BESSON Léonard
CORNAT Jacques
LUC Aymeric
RAULOT Adrien

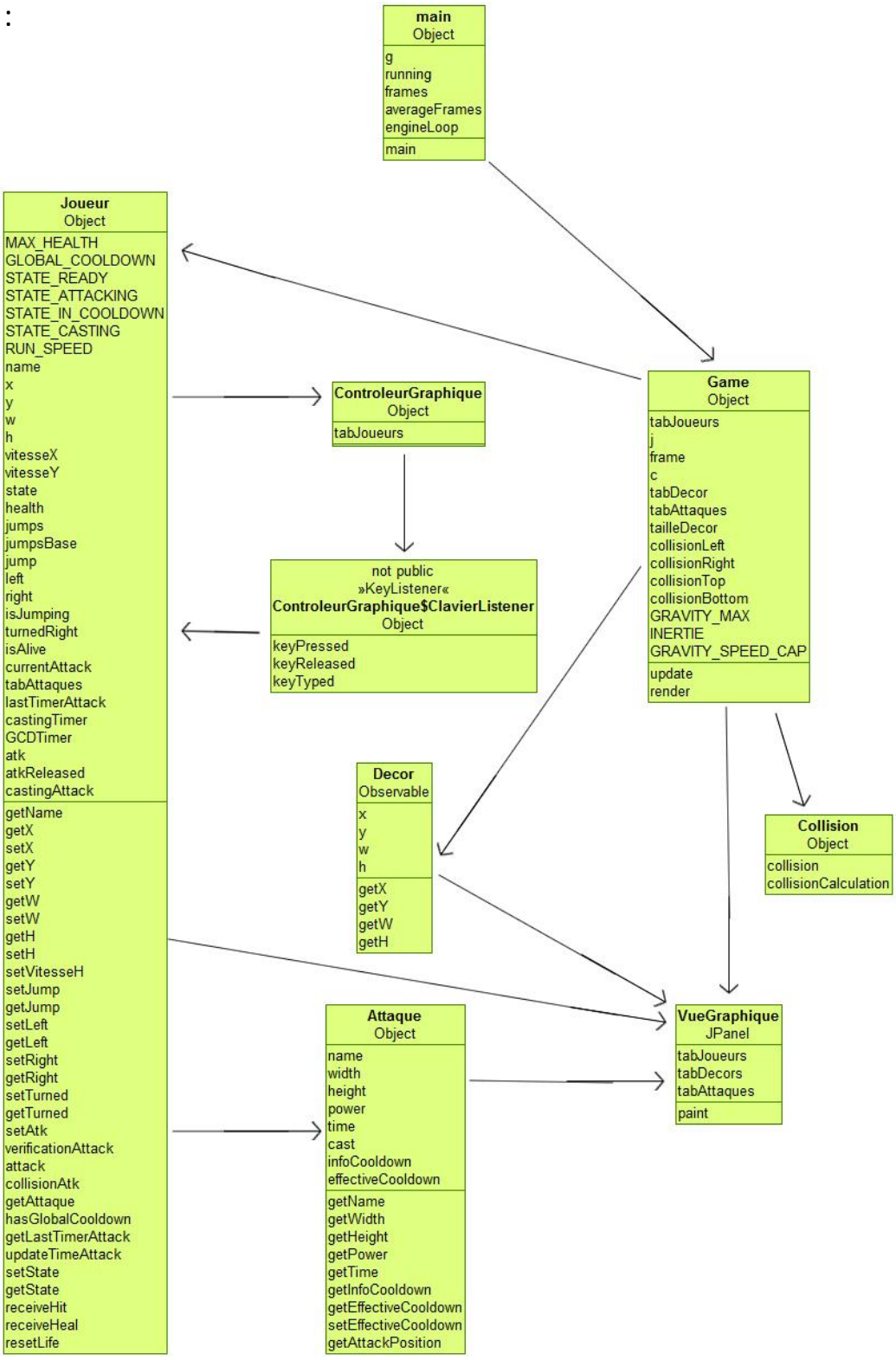
15/12/2013

# Dossier de Conception

# Jeu de combat en 2D

Dans ce dossier de conception, nous allons détailler le projet à son état actuel.

