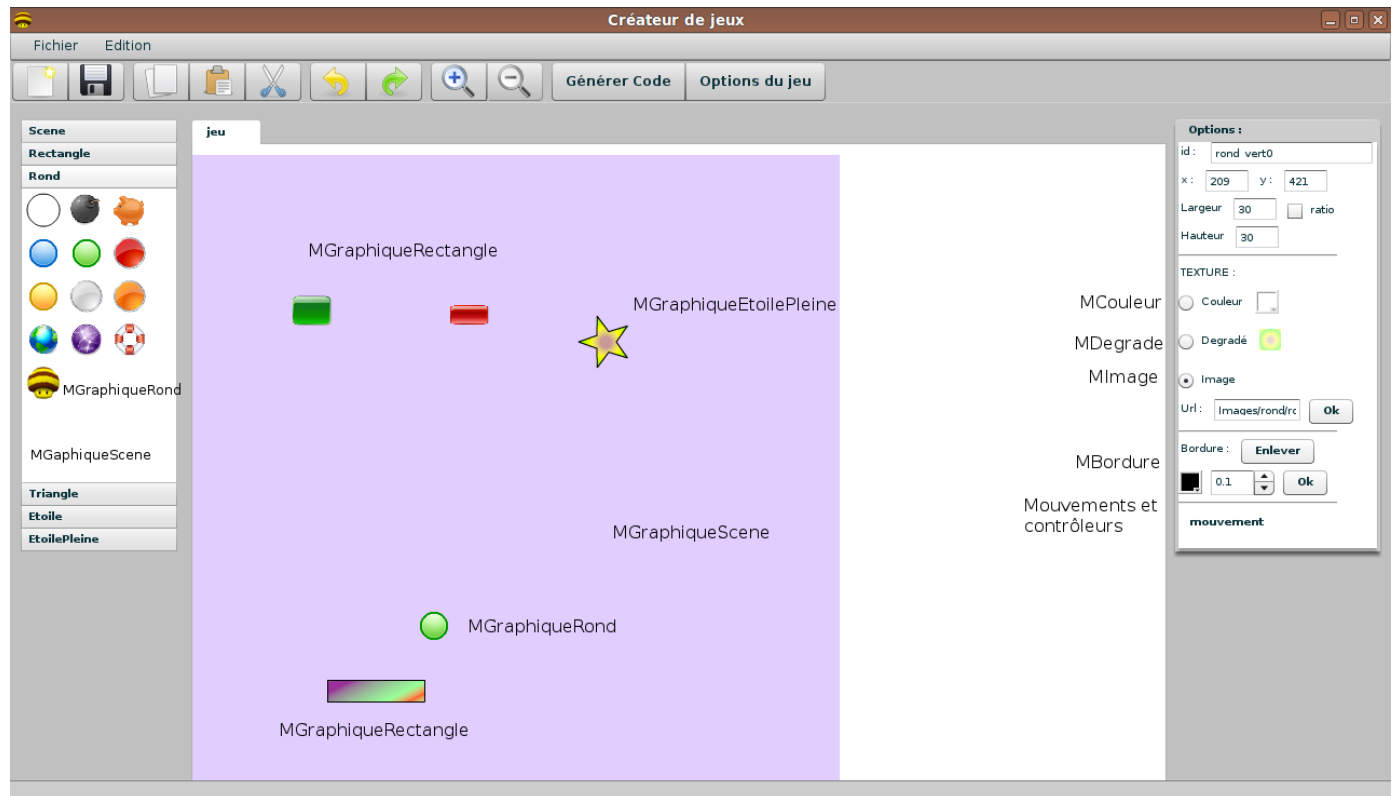
Le schéma nous montre aussi que chaque fabrique possède plusieurs fonctions pour créer les objets en question. En effet, il nous a paru nécessaire de pouvoir créer simplement tous types d'éléments, mais aussi et surtout de pouvoir en créer avec le plus de détails possible. De plus, la plupart du temps nous avons opté pour une fonction de création par défaut qui initialise les objets avec des valeurs de base (surtout utile pour le débogage). Nous avons également programmé une fonction qui permet de créer les objets en détaillant leur côté modèle, puis une autre fonction détaillant le graphisme et une dernière fonction qui mêle les deux côtés.

## 6 Créateur de jeu

Tout comme son nom l'indique le créateur de jeux est un logiciel permettant de créer facilement des jeux vidéos de types arcade. L'objectif du départ était de créer un logiciel avec lequel on pourrait créer un jeu en quelques clics, nous avons finalement produit un logiciel dans lequel il suffit de glisser les différents objets qui constitueront le jeu et de leur associer une texture, un ou plusieurs types de mouvement et de contrôle, puis de générer le code qui constituera le jeu. L'aide concernant l'utilisation du logiciel se trouve dans l'annexe de ce rapport.

