Dossier de Conception Version 6

Sommaire :

Changements divers apportés

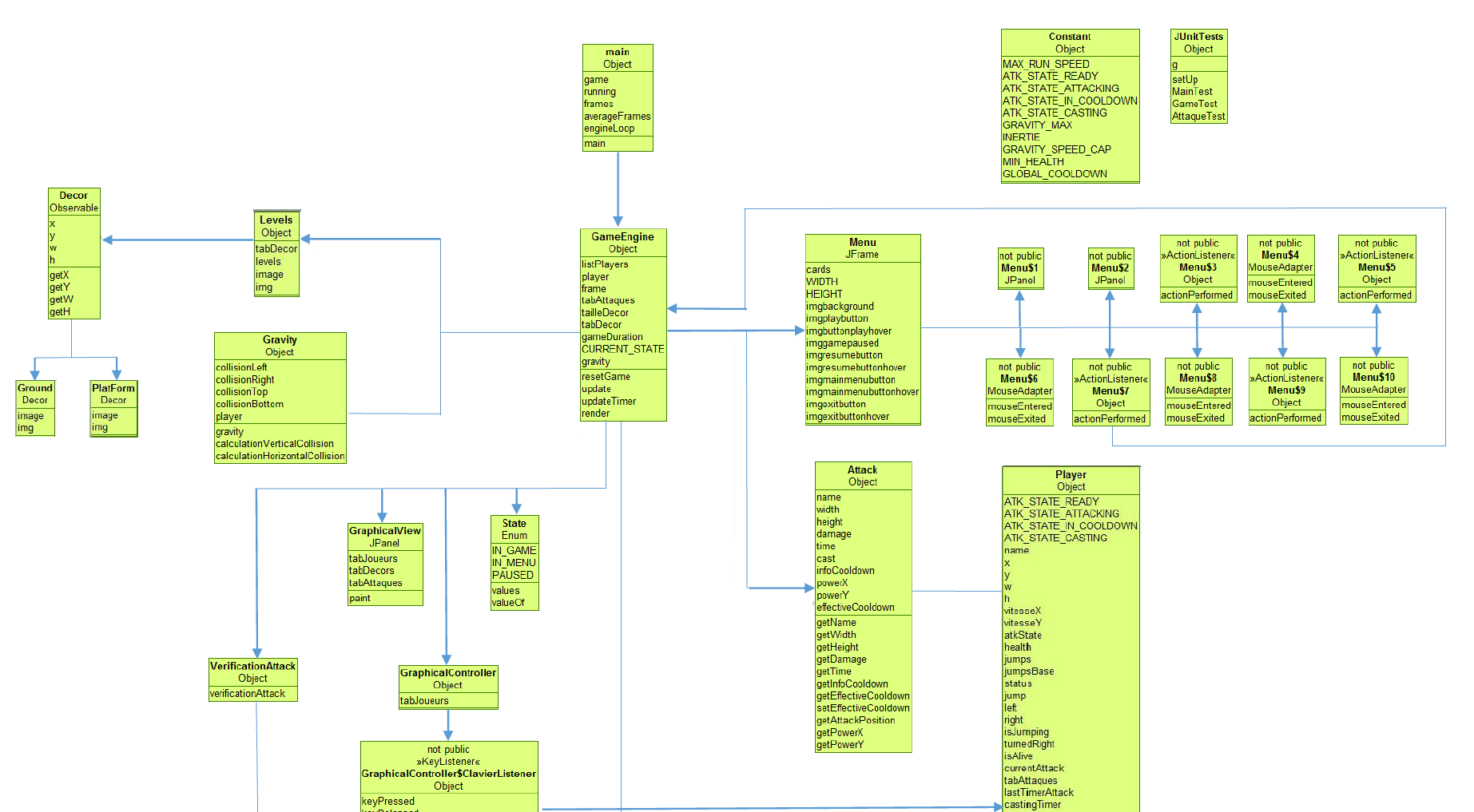
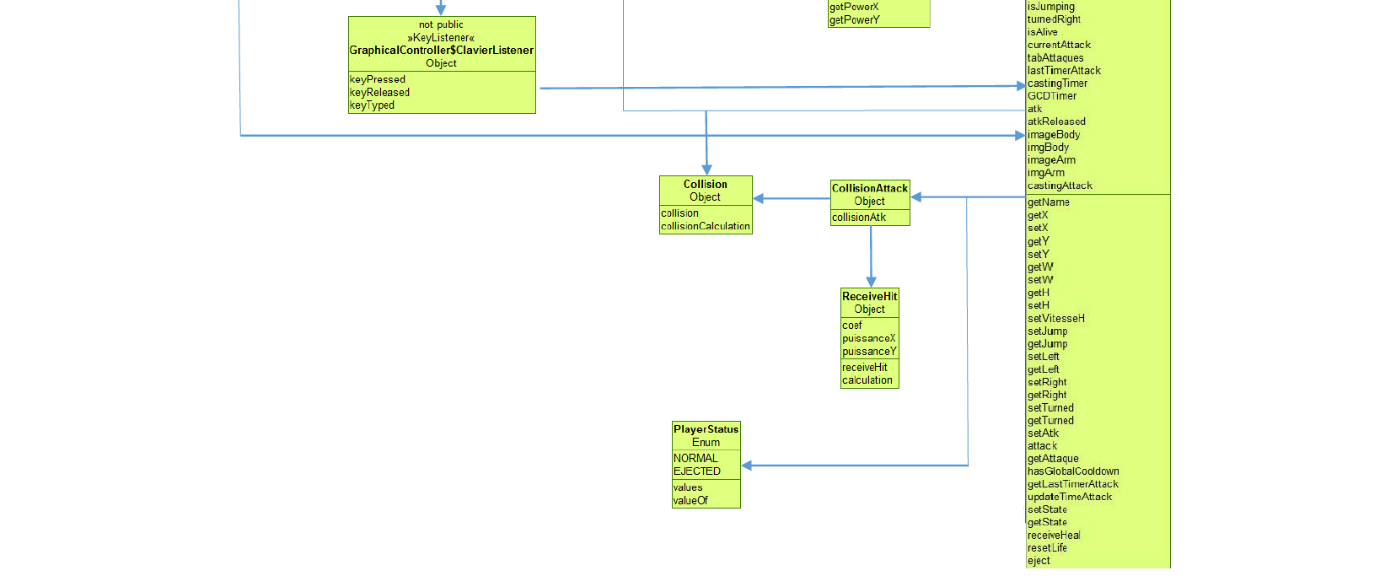
Sprites attaques basiques et sprites perso par Jacques Cornat & Adrien Raulot

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 :

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 n’y a pas 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 avons donc ajouté une autre image correspondant aux bras, et au niveau de l’affichage graphique, nous décidons de la position des bras en fonction de l’attaque utilisée.

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

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

**Limite du terrain et nombre de vies :**

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).

**Partie sons :**

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.

**Conception du contrôleur pour manettes (et sticks) :**

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 :

