RAPPORT : PROJET ORIENTÉ OBJET

Jun Nuo Chi - Nathan Dwek - Thomas Vandamme

1 Sujet choisi et fonctionnalités

Nous avons essayer de programmer un jeu de type run & jump basé sur l'univers de Sonic. Les principales fonctionnalités que nous avons reussis à implémenter dans notre jeu sont :

- Moteur physique et hitbox de formes simples entourant les personnages et objets de façon plus ou moins «précise>.
- Héros unique capable de marcher, de courir, de sauter, de récuperer des pièces mais également de se mettre en boule permettant ainsi de changer ses capacités : vitesse maximale accrue et invincibilité face aux monstres.
- Deux types de monstres se déplaçant de façon autonome et interragisant avec les blocs.
- Une interface graphique capable de suivre le héros dans sa progression dans le niveau.
- Une interface graphique avec lequel l'utilisateur peut interragir : bouton pour mettre en pause, redémarrer le niveau, revenir au menu ou encore choisir le terrain de jeu voulu(niveau).

Le choix du niveau se fait au menu de démarrage, le programme permet de choisir un terrain écrit en XML, celui-ci est retranscrit en language java par le programme. La création d'un niveau est ainsi considérablement facilitée, puisqu'il n'est plus nécessaire d'instancier un par un les monstres et autres objets intervenant dans le jeu.

2 Structure et implémentation du programme

Au niveau le plus élevé, le programme est structuré autour du pattern Modèle-Vue-Contrôleur avec modèle passif. Ce dernier s'occupe uniquement de la représentation et du traitement des informations (bien évidemment à travers une hiérarchie objet). Ces informations sont présentées à l'utilisateur par la vue qui se charge elle-même de questionner à intervalle régulier le modèle sur l'état du système. Le contrôleur reçoit les inputs de l'utilisateur et envoie les messages adéquats au modèle. Il émet aussi les ticks d'horloge, contrôle l'exécution ou non de la boucle principale du jeu proprement dit et peu diffuser certains messages d'importance globale.

2.1 Organisation du modèle

Les fonctionnalités principales à implémenter font logiquement apparaître trois interfaces qui regroupent des fonctions essentielles remplies par des groupes d'objets.

- Les objets Drawables : qui peuvent se représenter à l'écran.
- Les objets Hittables : qui jouent un rôle physique et possèdent donc une hitbox et sont capables de gérer leurs collisions.
- Les objets SelfUpdatables : qui sont dynamiques et agissent d'eux-même, sans intervention extérieure (contrairement à un bloc bonus par exemple, qui est aussi dynamique, mais qui ne peut pas décider de lui-même de s'animer)(Toute séquence est donc originellement initiée par un SelfUpdatable).

Ceci permettrait de réduire le modèle du jeu à quelque chose comme :

