Plateforme de création de mini jeux 3D sur le WEB

Analyse

10 décembre 2010

Berlon Antoine Bouzillard Jerôme
Chegham Wassim Clergeau Thomas Faghihi Afshin
Guichaoua Mathieu Israël Quentin
Kien Emeric Le Corronc Thibault Le Galludec Benjamin
Le Normand Erik Lubecki Aurélien
Marginier David Sanvoisin Aurélien Tolba Mohamed Amine
Weinzaepfel Philippe Zadith Ludovic

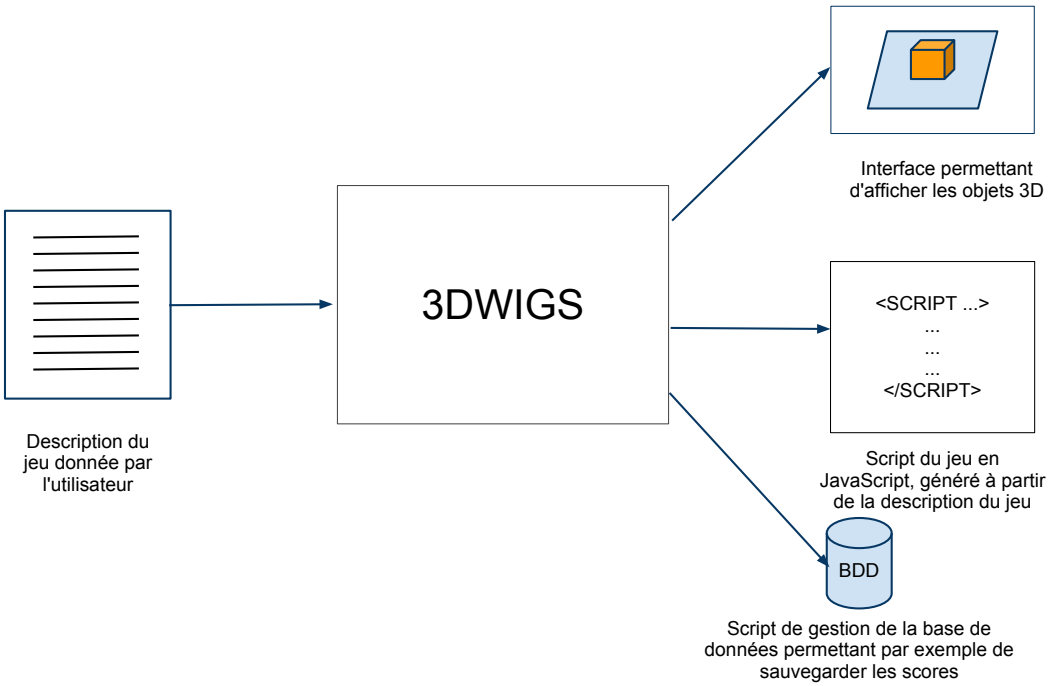
Résumé

Ce document présente la partie analyse du projet de master 1 intitulé "Plateforme de création de mini-jeux 3D sur le WEB". Il consiste à concevoir et à développer un outil auteur pour créer des contenus 3D très interactifs dans les pages internet. En particulier, il s'agit de définir un langage permettant de générer le code de mini-jeux affichés en 3D. Le rapport est divisé en 4 parties : la partie 1 présente de manière plus précise le sujet. La seconde utilise l'exemple de différents mini-jeux pour appuyer l'analyse du contenu d'un mini-jeu. La partie 3 définit les concepts récurrents dans l'univers des jeux. Enfin, la dernière partie présente les grammaires proposées.

1 Présentation du sujet

Ce projet consiste à créer un outil auteur de mini-jeux 3D destinés au Web. En effet, les récentes arrivées de HTML 5 et d'outils 3D comme WebGL permettent désormais l'affichage d'objets 3D directement intégrés dans les pages internet. Cependant, les contenus 3D dans les pages Web ne sont actuellement que très peu interactifs. Fort du succès des jeux flash, les contenus interactifs 3D comme les mini-jeux ont naturellement leur place sur le Web. Or s'il existe de nombreux outils tel que Google SketchUp pour créer et manipuler les objets 3D en eux-mêmes. Il reste tout de même un effort important à faire en ce qui concerne les interactions avec ceux-ci et en particulier la création de mini-jeux en 3D.

La conception d'un outil de création de mini-jeux 3D pour le Web nécessite, d'une part la description des objectifs, des règles, des interactions, du scénario du jeu qui permettra de générer le code du jeu, et d'autre part les éléments 3D constitutifs. Il peut également être souhaitable de pouvoir sauvegarder une partie : un jeu ne se finit pas nécessairement en quelques minutes, le fait de pouvoir reprendre une partie commencée un autre jour est alors nécessaire.



La figure ci-dessus présente le schéma général du projet. Ce dernier porte le nom de 3DWIGS pour 3D Web Interactive Game Studio. La partie gauche représente un fichier écrit par l'utilisateur contenant les règles du jeu qu'il souhaite créer respectant une certaine syntaxe qu'il faut définir.

Il s'agit de créer un langage, à la fois suffisamment abstrait pour être accessible à n'importe quel utilisateur, mais également suffisamment riche pour pouvoir décrire un maximum de jeux. Le langage est défini par une grammaire. Cette dernière doit permettre de décrire à la fois :

- les objectifs ;
- les règles ;
- les interactions avec le(s) joueur(s).

La création de la grammaire est complexe. En effet, il existe de nombreuses catégories de jeux. Par exemple, un jeu de gestion n'a, à première vue, aucun point commun avec un jeu de volley ou un jeu de plateforme.

Il serait illusoire de vouloir décrire tous les jeux à l'aide d'une seule et unique grammaire, à cause de leurs différences. Toutefois, de nombreuses similarités existent entre plusieurs jeux. Il s'agit donc de les exploiter afin de définir un langage général de description de jeux.

Le fichier définissant le jeu via la grammaire se fera à l'aide d'un éditeur spécial, créé à l'aide d'Eclipse par exemple.

La partie centrale du schéma correspond à un compilateur. Il permet de convertir le fichier de description du jeu en code JavaScript exécutable dans une page Web. En effet, le langage JavaScript a été choisi pour la génération du jeu car toutes les interactions se font côté client.

La création et l'édition des objets 3D se fera à l'aide d'outils déjà très complets tel que Google SketchUp. Leur affichage sera géré par WebGL, une spécification d'affichage 3D pour les navigateurs. Elle permet d'utiliser le standard OpenGL depuis le code JavaScript d'une page web, exploitant les accélérations matérielles 3D à l'aide des pilotes OpenGL de la carte graphique

2 Les mini-jeux

Les mini-jeux ont des genres très variés comme les jeux de gestion, de plateforme, de course. L'analyse des différences et des ressemblances entre ceux-ci est d'autant plus complexe.

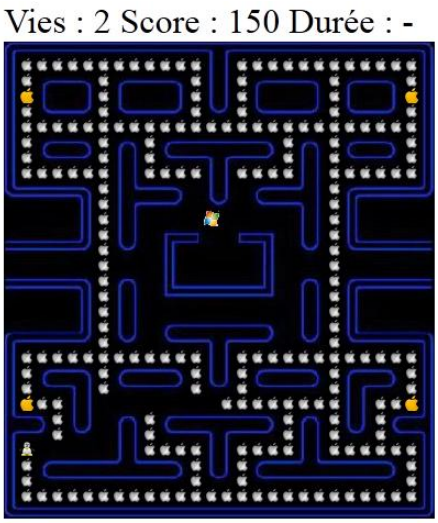
Les difficultés d'implémentation des mini-jeux sont elles aussi différentes. Huit exemples de jeux sont décrits par la suite. Cela permet d'appuyer l'analyse du contenu d'un mini-jeu. Ces jeux n'ont pas été choisi par hasard. Chacun possède un intérêt dans la description de ses règles et son implémentation.

Jeu	Intérêt majeur
Pacman	Système de bonus
1942	Tirs
Volley	Logique de score
Course	Intelligence artificielle
Mario	Niveaux
Watch'N'Droid	Vague d'ennemis
Billard	Collisions
Gestion	Ressources

Tout d'abord, chaque jeu est présenté succinctement, illustré par la capture d'écran d'une implémentation effectuée. Dans un second temps, une analyse de leur contenu sera effectuée, appuyée par un tableau comparatif et les diagrammes UML de ces jeux.

2.1 Présentation des mini-jeux

Pacman

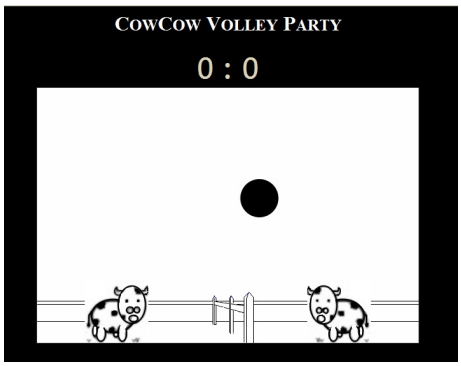


Ce jeu est une adaptation du jeu classique de Pacman. Le joueur contrôle le Tux via les touches du clavier. Son but est de manger toutes les pommes sur le terrain tout en évitant les Microsoft qui essayent de l'attraper. Pour arriver à cet objectif, le joueur dispose de 3 vies. Le Tux peut manger des pommes en or pour pouvoir détruire les Microsoft pendant une courte durée et marquer des points supplémentaires. Concernant l'affichage, le joueur voit le nombre de vies restantes, son score ainsi que le temps durant lequel les Microsoft restent vulnérables. Le jeu est adaptable car il est facile d'initialiser la carte à partir d'images et d'un fichier JSON.

1942

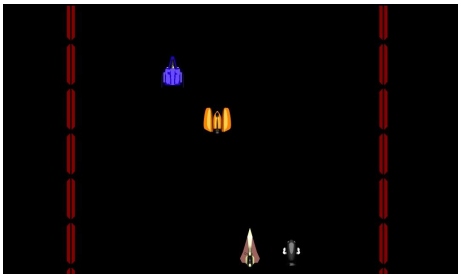
Dans ce shoot'em up (jeu d'action où le joueur fait face à une multitude d'ennemis), le joueur contrôle un vaisseau armé de deux canons pour détruire tous les véhicules adverses et ainsi gagner des points. Le joueur possède 3 vies et doit engranger le maximum de points. Lorsque le joueur perd une vie, il devient invulnérable durant une courte durée pour reprendre la main. Le vaisseau est contrôlé via le clavier. En ce qui concerne les ennemis, ils suivent des déplacements prédéfinis qui peuvent être paramétrés. Le joueur n'est pas obligé de tuer tous les ennemis mais le but est de faire le plus grand score.

Volley



CowCow Volley Party est un mini jeu humoristique mettant en scène deux vaches jouant au volley. On peut jouer à ce jeu en mode solo, contre une IA ou à deux joueurs. Le but est de remporter 2 sets, sachant que remporter un set revient à marquer 21 points. La vache dispose de 3 coups différents : passe courte, passe longue et un smash. Les règles de ce jeu sont les mêmes que celles du volley classique. La vache est contrôlée au clavier aussi bien en mode multijoueur qu'en mode solo.

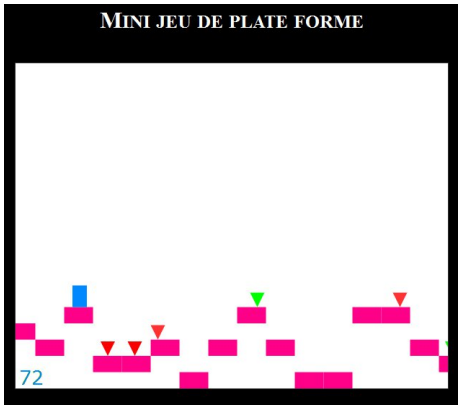
Course



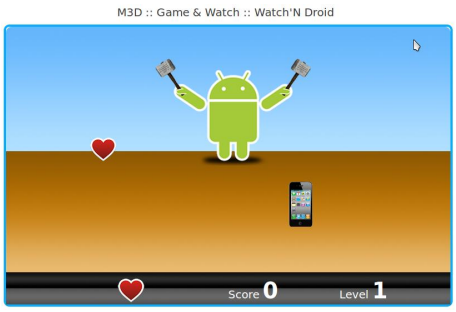
Ce jeu de course futuriste pour le web permet au joueur de se frotter à des intelligences artificielles pour faire le meilleur temps. Ce n'est pas un jeu de course classique, des bonus se trouvent sur le circuit : un turbo, un bonus permettant d'augmenter le chrono des concurrents, un autre permettant l'échange de sa position avec celle d'un adversaire, etc. De plus, il existe différentes zones de circuit ayant des effets divers : inversion des commandes, réduction de vitesse, etc. Le joueur peut choisir son véhicule parmi une sélection de 12 vaisseaux différents contrôlés au clavier. Ce jeu est adaptable : en effet, de nouveaux bonus et de nouveaux circuits peuvent être ajoutés facilement. Il est aussi possible de mettre en place un système de lecture de fichier pour configurer les paramètres des différentes courses.

Mario

Ce mini-jeu de plateforme reprend le principe de jeux comme Mario Bros. Le personnage est représenté par un rectangle et contrôlé au clavier. Il peut se déplacer sur les côtés ou sauter. Il doit avancer au maximum sans tomber dans les trous ni être touché par des ennemis. Ces derniers sont symbolisés par des triangles. Le personnage peut faire perdre des points de vie aux ennemis en leur sautant dessus jusqu'à les tuer. S'il les touche sur les côtés, il meurt et la partie est terminée. La caméra avance lorsque le joueur avance suffisamment mais elle ne lui permet pas de revenir en arrière. Le terrain est généré aléatoirement et est infini. Le but du jeu est d'obtenir le plus grand score. Le score diminue avec le temps et augmente lorsqu'un ennemi est tué ou que le personnage avance. Il est facilement possible de changer le type de fonctionnement du jeu pour passer dans un mode où le but est de passer d'un niveau à un autre avec des niveaux prédéfinis au préalable.



Watch'N'Droid



Watch'N'Droid est un mini-jeu du style Game & Watch. Le joueur doit chasser ses ennemis avant qu'ils ne l'atteignent. Ils apparaissent un par un en bas de l'écran et montent pour l'atteindre à une certaine vitesse. Cette dernière varie en fonction du niveau dans lequel le joueur se trouve. Le personnage est armé de deux marteaux pour détruire les ennemis et marquer des points. Lorsque l'ennemi atteint le joueur, ce dernier perd une vie. Lorsqu'il ne lui reste que deux vies, des vies bonus apparaissent sur l'écran et le joueur peut les ramasser. Lorsque le nombre de vie arrive à zéro, le joueur a la possibilité d'enregistrer son score s'il fait partie des cinq meilleurs qui sont actuellement enregistrés.

Billard

Ce mini-jeu de billard est un jeu multijoueurs en tour par tour. Il reprend les règles classiques du billard anglais : chaque joueur doit rentrer toutes les boules d'une couleur puis la noire. D'un point de vue gameplay, le joueur actif (désigné par une icone verte) contrôle la queue de billard avec la souris. Cette dernière pointe automatiquement vers la boule blanche. Il peut ainsi choisir l'angle avec lequel il compte frapper la boule blanche. En fonction du temps pendant lequel il laisse le bouton de la souris enfoncé, le tir est plus ou moins fort. Pour l'affichage, chaque joueur voit le nombre de boules rentrées ainsi que sa couleur. Enfin, le cadre bleu en bas de l'écran affiche la personne ayant gagné la partie.

