R3.04 – Qualité de code
TP 5 : Système d'entité/composant

Samuel Delepoulle
Franck Vandewiele

2024-2025

1 Objectif

Sur Moodle, vous pourrez récupérer les sources d'un petit jeu codé en Java. Pendant le jeu, des cibles apparaissent aléatoirement à l'écran. En cliquant sur une cible avant qu'elle ne disparaisse, le joueur marque des points. Le code de ce jeu repose sur un moteur d'entité/composant (Entity Component System), Ashley.

Dans un tel moteur, les éléments du jeu sont des *entités* munies de *composants* qui en décrivent les caractéristiques. La logique du jeu est prise en charge par un ensemble de *systèmes*, qui sont des objets qui parcourent à intervalle de temps régulier les entités disposant d'un ensemble cohérent de composants, afin d'en modifier l'état pour refléter l'évolution du jeu.

Votre objectif est d'ajouter de nouvelles fonctionnalités.

Avant toute chose, documentez-vous sur le moteur Ashley (`https://github.com/libgdx/ashley`) En particulier, consultez l'article du wiki qui rappelle le principe d'un ECS.

2 Classe App

La classe `App` est une classe exécutable pour la plate-forme JavaFX. Sa méthode `start()` réalise les initialisations suivantes :

- elle prépare une `Scene` JavaFX contenant un `Canvas`. Cette classe représente une zone sur laquelle on peut dessiner des formes et afficher des images ;
- elle initialise le moteur de Ashley en instanciant la classe `Engine`. C'est ce moteur qui gère la logique du jeu ;
- elle initialise des objets auxiliaires pour gérer le clavier, la souris et faciliter la création d'entités.
- elle enregistre un ensemble de systèmes auprès du moteur d'Ashley ;
- elle lance une instance de `GameLoopTimer` qui va mettre régulièrement à jour le jeu en appelant la méthode `tick()` 60 fois par seconde. Cette instance de `GameLoopTimer` fait office de boucle de jeu : au cours d'un tick d'horloge, l'appel à la méthode `update()` de `Engine` ddéclenche l'activation les uns après les autres des différents systèmes préalablement enregistrés auprès du moteur de jeu.

3 Composants

Les composants contiennent les données qui caractérisent les éléments du jeu.

Les composants Ashley implémentent l'interface `Component`. Cette interface ne prévoit aucune méthode abstraite et sert de marqueur pour identifier les composants. Les classes des différents composants sont situées dans le package `fr.iutlittoral.components`. Ce sont des classes de données pures, munies seulement d'attributs publics, de constructeurs et de méthodes techniques comme `toString()`. Elles sont destinées à représenter l'état des entités.

On y trouve :

— `AlphaDecay`, représentant qu'une entité va progressivement disparaître ; ce composant n'a pas d'attribut. S'il est présent, il agit comme un marqueur sur l'entité à laquelle il est attaché ;
— `BoxCollider`, représentant qu'une entité a une boîte englobante susceptible d'entrer en collision avec d'autres objets ;
— `BoxShape`, représentant qu'une entité est représenté par un rectangle dans l'écran de jeu ;
— `Bullet`, représentant qu'une entité est une balle tirée par le joueur dans la zone de jeu ;
— `CircleShape`, représentant qu'une entité a une vignette graphique circulaire dans l'écran de jeu ;
— `LimitedLifespan`, représentant qu'une entité a une durée de vie limitée ;
— `Position`, représentant les coordonnées d'une entité dans la zone de jeu ;
— `Shade`, représentant la couleur d'une entité destinée à être représentée à l'écran ; deux attributs, `color` et `currentColor` représentent une couleur de base et une couleur actuelle ;
— `Spawner`, représentant qu'une entité est un générateur de cibles ;
— `Target`, représentant qu'une entité est une cible qui peut rapporter des points lorsqu'elle est détruite ; le nombre de points accordés est porté par l'attribut `value` ;
— `Velocity`, représentant qu'une entité a une vitesse de déplacement dans la zone de jeu ;

4 Entités

Les entités sont les éléments du jeu. On peut leur attacher des composants. Les cibles, les balles tirées par le joueur et les générateurs de cibles sont des entités. Les propriétés d'une entité sont déterminées par les composants qui lui sont attachés. En ajoutant ou en retirant des composants à une entité, on change son comportement.

