CS-107 : Mini-projet 2
Jeux de plate-forme

J. Berdat, J. Sam, B. Jobstmann

Version 1.1

Table des matières

1 Présentation **3**

2 Schéma général de l'architecture **5**

3 Mise en place (étape 1) **7**

3.1 Affichages et points de vue 7

3.1.1 Premier affichage 7

3.1.2 Point de vue 9

3.2 Les acteurs 12

3.2.1 Ordre de priorité 13

3.2.2 Naissance et mort 14

3.3 Collisions et physique 16

3.3.1 Gestion des collisions 16

3.3.2 Premier acteur immobile : les blocs 18

3.3.3 Premier acteur mobile : les boules de feu 19

4 Noyau de base (étape 2) **23**

4.1 Le joueur 23

4.2 Instanciation d'un jeu 25

4.3 Le système d'interaction 27

4.3.1 Dégâts/effets . . . 27
4.3.2 Application des dégâts . . . 28
4.3.3 Les zones d'activation . . . 30
4.3.4 Relancement . . . 30
4.3.5 Ajout de nouveaux acteurs . . . 31
4.4 Validation de l'étape 2 . . . 34

5 « Puzzles et énigmes » (étape 3) 35
5.1 Les signaux . . . 35
5.2 Combinaison de signaux . . . 36
5.2.1 Négation . . . 36
5.2.2 Conjonction/disjonction . . . 36
5.3 Composants dépendant de signaux . . . 37
5.3.1 Portes et clés . . . 37
5.3.2 Leviers . . . 38
5.3.3 Ascenseurs . . . 38
5.3.4 Porte de sortie . . . 39
5.4 Validation de l'étape 3 et résultat du projet . . . 40

6 Aller plus loin 43
6.1 Variation autour des composants existants . . . 43
6.2 Enrichissement du visuel . . . 44
6.3 Composants avancés pour puzzle . . . 45
6.4 Divers projectiles . . . 47
6.4.1 Bombe et missile . . . 48
6.5 Gestion de la transition de niveau . . . 49
6.6 Bonus et doses de soins . . . 49
6.7 Ennemis et alliés . . . 49

1 Présentation

Ce projet a pour objectif de vous faire programmer un petit moteur de jeux vous permettant de créer des jeux de plate-forme[Lien]. La concurrence est certes rude :

https://en.wikipedia.org/wiki/Spelunky

https://fr.wikipedia.org/wiki/Super_Meat_Boy

(et tant d'autres ...) mais nous nous bornerons, au vu des temps impartis, à des déclinaisons simples constituées des composants illustrés par la figure 1.

Vous pourrez, une fois l'outil à votre disposition, créer des réalisations concrètes de petits jeux de plate-forme au gré de votre fantaisie et imagination.

La mise en oeuvre d'un moteur de jeux, outre son aspect ludique, permet de mettre en pratique de façon naturelle les concepts fondamentaux de l'orienté-objet. Il s'agira de concevoir progressivement cet outil en complexifiant étape par étape les fonctionnalités souhaitées ainsi que les interactions entre composants. L'accent sera mis sur les problématiques nouvelles rencontrées à chaque étape et comment y faire face en se plaçant au bon niveau d'abstraction et en créant des liens adaptés entre les composants. Le but est de tirer parti des avantages de l'approche orientée objets pour produire des programmes facilement extensibles et adaptables à différents contextes.

Le projet comporte trois étapes :

- Étape 1 (« Mise en place ») : au terme de cette étape vous aurez mis en place l'essentiel de l'architecture de base de votre projet. Vous disposerez d'un outil basique permettant de créer des objets statiques ou mobiles simples, interagissant entre eux par simple collision.
- Étape 2 (« Noyau de base ») : au terme de cette étape vous disposerez d'un noyau de jeu vous permettant de créer des jeux de plate-forme élémentaires : un joueur se déplaçant sur des plate formes et interagissant de façon simples avec d'autres composants)[1].
- Étape 3 (« Jeux d'énigmes »)[Lien] : durant cette étape le moteur sera enrichi de composants aux interactions plus complexes, ce qui permettra de créer des jeux d'énigmes (le joueur doit par exemple ouvrir des portes au moyen de clés, allumer des torches, activer des leviers et/ ou utiliser des ascenseurs pour parvenir au but souhaité).

[1] Comme un jeu célèbre avec un plombier !

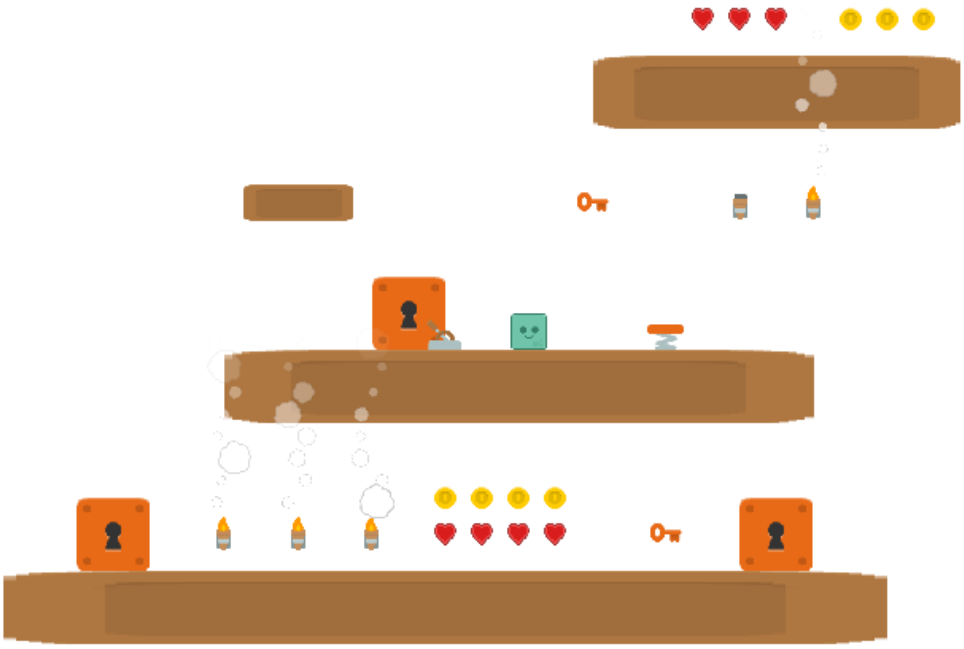


Fig. 1 : Le joueur (petit cube bleu) devra ramasser des clés, collecter des points de vie (cœur et pièces), allumer des torches, ouvrir des portes, prendre des ascenseurs et actionner des leviers pour parvenir à des objectifs (typiquement des portes de sortie lui permettant d'accéder à des niveaux de jeux).

Les deux premières étapes sont volontairement très guidées. Il s'agira essentiellement de prendre en main les outils fournis, de bien comprendre les problématiques soulevées à chaque fois et comment nous vous proposons d'y répondre[2].

Nous aurons bien sûr, des ambitions limitées quant aux capacités du moteur de jeu ; notamment pour ce qui touche aux aspects physiques et à la gestion des collisions. Une partie du matériel sera évidemment fournie.

[2]L'idée étant qu'en programmation, on apprend aussi beaucoup par l'exemple.

2 Schéma général de l'architecture

Le temps et les connaissances nécessaires pour implémenter l'entièreté du code nécessaire sont hors de portée de ce projet. Nous vous fournissons notamment quelques outils dans `platform.util`, qui vous seront présentés au fur et à mesure. Le code et la documentation devraient répondre aux détails, vous donnant ainsi l'occasion d'accéder à un code émanant de programmeurs plus expérimentés[3].