Gestion

Commissariat est un jeu de gestion du style Farmville. Le but est de maintenir et améliorer le commissariat en fonction des ressources disponibles. Les ressources sont au nombre de quatre : le nombre de policiers, l'argent, l'indice IGPN et l'alcool. Il y a trois actions disponibles pour le joueur. Il peut envoyer des policiers en mission dans un quartier choisi pour ramener de l'argent ainsi qu'un prisonnier. Si un prisonnier est ramené, le joueur a la possibilité de libérer le prisonnier contre de l'argent ou de le tabasser (avec un fort risque de pénalité). Il peut aussi acheter des équipements ainsi que de l'alcool avec son argent. L'alcool est une ressource qui diminue constamment, le joueur doit veiller à toujours en avoir pour ne pas perdre. Il peut aussi perdre avec un indice d'IGPN trop élevé, cet indice monte avec toutes les mauvaises actions comme tabasser un prisonnier.

2.2 Analyse des mini-jeux

Ces mini-jeux ont été développés et leurs diagrammes UML sont disponibles en annexe. De plus, ils permettent d'appuyer l'analyse de la description d'un mini-jeu. Le tableau suivant récapitule les différents objectifs des jeux et leur style de terrain.

Jeu	Niveaux	Fin de niveau	Victoire	Score	Terrain
Pacman	Oui	survie + tout ramassé	tous niveaux	Oui	Grille
1942	Oui	survie + ligne d'arrivée	tous niveaux	Oui	Progression verticale
Volley	Non	score	score	Oui	Plateau
Course	Oui	ligne d'arrivée	tous niveaux	Oui	Ruban
Mario	Oui	survie + ligne d'arrivée	tous niveaux	Oui	Progression horizontale
Watch'N'Droid	Oui	tout tué	tous niveaux	Oui	Plateau et grille
Billard	Non	toute boule rentrée	score	Oui	Plateau
Gestion	Non	Non	Non	Oui	Grille

Le second tableau permet de comparer les aspects concernant le contrôle et les personnages.

Jeu	Personnage	Contrôle	Autres actions	PNJ
Pacman	Oui	direct via clavier	Aucune	Oui
1942	Oui	direct via clavier	Tir	Oui
Volley	Oui	direct via clavier	Tir, Saut	Oui
Course	Oui	direct via clavier	Utilisation bonus	Oui
Mario	Oui	direct via clavier	Saut	Oui
Watch'N'Droid	Oui	direct via clavier	Frappe	Oui
Billard	Oui (si queue=joueur)	direct via clavier	Force	Oui
Gestion	Oui mais inutile	clic, sélection et action	Clic pour actions	Non

Ces tableaux permettent de mieux identifier les différences et points communs entre les jeux proposés. Dans le second tableau, PNJ signifie Personne Non Joueur, soit toute entité qui a un comportement autonome.

Par exemple, les objectifs au cours d'un niveau (ou pour un jeu sans niveau) se limitent souvent à atteindre une certaine zone appelée ligne d'arrivée, ou éventuellement à remplir des conditions de temps ou de ressources. En effet, la vie et le score peuvent être vus comme des ressources. Une condition sur celles-ci est nécessaire à la victoire. En faisant des conjonctions et des disjonctions de ces différentes possibilités, les conditions de victoire pour tous les jeux (sauf le jeu de gestion) peuvent être définies.

En revanche, si on observe la façon dont le monde est construit, il est très différent d'un jeu à l'autre. Pour certains comme Pacman, il est défini par une grille, pour d'autres comme le jeu de course, il est représenté via un ruban.

Le tableau permet de mettre en évidence le jeu de gestion par rapport aux autres jeux. En effet celui-ci ne dispose pas de niveau et de conditions de victoire. De plus, le personnage n'est pas directement contrôlé par l'utilisateur pour lui faire faire des actions. La grammaire proposée ne traitera pas le cas des jeux de gestions. Il serait possible de définir plusieurs grammaires, chacune couvrant une catégorie de jeux afin de pouvoir en créer plusieurs types. Ainsi, une autre grammaire pourrait spécialement être créée pour les jeux de gestion.

La comparaison se base donc sur les autres jeux, pour définir des concepts qui seront détaillés dans la partie suivante.

On remarque tout d'abord que tous les mini-jeux possèdent une boucle de rafraîchissement. Cette dernière permet à la fois d'effectuer les différentes actions telles que le déplacement des personnages non joueurs et de gérer les différents compteurs du jeu (durée d'une course, d'un bonus).

Ensuite, dans les mini-jeux se retrouvent souvent la notion d'avatar (personnage que le joueur peut déplacer). Différentes actions peuvent se greffer à cet avatar selon les jeux : souvent le déplacement, le saut, le tir. Cet avatar a des caractéristiques que l'on retrouve dans plusieurs jeux comme la vie, d'autres plus rares comme une quantité de points de magie. De même, dans de nombreux jeux se trouvent la notion d'ennemis. Ce sont des personnages non-joueurs dont le comportement est prédéfini et qui ont pour but d'empêcher le joueur de gagner.

De plus, la notion de ressources est très utile. Il s'agit d'une variable qui définit un état, par exemple, sur les entiers tel que le score. Celle-ci peut être modifiée selon les évènements du jeu.

Enfin, dans tous les mini-jeux, le concept de collision est nécessaire. Celle-ci déclenche, la plupart du temps, de nombreux évènements comme la mort d'ennemis, de joueurs ou l'incapacité de passer un obstacle.

3 Définition de concepts

Certains concepts se retrouvent dans beaucoup de mini-jeux. Le but de cette partie est de les définir de manière précise.

Définition générale d'un jeu

Jeu : Activité de loisir soumise à des règles et ayant des objectifs. Il s'agit ici d'un contenu numérique disponible via un site internet.

Règles du jeu : Ensemble de principes qui décrivent le fonctionnement du jeu et la façon dont les éléments qui le composent interagissent entre eux. Ces règles définissent également les conditions nécessaires à la victoire (objectifs).

Niveau : Sous-division d'un jeu. Elle permet de gérer la difficulté de jeu de manière progressive. Elle peut aussi être utilisée pour en définir la fin.

Élément d'un jeu

Environnement : Univers du jeu. Il s'agit des décors qui le composent, des entités et des objets qui s'y trouvent ainsi que des règles physiques qui s'y appliquent comme la gravitation.

Terrain : Socle de l'environnement. Il peut être défini via une grille (terrain fixe et positionnement par case), un ruban (typiquement pour représenté un circuit ou un jeu avec une progression qui amène à un défilement).

Entité : Elément présent dans l'environnement pouvant changer d'état. Elle dispose de caractéristiques et d'un panel d'actions prédéfinies qui lui sont propres.

Décor : Elément inerte présent dans l'environnement.

Ressource : Elément virtuel d'un jeu exprimé sous forme numérique, pouvant servir de monnaie, de matière première, d'état ou autres.

Groupes d'Entités

Avatar : Entité qui agit avec un comportement particulier. Un personnage peut être le joueur, un allié, un ennemi ou neutre.

Joueur : Le joueur est une entité contrôlée par l'utilisateur grâce au clavier ou à la souris.

Allié : L'allié est une entité contrôlée par l'IA. Elle possède un comportement visant à aider le joueur-lors de sa progression dans le jeu.

Ennemi : L'ennemi est une entité contrôlée par l'IA. Elle possède un comportement visant à faire perdre le joueur.

Neutre : Le neutre est une entité contrôlée par l'IA. Elle possède un comportement précis qui ne tient pas compte des autres groupes.

Liste de Ressources

Inventaire : Ensemble des objets dont dispose une entité et qu'elle transporte.

Bonus/Malus : Objet modifiant l'état d'une entité (ou plusieurs) de façon positive/négative. L'effet peut-être déclenché par une Entité ou par l'action de ramasser l'objet.

Vie : Ressource particulière associée aux entités mortelles et aux objets destructibles du jeu.

Temps : Ressource particulière qui se calcule par rapport à la boucle de rafraichissement et évolue de façon cyclique (+1 tous les 3 rafraichissements par exemple)

Score : Ressource particulière qui se calcule par rapport aux actions du joueur et leurs conséquences sur la partie. En fin de partie, le score permet un classement de tous les utilisateurs ayant déjà joué auparavant.

Contrôle direct/indirect

Contrôle direct : Capacité du joueur à interagir sur l'environnement, limitée à une seule entité à un instant donné. Les actions dont il dispose sont celles dont l'entité dispose.

Contrôle indirect : Capacité du joueur à interagir sur l'environnement, plus complexe. Les actions dont il dispose sont définies à la création du jeu et ne se limitent pas au contrôle et aux actions d'une unique entité. Il faut souvent une sélection de ou des entités avant d'agir dessus (ex : sélection de plusieurs soldats dans un jeu de type stratégie en temps réel).

Environnement linéaire/non linéaire

Environnement linéaire : Environnement dans lequel le joueur doit suivre un cheminement prédéfini (ex : Tomb Raider).

Environnement non linéaire : Environnement dans lequel le joueur est libre d'évoluer comme il le souhaite (ex : World of Warcraft). Il n'y a donc pas d'obligation à accomplir les objectifs dans un ordre précis.

4 Grammaires

Trouver une grammaire riche, complète et accessible à n'importe quel utilisateur est difficile. C'est pourquoi, deux grammaires sont présentées.

- La première est dite de 'haut-niveau'. Elle permet de décrire la majorité du contenu du jeu dans un langage simple et accessible pour les non-programmeurs.
- La seconde est dite de 'bas-niveau'. Elle permet de manipuler chaque attribut et est beaucoup plus proche de l'implémentation finale que la première grammaire. C'est, par exemple, elle qui permettra de définir le comportement de l'intelligence artificielle, chose très difficile à mettre en œuvre dans la grammaire de haut-niveau.

Le schéma de compilation se complexifie alors : un fichier décrivant le jeu dans la grammaire de haut-niveau est compilé afin de donner un fichier respectant la grammaire de bas-niveau. A ce niveau-là, l'utilisateur peut effectuer de nouveaux ajouts ou modifications. Le second compilateur produit alors le script final du jeu en JavaScript.

4.1 Grammaire haut-niveau

La description du jeu se déroule en plusieurs phases. Au début, on peut spécifier les attributs du jeu, puis déclarer ses entités : personnages, véhicules... et leurs attributs. Viennent ensuite les définitions d'actions de base et des commandes. Enfin, les règles du jeu sont spécifiées. Il est possible d'ajouter une intelligence artificielle (IA) basique, utilisant les entités et les actions déjà définies.

4.1.1 Attributs du jeu

Les attributs du jeu sont gérés principalement par les entités dans la grammaire haut-niveau, ce n'est donc pas obligatoire de préciser cette partie lors de l'écriture du code. Cependant il peut être plus rapide de les définir directement ici, comme : `game has gravity at 9.81`.

4.1.2 Définition des entités du jeu et de leurs attributs

Il s'agit de lister les différents personnages ou objets qui apparaissent dans le jeu. Pour faciliter le travail de l'utilisateur, diverses classes sont déjà créées pour définir les entités. La classe de base est la classe Object avec des attributs de base comme la position, l'orientation, la taille, etc. De celle-ci héritent de nombreuses autres classes comme Character, Vehicle ou Weapon, chacune ayant de nouveaux attributs spécifiques. L'ensemble de ces classes et leurs attributs sont en annexe.

La déclaration d'une nouvelle entité se fait alors via un identificateur et le mot-clef 'is'. Par exemple `Mario is Character` permet de déclarer Mario comme un personnage de classe Character, et peut donc posséder tous les attributs de cette classe. Ces derniers sont initialisés à des valeurs par défaut. L'utilisateur peut les modifier. Par exemple, pour changer l'attribut `lifeMax` de Mario, représentant le niveau le plus haut possible de son attribut `life` les mots-clés 'has' et 'at' sont utilisés : `Mario has lifeMax at 3`. De même, il est nécessaire de pouvoir rajouter de nouveaux attributs. La syntaxe est la même. Ainsi, `Mario has energie at 8` ajoute un attribut energie (qui n'existe pas déjà dans Character) à Mario, initialisé à 8. Les nouveaux attributs doivent obligatoirement être initialisés avec 'at'.

De plus, s'il existe divers personnages qui possèdent aussi cette énergie, il est souhaitable de ne pas avoir à répéter cette ligne. Ainsi, de nouveaux types peuvent être créés grâce au mot-clef 'type' : `type Plombier is Character. Plombier has energie at 8. Mario is Plombier. Luigi is Plombier.` L'héritage multiple

est possible ainsi que l'héritage de nouveaux types déclarés par l'utilisateur. Il suffira d'écrire : `type BillBall is Character and Projectile. type ennemiDangereux is BillBall` Si un nouveau type ou objet est fils de plusieurs classes ayant un même attribut, ils sont fusionnés en un seul.

Ensuite, en même temps que la déclaration de la classe d'une nouvelle identité, il est possible de la définir comme commandée par le joueur, ennemie, alliée ou neutre. Si rien n'est précisé, la valeur est mise à neutre par défaut. `Mario is Plombier player. Luigi is Plombier ally. type Boss is Character ennemy. Bowser is Boss.` De plus, dans le cas où ce n'est pas un personnage joué, il peut être déclaré comme 'duplicable'. Cela signifie qu'il est possible d'en générer plusieurs avec la seule commande 'generate'. Par exemple `Koopa is Character enemy duplicable. generate 5 Koopa in zone`. Permet de créer 5 Koopa dans `zone` qui aura été définie auparavant : `zone is Zone`. Outre 'in' qui est associé aux objets de type Zone, le mot-clef 'on' permet de placer des objets juste au-dessus de l'objet défini : `generate Mario on Luigi`, et le mot-clef 'at' génère le premier objet à l'endroit indiqué et les autres autour de celui-ci, en évitant les collisions.

Enfin, il est possible de manipuler des listes.

```
type humain is Character.
aragorn is humain player.
boromir is humain.
type elfe is Character.
legolas is elfe.
type nain is Character.
gimli is nain.
type hobbit is Character duplicable.
type magicien is Character.
gandalf is magicien.
CommunauteDeLAnneau is list of aragorn with legolas with gimli with aragorn with boromir with gandalf with
    4 hobbit.}
```

4.1.3 Autres classes prédéfinies

D'autres classes sont prédéfinies : une classe Game concernant l'ensemble général du jeu, une classe Camera contenant les informations sur les caméras : certains comportements classiques sont prédéfinies comme une caméra libre ou bien un suivi à la première ou troisième personne.

De plus, 2 classes permettent de gérer respectivement les compteurs et les ressources de type temporel.

Enfin, une dernière classe permet de gérer les fichiers multimédias tel que les sons ou les vidéos qui peuvent intervenir au cours d'un jeu avec la possibilité de les jouer, de les arrêter, de les mettre en pause, d'activer le mode muet ainsi que de définir s'ils seront joués une fois pour des sons (mot-clef 'once'), ou en boucle (mot-clef 'loop') pour des musiques par exemple.

Le détail sur ces classes est disponible en annexe.

4.1.4 Définition de nouvelles actions et assignation des commandes

Il est ensuite possible de définir de nouvelles actions via le mot-clef 'definition' et 'means'. Par exemple, en reprenant l'exemple de Mario : `Mario is Character player. definition sursautMario means mario jump 15`. Les commandes sont ensuite définies après le mot-clef 'command'. Elles sont générées soit par l'appui d'une touche X du clavier ('key X') soit par une action Y sur la souris ('mouse Y'). `command mouse rClick for sursautMario` permet par exemple de causer l'action nommée sursautMario lorsque le joueur clique sur le bouton droit de la souris.