## Schéma UML :



BESSON Léonard
CORNAT Jacques
LUC Aymeric
RAULOT Adrien

15/12/2013

# Explications des principaux algorithmes du projet :

- **Collision.java :**

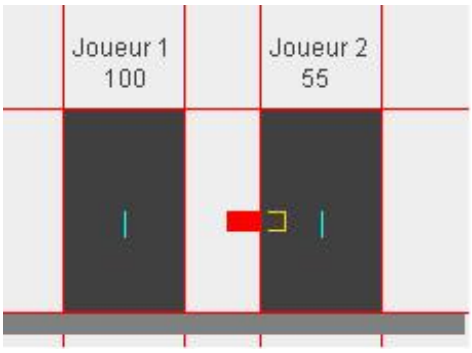
La classe Collision permet de détecter s'il y a collision ou non avec un élément (décor, joueur…), à travers deux méthodes. Concrètement, cette classe et ces méthodes sont utilisées pour vérifier s'il y a collision du joueur avec un ou plusieurs éléments du décor, mais également pour savoir si une attaque touche un joueur.

La première méthode se nomme collision() et permet de savoir si deux rectangles sont en collision ou non. La méthode renvoie vrai s'il y a collision, faux si non. Elle est principalement utilisée pour savoir si une attaque touche un joueur.

La seconde méthode se nomme collisionCalculation() et permet de calculer la collision du joueur avec un ou plusieurs éléments du décor. La méthode renvoie l'indice du décor dans la liste des décors avec lequel il y a collision, et -1 le cas échéant. Cette méthode est principalement utilisée lorsqu'un joueur se déplace en jeu, afin de calculer à l'avance les possibles collisions avec les différents éléments du décor.

Les deux méthodes en détail :

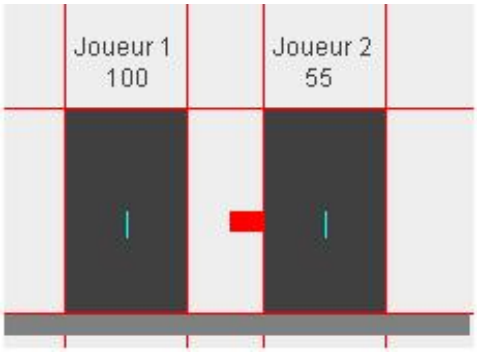
La méthode "collision" :




Sur cette image par exemple, nous détectons la collision de l'attaque avec un joueur en utilisant cette méthode.

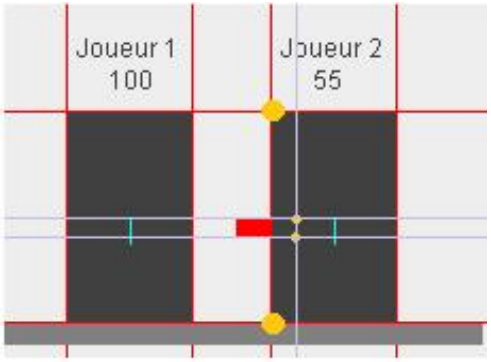
Le résultat nous donc vrai.

Nous pouvons voir sur l'image ci-contre (à droite) jusqu'où va la largeur de l'attaque, grâce aux contours imaginaires colorés en jaune.



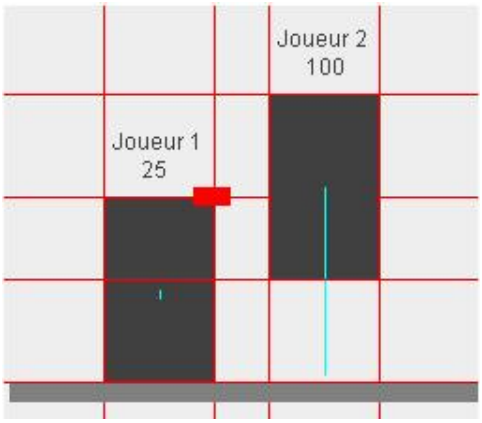
BESSON Léonard
CORNAT Jacques
LUC Aymeric
RAULOT Adrien

