RAPPORT : PROJET ORIENTÉ OBJET

Jun Nuo Chi - Nathan Dwek - Thomas Vandamme

1 Sujet choisi et fonctionnalités

Nous avons programmé un jeu de type run & jump basé sur l'univers de Sonic. Les principales fonctionnalités que nous avons implémentées sont :

- Moteur physique et hitbox «précises». Jeu réactif, accélérations et gravité réalistes.
- Héros unique capable de marcher, courir, sauter, de récupérer des pièces et surtout de se mettre en boule permettant ainsi de changer ses capacités : vitesse maximale accrue et invincibilité face aux monstres.
- Architecture des niveaux : blocs de taille totalement arbitraire et possibilités de pentes.
- Un modèle physique d'obstacle (fixe) et un modèle physique de monstre (dynamique). Plusieurs aspects facilement possibles grâce au pattern décorateur.
- Une interface graphique capable de suivre le héros dans sa progression dans le niveau. Le scrolling est satisfaisant et rend compte de la vitesse et des accélérations du héros.
- Ecran d'accueil et menus disponibles en jeu. Actions possibles : pause-depause, restart, retour à l'écran d'accueil, choix du niveau (voir point suivant).
- Niveaux décrits en XML : lisibilité, facilité d'édition, format standardisé.

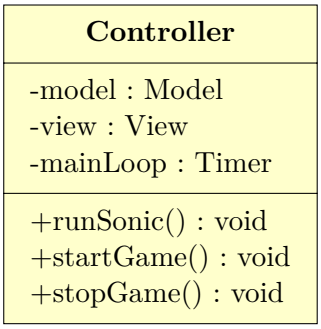
2 Structure et implémentation du programme

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 contrôle aussi l'exécution ou non de la boucle principale du jeu proprement dit et émet donc les ticks d'horloge.

```
public class Main {
  public static void main(String[] args){
    Model m = new Model();
    View v = new View(m);
    Controller c = new Controller(m,v);

    c.runSonic();
  }
}
```



A chaque tick, deux actions ont lieu : le modèle est «intégré» d'un pas de temps et la vue est rafraîchie. Nous allons maintenant voir comment a lieu l'intégration du modèle, ce qui va directement mener à son diagramme de classe.

2.1 Organisation du modèle

Pour avancer d'un pas de temps, le modèle effectue deux opérations : il gère les collisions entre les objets et demande aux objets dynamiques d'avancer à leur tour d'un pas de temps. A cet effet nous avons créé une classe abstraite `Hittable` permettant au modèle de dialoguer avec tous les objets quels qu'ils soient pouvant être impliqués dans une collision, ainsi qu'une interface `SelfUpdatable` permettant au modèle de demander à tous les objets dynamiques de s'updater. Il est à noter que pour éviter de vérifier des collisions qui ne peuvent normalement avoir lieu, ou qui n'auront jamais de conséquences (collision bloc-bloc par exemple) les objet `Hittable` sont répartis en deux listes afin d'effectuer à chaque pas uniquement :

- une seule vérification par paire d'objets a priori mobiles
- une vérification par paire objet a priori mobile - objet statique

2.1.1 Gestion des collisions

Nous approximons toutes les hitbox par des rectangles de taille entièrement libre. Pour permettre certaines collisions plus complexes, les hitbox sont aussi définies par les normales à leurs quatre côtés (pas forcément celles d'un rectangle même si la hitbox en est un). Un objet `Hittable` dispose donc d'une position et peut fournir sa taille et sa normale dans une des quatre

directions. Il est capable de gérer une collision avec un autre objet `Hittable`, en utilisant la normale à la face touchée.

Lorsque le modèle détecte une collision entre deux `Hittable`, il appelle la méthode `handleCollision` sur chacun des deux objets en leur passant l'autre objet et la normale correspondante. Cependant, les objets passés en arguments sont toujours seulement des `Hittable` puisque c'est sous cette forme que le modèle interagit avec eux. Pour permettre aux objets de gérer de manière différenciée les collisions selon le type d'objet touché, nous utilisons le pattern visiteur.

La méthode `handleCollision` d'une classe `A` demande simplement à l'autre objet (toujours vu comme un `Hittable`) de gérer une collision avec un objet `A` en passant toujours la normale adéquate en argument. Ceci permet en utilisant le polymorphisme de différencier les collisions tout en restant parfaitement orienté objet.