De plus, de nombreuses actions sont prédéfinies (pour les personnages notamment) comme jump ou move left ; ou concernant le jeu en général comme pause, save ou victory. `command mario is key space for jump 15, key Z for move forward, key Y for move backward, key S for move left, key D for move right.`

Il est possible de désactiver (respectivement d'activer) certaines touches du clavier ou action de la souris au cours du jeu. Pour cela, il faut utiliser le mot-clef 'disable' (respectivement 'activate'). `disable key Z` désactive les actions lors de l'appui sur la touche Z. Il est également possible de désactiver toutes les commandes `disable commands`, seulement celles du clavier `disable keyboard` ou celles de la souris `disable mouse`.

4.1.5 Déclaration des règles du jeu

Les règles du jeu sont définies par le mot-clef 'rule' et 'then'. Par exemple `rule mario dies then defeat.` 'mario dies' est un déclencheur et 'defeat' est une conséquence. Plusieurs conséquences peuvent suivre le 'then', elles sont exécutées séquentiellement.

Il est également possible de manipuler les attributs des différentes entités. Prenons le cas où un ennemi a été défini `bowser is Character ennemy`. Si mario entre en collision avec lui, il perd un point de vie, réduit son score de 100 et provoque un saut : `rule mario touches bowser then sub 1 for life of mario, sub 100 for score of game, mario jump 15.`

Avec 'sub', d'autres mots-clés existent pour les autres opérations arithmétiques élémentaires, ainsi qu'un 'assign' qui change directement la valeur de l'attribut. Il est également possible d'exécuter des actions seulement sous certaines conditions avec un 'If ... then ... else ... endIf'.

Certains concepts sont déjà définis : ainsi 'touches' est causé par une entrée en collision entre les deux entités. 'dies' signifie l'arrivée de l'attribut 'life' à 0 pour un objet de type Character, 'kills' se produit lorsque le premier personnage tue le second.

4.1.6 IA

Une IA basique peut être créée à partir des éléments déjà existants et définis lors des lignes de codes précédentes. La grammaire haut-niveau ne permet en aucun cas de créer une IA améliorée à moins d'écrire un nombre important de lignes de code. Mais là encore, les possibilités resteraient limitées. Voici un exemple :

```
ia goomba is
        move left during 2 sec
        jump 12
        move right during 2 sec
        jump 12
```

4.1.7 Exemple de mini-jeu décrit dans la grammaire 'haut-niveau'

Voici un exemple de code utilisant la grammaire haut-niveau. Supposons que l'utilisateur crée un mini-jeu avec les éléments suivants :

- des représentations en 3D d'un shérif, d'un zombie, d'une pièce de monnaie par exemple au format Collada (.dae) dans les fichiers 'sheriff.dae', 'zombie.dae' et 'piece.dae'
- d'un sol
- d'un volume englobant représentant une zone d'arrivée
- d'une musique nommée 'musiqueAngoissante.mp3' et de deux musiques de fin 'musiqueVictoire.mp3' et 'musiqueDefaite.mp3'

Le but est, pour un shérif contrôlé par le joueur, d'éviter tous les zombies et de récupérer toutes les pièces d'or sur la carte. Il gagne lorsque toutes les pièces d'or ont été ramassées et qu'il se place sur la zone d'arrivée. S'il parvient à la ligne d'arrivée avant d'avoir récupéré toutes les pièces, une nouvelle nuée de zombies affamés apparaît. Il perd lorsqu'un zombie parvient à le toucher.

```
game has score at 0.

cam1 is Camera free.         //free is not necessary because it is the default behavior for a camera

sol is Ground.

zombie is Character enemy duplicable.
zombie has attack at 1.                 //if a zombie touche the player, he loses one life

sheriff is Character player.
sheriff has life at 1.                  //he has only 1 life, so, at the first shot, he dies
sheriff has position at 0 300 0.

piece is Bonus duplicable.
piece has value at 10.
piece has collectors at sheriff.        // the only character able to collect the bonus

arrivee is Zone.
arrivee has position at 300 500 0.

musiqueAngoissante is media loop.
musiqueVictoire is media once.
```

```
musiqueDefaite is media once.


definition generer10Zombies means
        generate 9 zombie on sol,     //10 new zombies are generated
        generate 1 zombie in zone.      //1 more in the zone where the player is


command sheriff is
        space for jump 15,       //jump must be higher than game's gravity
        key Z for move forward,
        key S for move backward,
        key Q for move left,
        key D for move right.

command key P for pause.

command key enter for generer10Zombies.     //for more challenge


rule game starts then
        generate sol,
        generate 30 zombie on sol,
        generate 10 piece on sol,
        generate sheriff,
        play musiqueAngoissante.

rule sheriff touches arrivee then
        if score of game = 100 then
                play musiqueVictoire,
                victory
        else generer10Zombies
        endIf.

rule zombie touches sheriff then
        sheriff dies,
        play musiqueDefaite,
        defeat.
```

4.2 Grammaire bas-niveau

La grammaire bas-niveau est beaucoup plus proche de l'implémentation finale que la haut-niveau. Ainsi, toutes les ressources doivent avoir un identifiant unique. Les évènements, qu'ils proviennent du joueur (clavier, souris) ou de la modification de l'état d'un personnage sont gérés par des signaux. La description d'un jeu est constituée de ressources, d'entités, de caméra, d'une boucle de rafraîchissement, d'un gestionnaire d'évènements. Un moteur physique permettant de tester les collisions et contenant également les forces générales du jeu (gravitation, vent, etc) est également disponible.

4.2.1 Définition des ressources

Une ressource est une variable du jeu. Elle est identifiée par un nom unique.
`marioLife 1` permet de créer une ressource marioLife initialisée à 1.
Il existe un type spécial de ressource : les ressources temporelles. Celles-ci sont mises à jour via la boucle de rafraîchissement générale du jeu. Pour la définir, cette fois 2 valeurs sont données suite à la ressource : le pas du timer, et son initialisation : `timeStarBonus 10000 0` permet de créer une ressource de nom 'timerStarBonus' (pouvant correspondre à la durée du bonus donnée par une étoile) avec un pas de 10 secondes et initialisée à 0 millisecondes.
Il est également possible de faire des énumérations de ressources. `foret{chene,olivier,peuplier}`

4.2.2 Définition des entités et caméras

L'ensemble des entités d'un jeu est alors constitué d'un terrain et d'un ensemble d'objets. Chaque objet est identifié par un nom, ainsi que par un fichier créé par des outils de dessin d'objets 3D, par exemple au format Collada (file.dae). De plus chaque objet contient un ensemble de paramètres comme la vitesse ou la position.
Les caméras sont définies de manière similaire aux entités : chaque caméra est définie par un nom, une position et une orientation.

4.2.3 Boucle de rafraîchissement et gestionnaire d'évènements

Les divers évènements du jeu sont gérés via un système de signaux.

Le jeu dispose d'une boucle de rafraîchissement principale. Celle-ci génère un signal signalUpdateCounter émettant un signal régulièrement. De même, il permet de vérifier les touches enfoncées et actions de la souris à chaque tic du timer.

Enfin, chaque mise à jour de ressource nommée X émet un signal updateX.

Les actions lors de la réception d'un signal sont gérées par un gestionnaire d'évènements. Ce dernier est défini par un ensemble de signaux et d'instructions. Les instructions sont données comme dans un langage classique. Elles sont composées des mises à jours de variables en utilisant les opérateurs arithmétiques classiques, les constantes, les autres ressources et des nombres aléatoires. Des instructions conditionnelles sont également disponibles, les booléens étant générés via des comparaisons d'expressions.

Les instructions peuvent également reprendre des concepts classiques de mini-jeux : pause, nouvelle partie, fin de partie, sauvegarde de la partie.

Voici l'exemple d'un jeu où un personnage `perso` et un ennemi `boss` ont été définis, ainsi que des ressources `persoLife` et `bossLife`. Dans le cas où le signal `bossAttack` est lancé, il est possible d'avoir les règles : `bossAttack persoLife -1` qui retranche 1 à la ressource `marioLife`. Puisque `persoLife` est modifié, le signal `updatePersoLife` est émis et on peut avoir une règle `updatePersoLife if persoLife==0 then gameOver`.

4.2.4 Exemple de mini-jeu dans la grammaire bas-niveau

Le même mini-jeu que celui décrit dans la grammaire haut-niveau est maintenant présenté dans la grammaire bas-niveau.

A venir ...

4.3 Avantages et limites de ces langages

Ces grammaires ont l'avantage de donner à l'utilisateur la possibilité de décrire simplement le jeu. La séparation nette entre la définition des entités, des interactions et des règles du jeu donne plus de clarté au fichier de description.

De plus, les deux grammaires sont complémentaires. La première permet de décrire le fonctionnement du jeu en général. La seconde permet de détailler plus facilement le comportement concret d'entités, notamment pour l'intelligence artificielle.

Cependant, la capacité des deux grammaires est limitée. La réalisation d'un UML commun aux jeux n'a pas été possible. Ceux-ci ont une diversité importante au niveau de leur catégorie. Tous les jeux ne peuvent pas être codés avec les langages proposés.

5 Stratégie de Développement

Bon courage Aurélien.

Conclusion

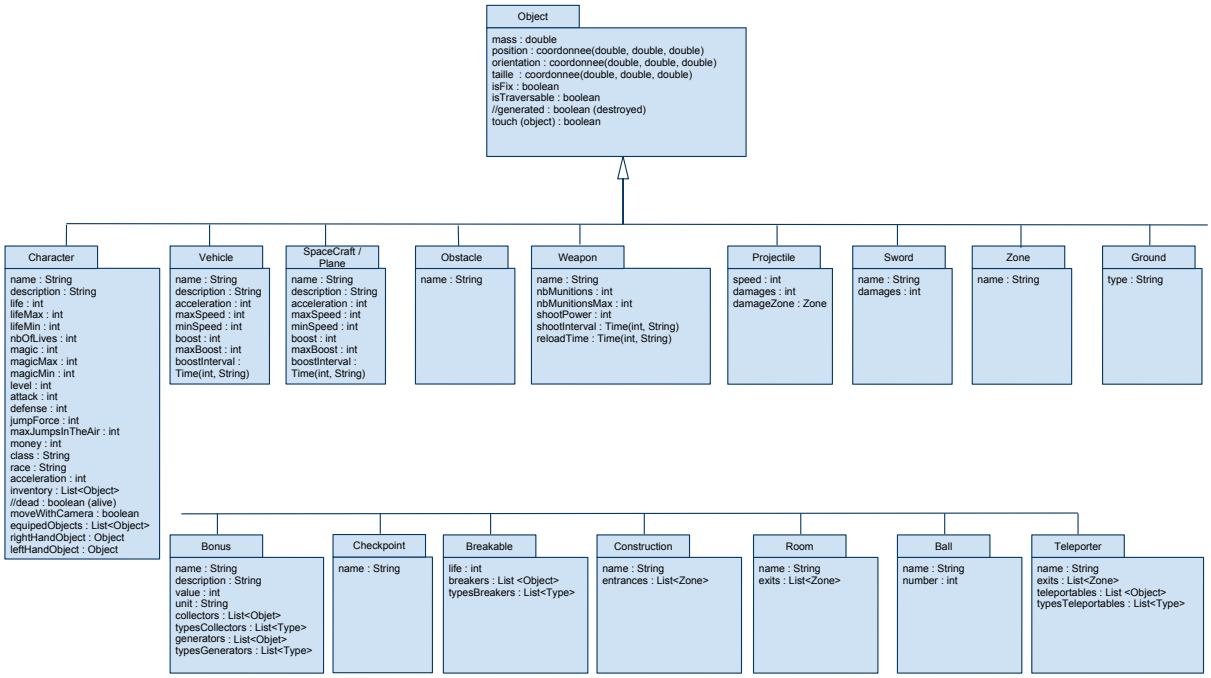
L'analyse d'exemples variés de mini-jeux a permis de mettre en évidence les nombreuses différences qu'il existe entre ceux-ci. Il a toutefois été possible d'en resortir des concepts récurrents. Deux langages ont été proposés, ils permettent à l'utilisateur de décrire simplement un mini-jeu à des niveaux différents. D'un côté, il est possible de décrire simplement les interactions avec l'utilisateur, les entités et les règles du jeu. De l'autre, le comportement de certains objets dans le jeu peut être précisés.

Pour couvrir un plus grand nombre de jeux, il serait possible de définir parallèlement de nouveaux langages, adaptés à d'autres types de jeu. Le compilateur s'adapterait selon le type de jeu à l'une ou l'autre des types de grammaires.

A Grammaire haut-niveau

A.1 La classe Object et ses filles

Pour rappel, la classe object permet de définir toutes les entités du jeu. Tout type définit par l'utilisateur hérite de cette classe ou de l'une de ces classes filles. Elles sont présentées dans le schéma ci-dessous :



Les divers attributs 'name' pourront servir lors de l'affichage d'information. `gandalf` **`is`** *`Character`*. `gandalf` **`has`** *`name`* **`at`** `'Gandalf le Blanc'`.

Les classes Character, Vehicle et Plane/SpaceCraft ont des comportements différents au niveau des commandes.

Pour un objet de type Character, le déplacement se fait directement lors de l'appui sur une touche. Il est possible d'initialiser l'attribut 'acceleration' pour avoir une physique plus réaliste comme Sonic ou Mario. Un attribut 'moveWithCamera' permet d'indiquer si le personnage se déplace en fonction de la caméra courante. Par exemple, indiquer au personnage 'move backward' sans cet attribut le fera reculer. Avec 'moveWithCamera', le personnage avancera vers la caméra.

Pour un ojet de type Vehicle, son délacement se fait uniquement avec son accélération et le fait de tourner dans une direction se fera uniquement si le véhicule n'est pas à l'arrêt.

Enfin pour un objet de type Plane, SpaceCraft, sa masse est nulle s'il avance, et tourner dans une direction signifie effectuer une rotation suivant l'axe que forme sa trajectoire.

Pour les Obstacles, ils héritent des attributs isFix et isTraversable de Object et sont mis respectivement à true et false.

Concernant les Projectiles, ils se déplacent dès qu'ils sont générés et peuvent avoir une zone de dégâts dans le cas d'un missile par exemle.

Une Zone correspond à un volume englobant invisible et traversable par tous.

La classe Ground peut avoir plusieurs types comme la neige, qui a donc un coefficient de frottement faible, ou l'eau qui est traversable.

Un Bonus possède une liste d'objets ou de types d'objets qui peuvent le ramasser. Dès qu'une entité de ce type entre en contact avec le bonus, celui-ci disparaît et a un effet sur elle. Il contient aussi une liste d'objets ou de types d'objets qui peuvent le générer lorsqu'ils disparaîssent.

Le Checkpoint est une Zone particulière : si le joueur meurt, il réapparaît à cet endroit. Le checkpoint peut aussi servir dans un jeu de course pour obliger le joueur à passer à cet endroit (exemple : TrackMania).

Les objets de la classe Breakable correspondent à un Object à 2 états, représenté par 2 fichiers 3D différents. Les deux états correspondent à une vie nulle ou une vie strictement positive. Un exemple concret est une fenêtre. Dès qu'une balle (Projectile ayant un attribut 'damages') la traverse, elle passe dans l'état cassée avec la représentation 3D associée.

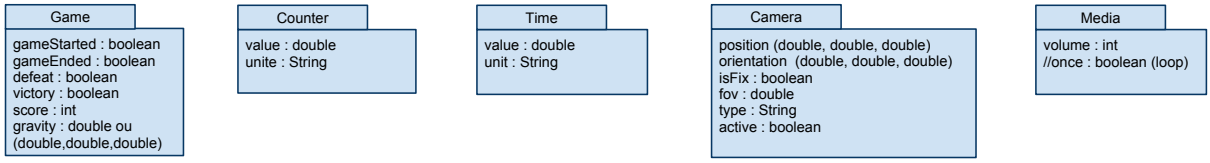
Une Construction correspond à une vue extérieure d'un bâtiment, d'une maison... Ce type d'objet est associé à Room pour des jeux de rôles ou il n'est pas nécessaire de représenter une maison à sa taille réelle pour y entrer ensuite. Une liste d'entrées correspond à des zones de téléporteurs associés aux sorties de Room le plus souvent.