Votre code va donc s'insérer dans une architecture fournie, schématisée dans les grandes lignes par le diagramme de la Figure 2.

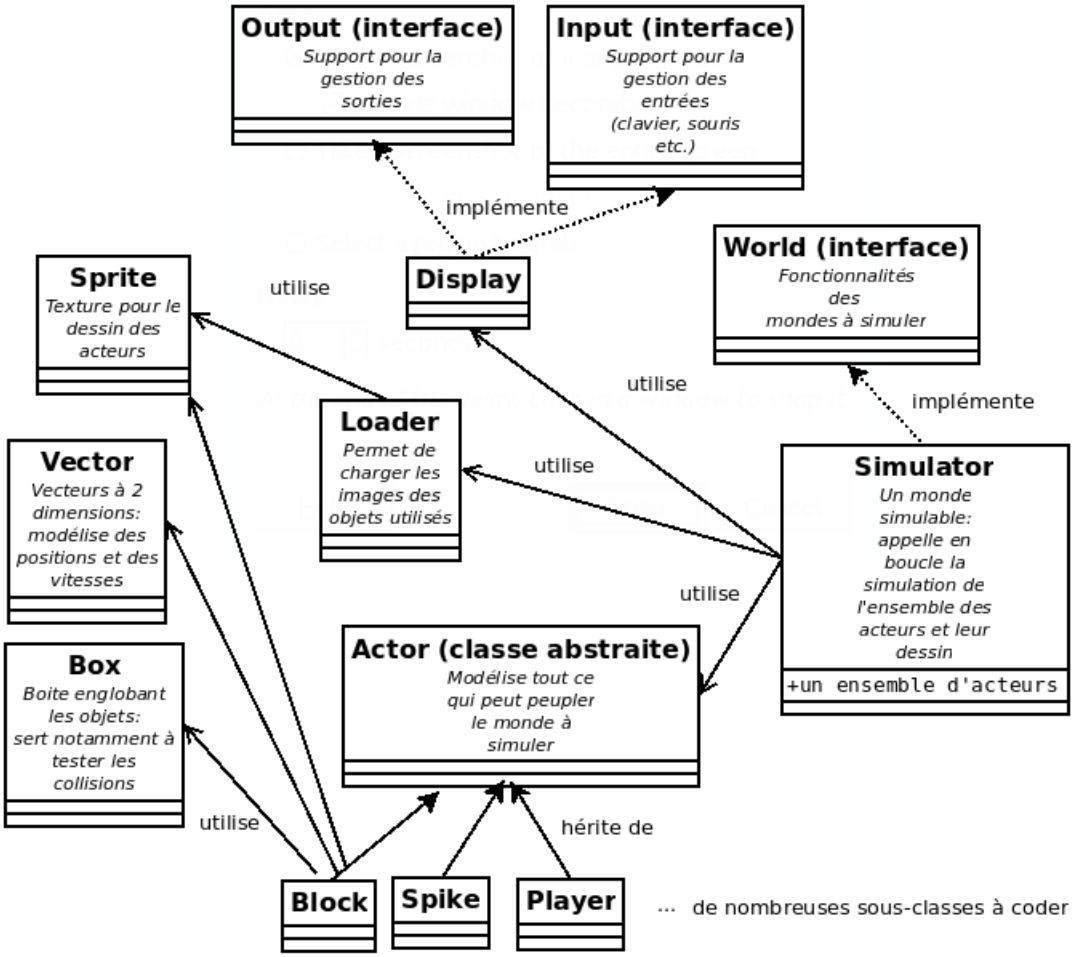


Fig. 2 : Classes principales du projet

- La classe `Simulator` joue le rôle de chef d'orchestre. Elle représente un monde à simuler et contiendra pour l'essentiel une collection d'acteurs (le joueur, les murs, et tout autre élément présent dans notre monde). La classe `Simulator` dispose donc d'une fonctionnalité permettant de faire évoluer le monde au cours du temps (méthode `update`). Dans ce projet, le programme principal, `Program.java`, créera

[3]Toujours dans l'esprit d'apprendre par l'exemple, même s'il n'est pas demandé de comprendre le code fourni dans les détails.

un objet de type `Simulator` et appellera en boucle sa méthode `update` jusqu'à que l'utilisateur décide d'y mettre fin en fermant la fenêtre. La méthode `update` mettra à jour l'état de chacun des acteurs et appellera leur méthode de dessin.

- Les objets du monde seront dessinés au moyen de petites images/textures (classe `Sprite`).
- La classe `Loader` permet de créer les `Sprite` nécessaires à partir de fichiers.
- Les classes `Vector` et `Box` seront omniprésente dans le code à réaliser. Elles permettront respectivement de modéliser des positions/vitesses et de gérer des collisions dans un environnement bi-dimensionnel.
- La méthode `update` de la classe `Simulator` doit par ailleurs pouvoir réagir à des événements extérieur (touches ou clics de souris) et disposer d'un support sur lequel dessiner le monde. Les interfaces `Input` et `Output` regroupent respectivement les fonctionnalités requises pour l'interaction avec le monde extérieur et pour le dessin sur un support graphique. Dans le code fourni, la classe `Display` qui implémente ces deux interfaces, va couvrir ces deux types de besoins.
- Enfin l'interface `World` dicte les fonctionnalités de base requises mettre en oeuvre un monde peuplé d'acteurs et simulable graphiquement. La classe `Simulator` implémente cette interface.

Notez qu'il ne vous est pas demandé de lire/comprendre la classe `Loader`, ainsi que les classes en charge du graphisme (notamment `Display`).

Pour commencer, nous ne disposerons pas encore d'un moteur de jeu, à proprement parler, mais d'une petite application graphique que nous ferons évoluer.

Le code du moteur de jeu sera placé dans `platform.game`. Ce paquetage contient l'ébauche des classes `Simulator` et `Actor` où vous allez principalement intervenir.

A ce stade, la classe `Simulator` est encore dépourvue d'acteurs. Son contenu se limite à une méthode `update` qui peut recevoir des signaux d'un objet `Input` et faire des affichages sur un objet `Output`. Elle implémente une interface `World` qui n'impose l'existence que d'une seule méthode pour le moment (`getLoader`, indiquant quelle objet va être en charge de créer des `Sprite` à partir de fichiers). Ouvrez les fichiers `Simulator`, `World`, `Input` et `Output` et familiarisez vous avec leur contenu.

D'autres classes plus avancées sont également fournies. Nous y reviendront en temps voulu.

3 Mise en place (étape 1)

Cette première partie du projet vise à vous faire prendre en main l'architecture fournie et à l'enrichir en commençant à y insérer votre propre code. Nous vous fournirons assez souvent du code "clé en main" qu'il suffira de placer aux bons endroits. Au fil du projet, nous indiquerons de moins en moins le code à ajouter, vous laissant plus de liberté et de responsabilités.

3.1 Affichages et points de vue

L'affichage graphique est évidemment un point essentiel puisqu'il nous permettra de *voir* concrètement ce qui se passe dans le monde. Voyons donc comment l'outillage existant vous permettra d'en réaliser. Et soyons fous, fixons nous l'objectif de dessiner un cœur...

Commencez par ouvrir le fichier `Simulator.java` : c'est sa méthode `update` qui va être en charge de réaliser l'affichage du cœur sur un support graphique `Output`.

Mais qui appelle `Simulator.update` ?

En fait, la méthode `main` de la classe fournie `Program` s'occupe de créer l'`Ouptut` (sous la forme d'un objet `Display`), crée un objet `Simulator` en lui associant un `Loader` et appelle dans une boucle infinie la méthode `update` de cet objet.

