Dossier de Conception Version 6

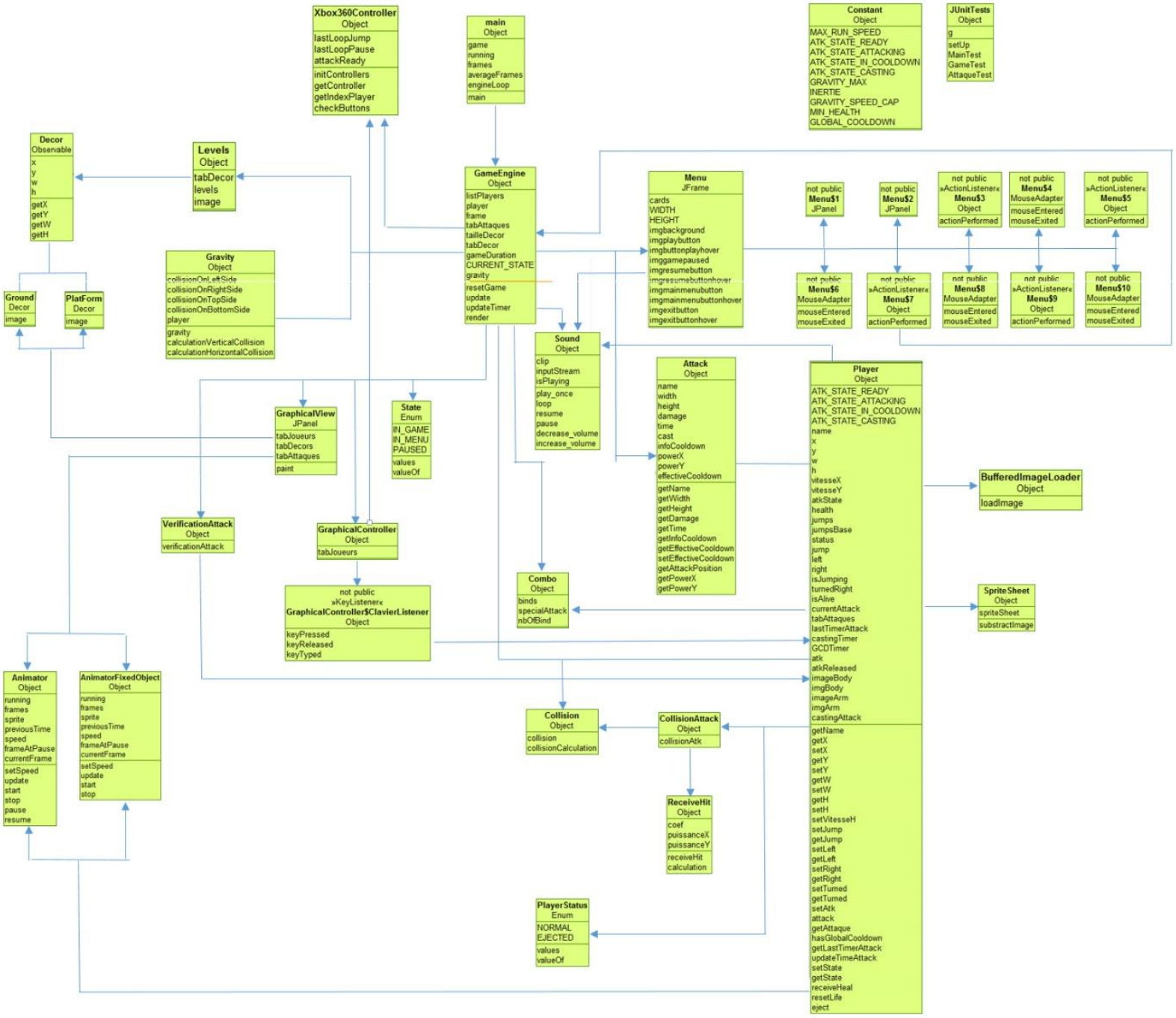
Sommaire :

Changements divers apportés :

- Changement de noms de classe et de nom de méthodes : traduction en anglais.
- Changement de noms de paramètres pour une meilleure clarté
- Apport de nouvelles classes pour diviser les fonctionnalités :
    - Ajout de CollisionAttack, PlayerStatus, ReceiveHit, VerificationAttack
    - Ajout de Ground et PlatForm héritant de Décor

L'ajout de ces deux dernières classes ont été faits pour permettre au joueur de réagir différent en fonction du type de Décor sur lequel il se trouve.

Voici le schéma UML du projet actuellement :



Sprites attaques basiques et sprites perso :

(Cornat Jacques et Raulot Adrien)

Pour cette partie, nous avons décidé d'ajouter un attribut (ou deux pour la classe Player, nous expliquerons juste après), Image pour les classes Player, Level, Ground et PlatForm.