Un objet de type Room reprend le principe de Construction mais les collisions sont gérées via des plans car un personnage à l'intérieur est considéré comme déjà en collision. Elle contient des sorties associées aux entrées d'une Construction ou d'une autre Room.

Un objet de type Ball a un volume englobant sphérique.

A.2 Les autres classes

Ci-dessous se trouvent les autres classes prédéfinies pour la grammaire haut-niveau.



La classe Game ne peut être instanciée qu'une seule fois.

Les ressources temporelles peuvent avoir plusieurs unités telles que les secondes, les millisecondes, les minutes ou les frames.

On peut définir la caméra selon un suivi à la première personne ('firstPerson') ou à la troisième personne ('thirdPerson') par rapport à l'objet qui aura été défini comme étant 'player'. On peut aussi ne rien définir ou mettre 'free' pour une caméra libre.

Un Media a des mots-clefs dédiés comme play, mute on, mute off, pause ou stop. On peut choisir entre un Media répété en boucle ('loop') ou lu une unique fois ('once').

A.3 La grammaire

L'intégralité de la gramaire haut-niveau est maintenant présenté.

```
/*------------------------------------------------------------------
* PARSER RULES
*------------------------------------------------------------------*/

jeu :
  (infosJeu '.')?
  (nouveauType '.')*
  (init '.')+
  (definition '.')*
  (commande '.')*
  (reglesJeu '.')*
  (iaBasique '.')*;


////////////////////////////// ( informations about the game)  ////////////////////////////////

infosJeu :
  'Game' 'has' ('gravity' | 'score' /*| ...*/ ) 'at' (NUMBER | NUMBER NUMBER NUMBER)
  ;


////////////////////////////// ( Inheritance )  //////////////////////////////

nouveauType :
  'type' ident 'is' (ident | typeObjet) ('and' ident | typeObjet)* // to declare a new type
  ;

// ident | typeObjet : if it is an ident, check that it is defined before by the user and that is an
     inherited Object.
```

```
//////////////////////////// ( Initializations )  ////////////////////////////

init :
  ident 'is' declarationObjet
  | accesClasse 'has' affectationObjet (',' affectationObjet)* // check the types and its attributes
  ;

declarationObjet :
  (ident | typeObjet3D) ('player' | interaction ('duplicable')? )?         // interaction is neutral by
      default
  | 'list' ('of' (operation)? (ident) ('with' (operation)? (ident))* )?  //operation if the object is
      duplicable
  | 'Camera' (('first' | 'third') 'person' | 'free')?
  | 'Media' ('loop' | 'once')?                                          // sound, music or video played in
       loop or once
  | 'in' ident                                                  // ident of a list to add an element
;

interaction :
  'ally' | 'enemy' | 'neutral'
  ;

affectationObjet :
  ident ('at' (operation (uniteTps)? | ident) )?       //aggregation
  | attribut 'at' (operation | STRING | BOOL)          //life at 5, name at "Gandalf Le Gris"
  | typeCoordonnees 'at' coordonnees            //size at 20 30 40
  | attributListeOuObjet 'at' ident             //inventory at listeArmesJoueur
  | attributTps 'at' operation uniteTps         //
  ;

// has ident at ... : to declare a new attribute
// Attributes of predefined class have default initialized
// it is not necessary to initialize not used attribute

typeObjet :
  'Camera'
  | 'Media'
  | 'Counter'
  | 'Time'
  | typeObjet3D
  ;

// every predefined classes
typeObjet3D:
  'Object'                      // -> position(x,y,z), orientation(x,y,z), size(x,y,z)
  | 'Character'                 // -> life, lifeMax, magic, magicMax , level, experience, attack, defense
  | 'Vehicle'                   // -> acceleration, speedMax,
  | 'Plane' | 'SpaceCraft'
  | 'Obstacle'                  // a fixed entity, used for collisions
  | 'Weapon'                    // -> nbMunitions, nbMaxMunitions, intervalleTirs, timeRecharge
  | 'Sword'                     // -> damages, level
  | 'Projectile'                // -> vitesse, damages, level(pourquoi pas)
  | 'Zone'                      // an invisible and traversable entity
  | 'Ground'                    // -> type of ground (water, snow ...)
  | 'Bonus'                     // an object which disappears when something touches it-> valeur(entier),
      nomObjet(type),listeObjets
  | 'CheckPoint'
  | 'Breakable'
  | 'Construction'
  | 'Room'
  | 'Ball'
  | 'Teleporter'
  ;

// every attributes of predefined classes
attribut :
  'mass'                  // attributes of object :
  | 'isFix'
  | 'isTraversable'
  | 'fov'                     // attributes of "camera"
  | 'type'
  | 'active'
  | 'name'                    // attributes of "character" :
  | 'description'
  | 'life'
  | 'lifeMax'
  | 'lifeMin'
  | 'nbOfLives'
  | 'magic'
  | 'magicMax'
  | 'magicMin'
  | 'level'
```

```
  | 'attack'
  | 'defense'
  | 'jumpForce'
  | 'maxJumpsInTheAir'
  | 'money'
  | 'class'
  | 'race'
  | 'acceleration'
  | 'speed'                 // attributes of "vehicle" :
  | 'maxSpeed'
  | 'minSpeed'
  | 'boost'
  | 'maxBoost'
  | 'nbMunitions'           // attributes of"weapon" :
  | 'nbMunitionsMax'
  | 'shootPower'
  | 'damages'               //attributes of "projectile"
  | 'value'                 // attributes of "bonus" :
  | 'unit'
  | 'objectname'
  | 'attributName'
  | 'volume'                  //attributes of "media"
  | 'number'               //attributes of "ball"
  | 'moveWithCamera'
  ;

attributListeOuObjet :
  'inventory'
  | 'equipedObjects'
  | 'entrances'
  | 'exits'
  | 'damageZone'
  | 'collectors'
  | 'typesCollectors'
  | 'generators'
  | 'typeGenerators'
  | 'breakers'
  | 'typesBreakers'
  | 'teleportables'
  | 'typesTeleportables'
  ;

attributTps :
  'boostInterval'
  | 'shootInterval'        //attributes of "weapon" :
  | 'reloadTime'
  ;


//////////////////////////// ( new definitions of actions ) ////////////////////

definition : 'definition' ident 'means' consequences;

consequences :
  consequ (',' consequ)*
  ;

consequ :
  siAlors
  | action
  | affectation
  | activCommande
  | appelDef
  | 'victory'
  | 'defeat'
  ;

appelDef :
  ident           //ident of a definition of an action (means)
  ;

activCommande :
  ('activate' | 'disable') ('commands' | 'mouse' (souris (',' souris)*)? | 'key' clavier (',' clavier)* | '
      keyboard' )
  ;
//disable commands              // all the commands
//disable key                   // all the key commands
//disable mouse up, down        // only move up or down with the mouse


//////////////////////////// ( Initialization of commands )  ////////////////////////////

commande :
```

```
  'command' (ident 'is' actionCommande (',' actionCommande)* | actionCommande)
  ;

actionCommande :
  ('mouse' souris | 'key' clavier) 'for' (ident | actionCommandePressee | actionCommandeMaintenue) // ident
       : what is defined with means
  ;

souris :
       'up' | 'down' | 'left' | 'right' | 'lClick' | 'cClick' | 'rClick' | 'scrollUp' | 'scrollDown'
       ;

clavier :
       CHAR | 'up' | 'down' | 'left' | 'right' | 'space' | 'echap' | 'enter'          //CHAR : Z,Q,S,D,...
       ;

actionCommandePressee :
  'jump' operation
  | 'pause'
  | 'stop'
  ;
actionCommandeMaintenue :
  'move' ('left' | 'right' | 'forward' | 'backward')
  | 'turn' ('left' | 'right')
  | 'accelerate'
  | 'brake'
  ;


//////////////////////////// ( Rules of the game + conditions of victory / defeat )
    ////////////////////////////

reglesJeu :
  'rule' (ident 'is')? declencheur 'then' consequences ','
  ;
conditions :
  conditionEt ('or' conditionEt)*
  ;

conditionEt :
  conditionsNot ('and' conditionsNot)*
  ;

conditionsNot :
  'not' cond
  ;

cond :
  '(' conditions ')'
  | etat
  | operation comparaison operation
  ;

etat :
  accesClasse 'is' ('not')? ('dead' | 'alive' | 'effaced' | 'generated' | 'touching' (('other')?
       accesGlobal | accesLocal))  // for an object
  | (ident | 'game') 'is' ('not')? ('finished' |'started' | 'paused' | 'muted' ('on' | 'off') | 'played' |
       'stopped' )  // game,counter,media
  | 'true'
  | 'victory'
  | 'defeat'
  ;

declencheur :
  accesClasse ('dies' | ('touches' | 'kills') (('other')? accesGlobal | accesLocal) | ('killed' | 'touched'
       ) ('by' (('other')? accesGlobal | accesLocal))? )
  | (ident | 'game') ('ends' |'starts')          //ident if it is a counter
  | variable 'becomes' varOuNb
  | ident 'becomes' ('player' | interaction)
  ;

siAlors :
  'if' conditions 'then' consequences ('else' consequences)? 'end'
  ;


action :
  accesClasse actionObjet
  | (ident | 'game') ('ends' |'starts')
  | ('pause' | 'mute' ('on' | 'off') | 'play' | 'stop' ) ident
  | 'block' transformation 'of' accesClasse coordonnees
  | ('efface' | 'generate') (accesClasse | operation accesClasse ('in' accesLocal | 'on' accesLocal | 'at'
       coordonnees)?)
```

```
  | 'wait' operation uniteTps 'then' consequences 'endWait'
  | 'save'
  ;

actionObjet :
  'dies'
  | actionCommandePressee
  | actionCommandeMaintenue ('during' operation uniteTps | 'until' conditions)
  | 'equip' (accesLocal | 'next' | 'previous')
  ;

affectation :
  (('assign' | 'add' | 'sub') operation) 'for' variable | 'invert' variable 'with' variable
  ;

//add : a += b, remove : a -= b, assign : a = b, invert : tmp=a; a=b; b=tmp;

coordonnees :
  operation operation operation
  ;

comparaison :
  '=' | '<' | '>' | '<=' | '>='
  ;

transformation :
  'translation'
  | 'rotation'
  | 'scale'
  ;

uniteTps :
  'mn'
  | 'sec'
  | 'ms'
  | 'frames'
  ;

operation :
  ('random' 'between' operationPlus 'and')? operationPlus
  ;

operationPlus :
  operationMul (operateurPlus operationMul)*
  ;
operationMul :
  operationPuiss (operateurMul operationPuiss)*
  ;

operationPuiss :
  operationparentheses ('^' operationparentheses)*
  ;

operationparentheses :
  '(' operation ')'
  | varOuNb
  ;

varOuNb :
  variable
  | NUMBER
  ;

variable :
  (('x' | 'y' | 'z') 'of' (typeCoordonnees | ident | attribut)) 'of' accesClasse
                      //x of size of num 10 in listeWeapon
  ;

accesClasse : accesLocal | accesGlobal;

accesGlobal :
  typeObjet
  | interaction
  | 'not' (typeObjet | interaction | 'player')
  | 'all'
  ;

accesLocal :
  ident
  | 'num' operation 'in' ident
  | 'player'
  ;
```

```
typeCoordonnees :
  'positition' | 'rotation' | 'scale'
  ;

operateur :
  operateurMul
  | operateurPlus
  ;

operateurMul :
  '*'
  | '/'
  | '%'
  ;

operateurPlus :
  '+'
  | '-'
  ;

ident :
  STRING
  ;

//////////////////////////// ia ////////////////////////

iaBasique : 'ia' accesClasse 'is' actionObjet (',' actionObjet)*;


/*------------------------------------------------------------------
* LEXER RULES
*------------------------------------------------------------------*/
BOOL        : ('true' | 'false');
STRING        :  ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | '0'..'9')+ ;
CHAR   : 'a'..'z' | 'A'..'Z';
NUMBER : ('0'..'9')+ (',' ('0'..'9')+ )? ;
WS  :   ( ' '
            | '\t'
            | '\r'
            | '\n'
            | '\u000C'
            )+ {$channel=HIDDEN;}
        ;
```

B Grammaire bas-niveau

Cette partie présente la totalité de la grammaire bas-niveau

```
 Game ::= refreshLoop eventsManager ressourcesSets camera entities physicsEngine

refreshLoop ::= signalUpdateCounter keyListener{ keyboardCommands } mouseListener{ mouseCommands }
//signalUpdateCounter is triggered by the refreshLoop and updates counters of type times

// pressing a key or action with mouse triggers an event
keyboardCommands ::= (keystroke : signalSets)*

mouseCommands ::= (typeOfClick : signalSets)*

signalSets ::= signal (| signal)*

-----------------------------------------------------------------------------------------------------------
// Events Managers

eventsManager ::= signal (@ signal)* (instructions)+ (| signal (@signal)* (instructions)+)*

instructions ::=
                | resourceApply resource (expression | applyExpression ) // expression pour resource =
                    expression ; applyExpression pour resource += expression
                | if conditonnal then instructions (& instructions)* (else instructions (& instructions)*)
                    ?
                | conceptsInstructions
                | mediaName mediaState

conceptsInstructions ::= gamOver | pause | newGame | saveGame

conditionnal ::= testExpression (booleanOperator testExpression)*

booleanOperator ::= and | or
```

```
testExpression ::= expression comparisonOperator expression

comparisonOperator ::= < | > | <= | >= | == | !=

applyExpression ::= arithmeticOperator (metaExpression | expression)

// to reuse an expression without writing it
metaExpression ::= nameExp expression

nameExp ::= string
expression ::= (arithmeticOperator value)+
arithmeticOperator ::= + | - | * | / | %


//random(value,value) to generate a random number between these integers
value = ressource | constant | random(value, value) | random(0, value)
constant ::= int | double

-----------------------------------------------------------------------------------------------------------
// Resource Manager

// An event tiggers when a resource is modified
resourcesSets ::= (enumResource | resource)+


enumResource ::= name { nameEnumResource (, nameEnumResource)*}
nameEnumResource ::=  String

resource ::= (# nameEnumResource)? name (signal (@ signal)*)? (timer|initValue)

name ::= string

timer ::= step initTimer

initValue ::= int | double

-----------------------------------------------------------------------------------------------------------
// camera and entites manager


camera ::= name position

position ::= vector | angle

entities ::= map with object+

// With a map like a points matrix with a unique texture
map ::= matrix texture


object ::= object = (name object3D parameters | media)
parameters ::= coeffOfFriction = double weight = double speed = vector position = vector  isFixed = boolean
      isTraversable = boolean
object3D ::= colladaFile boundingBox
colladaFile ::= [a-zA-Z]*.dae

media ::= mediaName soundLevel mediaState
mediaName ::= string
mediaStat ::= isMute | isPlayed | isStopped | isPaused

-----------------------------------------------------------------------------------------------------------
// physics engine

physicsEngine ::= forces+ collision
forces ::= gravity | wind |...

// Bounded box are provided by the user
collision ::= collision{ name name signalSets (, name name signalSets)*}
gravity ::= gravity = vector
wind ::= wind = vector
```

C Diagrammes UML des mini-jeux

non inclus : 2 diagrammes de séquences du jeu de volley et 2 de WatchNDroid, comme ca c'est joli, ca fait 2 pages par jeu. Pour les remettre juste à décommenter

Cette partie présente les diagrammes UML pour les mini-jeux exposés dans la seconde partie.