4.1 Cibles et balles

Les cibles et les balles tirées par le joueur ont de nombreux points communs. Elles sont représentées par des entités qui ont des composants :

— pour représenter leur position (`Position`) ;
— pour représenter la couleur avec laquelle elles sont dessinées (`Shade`) ;
— pour représenter leur durée de vie limitée (`LimitedLifespan`) ;
— pour représenter qu'elles disparaissent progressivement (`AlphaDecay`).

4.1.1 Cibles

Une cible dispose en plus :
— d'un composant qui la représente graphiquement par un rectangle (`BoxShape`) ;
— d'un composant représentant la boîte englobante avec laquelle une balle peut entrer en collision (`BoxCollider`) ;
— d'un composant qui l'identifie comme une cible rapportant des points (`Target`).

4.1.2 Balles

Une balle, en plus des quatre composants qu'elle a en commun avec une cible, dispose en plus :
— d'un composant qui la représente graphiquement par un cercle (`CircleShape`) ;
— d'un composant qui l'identifie comme une balle (`Bullet`).

4.2 Générateurs

Les générateurs de cibles sont des entités qui disposent de deux composants :
— un composant qui indique dans quelle zone des cibles peuvent générées et à quelle fréquence (`Spawner`) ;
— un composant qui identifie le type de cible qu'il génère (`SimpleBoxSpawnType` ou autre composant type.)

Remarque : Chaque entité ne peut disposer de plus d'un seul composant de chaque type. Une entité ne peut pas avoir plus d'une `Position`, plus d'une `LimitedLifespan`, etc. Cette limitation correspond à la façon intuitive de se représenter les composants.

5 Systèmes

Les systèmes Ashley contiennent la logique du jeu. Chaque système en représente une petite partie.

Les systèmes héritent de `EntitySystem`. La plupart du temps, les systèmes parcourent les entités et appliquent un traitement aux entités qui disposent de certains types de composants. Ces systèmes peuvent alors hériter de `IteratingSystem`, qui facilite l'écriture de leur code. Par exemple, le système qui gère le déplacement des entités s'intéressera aux entités qui ont une `Position` et une `Velocity` et mettra à jour la `Position` à chaque tick de la boucle de jeu en fonction de la `Velocity`.

Voici deux exemples de systèmes de ce jeu :
— `AlphaDecaySystem` modifie le composant `Shade` d'une entité en fonction de la durée de vie portée par son composant `LimitedLifespan` ;
— `BoxShapeRenderer` dessine dans un `Canvas` les composants de forme rectangulaire (`BoxShape`) à la bonne position (`Position`).

6 Fonctionnement d'un système : AlphaDecaySystem

`AlphaShadeSystem` hérite de la classe abstraite `IteratingSystem`, qui représente un système parcourant toutes les entités disposant d'un certain sous-ensemble de composants.

```
public class AlphaDecaySystem extends IteratingSystem {
        ComponentMapper<LimitedLifespan> lifespans = ComponentMapper.getFor(
            LimitedLifespan.class);
        ComponentMapper<Shade> shades = ComponentMapper.getFor(
            Shade.class);

        public AlphaDecaySystem() {
                super(Family.all(AlphaDecay.class,
                      LimitedLifespan.class,
                      Shade.class).get());
        }

        @Override
        protected void processEntity(Entity entity, float deltaTime) {
                LimitedLifespan lifespan = lifespans.get(entity);
                double decay = 1 - ((double)lifespan.elapsedLifespan /
                              (double)lifespan.totalLifespan);
                Shade shade = shades.get(entity);
                Color color = shades.color;
                shade.currentColor = color.deriveColor(1., 1., 1., decay);
        }
}
```

FIGURE 1 – Code de la classe `AlphaShadeSystem`.

On a rappelé son code en figure 1. Cette classe gère l'évolution d'un niveau de transparence en fonction du temps qu'une entité a passé dans la zone de jeu avant de disparaître.

Ce système s'intéresse donc uniquement aux entités qui disposent de ces trois types de composants :

— `AlphaDecay`, qui identifie qu'une entité doit progressivement disparaître ;
— `LimitedLifespan`, qui représente un compte à rebours avant le retrait d'une entité du jeu ;
— `Shade`, qui représente la couleur avec laquelle dessiner une entité du jeu.