15/12/2013



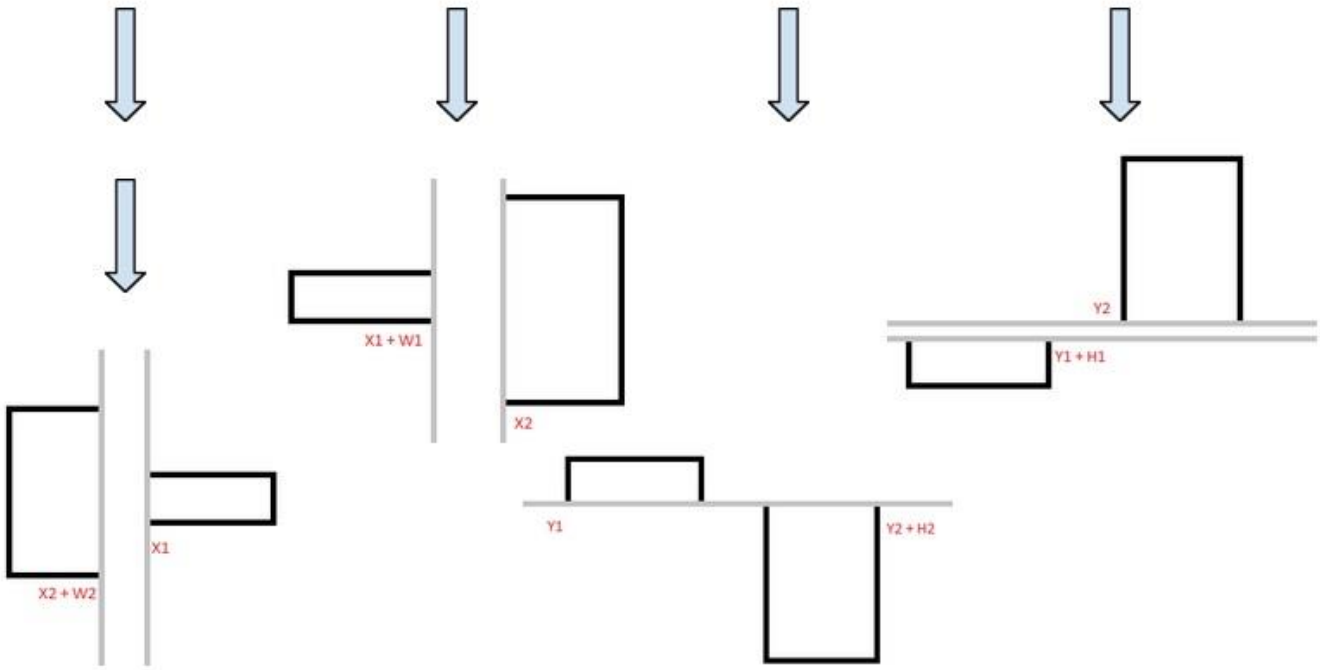

L'image de gauche montre que nous pouvons clairement distinguer les coins droits du rectangle d'attaque, leur abscisse et leur ordonnée (grâce aux ronds jaunes et aux lignes violettes dessinées), et ainsi voir qu'ils se situent bien dans le rectangle du joueur. C'est exactement ce que la méthode nous permet de calculer et vérifier.

Même si le rectangle d'attaque n'est pas totalement à l'intérieur du rectangle du joueur, il y a collision, ce qui est normal car le rectangle d'attaque touche le rectangle du joueur, donc cela signifie que l'attaque touche également. Cela montre que nous vérifions chaque coin du rectangle de l'attaque. Si l'un des coins se trouve à l'intérieur du rectangle du joueur, cela signifie qu'il y a collision.




Description graphique de l'algorithme de la méthode collision(), qui renvoie faux si :

(x1 >= x2 + w2) || (x1 + w1 <= x2) || (y1 >= y2 + h2) || (y1 + h1 <= y2))




… sinon retourne vrai.

La méthode “collisionCalculation” :

Elle parcours chaque élément de la liste de décor contenant tous les décors (c’est-à-dire tous les rectangle créés et utilisés afin de définir le décor) de manière itérative, et utilise la méthode “collision” expliquée précédemment sur le rectangle joueur et ces rectangles du décor à chaque itération.

Nous ne ferons aucune description graphique ici, car c’est simplement la méthode “collision” (et donc son algorithme) utilisée sur chaque rectangle du décor. Pour rappel : la méthode renvoie l’indice du décor dans la liste des décors avec lequel il y a collision, et -1 le cas échéant.

- **Game.java**

Explication de la méthode Gravity() :

Cette méthode reçoit en paramètre un Joueur j, et va permettre de déterminer si un joueur peut se déplacer (que ça soit verticalement ou horizontalement), ainsi que s’il est soumis à une gravité ou non.