Table des figures

1 Pacman : Diagramme de classes . . . 23
2 Pacman : Diagramme de cas d'utilisation . . . 24
3 Pacman : Diagrammes de séquence . . . 24
4 1942 : En haut, diagramme des cas d'utilisation ; en bas, diagramme de classes . . . 25
5 1942 : Diagrammes de séquences . . . 26
6 Volley : En haut, diagramme des cas d'utilisation ; en bas, diagramme de classes . . . 27
7 Volley : Diagrammes de séquences . . . 28
8 Course : En haut, diagramme d'activité ; en bas, diagramme de classes . . . 29
9 Course : Diagramme de séquence . . . 30
10 Mario : Diagramme de cas d'utilisation . . . 31
11 Mario : Diagramme de classes . . . 32
12 Mario : Diagramme de séquences . . . 32
13 Watch'N'Droid : En haut, diagramme des cas d'utilisation ; en bas, diagramme de classes 33
14 Watch'N'Droid : Diagrammes de séquences . . . 34
15 Billard : En haut, diagramme des cas d'utilisation ; en bas, diagramme de classes . . . 35
16 Billard : Diagrammes de séquences . . . 36
17 Gestion : Diagramme de cas d'utilisation . . . 37
18 Gestion : Diagramme de classes . . . 38

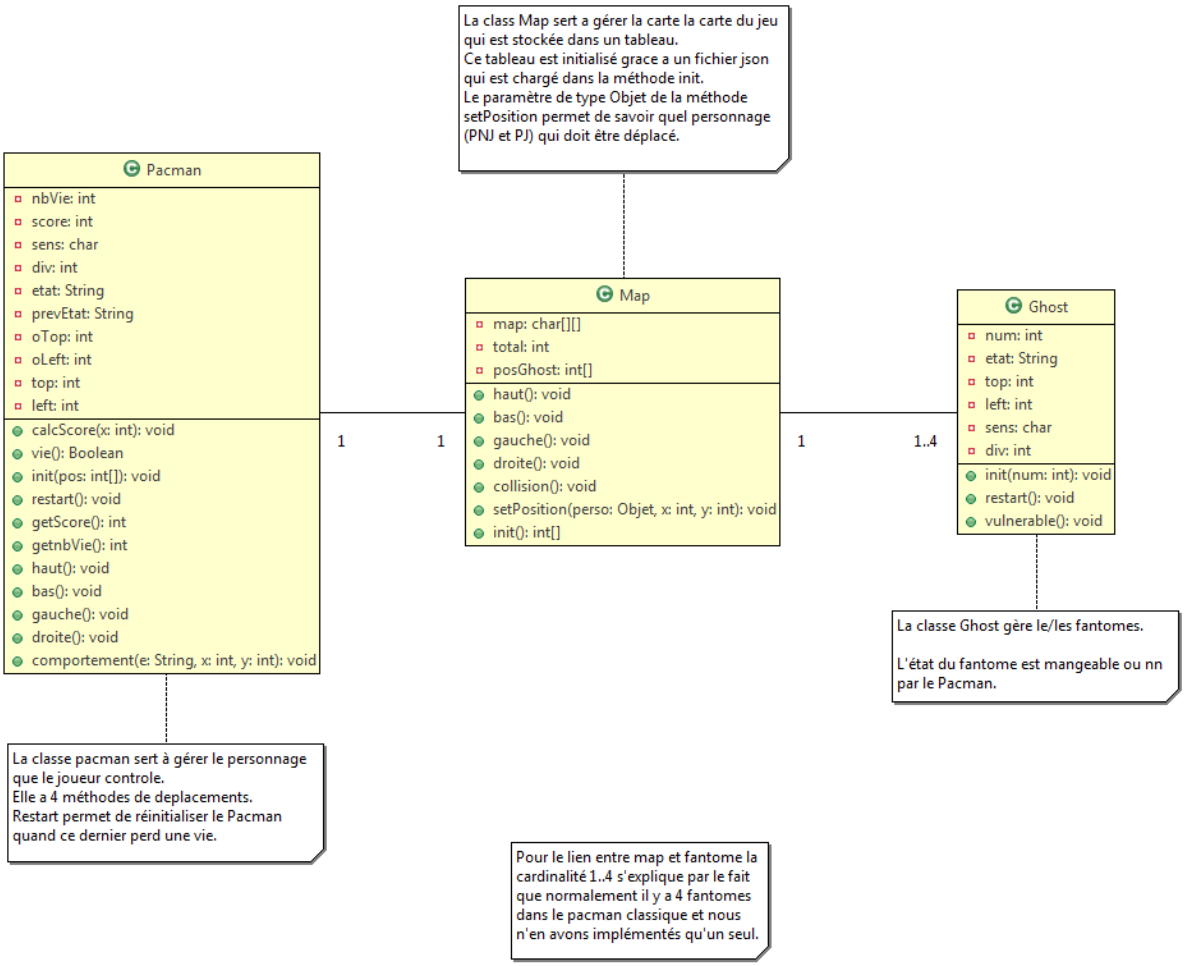


FIGURE 1 – Pacman : Diagramme de classes

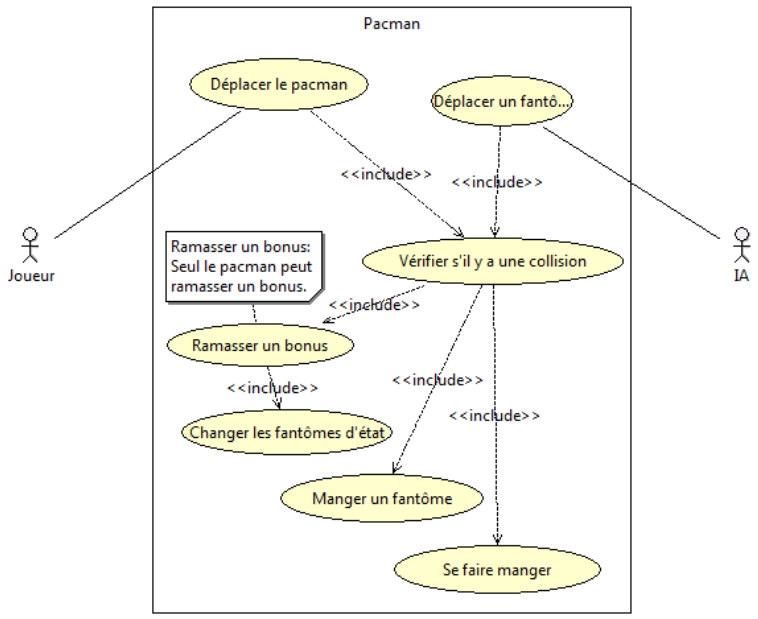


FIGURE 2 – Pacman : Diagramme de cas d'utilisation

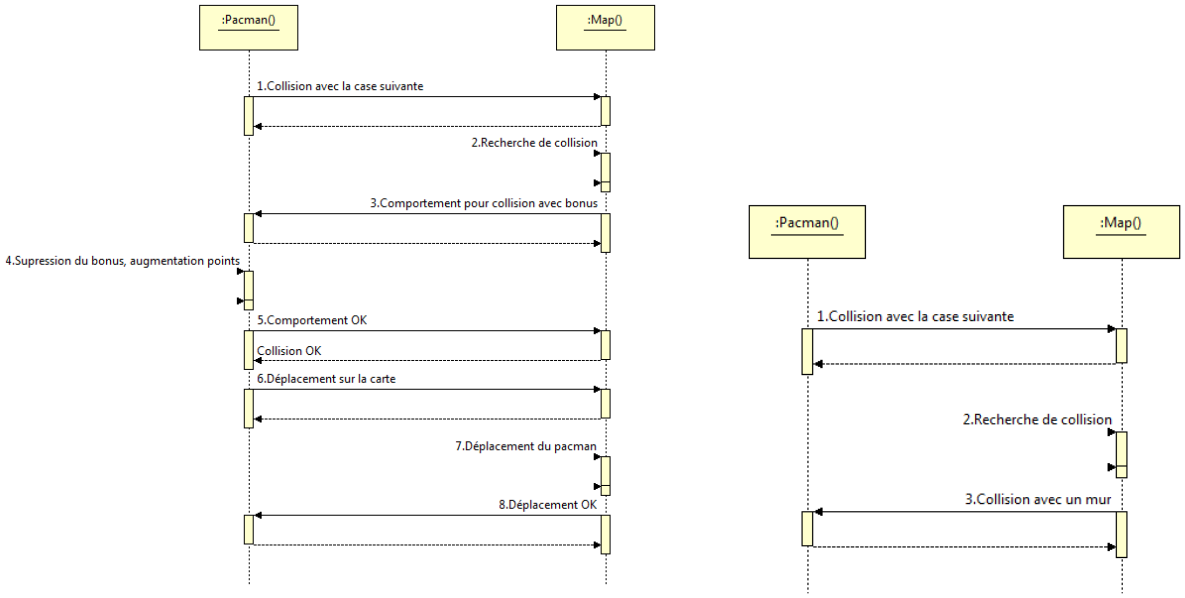


FIGURE 3 – Pacman : Diagrammes de séquence

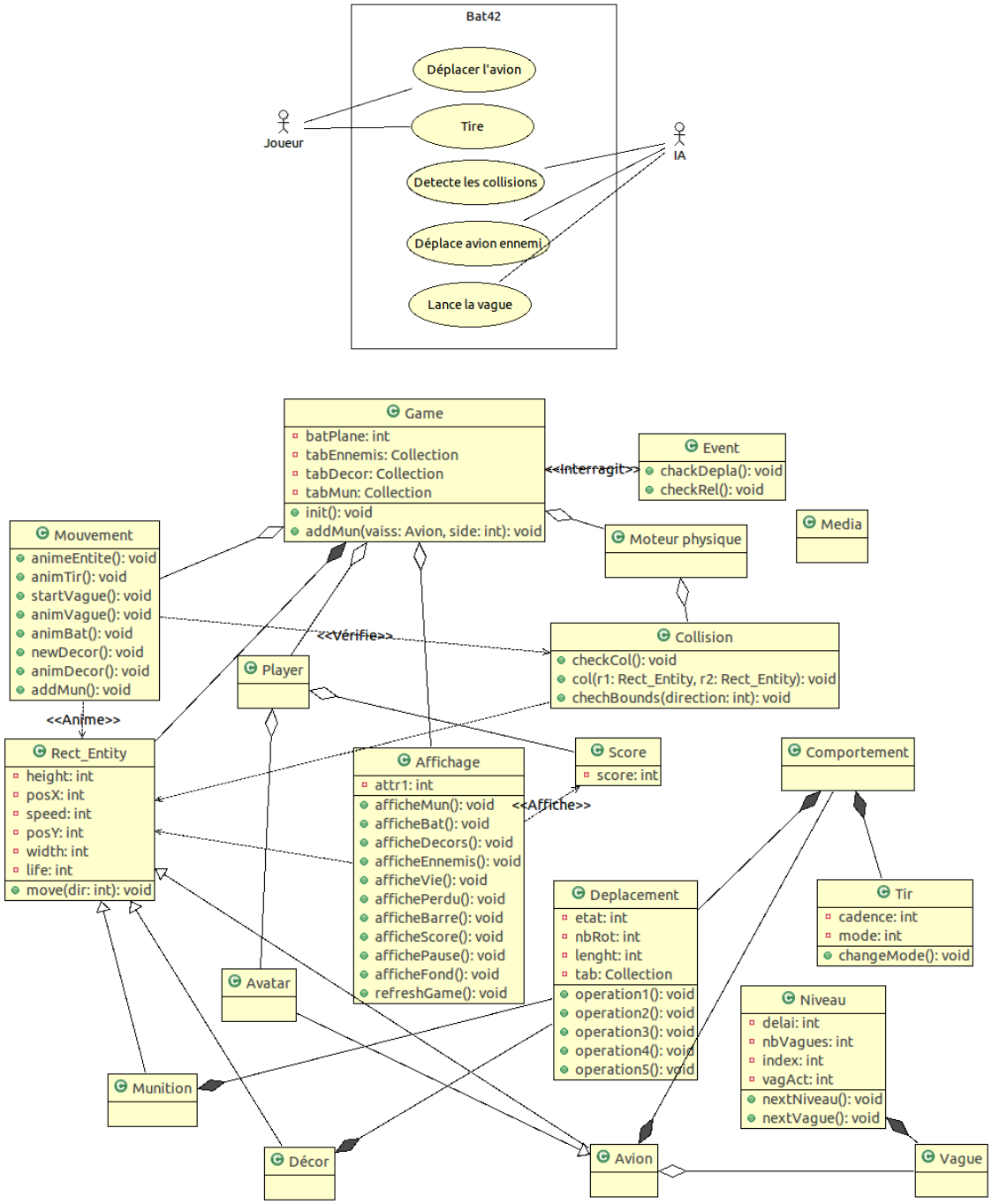


FIGURE 4 – 1942 : En haut, diagramme des cas d'utilisation ; en bas, diagramme de classes