### 6.1 Utilisation du framework

Le framework a été très utile pour confectionner le créateur de jeux. Dans cette image vous pouvez voir les types d'objets du framework que nous avons utilisés :



Créateur de jeux

#### 6.1.1 Créations des formes

Afin d'offrir à l'utilisateur un maximum de formes aux textures différentes, nous avons décidé de proposer des `MIObjetGraphique` dont les textures seraient des images toutes prêtes (avions, murs...).

Il a cependant fallu trouver un moyen simple de les représenter et de les stocker. Le framework propose plusieurs types de formes telles que rectangle, triangle ou encore étoile. Nous avons donc choisi de placer les images de créateur de jeux en fonction de ces formes. Ainsi, l'utilisateur peut choisir le type d'objet qu'il veut utiliser en fonction de sa forme. Pour cela, il a fallu que l'on trouve un moyen simple de stocker toutes ces images, nous avons alors créé

un fichier xml dont chaque noeud représente le nom d'une classe graphique du framework (ex : MGraphiqueRectangle, MGraphiqueTriangle...), avec des noeud enfants représentant les images qui lui sont associées et dont l'url pointe vers une image présente dans les sources du projet.

Ainsi, il nous suffit de parser ce fichier et pour chaque noeud père nous avons créé un menu-accordéon qui contient une MGraphiqueScene sur laquelle on positionne chacun de nos objets en leur associant pour texture une MImage dont le chemin se situe dans le fichier xml.

Dans la fonction de passage, nous appelons une fonction qui prend en paramètre un string qui correspond à un nom de classe de notre framework et qui retourne une instance de cet objet. De cette façon, si le noeud père est MGraphiqueRectangle, nous créons un menu-accordéon au nom de Rectangle qui ne contiendra que des objets de type MGraphiqueRectangle.

Le fait d'utiliser un fichier xml construit de cette façon, permet à l'utilisateur de rajouter facilement des images adaptées à ces objets en complétant simplement ce fichier. De plus, dans le cas où l'utilisateur aurait créé un nouveau type de MObjetGraphique, il n'aurait plus qu'à compléter de la même manière le fichier. L'aide concernant la modification du fichier xml, se trouve dans l'annexe de ce rapport. Lorsque flex permettra d'enregistrer des fichiers sur l'ordinateur de l'utilisateur, une extension possible du créateur de jeux serait une interface graphique permettant de rajouter des images au fichier xml plus simplement.

### 6.1.2 L'onglet central

L'onglet central est une MGraphiqueScene, c'est ici que l'utilisateur pourra créer son jeu. De plus, il aura la possibilité de changer le type de texture de la scène grâce au panel d'option situé sur la gauche.

Pour créer un jeu, il suffit simplement de placer des objets sur la scène et de leur donner une action. Nous utilisons le mécanisme du drag&drop pour ce qui concerne le déplacement des objets du menu-accordéon vers la scène. Lors du drop, nous utilisons la méthode clone() d'un MObjetGraphique. Ainsi, les objets du menu-accordéon sont conservés, et on crée un nouvel objet dans la scène qui possède les mêmes propriétés que celui du menu-accordéon.

Le framework ayant été construit sur le modèle MVC, lors d'un ajout dans la scène du côté graphique, le modèle est mis à jour automatiquement, ainsi dans le créateur de jeux, lors d'un drop, nous ne nous préoccupons pas du côté modèle de la scène.

### 6.1.3 Modifications des propriétés d'un objet

Afin que l'utilisateur puisse modifier le plus possible les objets comme s'il le faisait en programmant, nous avons installé un panel d'options contenant toutes les actions qu'il peut réaliser pour modifier les propriétés d'un objet. Ainsi, quelque soit le type de MObjetGraphique il peut changer :

- la taille de l'objet
- sa position sur la scène
- sa texture : MCouleur, MDegrade, MImage
- rajouter ou enlever sa bordure
- lui ajouter un ou plusieurs types de mouvements
- lui associer un type de contrôle : souris et/ou clavier
- lui associer ou modifier son id (identifiant)

De plus, et toujours grâce au modèle mvc, dès qu'une modification de ce panel est effectuée, elle se répercute immédiatement sur le modèle et sur le graphisme, on peut ainsi voir en direct

les modifications réalisées.

#### 6.1.4 Génération de code

Afin que l'utilisateur qui crée son jeu puisse pouvoir le tester, il faut qu'il génère le code de son application. Nous n'avons pas pu intégrer de compilateur flex/actionScript à notre créateur, donc il faut compiler les sources du jeu en passant par Eclipse. De plus, nous ne pouvons pas enregistrer le code que nous produisons avec le créateur, car flex ne le permet pas. De ce fait, nous ne pouvons pas sauvegarder ni restaurer ce que fait l'utilisateur. Pour parer à ce manque, lors de l'appui sur le bouton *GénérerCode*, nous ouvrons une fenêtre qui contient le code, et nous proposons à l'utilisateur de copier tout le code par le biais d'un bouton destiné à ça.

Nous générons un code flex, donc à balises, pour tout ce qui concerne la création des objets et leur propriétés, et un code de type *script* pour associer des écouteurs aux objets. Nous générons également les classes pour les différents écouteur : effets, claviers, souris.