Explication détaillée de l’algorithme :

Nous commençons par ajouter à sa vitesse actuelle, une gravité. Si cette vitesse dépasse 50, nous la ramenons à 50, c’est-à-dire que nous mettons en place une vitesse verticale cap, par pur choix de gameplay.

Pour la suite, si le joueur à une vitesse verticale > 0, cela indique qu’il tombe, donc nous calculons s’il y a un objet ou un décor contre lequel il ne devrait pas passer au travers, et nous lançons la méthode de collision. Si une collision est détectée, nous ramenons la vitesse verticale du Joueur à 0, et nous réinitialisons le nombre de saut restant, pour qu’il puisse de nouveau profiter de ses multi-sauts. Par la même occasion, nous calculons l’espace entre le joueur et le décor, et nous collons le joueur au décor. Il est important de préciser que par choix de gameplay, les joueurs n’ont pas de collision entre eux.

Si aucune collision n’est détectée, alors le joueur continue de chuter.

Pour une vitesse verticale < 0, le principe est sensiblement le même.

Pour la vitesse horizontale, le principe est conservé, mais il s’applique une inertie au joueur lorsqu’il commence ou s’arrête d’avancer.

BESSON Léonard
CORNAT Jacques
LUC Aymeric
RAULOT Adrien

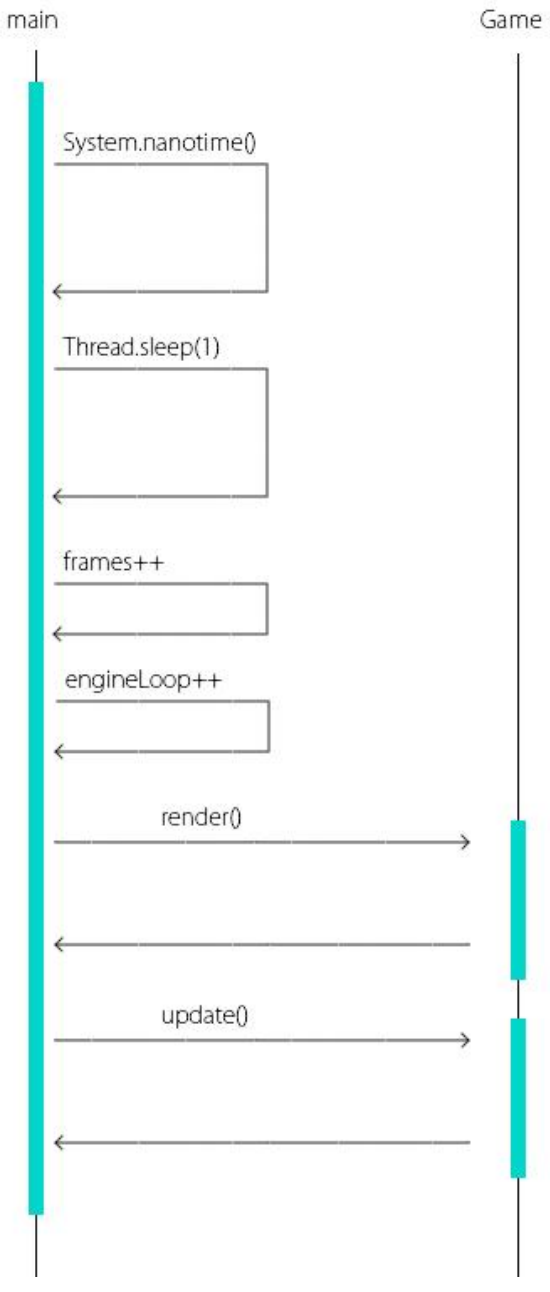
15/12/2013

# Liste des questions et changements effectués :

Diverses questions ont été posées et certains problèmes ont été soulevés lors de réunion avec notre professeur tutoré.

- Boucle principale fonctionnant sur le sleep
- ControleurGraphique qui fait appel à certaines fonctions directes du jeu
- Attaques fonctionnant dans un autre Thread que celui de la boucle principale

Voici un diagramme illustrant notre boucle principale, avec pour changement, le fait que d'autres fonctionnalités puissent se baser sur le nombre de tour de boucle, pour ainsi synchroniser l'ensemble des éléments du jeu :



Maintenant, les attaques se basent sur le nombre de frames, et donc, l'ensemble se retrouve à être totalement dépendant de la vitesse de la boucle principale.