Cette classe dispose d'attributs de type `ComponentMapper`. Il y en a deux : un pour les composants `LimitedLifespan` et un pour les composants `Shade`. Ces objets permettent de récupérer efficacement le composant d'un certain type d'une entité. Ils sont initialisés en faisant appel à la méthode `static getFor()` de `ComponentMapper`. Cette méthode prend en paramètre un objet de type `Class`, qui représente la classe d'un objet.

Le constructeur de la classe `AlphaDecaySystem` se contente d'appeler le constructeur de sa classe mère. Il lui transmet une instance de `Family`, qui représente un ensemble de contraintes que doit remplir une entité. Seules les entités remplissant ces contraintes seront traitées par ce système. Ainsi :

```
Family.all(AlphaDecay.class, LimitedLifespan.class, Shade.class).get()
```

représente la famille des entités qui possèdent un composant `AlphaDecay`, un composant `LimitedLifespan` et un composant `Shade`. La méthode `exclude()`, qui n'est pas utilisée ici, permettrait d'exclure les entités disposant de certains types de composants.

Une `Family` est un sélecteur qui joue le même rôle qu'une directive `WHERE` en SQL.

La méthode `processEntity(Entity entity, float deltaTime)` décrit le code à exécuter pour mettre à jour une entité répondant aux critères de recherche. La valeur `deltaTime` correspond au nombre de secondes écoulées depuis la dernière mise à jour (dans notre cas, $1/60^{\text{e}}$ de seconde.)

Le corps de la méthode illustre comment interroger les `ComponentMapper` pour obtenir les composants de l'entité à traiter.

L'objectif de `processEntity()` est rempli à la dernière ligne : modifier la couleur portée par le composant `Shade` pour la rendre transparente proportionnellement à la durée de vie écoulée de l'entité.

7 Travail à faire – partie 1

Exercice 1.

Examiner attentivement le code de la classe `SimpleBoxSpawnerSystem` et sa classe mère.

— À quoi correspond le paramètre transmis à la classe mère ?
— Quelle est la durée de vie des cibles qui apparaissent dans le jeu ? Essayer de la modifier et observer les effets pendant l'exécution.
— Dans la classe `App`, quelle entité est responsable de l'apparition des nouvelles cibles ? Que se passe-t-il si on la supprime ? Que se passe-t-il si on lui retire l'un de ses composants ? Expliquer.
— Que se passe-t-il si on retire le composant `AlphaDecay` aux entités créées dans la méthode `spawn()` de la classe `SimpleBoxSpawnerSystem` ? si on retire le composant `BoxCollider` ?

— Ajouter un composant `Velocity` aux entités créées dans la méthode `spawn()`. Que se passe-t-il en cours de jeu ?

Exercice 2.

Nous allons maintenant écrire le code qui permettra, en plus des cibles statiques qui apparaissent déjà, de faire apparaître des cibles qui se déplacent.

Comme pour toutes les nouvelles fonctionnalités dans un ECS, il faut procéder en trois étapes :

1. écrire le code des nouveaux composants dont nous avons besoin ;
2. écrire le code des nouveaux systèmes dont nous avons besoin ;
3. ajouter les entités dont nous avons besoin dans Ashley.

— Écrire un composant `MovingBoxSpawnType` dans `fr.iutlittoral.components.spawntypes`.
— Écrire un système `MovingBoxSpawnerSystem` qui générera des cibles mouvantes.
— Ajouter la ou les entités nécessaires au jeu pour que des cibles mobiles apparaissent en plus des cibles statiques.
— Enregistrer le système auprès du moteur Ashley.

8 Gestion du score

Dans l'état actuel des choses, lorsqu'une cible est détruite, le score n'est pas augmenté. Cette fonctionnalité n'a pas encore été implémentée. Le code qui affiche le « score » provient de la classe `App`, ligne 67 :

```
gc.fillText("Score 0", 10, 35);
```

C'est la classe `BulletCollisionSystem` qui est en mesure de détecter si une balle détruit un cible. Le code qui prend en charge la destruction d'une cible touchée se trouve en effet à la ligne 64 :

```
getEngine().removeEntity(targetEntity);
```

8.1 Score – v0.1a

Nous allons dans un premier temps mettre en œuvre une méthode simple pour que le score augmente à chaque cible détruite.

Exercice 3.

— déclarer un attribut entier `score` dans la classe `BulletCollisionSystem` ;
— ajouter un getter pour cet attribut ;
— à la toute fin de la méthode `update`, juste avant de supprimer la cible, aujoutez des points à l'attribut `score` ;
— dans la classe `App`, modifier `gc.fillText("Score 0", 10, 35);` de façon à tenir compte de la valeur du score connue par `BulletCollisionSystem`.