Nous sommes partis du principe que nous chargerons simplement l'attribut dans la classe GraphicalView, c'est-à-dire au moment d'afficher l'image en question.

Pour charger les images, nous utilisons le type ImageIcon, où nous donnons en paramètre l'adresse de l'image que nous voulons charger.

Ensuite, nous intégrons simplement l'ImageIcon dans un type Image pour mettre à l'échelle l'image qui sera affichée plus tard, grâce à la méthode « getScaledInstance() ».

Pour ce qui est du décor, il ne devrait pas y avoir de problème particulier, mais pour le personnage, il faut prendre en compte le sens d'orientation du personnage, ainsi que s'il est en train d'attaquer ou non.

Nous ajouterons donc une autre image correspondant aux bras, et au niveau de l'affichage graphique, et nous déciderons de la position des bras en fonction de l'attaque utilisée.

Pour ce faire, nous utiliserons une méthode de Graphics : drawImage() où nous pourrons facilement décider de la position et de l'orientation du sprite.

Ainsi, les objectifs selon lesquels nous considérerons accompli cette tâche (correspondant à : « 2 attaques basiques avec sprite » et « Simple sprite non animé ») seront donc accomplis.

Sprites animés :

(Cornat Jacques)

Au vu de notre avancement dans le projet, j'ai décidé de continuer le système de sprites, pour le rendre plus facile à utiliser et surtout à le rendre dynamique !

Pour se faire, après quelques recherches, j'ai décidé de me tourner vers des BufferedImage, qui ont la particularité de pouvoir être manipulées et modifiées facilement.

Ensuite, il est question de découper les images de manière faciles, donc une classe sera créée pour s'occuper des découpes des sprites.

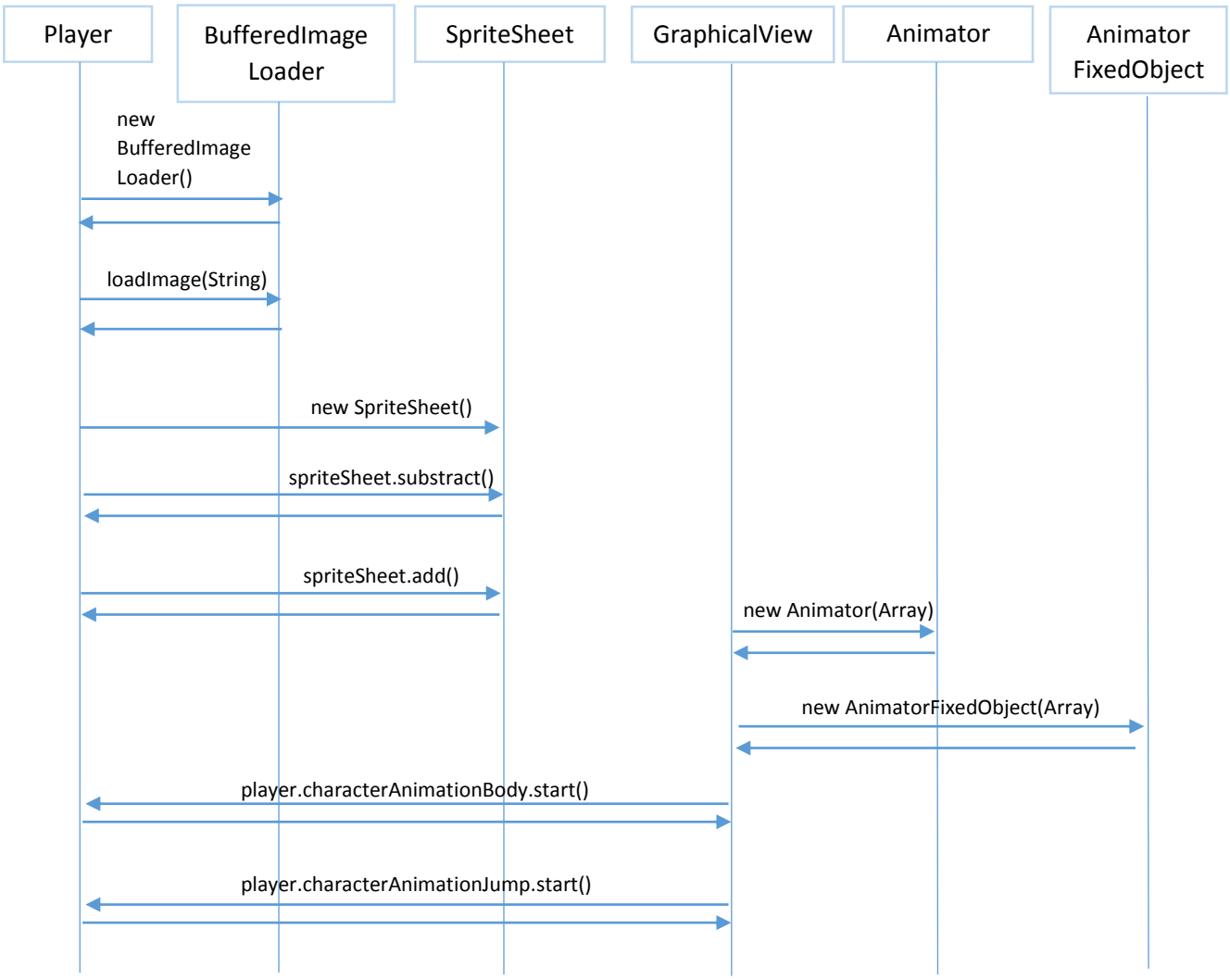
Les sprites seront ainsi de simples images contenant chaque frame de l'animation, où chaque frame sera de la même taille que les autres.