Au niveau de l'appel des attaques directement par le ControleurGraphique, ce problème a été résolu.

Au niveau de la méthode update de Game, une nouvelle méthode a été créée : verificationAttack, faisant partie de la classe Joueur.

Ainsi, le ControleurGraphique mets à vrai ou à faux des booléens d'attaque, et cette méthode ce charge dorénavant de lancer elle-même les attaques des joueurs, de manière synchronisée avec le moteur du jeu, car se basant sur le nombre de tour de boucles effectuées.

## Changements annexes effectués :

- Ajout d'un état d'attaque pour un joueur : STATE_CASTING.
  Cet état se rajoute à 3 autres états :
  STATE_READY, STATE_ATTACKING et STATE_IN_COOLDOWN.
  Cet état sera utilisé pour des coups qui auront le besoin d'être chargé.
  Il était important de faire une modification dès le début du développement, car c'est une fonction que nous avons considéré comme « de base ».
  Ainsi, une Attaque sera maintenant définie par une durée de cooldown aussi, et si le joueur se fait toucher par un coup durant le chargement du coup, le casting est annulé, et il sera incapable de donner un autre coup avant la fin du cooldown de l'attaque anciennement chargée.

- Un autre souci avait été détecté au niveau de l'attaque :
  En effet, lors de la durée où une attaque pouvait toucher et blesser un joueur, la vérification de collision entre l'attaque et le joueur adversaire ne se faisait qu'à la première frame où le coup se lançait.
  Dorénavant, une vérification est effectuée à chaque frame, et quand le coup touche un joueur adversaire, le joueur à l'origine de cette attaque voit son coup incapable de toucher de nouveau l'adversaire, pour laisser place au cooldown du coup.

- Simplification de la méthode Attack() du Joueur, qui ne prend plus qu'en paramètre un id, correspondant à l'attaque voulue.
  Une amélioration du code a été faîte, en enlevant le fait de rechercher le coup dans la liste des coups du Joueur.

## Validation du travail :

Pour le moment, nous mettons en place une classe de test JUnit, et nous nous efforçons de tester certaines fonctionnalités.

Ceci est délicat pour la plupart des méthodes, car cela dépend du moteur de jeu, et du placement des différents personnages, etc… mais nous imaginons plusieurs façons de faire ces tests.
Pour le moment, des tests basiques ont été réalisés, sur la classe main, Game ainsi qu'Attaque.

## Actions envisagées par la suite :

Nous avons en tête de mettre plusieurs choses en place pour la prochaine étape du Projet.

Tout d'abord, nous allons commencer par terminer la Version 5 de notre projet, qui consiste encore à :

- Mise en place d'un gameplay : Le joueur commence avec un certain nombre de vies, et 0% de dégâts
- Si le nombre de vies tombe à 0, le personnage a perdu. (Gestion d'un écran de « Game Over »)
- Si le personnage est expédié hors de l'écran, le joueur se voit enlever une vie. (Ajout d'une force aux attaques)
- Ajout de la fonction temps, qui permettra d'arrêter le jeu si la partie commence à durer.

Nous voyons ici comme difficulté de gérer les forces qui feront s'envoler le joueur en fonction des dégâts qu'il subit.

## Répartition des tâches jusqu'au prochain dossier de conception :

- BESSON Léonard : Si le nombre de vies tombe à 0, le personnage a perdu, avec la gestion de l'écran de Game Over.
- CORNAT Jacques : Si le personnage est expédié hors de l'écran, le joueur se voit enlever une vie, avec le fait de gérer une force sur les attaques
- LUC Aymeric : Le joueur commence avec un certain nombre de vies, et 0% de dégâts
- RAULOT Adrien : Ajout de la fonction temps, qui permettra d'arrêter le jeu si la partie commence à durer.

Une recherche et une conception, de la part des personnes du groupe ayant fini de faire la conception et la réalisation de leur tâche, sera attendue sur les points suivants :

- Gestion des différents menus du jeu (nous imaginons pour le moment réaliser ceci avec des CardLayout)
- Gestion des Sprites
- Affinement des collisions avec les « SAABB ».