3.1.1 Premier affichage

Commencez par ajouter les lignes suivantes dans `Simulator.update`, afin de dessiner un cœur centré en $(100, 100)$, de largeur et hauteur 32.

```
Sprite sprite = loader.getSprite("heart.full");
Box zone = new Box(new Vector(100.0, 100.0), 32, 32);
output.drawSprite(sprite, zone);
```

Notez que l'utilisation du type `Sprite` requiert l'importation `import platform.util.Sprite;` en début de fichier. Vous ferez pareil pour les types `Vector` et `Box`.

Décryptons ce que nous venons de faire :

- La première ligne crée une image/texture (objet de type `Sprite`) associé au fichier nommé `heart.full`[4].
- La seconde crée une surface rectangulaire (objet de type `Box`).
- La troisième affiche une `Box` texturée au moyen du `Sprite`

Les coordonnées sont représentées au moyen de la classe utilitaire fournie, `Vector`[5].

Le cœur va s'afficher dans un système de coordonnées bidimensionnel, en pixels, ayant son origine en bas à gauche de la fenêtre graphique.

[4] Tous les fichiers liés aux textures sont dans le répertoire fourni `res`.

[5] Il s'agit donc de `platform.util.Vector`, à ne pas confondre avec `java.util.Vector`.

Les classes `Sprite`, `Vector` et `Box` sont fournies dans `platform.utils`. Elle seront omniprésentes dans ce projet. Nous vous invitons à étudier plus en détail le contenu des deux dernières car certaines de leur fonctionnalités vous seront utiles.

Voici deux exemples d'utilisation typiques des classes `Vector` et `Box` :

Les `Vector` permettent typiquement de représenter des positions et des vitesses dans un monde bidimensionnel :

```
Vector position = new Vector(2.0, 1.5);
Vector velocity = new Vector(0.0, 1.0);
double deltaTime = 0.1;
// calcul de la nouvelle position d'un objet après le temps
//deltaTime s'il se déplace à la vitesse velocity :
Vector newPosition = position.add(velocity.mul(deltaTime));
```

Les `Box` sont des surfaces rectangulaires caractérisées par leur coin inférieur gauche et leur coin supérieur droit :

```
Vector lowerLeftCorner = new Vector(-1.0, -1.0);
Vector upperRightCorner = new Vector(1.0, 1.0);
Box zone = new Box(lowerLeftCorner, upperRightCorner);
Box shiftedZone = zone.add(new Vector(5.0, 0.0));
```

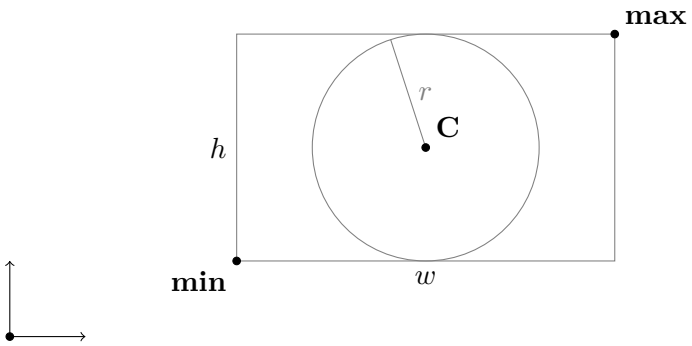
Maintenant que tout est compris, vous pouvez **exécuter** `platform.Program`, et ainsi voir un petit cœur sur fond blanc. Moment de grâce...

Notez que ce code est temporaire et sera remplacé par la suite. Il sert uniquement de test pour les premières étapes.

3.1.2 Point de vue

Nous savons maintenant dessiner un rectangle texturé. Nous ne l'avons cependant pas fait de manière très satisfaisante : le fait de définir nos coordonnées en pixels n'est pas du tout commode (un pixel est petit !) et nous rend dépendant de la taille de la fenêtre ce qui est contre-intuitif ; notre univers étant probablement plus grand que ce que l'on souhaite afficher. Il est donc assez vite nécessaire de pouvoir travailler avec des unités plus adaptées à nos besoins.

La classe `View` fournie permet de se focaliser sur une zone du monde caractérisée par un centre et un rayon. Il s'agit du rayon du cercle inscrit dans le rectangle de la fenêtre. Si cette dernière n'est pas carrée, il sera donc possible de voir plus loin.



La classe `View` fournie se charge de faire les changements de coordonnées requis, déléguant le réel travail à notre objet `Display`[6].

Concrètement, il suffit de créer un objet `View`, de choisir le point de vue désiré, et de l'utiliser de façon transparente. Par exemple, si l'on voulait que `Simulator.update` affiche le cœur dans une vue centrée en (0,0) et de rayon 10 on écrirait :

```
View view = new View(input, output);
Vector center = new Vector(0.0,0.0);
double radius = 10.0;
view.setTarget(center, radius);
Sprite sprite = loader.getSprite("heart.full");
// on exprime les coordonnées et la taille dans le
   nouveau référentiel :
Box zone = new Box(new Vector(0.0, 0.0), 2, 2);
// on dessine dans la vue :
view.drawSprite(sprite, zone);
```

Il faut au préalable avoir supprimé l'ancienne façon de dessiner le cœur.

Essayez ce code et jouez un peu avec les coordonnées du cœur dans la vue pour bien comprendre comment elles sont exprimées.

Bien entendu nous n'allons pas coder la création de la vue courante de façon aussi ad hoc.

[6] Tout comme ce dernier, elle implémente les interfaces `Input` et `Output`, permettant une utilisation similaire.

L'idée est maintenant de refléter dans la conception le fait qu'au monde à simuler soit associée une vue courante caractérisée par un rayon et un centre. Comme l'interface `World` dicte les fonctionnalités de base requises pour mettre un oeuvre un monde simulable graphiquement, il semble naturel qu'elle impose la fonctionnalité permettant de se focaliser sur une vue particulière (la gestion de la vue doit faire partie des fonctionnalités requises pour les mondes à simuler).

Ajoutez donc la méthode suivante à l'interface `World`.

```
/**
 * Set viewport location and size.
 * @param center viewport center, not null
 * @param radius viewport radius, positive
 */
public void setView(Vector center, double radius);
```

Afin d'implémenter cette méthode, la classe `Simulator` a besoin de mémoriser les valeurs souhaitées en guise de centre et de rayon dans des attributs, correctement initialisés dans le constructeur.

```
private Vector currentCenter;
private double currentRadius;

/**
 * Create a new simulator.
 * @param loader associated loader, not null
 */
public Simulator(Loader loader, String[] args) {
  if (loader == null) {
    throw new NullPointerException();
  }
  this.loader = loader;
  currentCenter = Vector.ZERO;
  currentRadius = 10.0;
}

@Override
public void setView(Vector center, double radius) {
  if (center == null)
    throw new NullPointerException();
  if (radius <= 0.0)
    throw new IllegalArgumentException("radius must be
       positive");
  currentCenter = center;
  currentRadius = radius;
}
```

Notez l'utilisation d'exceptions[7] pour indiquer une mauvaise condition d'utilisation de la méthode. Pour des raisons de brièveté, nous n'indiquerons plus l'entièreté du code à ajouter, ni les commentaires et documentation.

[7] Nous expliquerons ces tournures en semaine 12 du cours.

Enfin, remplacez les lignes qui créaient la vue "en dur" par les lignes suivantes au début de la méthode `update` :

```
View view = new View(input, output);
view.setTarget(currentCenter, currentRadius);
```

ce qui permet de configurer la vue pour cette étape de simulation.

Le cœur devrait apparaître au centre de l'écran, comme auparavant.