#### Mécanisme de génération du code

Comme nous l'avons vu précédemment, à chaque fois qu'un objet est ajouté à la scène, supprimé, ou modifié, le modèle est lui aussi modifié, grâce au MVC. Afin de générer du code, nous parcourons simplement la liste des enfants de la scène et pour chacun d'entre eux, nous parcourons la liste de ses textures, de ses mouvements et de ses types de contrôle. En fonction de tous ces paramètres, nous générons le code qui lui correspond. De plus, si l'utilisateur ajoute un mouvement ou un type de contrôle, la classe correspondante est elle aussi automatiquement générée.

Il suffit alors que l'utilisateur copie tout le code dans Eclipse, et qu'il rajoute les différentes actions qu'il souhaite donner à ses objets.

## 6.2 petite conclusion

Le créateur de jeux a été programmé en même temps que le framework, il a permis notamment de pouvoir tester les différentes textures, mais aussi de mettre en évidence des problèmes de programmation qui ont été réglés par la suite.

## 7 Jeux

Nous avons décidé de faire un premier jeu qui servira d'exemple aux futurs utilisateurs de notre framework. Pour cela nous avons choisi un jeu qui utilise beaucoup de fonctionnalités. Il s'agit d'une forme de Ping-pong.

### 7.1 Le Menu

Pour commencer, nous avons fait un menu principal qui permettra de naviguer dans le jeu, modifier les commandes, voir les règles...



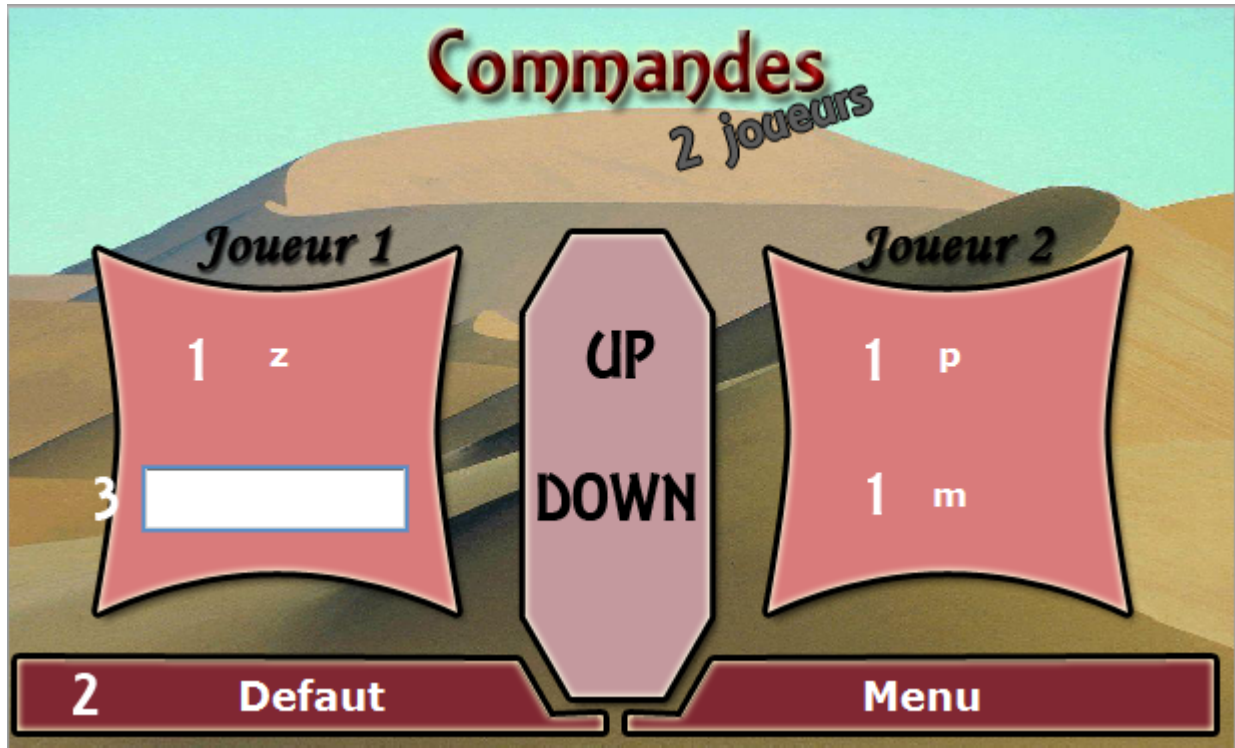
Il s'agit donc d'une simple *MGraphiqueScene* dans laquelle on a ajouté des boutons. Ces boutons ont été personnalisés grâce à un fichier css externe. Il a suffi de rajouter des actions sur le clic.

Légende :

1. Démarre le jeu en affichant la *MGraphiqueScene* du jeu
2. Affiche la *MGraphicScene* des Commandes

### 7.2 Les Commandes

Ce panneau de commande permet aux deux joueurs de personnaliser leurs commandes et ce quelque soit le moment. En effet, en pleine partie, l'utilisateur peut accéder à ce panneau.