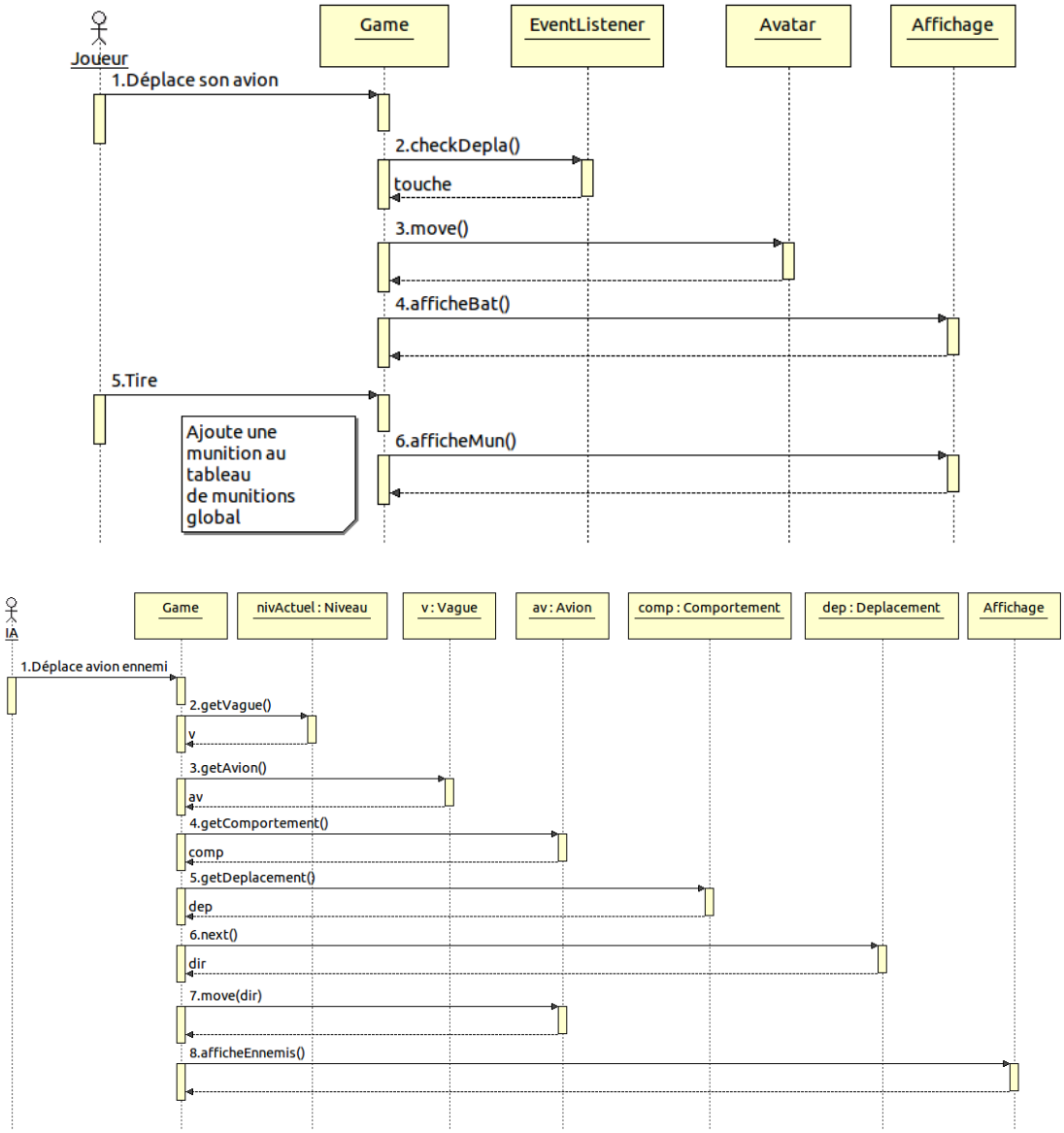


FIGURE 5 – 1942 : Diagrammes de séquences

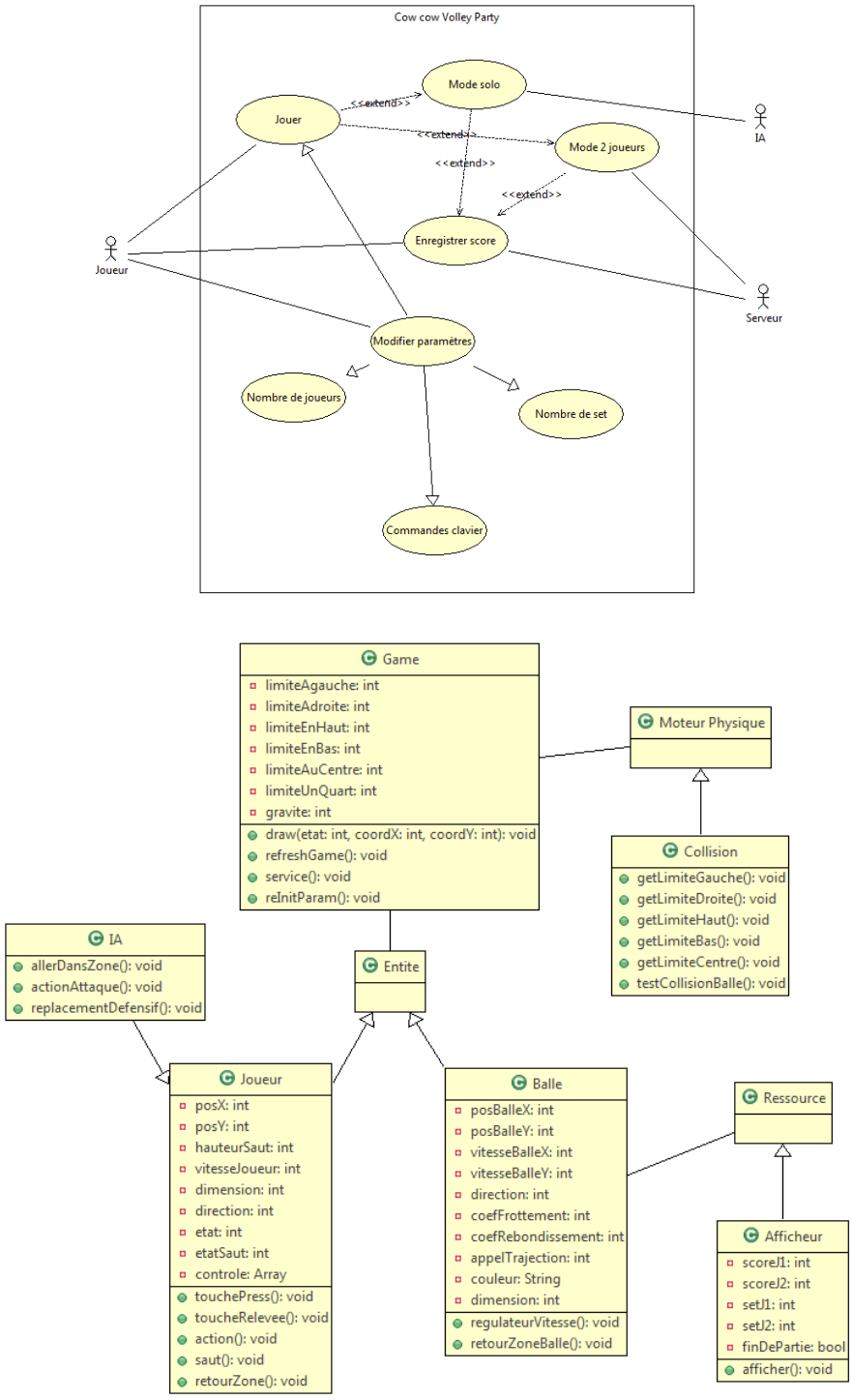


FIGURE 6 – Volley : En haut, diagramme des cas d'utilisation ; en bas, diagramme de classes