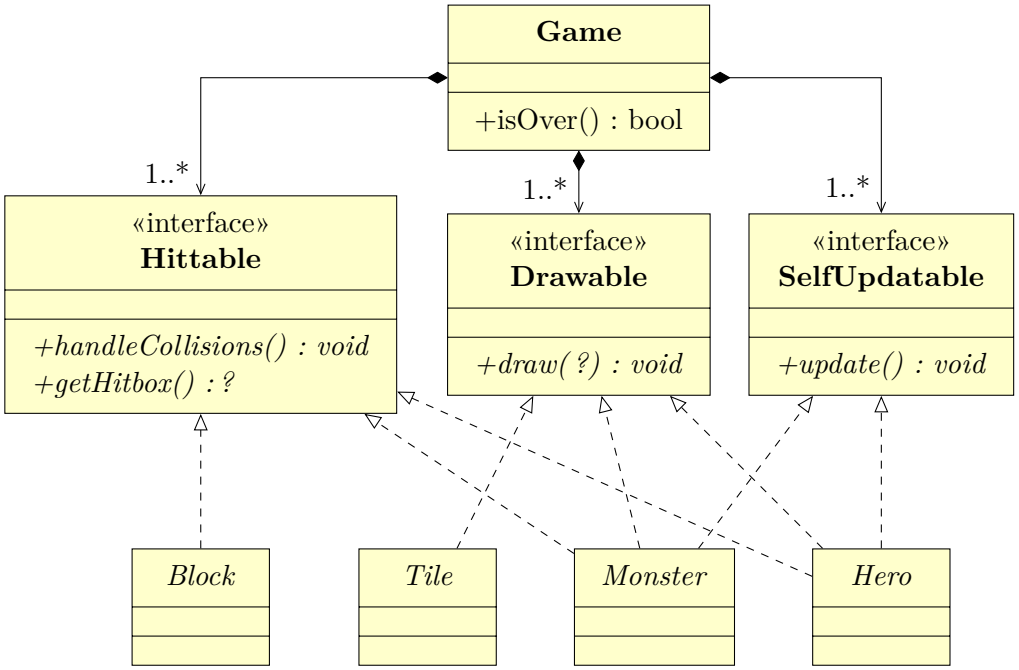
```
while (not game.isOver()){
  for (Hittable h : hittables){
    h.handleCollisions()
  }
  for (SelfUpdatable s : selfUpdatables){
    s.update()
  }
  for (Drawable d : drawables){
    d.draw(?)
  }
  wait(?)
}
```

Avec le méthode update de Hero étant certainement un peu particulière puisque contrairement à celle Monster, elle ne se limite pas à de la logique

mais attend aussi des inputs du joueur.

Un niveau plus bas, différentes classes abstraites, plus «parlantes» implémentent une ou plusieurs interfaces. Plusieurs problématiques restent encore ouvertes

- Les Blocks ne sont pas forcément Drawables car les blocs invisibles sont monnaie courante (pour détecter la mort par chute par exemple). Dès lors, on peut créer des blocs visibles soit en implémentant Drawable par certaines des sous-classes concrètes de Block, soit en créant des objets composites composés d'un Block et d'une Tile.
- La gestion des collisions est encore sujette à débat. En effet, si l'on ne veut pas doubler le nombre d'opérations nécessaires, il faut a priori introduire une asymétrie dans les objets Hittables (Pour éviter que monstre1 vérifie s'il entre en collision avec bloc2 en même temps que bloc2 vérifie s'il entre en collision avec monstre1). Si l'on y fait pas attention monstre1 pourrait même essayer de gérer deux fois une seule et même collision.
- De plus pour que les Hittables puissent détecter les collisions, il faudrait à priori qu'ils disposent de références vers les autres Hittables (ou probablement uniquement les Hittables SelfUpdatables). Dès lors, chaque Hittable disposerait d'une (grande) collection de ces autres agents. Une alternative probablement plus judicieuse serait qu'il existe une unique instance de cette collection existant de manière autonome et que les Hittables disposent simplement d'une référence vers cette collection. On pourrait même aller plus loin et attribuer à cette collection le rôle de rechercher (sans répétition) les collisions et de transmettre les messages appropriés aux intervenant impliqués.
- Comment représenter une hitbox ? (classe Hitbox envisageable ?)
- Lorque le jeu demande au Drawables de se dessiner, il doit probablement leur passer en argument des informations sur la position et la taille de l'écran (centré sur le héros et de dimension fixée ?).



3 Patterns mis en œuvre

La section précédente introduit déjà des patterns qu'il serait peut-être intéressant d'approfondir :

- La découpe en objets représentables, objets ayant un rôle physique, et objet agissant d'eux même/attendant des inputs du joueur nous incite à essayer de vraiment appliquer le pattern MVC.
- L'idée de créer un objet reprenant la liste des Hittables et s'occupant de checker les collisions et de transmettre les messages appropriés est une application du pattern médiateur.
- Certaines classes ne devraient avoir qu'une seule instance par partie (Héros et collection des Hittables). Le pattern singleton pourrait leur être appliqué pour rendre certains appels statiques.