Tester.

Que pensez-vous de cette solution d'un point de vue SOLID ?

8.2 Événements, signaux et écouteurs

Ashley propose une solution pour découpler la gestion des collisions de celle du score. Cette solution se présente comme une variante du patron de conception `Observer`.

Dans cette variante, les sujets observables sont des **signaux**. Les observateurs sont des **écouteurs** qui peuvent s'abonner à des signaux. Lorsqu'un signal notifie ses écouteurs, il leur transmet un objet **événement** qui décrit la raison pour laquelle ils ont été notifié.

Dans cette approche :

- les événements peuvent être de n'importe quelle classe. Nous avons prévu une classe `TargetDestroyed` pour représenter un événement qui annonce qu'une cible a été détruite. Cette classe se trouve dans le package `fr.iutlittoral.events`. Elle dispose de plusieurs attributs qui décrivent combien la cible détruite valait de points et sa position ;
- les signaux sont d'un type générique `Signal<T>`, où `T` est la classe d'événement émis. La syntaxe est la même que pour `ArrayList<>`. Ainsi, un signal qui émet un objet `TargetDestroyed` relève de la classe `Signal<TargetDestroyed>` ;
- les écouteurs implémentent l'interface `Listener<T>`. Là aussi, `T` représente la classe des événements reçus. Ansi, un écouteur qui reçoit des événements `TargetDestroyed` implémente l'interface `Listener<TargetDestroyed>` ;
- l'interface `Listener<T>` prévoit une méthode
  `public void public void receive(Signal<T> signal, T event)`
- le signal notifie ses écouteurs d'un événement `event` grâce à sa méthode `dispatch(T event)`, où `T` est toujours un type générique ;
- un écouteur peut être enregistré auprès d'un signal en utilisant sa méthode `add(Listener<T> listener)`.

Remarque Dans Ashley, les signaux sont des classes concrètes, qui sont utilisées comme sujets auxquels on peut abonner des écouteurs. Les signaux ne sont pas des classes métier, contrairement aux sujets du patron Observer classique. Les signaux sont utilisés par les classes métier lorsqu'elles ont besoin de notifier des écouteurs.

8.3 Score – v1.0

Exercice 4.

Corriger la gestion du score pour la rendre plus SOLID :

- nettoyer la classe `BulletCollisionSystem` pour la décharger de la gestion du score.
- écrire le code d'une classe `Score` qui implémente `Listener<TargetDestroyed>`. Elle gèrera un attribut entier, l'exposera à travers un getter et pourra être notifié qu'une cible a été détruite, augmentant alors le score d'un certain nombre de points conformément à l'événement reçu ;
- dans `App`, déclarer et instancier un objet de type `Score` ;
- dans `App`, corriger l'affichage du score de façon à ce qu'il utilise la valeur du score provenant de l'objet précédent.
- la classe `BulletCollisionSystem` expose déjà un `Signal<TargetDestroyed>` à travers un getter. Dans la classe `App`, enregistrer l'instance de `Score` auprès de ce signal en utilisant la méthode `add()`.

9 Explosions

Exercice 5.

Lorsqu'une cible est détruite, elle disparaît instantanément.

Ajouter au jeu des explosions sous forme de particules, c'est-à-dire de petits carrés de couleur qui s'écartent très rapidement autour du point d'explosion et disparaissent aussitôt après qu'une balle ait détruit une cible.

Pour produire l'explosion elle-même, on peut écrire le code d'un `SpawnerSystem` dédié. À chaque fois qu'une cible est détruite, on ajoutera une entité portant un composant `Spawner` permettant de produire une explosion. Pour que l'explosion ne se prolonge pas infiniment, cette entité sera munie d'un composant `LimitedLifespan`.

10 Pour aller plus loin...

— un générateur de cibles qui accélèrent ;
— un générateur de cibles qui tournent en rond ;
— un générateur de bonus qui, une fois ramassé, fige toutes les cibles et les générateurs pendant 10 secondes ;
— un indicateur de temps de bonus restant ;
— un générateur de bonus, qui une fois ramassé, transforme pendant 20 secondes l'arme du joueur en une explosion de forme carrée qui détruit toutes les cibles rencontrées.
— etc.