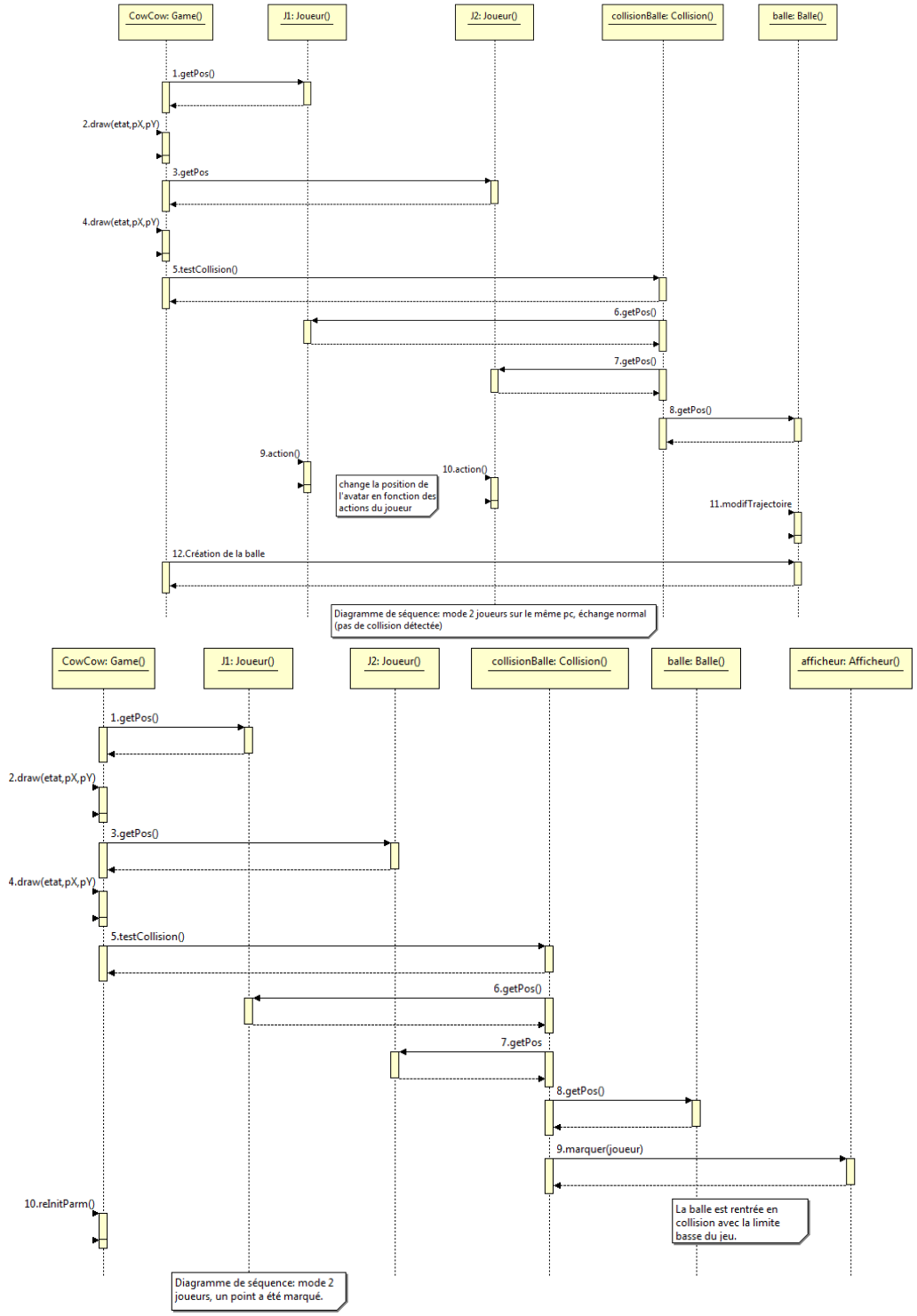


FIGURE 7 – Volley : Diagrammes de séquences

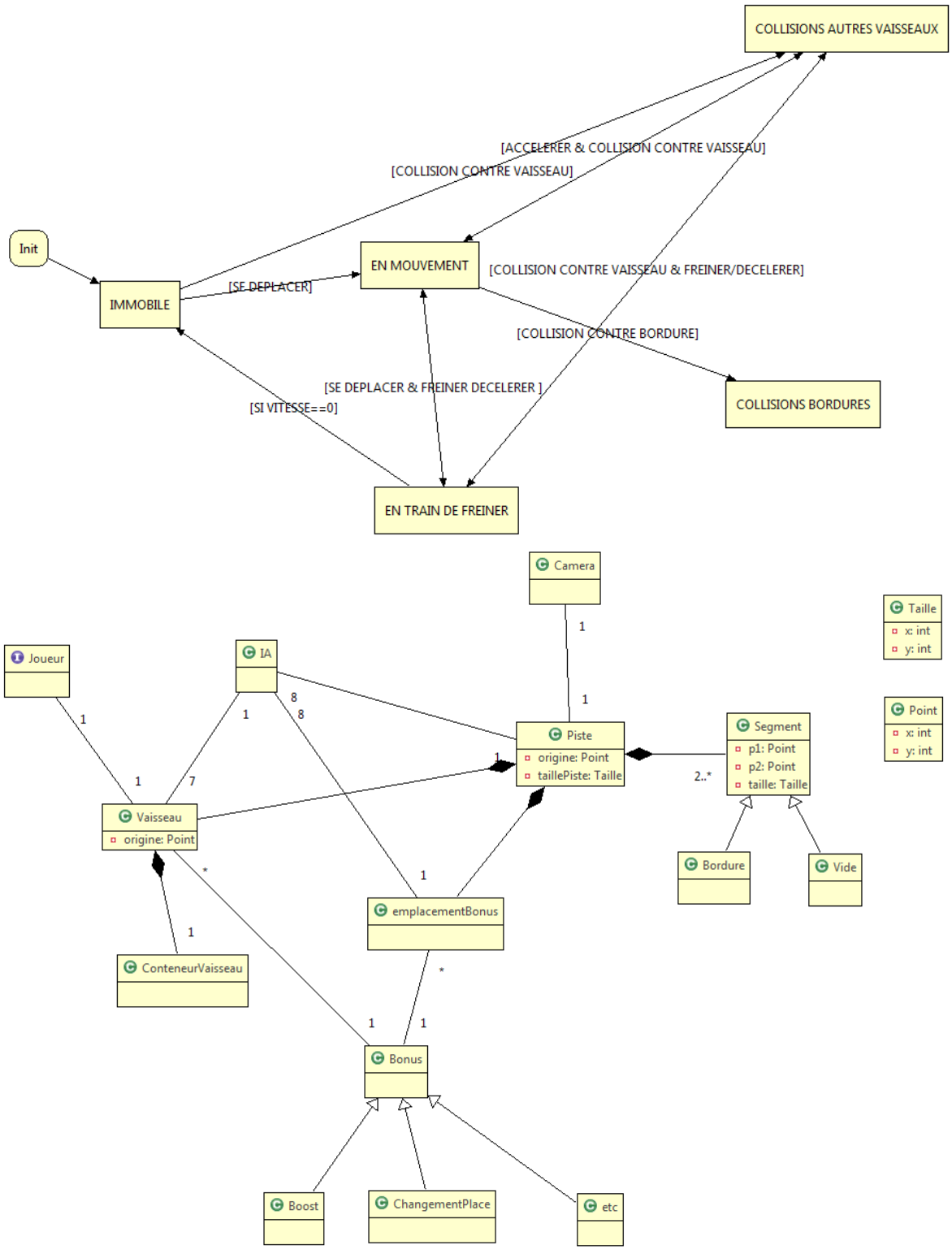


FIGURE 8 – Course : En haut, diagramme d'activité ; en bas, diagramme de classes

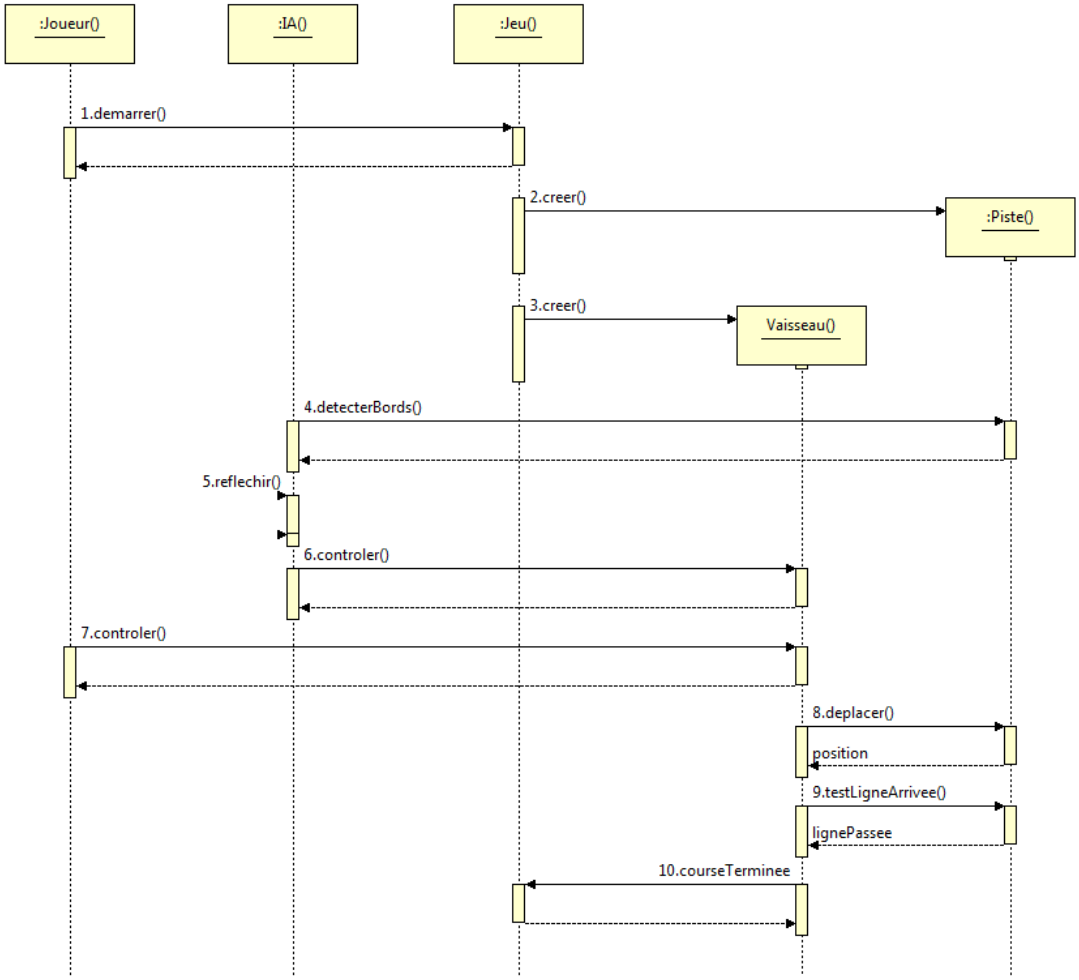


FIGURE 9 – Course : Diagramme de séquence

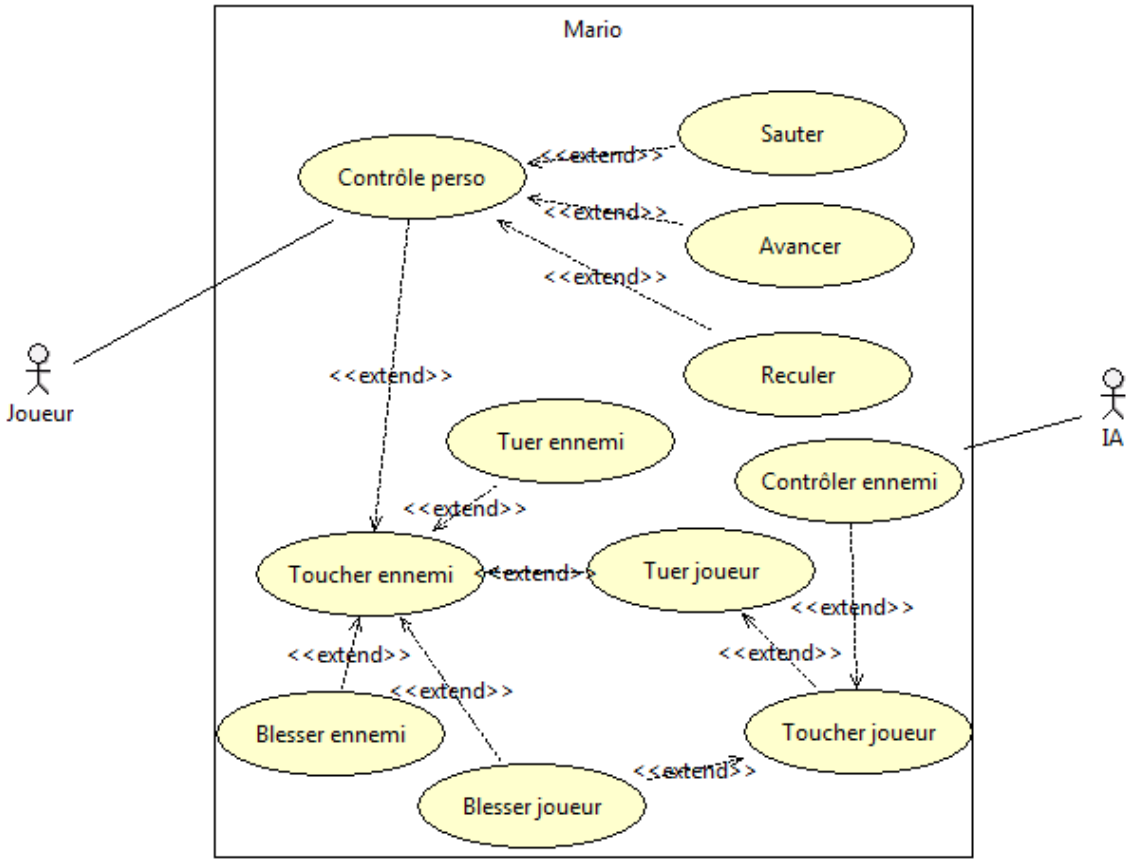


FIGURE 10 – Mario : Diagramme de cas d'utilisation

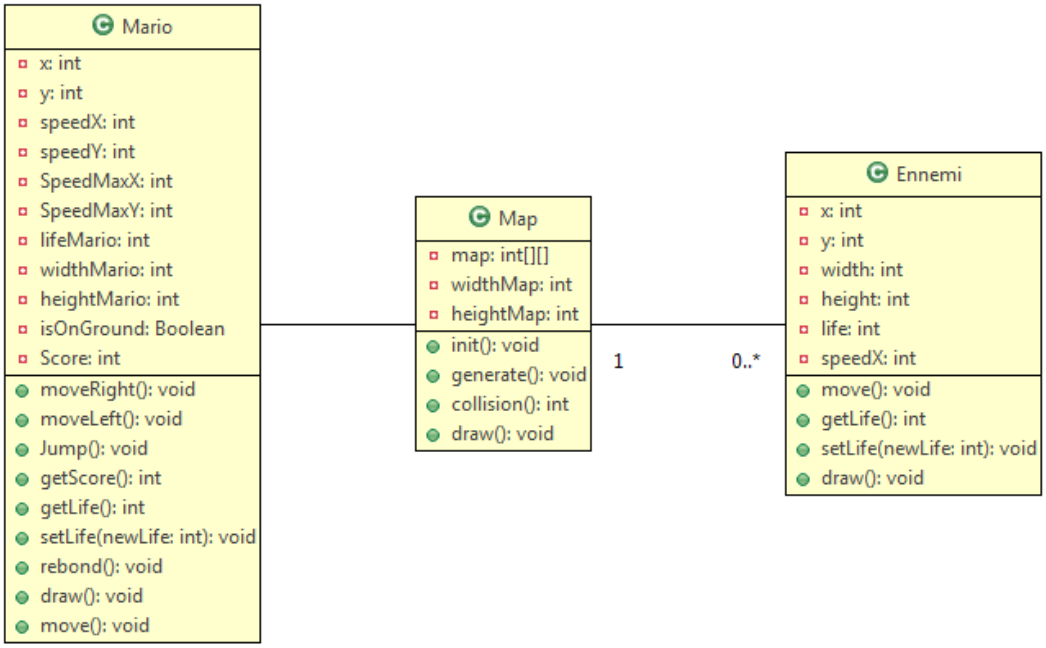


FIGURE 11 – Mario : Diagramme de classes

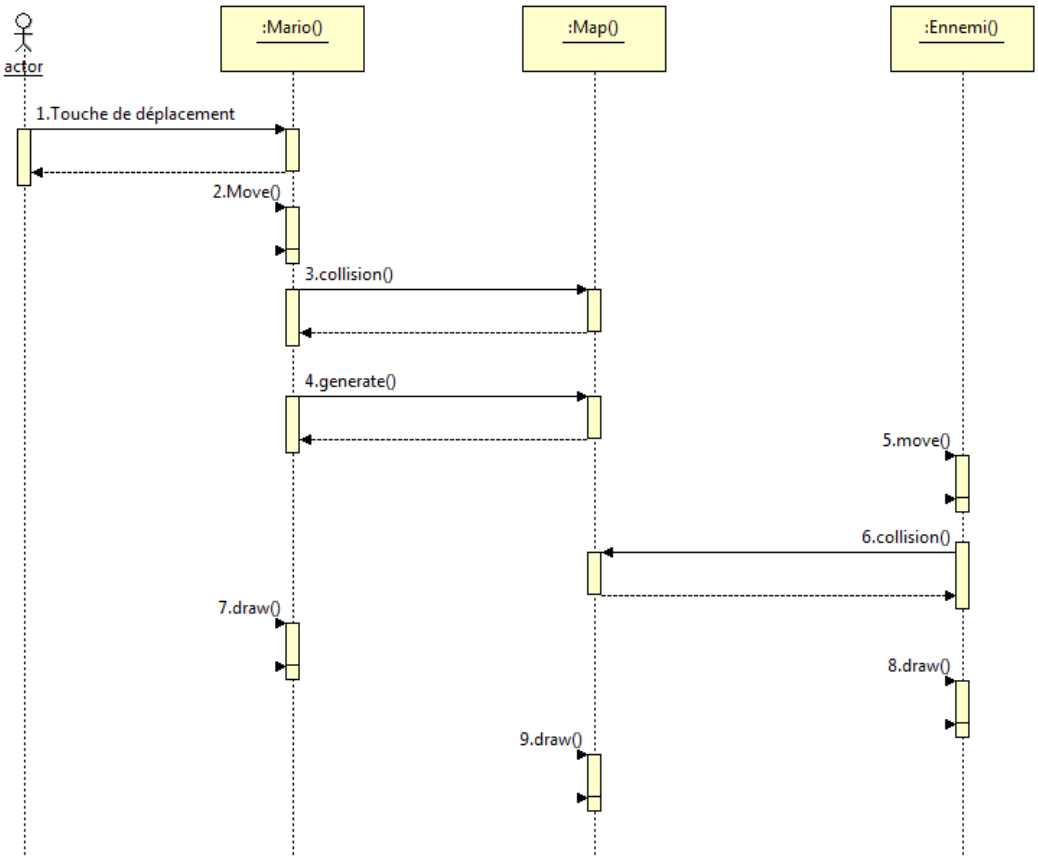


FIGURE 12 – Mario : Diagramme de séquences

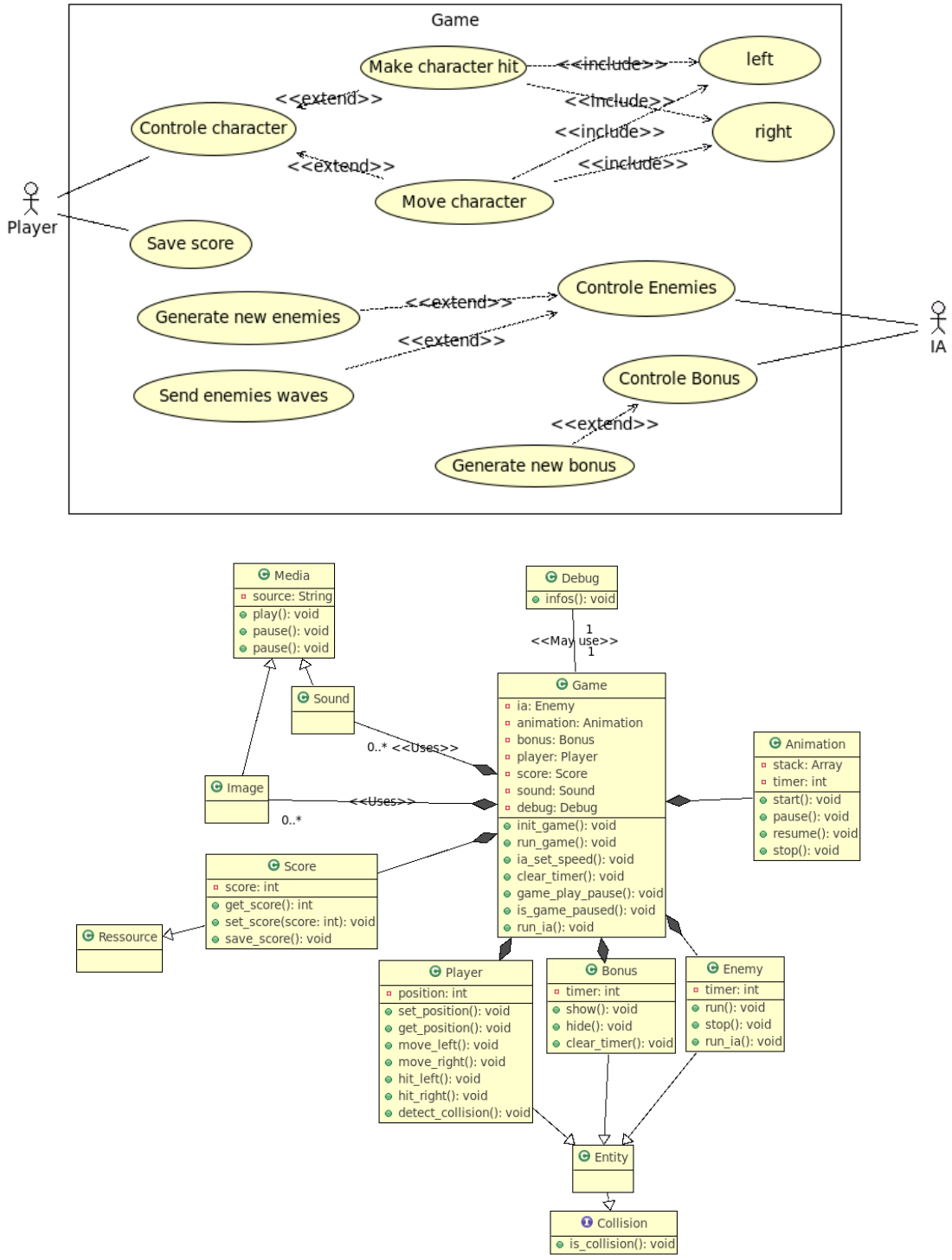


FIGURE 13 – Watch'N'Droid : En haut, diagramme des cas d'utilisation ; en bas, diagramme de classes