Il s'agit donc d'une simple "MGraphiqueScene" dans laquelle on a ajouté deux boutons personnalisés. Il y a également des "MImage", des "Label", et des "Textfield". Il a suffi de rajouter des actions sur le clic des boutons et des "Label", ainsi que des actions sur les touches (KeyEvent).

Le principe est le suivant :

1. On clique sur un label d'une touche
2. Cela ouvre un Textfield sur ce label
3. Puis l'utilisateur appuie sur une touche
4. La touche est reconnue puis le label se réaffiche avec pour texte cette nouvelle touche

Légende :

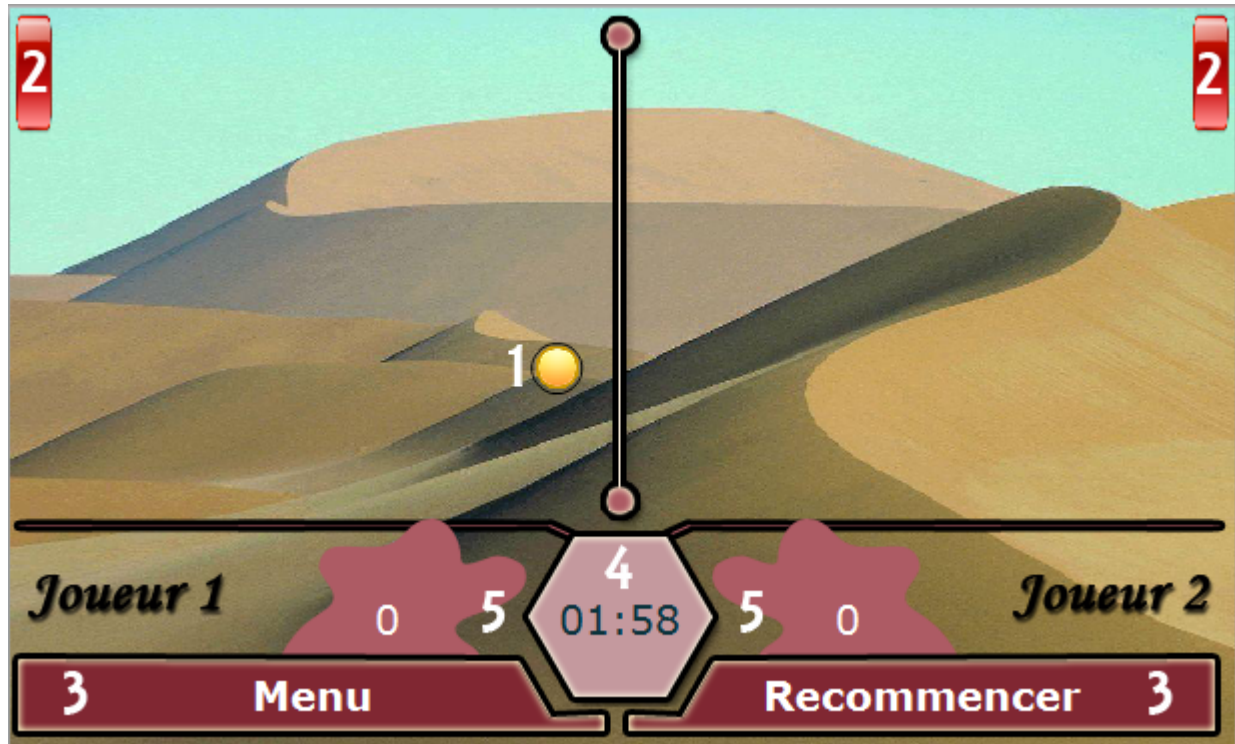
1. Label qui lors du clic affiche le textfield
2. Textfield affiché après le clic sur le label
3. Bouton permettant de remettre les commandes par défaut

### 7.3 Le Jeu

Le panneau du jeu est le plus complexe. Mais le framework a vraiment simplifié les choses comme on le verra ultérieurement. Il s'agit d'une MGraphiqueScene contenant plusieurs éléments graphiques. Certains sont en mouvement grâce à des événements claviers, d'autres sont en mouvement perpétuel et d'autres sont statiques.

La balle est un MGraphiqueRond avec une MTexture qui permet de lui mettre une image de fond ainsi qu'une MBordure permettant de mettre une bordure autour de la balle.

Ce MGraphiqueRond implémente MIObjetGraphiqueEcouteur ce qui permet de réimplémenter certaines méthodes telle que graphiqueCollision(...). Celui-ci contient également un MMouvementPerpetuel qui permet notamment de gérer par exemple les trajectoires.