Une classe d'animation est prévue pour retourner les différentes images lors de l'affichage pour afficher l'animation. Cette classe prendra en paramètre une liste de BufferedImage, ce qui fera le lien avec les méthodes de découpages et de conversion

Une autre class d'animation a aussi été prévue pour ne fonctionner qu'une fois, pour les animations qui ne devront être lue qu'une fois (attaques, saut, etc…), et ne pas faire de boucle.

Des méthodes de pause, reprise, démarrage et d'arrêt dans les classes d'animations seront aussi prévues pour contrôler les animations facilement.

Diagramme de séquence :



Combos :

(Adrien Raulot)

Tout d'abord, un combo est une attaque spéciale déclenchée par un enchaînement d'attaques basiques. Je pense que cette fonctionnalité ne sera pas facile à mettre en place.

Mon but est d'initialiser un « timer » lors du lancement d'une attaque, qui laissera un temps très court entre l'appui sur la touche ou l'appui sur le bouton de lancement d'une attaque, pour que le joueur ai le temps d'appuyer sur d'autres touches d'attaques. Chaque attaque voulant être lancée par le joueur pendant ce laps de temps sera sauvegardée. À la fin du temps, je créerai une méthode qui regardera quelles attaques auraient dû être lancées pendant ce temps, ainsi que leur ordre de lancement, pour enfin lancer le combo correspondant.

Je pense créer une classe « Combo », qui contiendra plusieurs attaques en attribut :

• un tableau des attaques devant être lancées pour pouvoir déclencher le combo (donc triées par ordre de lancement)

• une attaque spéciale, qui ne sera rien d'autre qu'une attaque simple (objet de la classe Attack), mais plus puissante qu'une attaque normale et sans temps de cast

• une variable définissant le nombre d'attaques devant être lancées par le joueur pour déclencher le combo

Les problèmes que je pense rencontrer se situeront au niveau du timer et du temps de cast des attaques. Il ne faut pas que le temps d'attente d'un combo soit trop long, ni trop court, et il ne doit pas empiéter sur le temps de cast. Je veux qu'il soit quasi-invisible au joueur.

De plus, je vais faire en sorte que les combos soit réalisables en appuyant successivement sur deux touches et trois touches. Les combos à trois touches seront évidemment plus puissants que ceux à deux touches. Une des difficultés sera également de faire en sorte que si le joueur lance une seule attaque, ou effectue un enchainement d'attaques ne correspondant à aucun combo déterminé dans le jeu, cela lance la première attaque de cet enchaînement.

Enfin, les combos devront bien sûr pouvoir être lancés avec une manette de jeu ou un stick.

Limite du terrain et nombre de vies :

(Besson Léonard)

Pour cette partie, nous avons décidé de fixer les limites du terrain aux limites de l'écran. Nous prévoyons une refonte du terrain (avec des attributs de limites) pour plus tard.

Les tests pour savoir si le joueur à dépassé les limites de l'écran se feront dans la méthode update() de GameEngine. Si le joueur quitte l'écran (par le haut, le bas, la gauche ou la droite) toutes les forces qui lui sont appliquées sont annulées, il est également repositionné à un endroit prédéfinit et une vie lui est enlevé. Si le nombre de vie d'un joueur tombe à 0 , la partie est terminée et l'écran du menu principal s'affiche (création d'un écran GameOver par la suite).

Conception du contrôleur pour manettes (et sticks) :

(Besson Léonard)

Pour rendre notre jeu compatible avec les manettes, il nous faut pouvoir communiquer avec la manette. Il existe pour cela plusieurs librairies. Le projet Kenai propose la librairie "Jinput" qui semble être la plus populaire, c'est pour cela que nous l'avons choisie.