Il est à noter que la méthode `handleCollision` renvoie une valeur booléenne selon que l'autre objet doit être détruit lors de la collision ou non. Si c'est le cas, cet objet est ajouté a une liste temporaire d'objet a détruire et ne sera détruit qu'à la toute fin du pas de temps courant.

2.1.2 Objets Dynamiques

Les objets dynamiques disposent de la méthode `selfUpdate` qui est appelée par le modèle après la gestion des collisions et qui s'exécutent différemment selon la classe et l'état de l'objet.

2.1.3 Diagramme de classe du modèle

Pour ce qui est du «moteur physique» du jeu, les classes abstraites et interfaces essentielles ont été décrites. Toutes les autres classes implémentent une ou plusieurs de ces superclasses pour définir leur comportement. L'organisation de ces sous-classes ne dépend plus maintenant de la manière dont le modèle est implémenté mais de la manière dont l'univers du jeu s'organise lui-même. Voici le diagramme de classe final du modèle :

2.1.4 Diagramme de séquence du modèle

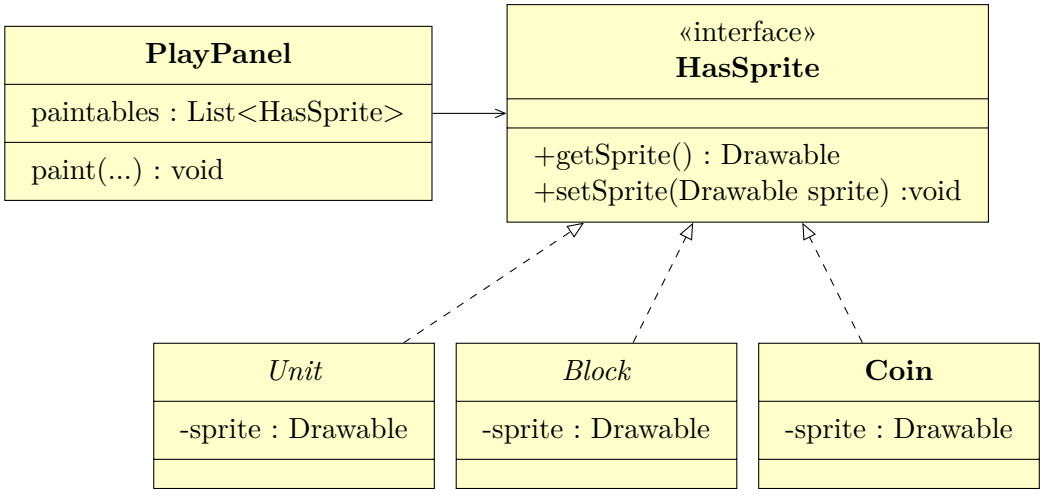
2.2 Implémentation de la vue

A chaque tick d'horloge, la vue doit présenter à nouveau à l'utilisateur tous les objets devant être représentés à l'écran. Ces objet dépendent bien sûr du modèle, et comme dit dans l'introduction, la vue questionne donc

celui-ci à intervalles réguliers. Cependant, nous avons préféré éviter que la vue demande directement à des objets du modèle de se représenter eux-même car ceci nuirait à l'indépendance des divers composants du jeu. Nous avons donc utilisé le pattern décorateur/pont pour garder une approche MVC rigoureuse : les objets représentables du modèle implémente l'interface `HasSprite` qui garantit simplement qu'ils sont capable de pointer vers un objet de la vue capable des les représenter adéquatement. Cet objet implémente l'interface `Drawable` et peut donc être appelé par la vue.

Ceci comporte aussi l'avantage habituel du pattern décorateur qu'il est possible de très facilement changer le sprite d'un objet, ce qui signifie que nous pouvons créer des objets ayant le même comportement physique côté modèle, mais des représentations différentes, et ce sans créer de nombreuses sous-classes superflues. Ceci a été fait pour créer deux monstres au comportement similaire, mais d'aspect différent.

```
public class PlayPanel{
  ...
  public void paint(Graphics g){
    ...
    for (HasSprite hS : paintables){
      hS.getSprite().draw(g, this, left,top, windowWidth, windowHeight);
    }
  }
  ...
}
```



2.3 Contrôle du Héros

3 Détails intéressants (?)