Par la suite, la position de la vue dépendra du jeu, et plus particulièrement du personnage contrôlé par le joueur. Cependant, afin de mieux tester, nous allons centrer la vue à l'endroit cliqué par le joueur. Pour cela, après le dessin du cœur, **ajoutez** les lignes suivantes.

```
if (view.getMouseButton(1).isPressed())
  setView(view.getMouseLocation(), 10.0);
```

À chaque clic du bouton gauche de la souris, la position de la vue est modifiée : la position cliquée devient le centre de la vue (toujours placée au centre de la fenêtre). Le déplacement instantané de la vue, ainsi occasionné, est toutefois peu agréable.

Nous allons terminer cette section en ajoutant une transition plus lente d'une vue vers l'autre. Pour cela, le nouveau centre de la vue (l'objectif à atteindre) sera séparé de son centre courant. La vue va ainsi voir sa position courante changer en direction de la position à atteindre, jusqu'à ce que cet objectif soit atteint. Nous ferons pareil pour le rayon.

Ajoutez pour cela deux attributs `expectedCenter` et `expectedRadius`, de type `double`, sans oublier de les initialiser (à `Vector.ZERO` et 10.0 respectivement).

La méthode `setView` doit désormais initialiser `expectedCenter` et `expectedRadius` plutôt que `currentCenter` et `currentRadius`.

Afin de permettre une transition douce de la vue, ajoutez les lignes suivantes au début d'`Simulator.update` :

```
double factor = 0.001;
currentCenter = currentCenter.mul(1.0 -
   factor).add(expectedCenter.mul(factor));
currentRadius = currentRadius * (1.0 - factor) +
   expectedRadius * factor;
```

La vue devrait désormais transiter de façon fluide vers ses nouvelles caractéristiques.

À la fin de cette section, le simulateur est prêt pour afficher des éléments, la méthode `setView` permettant de choisir le point de vue du monde.

3.2 Les acteurs

La classe `Simulator` n'est en réalité pas destinée à définir la mécanique de jeu ni ses modalités de dessin.

Son rôle doit être uniquement de déléguer les actions à entreprendre ainsi que le dessin aux objets composant le jeu, les acteurs[8].

Les acteurs seront des instances de la classe `Actor`, pour l'instant vide.

La première chose à faire ici est donc de doter le monde à simuler (`Simulator`), d'un attribut `actors` représentant l'ensemble des acteurs qui y évoluent.

Se pose alors le problème du type à donner à cet attribut. Laissons cette question en suspens pour le moment.

Chaque acteur doit pouvoir évoluer au cours du temps et bien sûr être dessinable, ce qui nous fait penser aux méthodes suivantes dans la classe `Actor` :

```
// pour évoluer au cours du temps :
public void update(Input input) {}

// pour être dessiné
public void draw(Input input, Output output) {}
```

Il est justifié ici de mettre un corps vide à la place de définir ces méthodes comme abstraites. En effet, de nombreux acteurs ne feront rien par défaut (murs, obstacles, portes, clés etc.) et seront invisibles (comme les signaux, que nous décrirons plus tard). Nous partons donc d'un choix de conception selon lequel, par défaut, un acteur ne fait rien et est invisible. Ici on ne met donc pas un corps vide parce qu'on ne sait pas définir, mais un corps vide pour dire "ne fait rien".

Libre à vous cependant de concevoir cet aspect différemment si vous l'estimez plus pertinent quand votre projet évoluera.

Pour ce qui est du choix des paramètres des méthodes `Actor.update` et `Actor.draw` :

- L'évolution d'un acteur peut dépendre d'événement extérieurs (d'où le paramètre `Input`).
- Le dessin aussi (on peut imaginer de dessiner à la main certaines choses).
- Le dessin va devoir en outre se faire sur un support (`Output`).

La vue courante jouera pour nous le rôle d'`Input` et d'`Output`. Souvenez-vous que `View` implémente les deux interfaces.

[8] Déléguer les tâches aux classes appropriées est une nécessité pour une bonne conception orientée objet.

La méthode `Simulator.update` aura donc pour rôle d'appeler en séquence les méthodes `update` et `draw` de chacun de ses acteurs :

```
// Apply update
for (Actor a  : actors)
  a.update(view);

// Draw everything
for (Actor a  : actors) // sera retouchée un peu plus loin
  a.draw(view, view);
```

Cela garantira que les acteurs soient dessinés une fois que tout le monde a été mis à jour, et non au milieu de changements.

Rappelez vous que la méthode `Simulator.update` est appelée en boucle par le programme principal : une itération de la boucle est ce que l'on appelle *un pas de la simulation*.

Vous pouvez maintenant supprimer ou commenter le code dessinant le cœur.

Libre à vous de conserver le code qui déplace la vue, jusqu'à ce que nous l'attachions au personnage[9].

3.2.1 Ordre de priorité

Revenons à la question du choix du type pour l'attribut `actors`. Si on utilise un tableau quelconque, comme un `ArrayList`, les mises à jour (`update`) se feront dans l'ordre de stockage dans l'attribut. Or cet ordre peut avoir une influence non négligeable sur l'enchaînement des événements : simuler le méchant avant le gentil fera que le premier pourra tirer son épée avant que le second ne puisse se protéger de son bouclier et cela n'aura pas le même effet que si ces deux acteurs sont simulés dans l'ordre inverse !

Il y a donc intérêt à ce que l'ensemble `actors` soit trié selon un critère de priorité associé aux acteurs. La classe `platform.util.SortedCollection` permet de modéliser une liste ordonnée. Vous pouvez donc remplacer le `ArrayList` par `SortedCollection`. Les éléments de la collection sont toutefois alors tenus d'implémenter l'interface `Comparable<Actor>` : pour que la collection soit ordonnée il faut en effet que ses éléments soient comparables entre eux !

Implémenter l'interface `Comparable<Actor>` impose donc à la classe `Actor` de redéfinir la méthode suivante :

```
@Override
public int compareTo(Actor other) {..}
```

Nous allons pour cela tout simplement instaurer un système de priorité. Un acteur avec une priorité haute sera simulé avant un acteur de basse priorité. Il vous est donc demandé maintenant :

- d'associer aux acteurs une méthode `getPriority()` retournant un entier qui représente le niveau de priorité de l'acteur. Vous considérerez que cette méthode ne peut

[9] Par abus de langage nous l'appellerons parfois « le joueur ».

être définie concrètement pour un acteur quelconque.

- de définir la méthode :

  ```
  public int compareTo(Actor other)
  ```

  retournant -1 si `this` a un niveau de priorité supérieur à celui de `other`, 0 si `this` et `other` ont la même priorité et 1 sinon.

De ce fait, un acteur avec une priorité haute sera placé avant celui avec priorité faible dans la collection d'acteurs, ce qui permettra de le simuler en premier.

Les priorités utilisées seront choisies afin que les petits éléments actifs soient prioritaires sur les gros acteurs peu réactifs.

Il faut alors aussi faire en sorte que les objets prioritaires se dessinent par-dessus les moins prioritaires : par exemple le joueur (plus prioritaire) sera dessiné sur les blocs qui composeront l'arrière plan (éléments statiques peu prioritaires).

Retouchez dans ce sens la méthode de dessin du monde à simuler (`Simulator.draw`), en la modifiant comme suit :

```
// Draw everything
for (Actor a : actors.descending())
  a.draw(view, view);
```

Notez ici qu'au vu de la structure de données désormais utilisée, les itérations sur ensembles de valeurs deviennent le seul outil (utilisable à ce stade) pour parcourir la collection.