Le fonctionnement est le suivant :

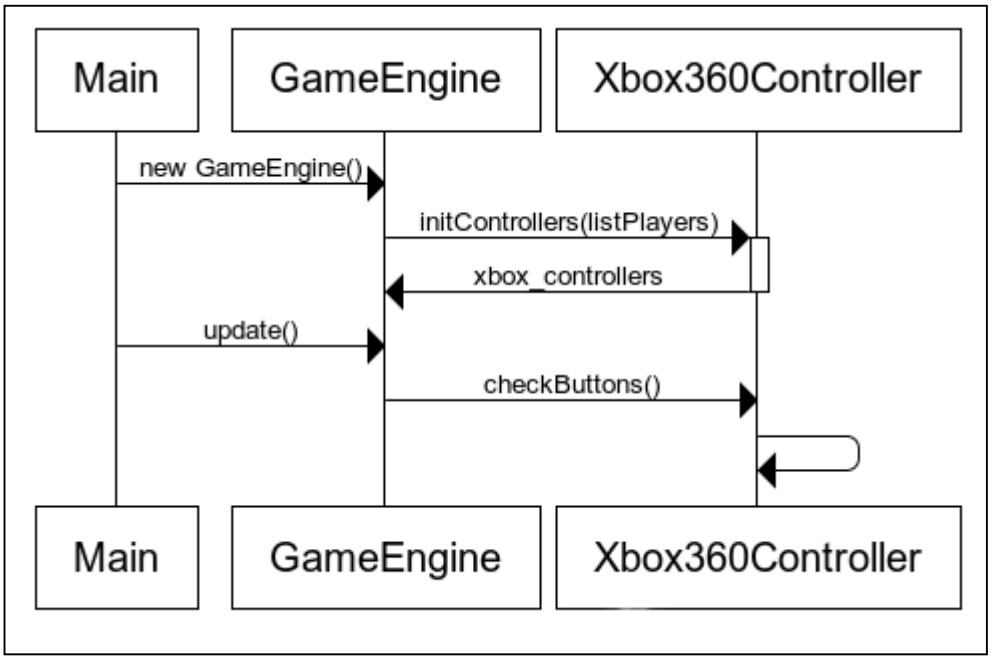
Un environnement doit être créé et permet de récupérer tous les périphériques de contrôle (souris, clavier, manettes, etc).

Chaque périphérique possède des "component" (c'est à dire les boutons, les gâchettes et les stick analogiques pour une manette par exemple) et le périphérique peut être interrogé sur l'état de ses "component". A partir de ces information nous pouvons savoir si les boutons sont poussés ou non par exemple.

Voici comment nous allons procéder :

Nous allons créer une classe “Xbox360Controller” qui s’occupera de gérer une manette. Cette sera dotée d’une méthode static permettant de récupérer et initialiser les manettes branchées et qui retournera la liste des manettes à la classe GameEngine. cette dernière pourra alors dans la méthode update() mettre à jour les touches pour chaque controleur grace à la méthode checkButtons(). Cette méthode regarde pour une manette si les boutons, les sticks et les gâchettes sont actionnés et réalise les actions en conséquence (comme le controleur pour le clavier).

Voici le diagramme de séquence :



Partie sons :

(Luc Aymeric et Besson Léonard)

Pour les sons deux choix s’offre à nous, utiliser un Clip ou une SourceDataLine. Clip semble être clairement la solution la plus simple et adaptée à notre type de besoin (SourceDataLine est beaucoup plus adapté aux fichiers volumineux qui ne peuvent pas êtres pré-chargés en entier dans la mémoire, il faut donc écrire un buffer lorsque l’on joue le son). Nos fichiers sont de faibles tailles et sont donc pré-chargés dans la mémoire et êtres joués instantanément ou bouclés.

Une nouvelle classe sera créée, “Sound”, elle prendra le fichier son (wav) en paramètre et comportera toutes les méthodes utiles à son exploitation, c’est à dire :

- le lire une fois.
- le lire en boucle.

- le mettre en pause

- reprendre la lecture.

- baisser le volume.

- augmenter le volume.

Pour ne pas surcharger les classes d'attributs, les sons seront des variables locales (sauf pour la musique de fond qui devra être mise en pause lorsque l'on n'est pas dans le jeu).

La classe Sound possède trois attributs :

- un flux audio de lecture (AudioInputStream)

- un clip qui ouvre le flux audio

- un booléen qui indique si le son est en train d'être joué.

Listes des choses à faire :

- Caméra dynamique
- Amélioration du confort utilisateur :
    - Choix de la carte
    - Choix des personnages
    - Attribution des touches
    - Changement de musique en cours de partie
- Modifier le moteur physique :
    - Rebondissement contre les murs dans certains cas précis
    - Accroches sur les plateformes
    - Traverser une plate-forme vers le bas lors de l'appui de la touche « bas »
- Gestion réseau
- Intelligence Artificielle