Légende :

1. La balle qui traverse la MGraphiqueScene
2. Les raquettes qui font rebondir la balle
3. Boutons permettant de recommencer la partie ou de retourner sur le menu
4. Timer affichant le temps restant
5. Points de chaque joueur

## 7.4 Les données

Ce jeu a été d'une grande aide au fur et à mesure de l'avancée du framework. Il a permis d'identifier différents bugs tant au niveau graphique que modèle. Le fait d'avancer le jeu en même temps que le framework a donné des idées supplémentaires pour simplifier la programmation de petits jeux d'arcades.

Le Ping-Pong réunit les statistiques suivantes :

- Création des différentes images et boutons personnalisés : 2 Jours
- 5 classes : Menu, Jeu, Commandes, Balle, Raquette
- Moins de 900 lignes de codes
- Peut être implémenté en une journée si la personne a une bonne connaissance du framework



## 8 Conclusion

Les fonctionnalités dont on parle dans l'introduction ont été implémentées dans notre framework, ainsi, grâce à notre framework, un utilisateur peut maintenant créer facilement un jeu d'arcade, à condition bien sûr qu'il ait quelques notions de programmation. Il pourra ainsi publier son jeu sur internet, ou l'utiliser en application de bureau. De plus grâce aux MElement que nous avons rajouté, un utilisateur peut facilement créer des éléments ayant un comportement particulier comme par exemple un petit personnage qui pourrait tirer des missiles, s'accroupir et sauter.

Nous avons utilisé de nombreux patrons de conception, ce qui fait que notre framework est facilement extensible, un utilisateur pourra donc facilement étendre notre framework à sa guise pour fabriquer par exemple de nouvelles formes, de nouvelles textures, ou de nouveaux mouvements, et ce en rajoutant simplement des classes dans notre hiérarchie.

Nous avons développé ce framework dans l'optique de l'utiliser pour programmer nos propres jeux d'arcades, ainsi, les principales fonctionnalités que nous avons implémenté sont apparues naturellement au cours de la programmation des jeux.

### 8.0.1 Difficultés

Comme vous pouvez le constater, tout le nécessaire à la création d'un jeu d'arcade est présent dans notre framework, cependant, nous aurions par exemple pu améliorer la gestion des ellipse. Nous avons une classe qui gère les ellipses, mais nous avons remarqué que la façon de traiter la collision pour ce type de forme était très complexe, nous avons alors décider de ne pas continuer de développer les ellipses. De plus, nous avons commencé à programmer des formes complexes composées d'autres formes, elles mêmes pouvant être complexes, mais par soucis de temps, nous avons préféré privilégié le développement des éléments plutôt que de continuer à élaborer les formes complexes. Il serait également possible de rajouter d'autres éléments ou d'autres actions. Néanmoins, le framework est utilisable puisque grâce à lui nous avons pu créer facilement : un jeu de ping-pong, un jeu de type "Space invader" et un "pac-man".

### 8.0.2 Connaissances acquises

Ce projet nous a apporté de nombreux bénéfices, tout d'abord, nous avons appris comment faire un framework, et surtout comment organiser nos classes afin qu'elles soient extensible et utilisables facilement. Ainsi, nous avons utilisé de nombreux patrons de conception, et le fait de les avoir implémenté nous a imposé une certaine rigueur dans notre programmation, le modèle MVC a également été d'une grande aide pour rendre notre programmation plus clair.

Ce projet a été réalisé en groupe, et nous a donc appris à travailler en groupe, à réfléchir ensemble, et surtout à programmer ensemble. L'utilisation de logiciels de SVN, ou du site googleCode a été d'une grande aide pour partager nos idées et notre travail.

Nous avons choisi de programmer en utilisant l'API flex. Nous ne connaissions pas les langages de l'API qui sont MXML et l'ActionScript, cela a d'abord été un léger frein dans le développement du framework, mais nous nous sommes énormément documentés et ces langages étant très intuitifs, nous avons réussi à comprendre ces mécanismes et à les maîtriser. Dans la partie bibliographie, nous vous fournirons les divers liens vers les sites internet qui nous ont été utiles.

Nous avons également créé un site internet sur lequel nous avons mis : les sources de notre projet ainsi que les jeux et le créateur de jeux, qui sont accessibles en ligne à l'adresse suivante : <http://mus-d.lydiman.net/>.

## **9 Bibliographie**

### **9.1 Général**

- <http://fr.wikipedia.org>
- <http://www.adobe.com/livedocs/flex/3/html/index.html>
- <http://www.adobe.com/devnet/flex/tourdeflex/web/>

### **9.2 Collision**

- <http://www.flashxpress.net/ressources-flash/la-detection-de-collision/>
- [http://www.javafr.com/codes/COLLISIONS-2D-AXES-SEPARATEURS\\_42788.aspx](http://www.javafr.com/codes/COLLISIONS-2D-AXES-SEPARATEURS_42788.aspx)