3.2.2 Naissance et mort

Il faut maintenant penser à une facette importante : comment faire venir au monde les acteur du jeu et les en faire disparaître[10] ? Il faut concrètement pouvoir ajouter ou enlever un `Actor` dans `actors`.

Se pose alors à nous un nouveau problème. Que faire lorsque la simulation d'un acteur cause la création de nouveaux acteurs : l'acteur représentant le joueur lance des boules de feu lorsqu'il rencontre un mur de glace par exemple. Que faire aussi lorsqu'un acteur est amené à disparaître.

Rappelons que la boucle simulant le comportement des acteurs à cette allure :

```
for (Actor a : actors)
  a.update(view);
```

un acteur lançant des boules de feux signifie ici que l'appel à `update` sur cet acteur, devrait causer la modification de l'ensemble `actors` (ajout des acteurs représentant les boules de feu) !

Or, modifier le contenu d'une collection pendant son parcours au moyen d'une itération sur ensemble de valeurs n'est pas possible[Lien].

[10] ... car nous nous adonnerons à certaines violences parfois :-/

L'idée est donc d'enregistrer les nouveaux venus, ou ceux à disparaître, dans des listes d'attente, et de mettre à jour `actors` après que tous ses éléments aient reçu les événements `update` et `draw`.

Supposons donc que l'on ait doté `Simulator` de deux attributs `registered` et `unregistered` (de type `List<Actor>`) représentant ces listes d'attentes. L'idée serait donc d'ajouter le code suivant à la suite de celui déjà présent dans `Simulator.update` :

```
// Add registered actors
for (int i = 0; i < registered.size(); ++i) {
  Actor actor = registered.get(i);
  if (!actors.contains(actor)) {
    actors.add(actor);
  }
}
registered.clear();

// Remove unregistered actors
for (int i = 0; i < unregistered.size(); ++i) {
  Actor actor = unregistered.get(i);
  actors.remove(actor);
}
unregistered.clear();
```

(les nouveaux venus seront alors simulés lors du prochain pas de simulation).

Cela présuppose bien entendu que l'on dispose désormais du moyen de « peupler » ces listes d'attentes quand un nouvel acteur apparaît dans le monde ou doit disparaître du monde et cela doit faire partie des fonctionnalités de base d'un monde simulable. Ainsi, **ajoutez** à `World` et `Simulator` les méthodes suivantes :

```
@Override
public void register(Actor actor) {
  registered.add(actor);
}

@Override
public void unregister(Actor actor) {
  unregistered.add(actor);
}
```

Le constructeur de `Simulator` pourra initialiser les attributs `registered` et `unregistered` au moyen de `ArrayList` vides.

Voilà, à ce stade, nous avons assis quelques points importants de la conception. Nous devons maintenant poursuivre un peu nos efforts pour mettre en scène des acteurs concrets.

3.3 Collisions et physique

Certains acteurs seront mobiles, sans quoi on s'ennuierait un peu. Leurs déplacements peuvent se faire selon des modèles physiques plus ou moins complexes (des volutes de fumées qui font des sinusoïdes, des projectiles à trajectoire paraboliques etc.).

Réalistes ou simplistes, les algorithmes définissant le comportement des objets peuvent grandement varier d'un jeu à l'autre. De plus, en se déplaçant, les acteurs seront amenés à faire des rencontres avec d'autres acteurs. Ce qui permettra de mettre en oeuvre toute la logique du jeu.

Un point important est donc aussi de pouvoir détecter les collisions entre acteurs, ce qui selon la forme des objets peut être non trivial.

Nous adopterons des modèles très simplifiés et ad hoc pour mettre en oeuvre ces aspects[11]. Ce qui limitera les possibilités mais permettra d'aller suffisamment loin dans les temps impartis.

3.3.1 Gestion des collisions

Pour la gestion des collisions nous nous simplifierons les choses en associant à chaque acteur un rectangle l'englobant qui sera aligné aux axes (càd sans rotation).

Tester l'intersection de leurs rectangles englobant permet alors de savoir si deux acteurs se touchent.

Se pose alors à nous le problème de savoir comment gérer la collision une fois détectée.

On pourrait partir sur un modèle totalement *ad hoc* où chaque paire d'acteurs fait l'objet d'un algorithme particulier : si le joueur rencontre un mur il rebondit, s'il rencontre un fantôme les deux se repoussent mutuellement etc.

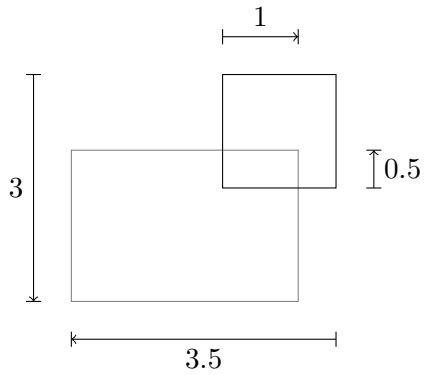
Nous ne disposons pas encore de tout l'outillage orienté objet nécessaire[12]. Nous allons donc essayer d'abstraire un peu les comportements possibles pour avoir un modèle suffisamment général sans pour autant nécessiter de traiter individuellement tous les cas particuliers. Les hypothèses simplificatrices (et de ce fait réductrices) seront adoptées :

- Un objet pourra être soit solide soit non solide, mais pas les deux en même temps : un objet solide est simplement un objet sur lequel on peut se poser : les projectiles et la fumée par exemple ne sont pas solides, tandis que les murs et les portes le sont.
- Si un objet solide se trouve en collision avec un objet non solide, alors ce dernier doit être repoussé (il rebondit). Pour ce faire, il s'agira de trouver la direction pour laquelle il y a le moins de distance à parcourir.

[11] Pas d'utilisation d'un moteur physique. On aurait pu songer à la bibliothèque externe, JBox2D, basée sur l'excellent Box2D.

[12] Comme les « design patterns » que vous aborderez en partie au second semestre.

Dans l'exemple suivant, le petit cube doit être repoussé vers le haut.



Un acteur n'interagira qu'avec les autres objets de priorités plus faibles : dans l'exemple précédent, le petit cube est prioritaire, et peut donc recevoir l'événement (être repoussé).

Le gros rectangle n'a pas connaissance du petit cube, car il n'a pas la priorité nécessaire.

À noter aussi que la comparaison est stricte, deux objets de même priorité n'interagissent pas. Vous noterez aussi qu'être solide n'implique pas l'immobilité, une plate-forme doit être solide pour que le joueur puisse s'y poser sans pour autant être immobile.

Pour ce qui est de la gestion des interactions entre acteurs, la classe `Simulator` continuera de déléguer le travail aux acteurs.

La classe `Actor` sera dotée d'une méthode lui permettant de gérer l'interaction avec un autre acteur :

```
public void interact(Actor other) {}
```

Au préalable, cette classe aura été dotée de l'outillage permettant d'associer un acteur à son rectangle englobant et de le répertorier comme solide ou non :

```
public boolean isSolid() {
  return false;
}

public Box getBox() {
  return null;
}

public Vector getPosition() {
  Box box = getBox();
  if (box == null)
      return null;
  return box.getCenter();
}
```

Par défaut nous considérons donc qu'un acteur est non solide, qu'il n'a ni rectangle englobant ni position.

La classe `Simulator` de son côté se contentera de considérer chaque paire d'acteurs et de les faire interagir dans le bon ordre, ce qui signifie que `Simulator.update` doit être enrichie du code suivant :

```
for (Actor actor  : actors)
  for (Actor other  : actors)
    if (actor.getPriority() > other.getPriority())
      actor.interact(other);
```

Ce code sera placé avant la boucle simulant le comportement de chaque acteur (`update`) afin que les changements d'états liés aux collisions éventuelles puisse être pris en compte. C'est donc ici l'acteur prioritaire qui vérifiera s'il y a collision et le cas échéant décidera de l'action à entreprendre.