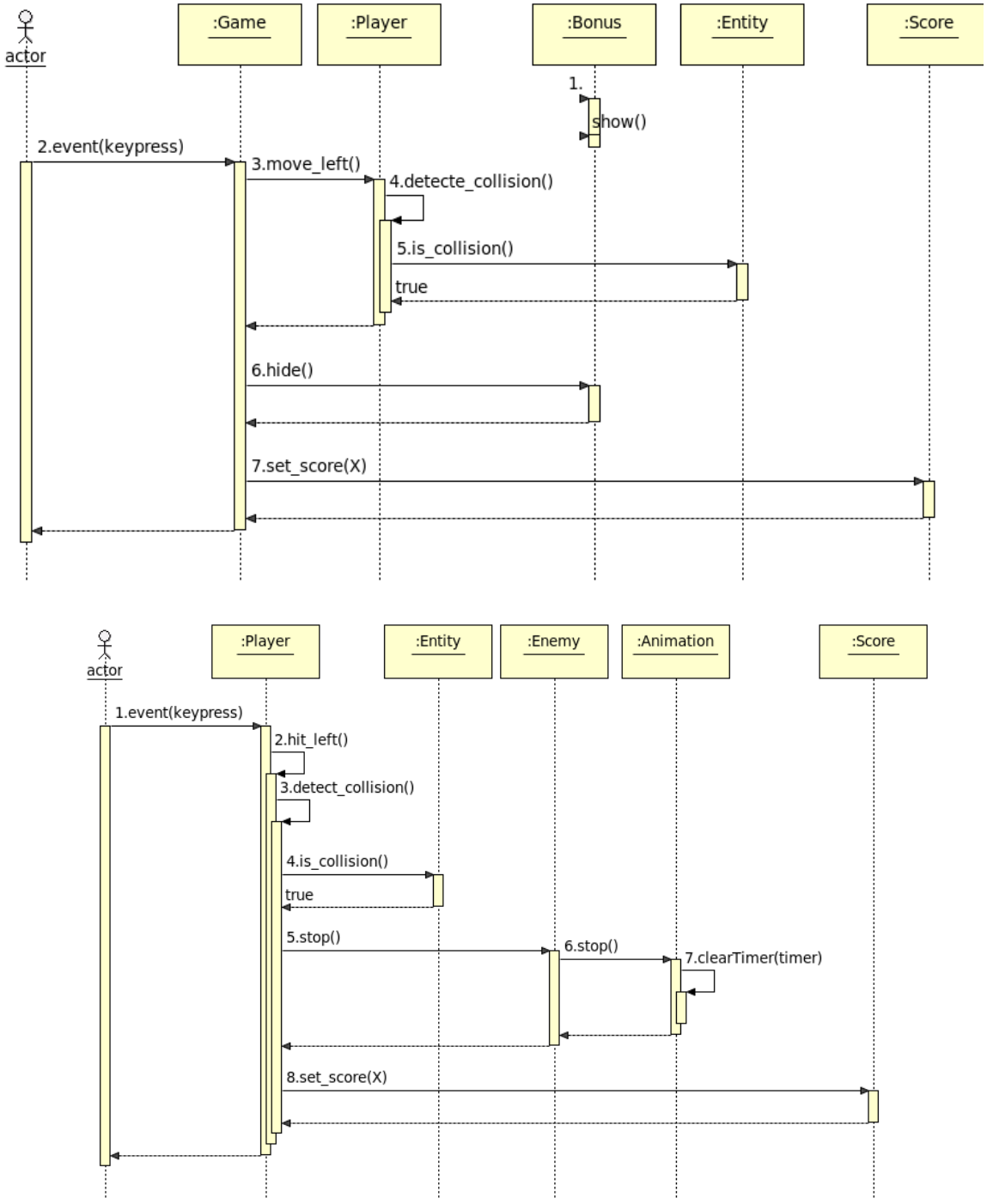


FIGURE 14 – Watch'N'Droid : Diagrammes de séquences

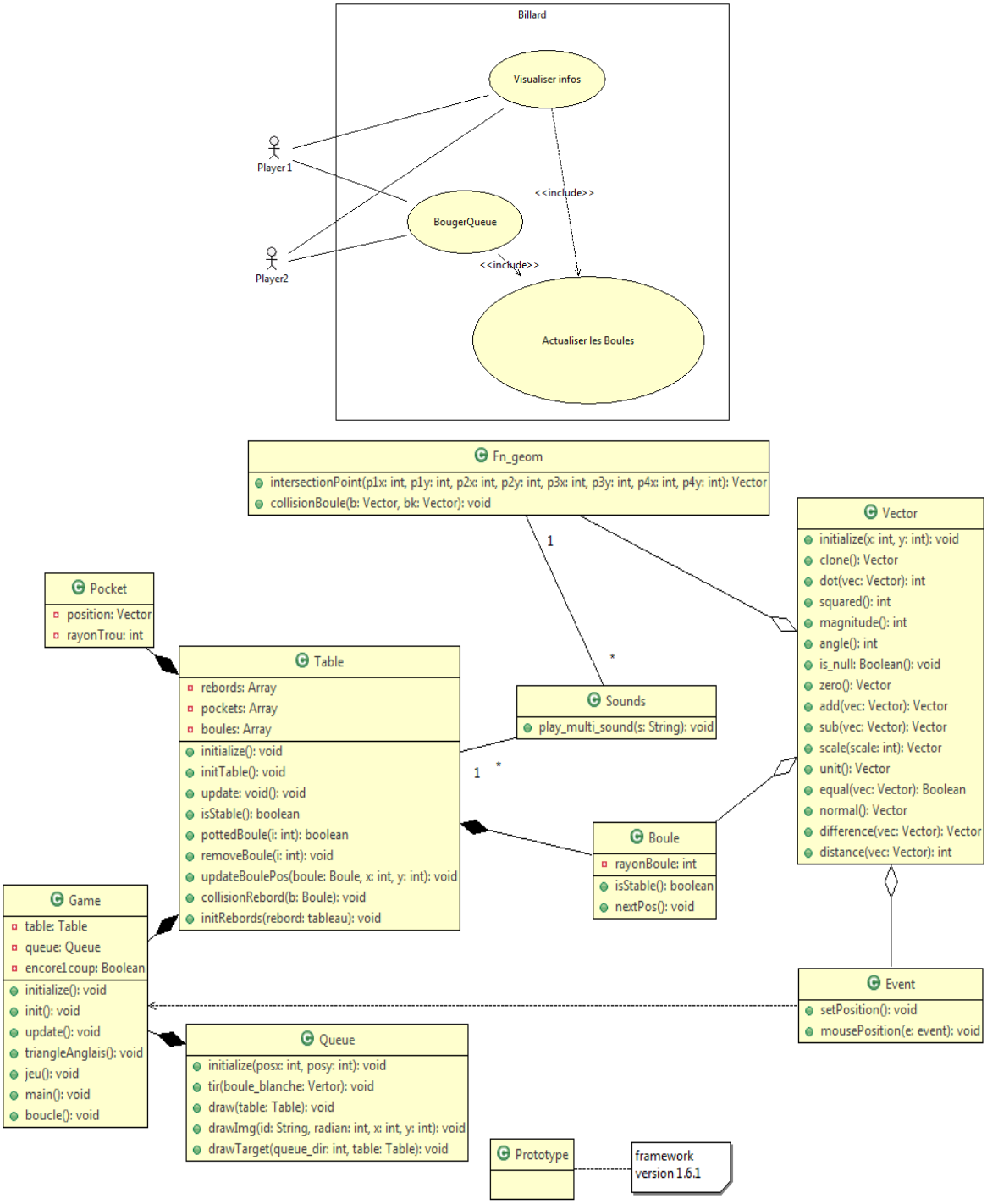


FIGURE 15 – Billard : En haut, diagramme des cas d'utilisation ; en bas, diagramme de classes

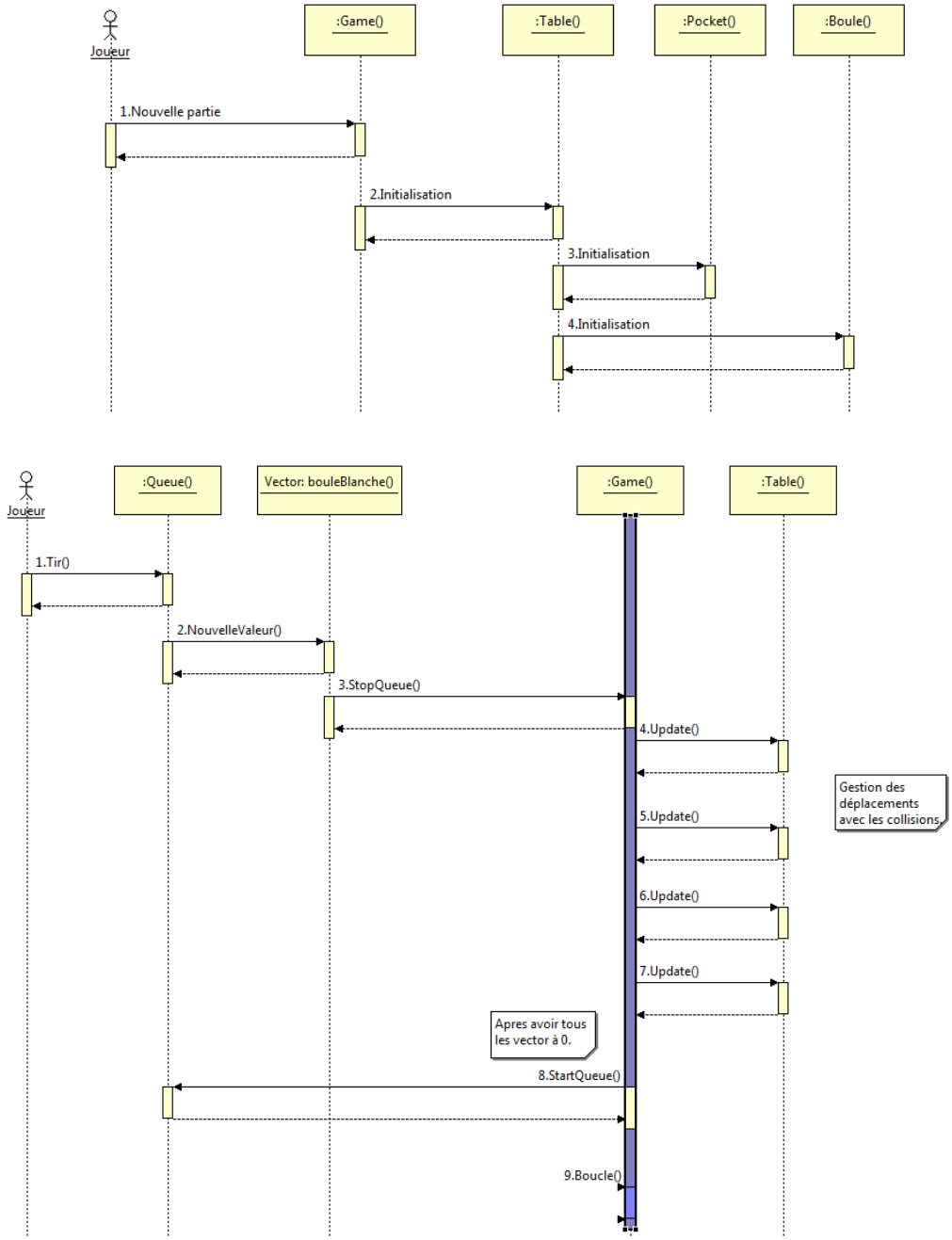


FIGURE 16 – Billard : Diagrammes de séquences

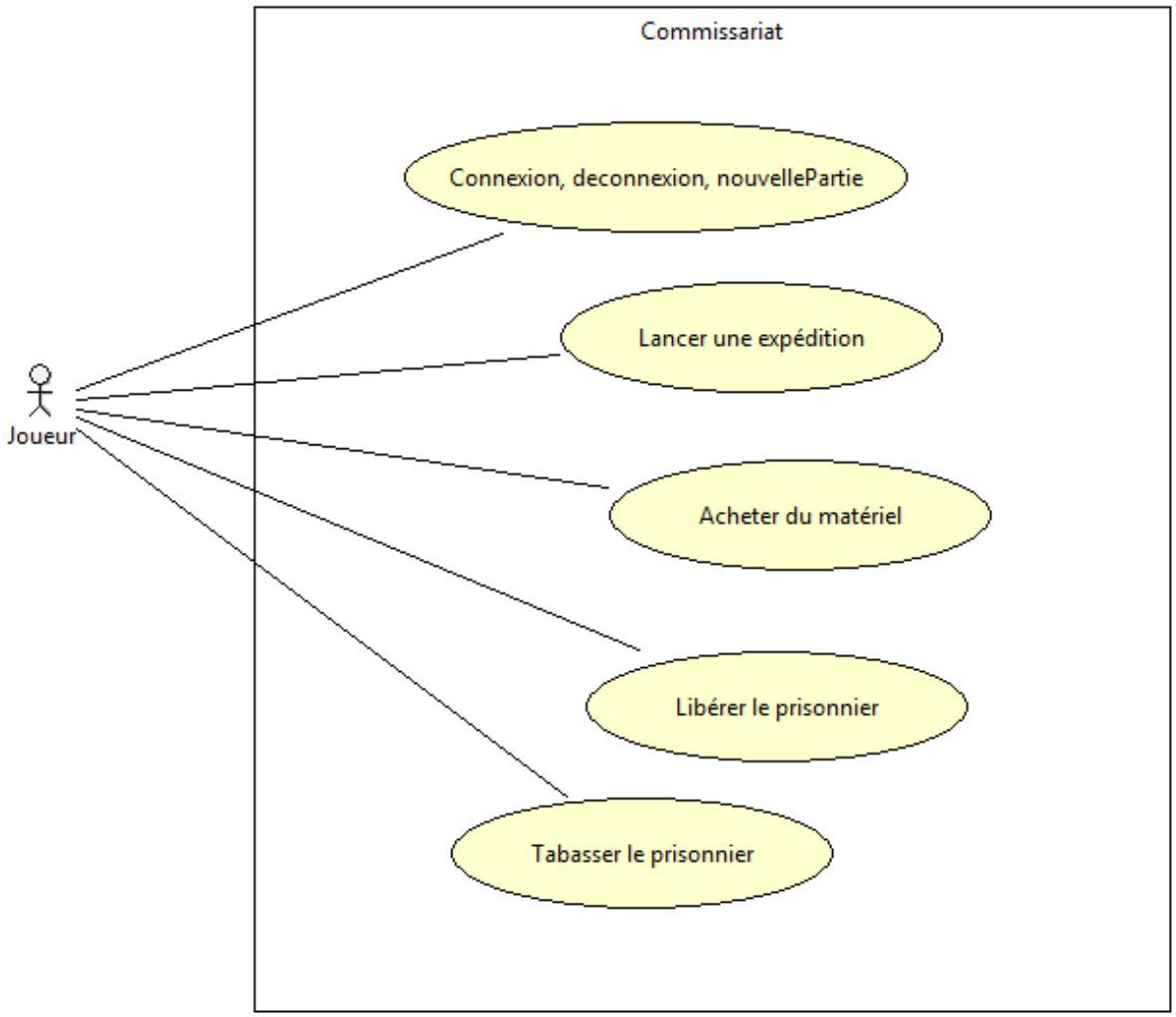


FIGURE 17 – Gestion : Diagramme de cas d'utilisation

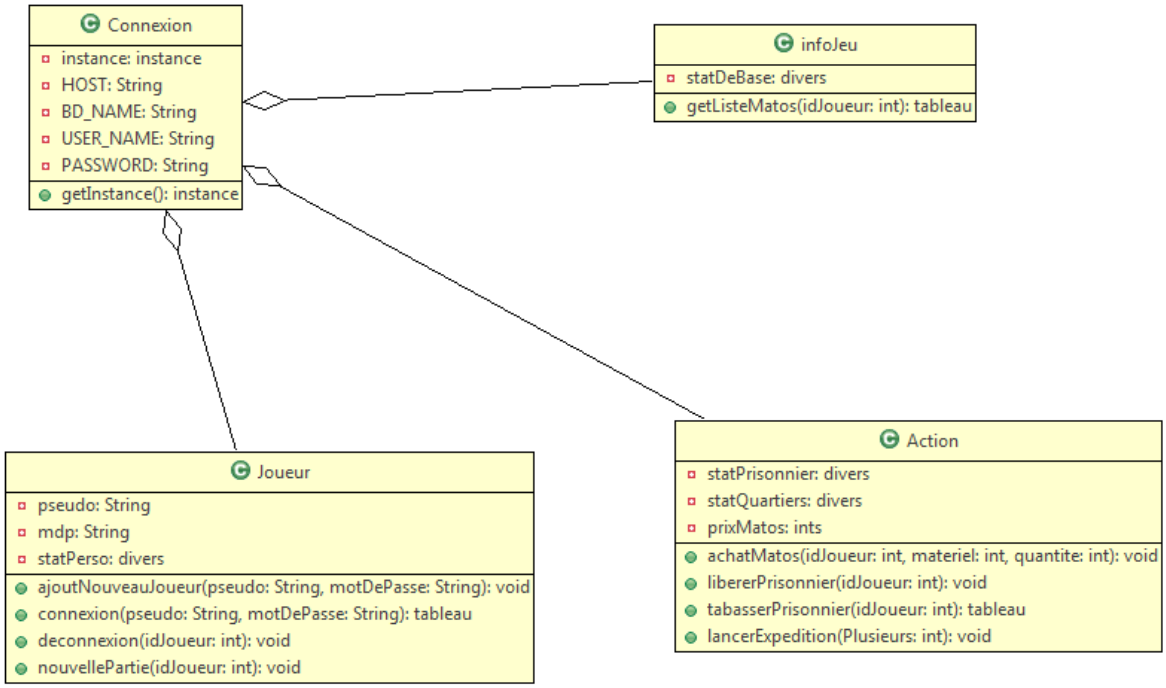


FIGURE 18 – Gestion : Diagramme de classes