### **9.3 Contrôleur**

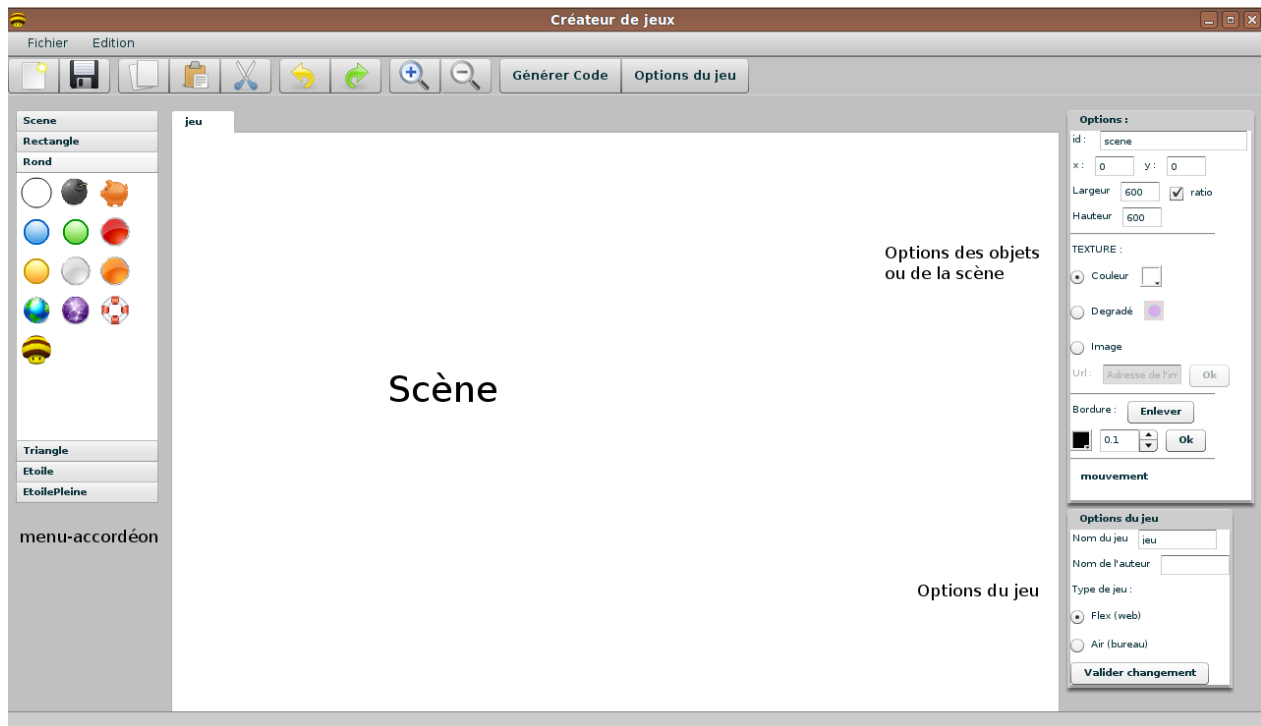
- <http://blog.flash-actionscript.com/actionscript-3-les-ecouteurs-d-evenements/>

### **9.4 Créateur de jeux**

- <http://examples.adobe.com/flex2/consulting/styleexplorer/Flex2StyleExplorer.html>

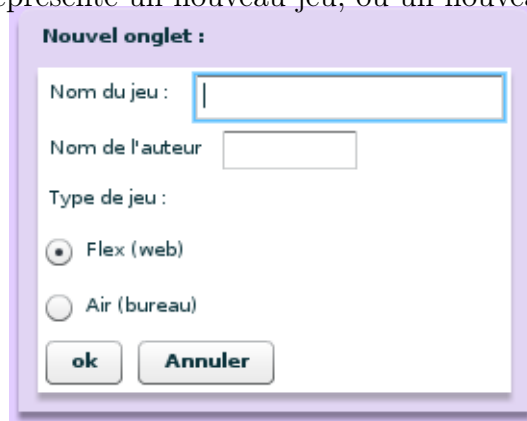
## 10 Annexe

### 10.1 Guide pour l'utilisation du créateur de jeux



#### 10.1.1 Création d'un premier jeu

Dès l'ouverture du logiciel, il est possible de créer son premier jeu dans l'onglet central qui est en fait la scène du jeu. Néanmoins, on peut également faire *Fichier -> Nouveau* ou cliquer sur la barre d'outil pour créer un nouvel onglet et ainsi créer un nouveau jeu, ou un nouveau niveau. Après avoir cliqué sur *Fichier->Nouveau*, une fenêtre s'ouvre demandant à l'utilisateur de choisir un nom pour son jeu, d'entrer le nom de l'auteur et de choisir le type de jeu qu'il désire. En effet, l'utilisateur a la possibilité de choisir si il veut créer un jeu à publier sur Internet, et pour cela il doit cocher la case Flex, ou si il veut créer une application de bureau, et dans ce cas il devra cocher la case Air, l'utilisateur aura à tout moment la possibilité de modifier ce choix. Tous les choix réalisés dans cette fenêtre seront utiles dans la génération du code, le titre apparaîtra dans l'onglet du navigateur web, ou dans la barre de titre de l'application bureau. Ces différents choix seront modifiables tout au long de la création du jeu, et ce en appuyant sur le bouton : *Options du jeu* ou en cliquant sur la scène de fond. Après avoir validé, un nouvel onglet apparaît, et l'utilisateur peut alors créer un autre jeu. Chaque onglet représente un nouveau jeu, ou un nouveau niveau.