Pour l'instant, trois types d'événements sont exécutés à chaque étape de la simulation : la gestion des interactions, la gestion de l'évolution (physique) et le dessin.

Afin de laisser la possibilité aux acteurs d'agir avant la physique et après le dessin, définissez les événements `preUpdate` et `postUpdate`, similaires à `update`.

N'oubliez pas d'invoquer ces événements dans `Simulator.update` aux bons endroits.

Par défaut, un acteur ne fera rien dans ses méthodes `preUpdate` et `postUpdate`.

La conception est maintenant suffisamment établie pour nous permettre de créer des acteurs concrets.

Dans ce qui suit vous allez créer deux types d'acteurs et les faire interagir par le biais de collisions.

3.3.2 Premier acteur immobile : les blocs

Notre premier acteur concret sera un simple cube immobile et solide.

Créez un type d'acteur particulier nommée `Block`, caractérisé par son rectangle englobant (une `Box`) et le `Sprite` permettant de le représenter graphiquement. Ces attributs seront initialisé dans le constructeur.

Il aura comme niveau de priorité la valeur zéro, ce qui est fixé de façon totalement arbitraire ici, pour indiquer qu'il est de basse priorité.

La méthode de dessin fera appel à la méthode de dessin de la super-classe.

Notez que pour tout type d'événement (`draw`, `update` etc.), il est important de faire appel aux méthodes de la super-classe afin de préserver les comportements hérités de plus haut. Si le `Sprite` associé est non `null` il sera dessiné au moyen de la `Box` de l'acteur.

Afin de tester, ajoutez à la fin du constructeur de `Simulator` des lignes permettant d'enregistrer deux `Blocks` dans le monde :

- l'un ayant une box de sommets extrêmes (-4, -1) et (4, 0)

- l'autre une Box de sommets extrêmes (-2, 0) et (-1, 1)

Les deux `Block` utiliseront l'image associée à *`"box.empty"`*.

Le lancement de `Program.Java` devrait vous donner ceci :

3.3.3 Premier acteur mobile : les boules de feu

Comme premier objet dynamique, nous allons ajouter une boule de feu qui rebondit[13].

Cet acteur aura une position et une vitesse qui évolueront au cours du temps. Pour l'instant, telle que déclarée dans `Actor`, la position dépend du rectangle englobant.

Pour des acteurs mobiles, dont la taille est fixe, il est plus aisé de plutôt calculer le rectangle englobant à partir de la position (qui évolue).

Créez la classe `Fireball`, avec pour attributs une position et une vitesse (de type `Vector`). La méthode `getBox()` peut y être redéfinie afin de calculer la `Box` englobante à partir de sa position :

```
@Override
public Box getBox() {
  // position est l'attribut position de l'objet
  // SIZE une constante choisie pour la taille, par exemple
     0.4
  return new Box(position, SIZE, SIZE);
}
```

Les position et vitesse seront initialisées à la construction et la priorité associée à l'objet sera très haute (pourquoi pas 666). La tentative de création d'un `Fireball` avec une position ou une vitesse valant `null` devra se solder par le lancement d'une `NullPointerException()` :

```
throw new NullPointerException();
```

Intéressons nous maintenant au codage de la méthode `update` qui doit prendre en charge la gestion du déplacement. Mathématiquement, si $\mathbf{x}(t)$ est la position en fonction du temps d'un point, la dérivée $\frac{\partial}{\partial t}\mathbf{x}(t) = \dot{\mathbf{x}}(t)$[14] est sa variation au cours du temps. Autrement dit,

[13] Nous avons le droit, nous ne sommes pas en section Physique :-)

[14] La notation avec le point symbolise la dérivée en fonction du temps, typique en physique. Vous êtes peut-être plutôt habitués à la notation $\mathbf{x}'(t)$.

la vitesse du point. De même, $\frac{\partial}{\partial t}\dot{\mathbf{x}}(t) = \frac{\partial^2}{\partial t^2}\mathbf{x}(t) = \ddot{\mathbf{x}}(t)$ est la variation de la vitesse, que l'on nomme accélération. L'opération inverse de la dérivée est l'intégrale, qui permet de déterminer $\mathbf{x}$ et $\dot{\mathbf{x}}$ à partir de $\ddot{\mathbf{x}}$. Mais tout ceci est au sujet de vos cours *Analyse I*, *Analyse II* et *Physique Générale I*. Nous allons donc ignorer les détails mathématiques et présenter la solution concrète.

À chaque instant t, nous connaissons l'accélération, c'est-à-dire la somme des forces divisée par la masse, selon la loi de Newton. Dans le cas de notre boule de feu, il s'agira uniquement de la gravité :

$$\ddot{\mathbf{x}}^{(t)} = \frac{1}{m}\sum_i f_i^{(t)} = \mathbf{g}$$

Intégrer n'étant pas une opération triviale, nous allons faire l'approximation suivante, valable si Δt est suffisamment petit :

$$\dot{\mathbf{x}}^{(t)} = \dot{\mathbf{x}}^{(t-1)} + \Delta t\ \ddot{\mathbf{x}}^{(t)}$$
$$\mathbf{x}^{(t)} = \mathbf{x}^{(t-1)} + \Delta t\ \dot{\mathbf{x}}^{(t)}$$

Simuler la physique de la boule de feu, revient donc à coder sa méthode `update` selon le modèle suivant :

```
@Override
public void update(Input input) {
  super.update(input);
  double delta = input.getDeltaTime();
  Vector acceleration = new Vector(0.0, -9.81);
  velocity = velocity.add(acceleration.mul(delta));
  position = position.add(velocity.mul(delta));
}
```

Il est vraisemblable que l'accélération soit utile dans d'autres contextes que les boules de feu (toute sorte d'objets imaginables peuvent en avoir besoin). Il est dès lors plus censé d'en faire une donnée relative au monde simulé.

Faites en sorte que l'interface `World` impose l'existence d'une méthode `getGravity` retournant le Vector `(0.0, -9.81)`.

L'idée est donc de remplacer la variable `acceleration` dans le code ci-dessus, par la valeur retournée par la méthode `getGravity` du monde simulé.

Se pose alors à nous le problème suivant : **un acteur a désormais besoin d'avoir des informations sur le monde auquel il appartient**.

Ce sera probablement vrai pour toutes sortes de raison et pour de nombreux acteurs. Pour qu'un acteur puisse savoir à quel monde il appartient on peut envisager la solution consistant à doter la classe `Actor` d'un attribut `world` de type `World`.

Cet attribut doit être initialisé et le bon moment de le faire est quand l'acteur s'enregistre dans le monde.

Définissez pour cela les méthodes suivantes dans `Actor` :

```
public void register(World world) {
    this.world = world;
}

public void unregister() {
    world = null;
}
```

Invoquez ensuite ces méthodes de façon appropriée dans `Simulator.update` : lorsqu'un acteur est ajouté à l'ensemble des acteurs, il doit initialiser son attribut monde en faisant l'appel `register(this)` (`this` est alors l'instance courante du monde). Un traitement analogue doit être fait lorsqu'un acteur est supprimé de l'ensemble des acteurs.

Mais .. nouvel os : l'attribut `world` de `Actor` doit être privé. On ne peut donc toujours pas accéder à la gravité par une tournure de type `world.getGravity()` dans `Fireball` !

Les classes `Actor` et `World` sont en fait intimement liées. Il est admissible ici qu'un getter `getWorld()` protégé permette à tous les programmeurs d'extension travaillant dans le paquetage `platform.game` d'accéder au monde d'un acteur.

Codez ensuite la méthode `draw` de `Fireball` en procédant de façon analogue à ce que vous avez fait pour les `Block`. Le `Sprite` associé à la `Fireball` doit être celui retourné par l'invocation de `getLoader().getSprite("fireball")` sur le monde de l'acteur.

Indications :

- Il est préférable ici de définir une méthode protégée `getSprite(String name)` dans `Actor` afin de faciliter l'accès aux textures dans les sous-classes de `Actor`.
- Pour que le `Sprite` tourne sur lui même au cours du temps, vous pouvez invoquez la méthode de dessin comme suit :

  ```
  output.drawSprite(sprite, getBox(), input.getTime());
  ```

A l'image de ce que vous avez fait pour les blocs, **ajoutez** une ligne dans le constructeur de `Simulator` pour créer une boule de feu en $(3, 2)$, avec une vitesse de $(-3, 5)$ (la taille de 0.4, par exemple, pourra être une constante de la classe).

Normalement, la boule de feu devrait se déplacer selon une parabole, traverser les blocs et s'en aller vers l'infini.

Introduisons maintenant comme premier exemple de gestion des collisions, celui de la boule de feu avec les blocs : la boule de feu doit pouvoir rebondir sur ces derniers.

La méthode `interact` de `Actor` ne fait rien par défaut. Il faut donc la redéfinir proprement dans `Fireball`. Comme il s'agit de la toute première fois que vous mettez en place une interaction concrète, le code vous est donné en guise d'exemple (vous vous en inspirerez pour les nombreux autres type d'objets que vous pourrez créer par la suite).

```
@Override
public void interact(Actor other) {
  super.interact(other);
  if (other.isSolid()) {
    Vector delta = other.getBox().getCollision(position);
    if (delta != null) {
      position = position.add(delta);
      velocity = velocity.mirrored(delta);
    }
  }
}
```

Rappelez-vous de ce qui a été présenté précédemment : la boule de feu n'est pas solide et va donc réagir face aux objets solides. Pour cela, la méthode `Box.getCollision` calcule la direction la plus courte pour sortir le point du rectangle. Si le point n'est pas dans le rectangle, alors `null` est retourné. Dans le cas contraire, la position est modifiée pour sortir de l'objet et la vitesse est reflétée pour simuler un rebond.

Pour valider les objectifs de cette étape, vous vous contenterez de vérifier qu'au lancement de `Program.java`, la boule de feu rebondit bien sur les blocs solides.

4 Noyau de base (étape 2)

Il est temps de prendre un peu le contrôle du jeu. Le but de cette étape est d'ajouter un acteur un peu particulier, le joueur. Lors de cette étape vous mettrez également en place un système d'interaction plus général entre objets (nos boules de feu ne se contenteront plus de rebondir sur les blocs, elles pourront avoir un effet sur le monde).

4.1 Le joueur

Le joueur sera aussi un acteur. La différence principale est qu'il doit pouvoir être contrôlé depuis l'extérieur : nous allons donc pouvoir faire usage du paramètre `Input` de sa méthode `update` !

Ajoutez la classe `Player`, caractérisée par une position et une vitesse, de manière similaire à `Fireball`. Surchargez `getPosition` ainsi que `getBox` (le rectangle englobant aura une taille 0.5×0.5 et sera centré sur la position du joueur).

Le joueur est non solide et aura une priorité de 42, pour montrer sa toute puissance... relative.

Enfin, utilisez l'image *`"blocker.happy"`* pour dessiner le personnage.

Dans le même esprit que pour la la boule de feu, **surchargez** les méthodes `update` pour simuler l'évolution et `interact` pour réagir aux objets solides.

On ne cherche pas encore à contrôler le joueur ici. La seule différence avec la boule de feu pour le moment est que le joueur ne va pas rebondir. Il doit s'arrêter ce qui, en clair, signifie de mettre à zéro sa vitesse selon les modalités suivantes :

```
@Override
public void interact(Actor other) {
  super.interact(other);
  if (other.isSolid()) {
    Vector delta = other.getBox().getCollision(getBox());
    if (delta != null) {
      position = position.add(delta);
      if (delta.getX() != 0.0)
        velocity = new Vector(0.0, velocity.getY());
      if (delta.getY() != 0.0)
        velocity = new Vector(velocity.getX(), 0.0);
    }
  }
}
```

Comme pour chaque objet que l'on veut tester, **ajoutez** une ligne dans le constructeur de `Simulator` pour créer un joueur (avec par exemple pour position $(2, 3)$ et pour vitesse $(0, -1)$).

Normalement, il devrait tomber et se poser sur un bloc, et rester immobile pour l'éternité.

Nous allons donc ajouter de quoi le contrôler pour le faire avancer.

Dans `update`, **ajoutez** le code suivant avant celui mettant à jour la position et la vitesse :

```
double maxSpeed = 4.0;
if (input.getKeyboardButton(KeyEvent.VK_RIGHT).isDown()) {
  if (velocity.getX() < maxSpeed) {
    double increase = 60.0 * input.getDeltaTime();
    double speed = velocity.getX() + increase;
    if (speed > maxSpeed)
      speed = maxSpeed;
    velocity = new Vector(speed, velocity.getY());
  }
}
```

Si la flèche droite est enfoncée et que le joueur n'a pas atteint la vitesse maximale, alors on augmente la vitesse horizontale, sans dépasser la limite. **Faites** de même contre la gauche.

Notez que l'importation `java.awt.event.KeyEvent` devient nécessaire.

Être cloué au sol est quelque peu dérangeant, surtout au vu du titre de ce projet. **Ajoutez** le code suivant à la méthode `update`, permettant de propulser notre personnage dans les airs, à volonté.

```
if (input.getKeyboardButton(KeyEvent.VK_UP).isPressed())
  velocity = new Vector(velocity.getX(), 7.0);
```

Il y a du mieux, mais notre objectif n'est pas de recoder *Flappy Bird*. Pour éviter d'avoir un oiseau voletant nerveusement, nous allons interdire le saut si le joueur n'est pas au sol ou contre un mur (pas en état de collision).

Nous allons pour cela faire usage de la possibilité de mémoriser des informations sur l'état du joueur avant la simulation de son évolution (physique), via la méthode `preUpdate`.

On peut en effet ici définir un attribut `colliding` de type `boolean` qui indiquera si le joueur a touché un objet solide.

Cet attribut sera réinitialisé à `false` lorsque `preUpdate` est appelé, et mis à `true` lors d'une collision avec un acteur solide, dans `interact`.

Ajoutez ces éléments et **modifiez** la condition de saut dans `update` pour tenir compte de `colliding`. L'absence de frottement, combiné à l'image de ce personnage, donne vraiment l'impression de contrôler un cube de savon. La physique derrière les frottements statiques et dynamiques sont bien trop compliquées pour ce projet, nous allons donc comme toujours approximer cela[15]. Dans `update`, juste avant de vérifier les flèches directionnelles, **ajoutez** le code suivant, réduisant la vitesse lors d'un contact avec un objet solide :

```
if (colliding) {
  double scale = Math.pow(0.001, input.getDeltaTime());
  velocity = velocity.mul(scale);
}
```

[15] Bienvenue en informatique !

On s'approche d'un résultat intéressant, mais il serait mieux que la vue suive notre personnage.

Supprimez le code qui permet de déplacer la vue à la souris, dans `Simulator.update` et il est temps de surcharger l'événement `postUpdate` du joueur.

Cette méthode doit invoquer la méthode `postUpdate` de la super-classe et faire en sorte que la vue sur le monde soit un cercle centré sur la position du joueur et de rayon 8.