### 10.1.2 Ajout des éléments à la scène

Après avoir ouvert un onglet, l'utilisateur peut alors positionner des objets qui seront des éléments de son jeu. Afin d'ajouter des éléments à la scène, l'utilisateur doit simplement choisir un des objets contenus dans le menu-accordéon situé sur la gauche, puis de le glisser à l'aide de la souris sur la scène à la position de son choix. L'utilisateur peut ajouter autant d'éléments qu'il le souhaite, et ce aux endroits qu'il désire.


L'utilisateur peut également utiliser le copier/coller pour copier un objet qui serait déjà présent sur la scène, pour ce faire, il doit utiliser les raccourcis clavier : *ctrl-c* et *ctrl-v* (seulement dans la version bureau de l'application) ou alors les boutons prévus à cet effet dans la barre d'outil.

Dans le cas où un utilisateur voudrait supprimer un objet de la scène, il peut également le faire, en appuyant sur la touche *suppr* de son clavier.

### 10.1.3 Modification des propriétés des objets

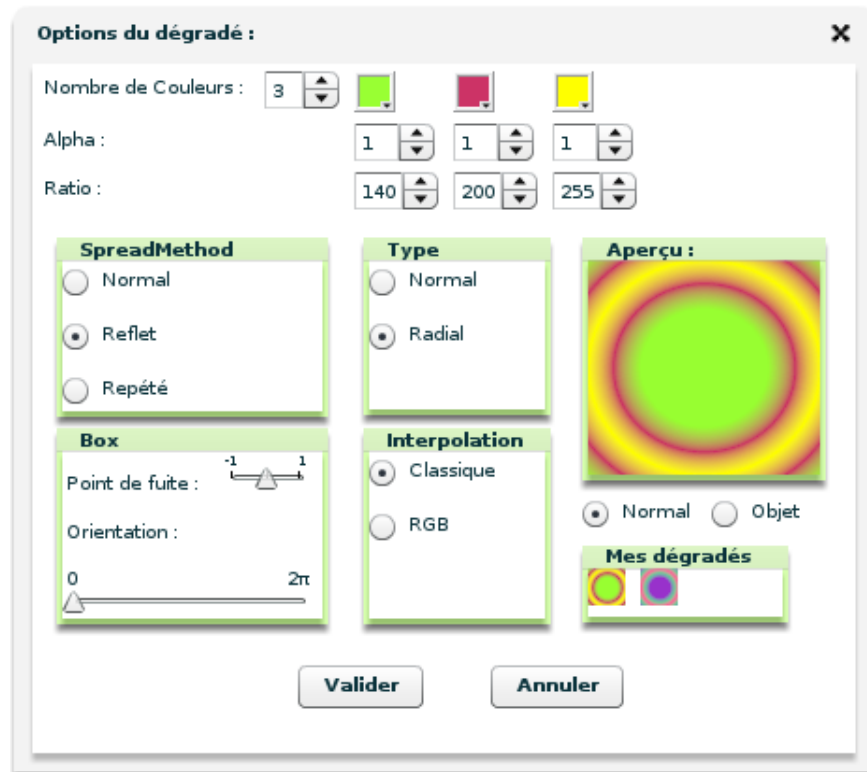
L'utilisateur peut modifier autant qu'il le souhaite les objets qui sont présents dans la scène. Tout d'abord, il doit cliquer sur un objet en particulier afin de lui donner le focus, dès lors, un carré rouge se met autour de l'objet pour signifier qu'il est sélectionné, et toutes les informations relatives à cet objet s'inscrivent dans le panel d'options (situé sur la droite). L'utilisateur peut alors modifier tous les champs de ce panel, il a la possibilité de modifier :

- l'identifiant de l'objet,
- sa position en x et en y, l'objet se place en fonction de ce que l'utilisateur entre comme paramètres,
- sa largeur et sa hauteur, il peut choisir de garder le ratio, donc la proportion entre la largeur et la hauteur, ou de ne pas le garder et ainsi, cela pourra par exemple modifier la forme de l'objet (un carré peut ainsi devenir un rectangle),
- sa texture : la couleur grâce à un ColorPicker, le dégradé, ou l'image en entrant le chemin vers l'image
- lui ajouter ou enlever une bordure
- lui affecter un ou des mouvements, des contrôleurs



## Modification du dégradé

En cliquant sur *Dégradé*, une fenêtre s'ouvre permettant de créer un nouveau dégradé. Elle propose de régler toutes les options nécessaires à instancier un MDegrade. Malgré tous les choix dont dispose l'utilisateur, la création d'un nouveau dégradé est assez intuitive.



Le panel d'option *SpreadMethod* permet de choisir le type de diffusion du dégradé : normal, répété, ou en reflet.

Le panel *type* permet de choisir si le dégradé sera normal, ou radial.

Le panel *Box* sert à définir le point de fuite du dégradé, dans le cas où il serait radial, et son orientation. La modification de l'orientation permet de créer des dégradés verticaux, obliques, horizontaux, en fonction d'un angle choisi grâce au slider.

La fenêtre de dégradé propose également une MGraphiqueScene dans laquelle on peut voir un aperçu du dégradé, en cliquant sur *objet*, l'utilisateur aura un aperçu de son dégradé appliqué à son objet. Après avoir validé, le dégradé est appliqué à l'objet, et le dégradé créé est ajouté à la liste des dégradés préférés de l'utilisateur.

## Ajout d'effets, de contrôleur

En cliquant sur le bouton *Mouvement*, une fenêtre permettant de choisir les différents mouvements et contrôles apparaît.

**Mouvements :**

**Redimensionnement** ▼ largeur finale :  hauteur finale :  temps :  ms

**Rotation perpétuelle** ▼ centre x :  centre y :  tours par secondes :

**Type de mouvement** ▼

Type de contrôle

☐ Souris

☐ Clavier

**Valider** **Annuler**

L'utilisateur peut tout d'abord choisir le type de mouvement à appliquer à son objet : perpétuel, fini, ou redimensionnement. Dès lors qu'il fait son choix, les différentes options du mouvement s'affichent à l'écran, ainsi, en fonction du mouvement l'utilisateur doit choisir :

- mouvement perpétuel : l'angle de lancement de départ, et la vitesse de l'objet
- mouvement fini : la valeur du x et du y d'arrivée, et le temps pour réaliser cette action, en milliseconde
- redimensionnement : la largeur et la hauteur finale, et le temps pour réaliser cette action.
- circulaire fini : le point autour du quel l'objet va tourner, l'angle et le temps en milliseconde
- circulaire perpétuel : le point autour du quel l'objet va tourner, le nombre de tour par seconde qu'il va réaliser
- rotation perpétuelle : le point autour du quel l'objet va tourner, le nombre de tour par seconde qu'il va réaliser

L'utilisateur peut ajouter autant de mouvements qu'il le souhaite. Il seront ensuite traités dans la génération du code.

Afin d'ajouter un type de contrôleur pour son objet : clavier ou souris, il suffit qu'il coche la ou les cases correspondantes.

### 10.1.4 Génération du code

En appuyant sur le bouton *Générer code*, une fenêtre contenant le code de l'application permettant de la lancer est générée.



Dans le cas où les objets auraient des mouvements ou des contrôleurs, nous générons également le squelette des classes contenant les méthodes des interfaces à réimplémenter. L'utilisateur n'aura alors qu'à compléter ces fichiers. Nous produisons des classes qui implémentent les écouteurs : *MObjetGraphiqueEcouteur*, *MIEcouteurSouris* et *MIEcouteurClavier*.

Cette fenêtre est composée de plusieurs parties : un éditeur de texte contenu dans un onglet, chaque onglet correspondant à la génération du code de l'application et/ou des classes à réimplémenter, et d'un bouton permettant de copier tout le code de l'onglet dans le presse papier.

### 10.1.5 Utilisation du code généré

Flex ne permet pas d'enregistrer, de modifier ou d'ouvrir des fichiers qui se trouvent sur l'ordinateur de l'utilisateur, donc pour qu'il puisse utiliser le code généré par l'application, il doit :

- Ouvrir Eclipse
- Créer un nouveau projet flex ou air, en fonction du type de jeu qu'il souhaite créer.
- Coller le code de l'application dans le fichier *.mxml*
- Coller le code des classes dans des classes *actionScript* portant le même nom
- Coller les différentes images qui constituent le jeu
- Implémenter les différentes fonctions du comportement des objets
- Compiler le projet avec Eclipse
- Lancer le jeu
- Jouer !

### 10.1.6 Ajouter des éléments au menu-accordéon

L'utilisateur peut facilement ajouter des éléments au menu-accordéon, il suffit pour cela qu'il modifie le fichier xml appelé *liste\_image.xml*.

Dans le cas où l'utilisateur voudrait ajouter des objets à un menu déjà existant, il doit simplement rajouter une ligne au fichier, à l'endroit où se trouve le noeud de la classe de l'objet à rajouter. Par exemple, si il souhaite ajouter un rectangle dont l'image est marron, il doit alors compléter le noeud "MGraphiqueRectangle".

Si en revanche l'utilisateur crée un nouveau type d'objet, il devra alors rajouter un nouveau noeud, dont le nom sera celui du nouveau type.

Voici un exemple de noeud :

```
<MGraphiqueRectangle>  
<carre id="carre_bleu" source="Images/carres/bleu.png" largeur="64" hauteur="64"/>  
</MGraphiqueRectangle>
```

Voici ici un exemple générique d'un noeud du fichier :

```
<NomDeLaClasseGraphique>  
<classe id="mon_id" – paramètres nécessaires à la création des objets – />  
</NomDeLaClasseGraphique>
```