Ajoutez maintenant ce qu'il faut à la méthode `update` pour lancer une boule de feu depuis la position du joueur en pressant la barre d'espace. On teste la barre d'espace est pressée avec la tournure suivante :

```
  if (input.getKeyboardButton(KeyEvent.VK_SPACE).isPressed())
     {..}
}
```

La vitesse de la boule de feu peut être calculée à partir de celle du joueur par une tournure comme ceci :

```
    Vector v = velocity.add(velocity.resized(2.0));
```

Nous y voilà, un acteur (le joueur) en crée un autre (la boule de feu), le nouvel acteur doit évidemment être enregistré dans le monde pour devenir visible.

4.2 Instanciation d'un jeu

Jusqu'à présent, les acteurs ont été créés dans le constructeur de `Simulator`. Ceci n'est pas une approche viable, d'autant que l'on souhaite pouvoir créer divers instances de jeux à partie de la boite à outils que constituera l'ensemble des acteurs codés.

A cette fin, le code fourni prévoit un acteur particulier codé au moyen de la classe `Level`. Décommentez le corps de cette classe dont vous aurez désormais besoin.

Les différentes instances de jeux que vous créerez seront en fait des sous-classes de `Level` dans le package `game.level`. Un exemple d'une telle sous-classe, utilisant les composants codés jusqu'ici est fournie dans le fichier `BasicLevel` du package `game.level`.

Vous vous en inspirerez pour coder vos propres instances de jeu.

L'idée est de faire en sorte qu'un objet de type `Level` soit le seul acteur à s'enregistrer dans le monde. Lors de son enregistrement, son rôle sera de construire le monde en enregistrant à son tour les acteurs qui lui sont spécifiques : examinez le code de `BasicLevel` pour voir concrètement à quoi cela correspond.

Le simulateur doit en outre pouvoir transiter d'un `Level` à une autre en cours de jeu.

Il est donc naturel d'envisager dans `Simulator` un attribut `next` de type `Level` indiquant quel est le prochain niveau vers lequel il faut transiter et de doter l'interface `World` des nouvelles méthodes suivantes :

```
// permet d'indiquer que la transition à un autre niveau
// doit se faire :
public void nextLevel();

// permet de passer au niveau level  :
public void setNextLevel(Level level);
```

La méthode `Simulator.setNextLevel(Level level)` se contentera de modifier l'attribut `next`.

La logique permettant de transiter d'un niveau à un autre va dépendre des acteurs enregistrés dans le monde. Les acteurs doivent pouvoir, le moment venu, aviser `Simulator` qu'un passage au niveau `next` doit se faire : par exemple, le joueur passant une porte de sortie doit pouvoir demander une transition vers le prochain niveau.

Une façon simple de mettre cela en place est de doter `Simulator` d'un attribut `transition` de type booléen mis à `true` par la méthode `nextLevel()`. Un acteur demandant la transition vers le le niveau suivant pourra simplement alors invoquer `nextLevel()`.

Pour que cela fonctionne, la méthode `Simulator.update` devra à la suite de tous les traitements qu'elle met déjà en oeuvre appliquer le traitement suivant pour permettre la transition à un autre état demandée par un acteur :

```
// si un acteur a mis transition à true pour demander le
   passage
// à un autre niveau :
if (transition) {
  if (next == null) {
        next = Level.createDefaultLevel();
  }
  // si un acteur a appelé setNextLevel, next ne sera pas
     null :
  Level level = next;
  transition = false;
  next = null;
  actors.clear();
  registered.clear();
  // tous les anciens acteurs sont désenregistrés,
  // y compris le Level précédent :
  unregistered.clear();
  register(level);
}
```

`Level.createDefaultLevel` définit quel niveau charger au départ. Il est configuré de sorte à prendre `BasicLevel` comme niveau de départ et vous devrez le changer à chaque fois que vous voudrez partir d'un autre niveau[16].

Dans le constructeur de `Simulator`, vous prendrez soin de supprimer toutes les créations d'objets "codées en dur" jusqu'ici et d'initialiser `next` ainsi que `transition` de sorte à ce

[16]Des mécanismes d'initialisation plus évolués vous seront suggérés en fin de projet si vous souhaitez parfaire ce point.

que le premier `Level` enregistré soit celui par défaut.

4.3 Le système d'interaction

Pour l'instant, nos acteurs peuvent se déplacer, mais leurs seules interactions sont les collisions. Qu'il s'agisse d'explosions détruisant des caisses ou du joueur appuyant sur un interrupteur, un vrai jeu requiert des modalités de collaboration plus étendues entre les acteurs.

Le plus grand défi est alors de définir un système général, fonctionnant pour tout type d'acteur.

Il serait peu conforme aux principe de l'orienté-objet que de devoir considérer de façon ad hoc toutes les combinaisons possibles. Et encore moins de devoir tester le type de l'objet rencontré !

4.3.1 Dégâts/effets

La solution proposée consiste à l'échange de « messages » entre acteurs, sous la forme de « dégâts ». Illustrée sur un cas simple, l'idée que nous utiliserons est qu'une boule de feu n'a pas besoin de savoir qu'elle brûle ce qu'elle rencontre.

Elle n'a qu'à indiquer qu'elle inflige des dégâts de feu. Libre alors aux acteurs qu'elle rencontre d'y réagir ou non.

Commencez par définir une énumération `Damage`, définissant les types de dégâts anticipés. Par exemple ici la valeur `FIRE`. Ajoutez-y d'autres types naturels à envisager comme `PHYSICAL`, qui caractérise un contact direct, `AIR`, qui représente une propulsion sans blessure ou encore `VOID` pour les dommages majeurs entraînant la fin d'un niveau de jeu par exemple.

Nous parlons ici de « dégat » car cela est parlant dans ce contexte, mais il s'agit plus généralement d'effets. Le modèle suggéré ne se restreint pas aux seules agressions.

En guise d'exemple de dégâts non dommageables, introduisez par exemple `ACTIVATION`, qui caractérisera l'interaction du joueur avec un interrupteur et `HEAL` qui sera l'opposé de la destruction, rendant de la vie au joueur.

L'intérêt de l'abstraction fournie est que tout acteur est libre de réagir aux dégâts à sa manière. Par exemple, un zombie pourrait se voir revigoré par `FIRE` et ravagé par `HEAL`!. De plus, dans le cas général, il ne sera pas forcément nécessaire de se soucier de la réelle source du dégât. Libre à vous d'ajouter à l'énumération de nouveau type de dégâts, mais veillez à ne pas dupliquer de concepts.

4.3.2 Application des dégâts

Il faut maintenant que nos acteurs soient capables de réagir aux dégâts émis par d'autres acteurs. Ils seront pour cela dotés d'une méthode `hurt` indiquant comment il réagit à un dégât d'un type donné :

```
public boolean hurt(Actor instigator, Damage type, double
    amount, Vector location) {
     return false;
  }
```

La méthode retourne `true` si le dégât a eu un effet et `false` sinon. Dans le cas général, on indique donc ici que quelque soit le dommage il n'y a pas d'impact. L'instigateur du dégât est quand même mis dans les paramètres car dans certains cas, il peut être utile d'y avoir accès ... par exemple pour lui rendre des coups :-)

L'instigateur doit de son côté être capable d'infliger les dégâts. Mettons le en oeuvre dans le cas concret de la boule de feu.

Dans `Fireball.interact`, nous avons traité le cas où le second acteur est solide. **Ajoutez** le code suivant, pour que la boule de feu, avant de rebondir éventuellement, tente de brûler sa cible :

```
  if (other.getBox().isColliding(getBox())) {
    if (other.hurt(this, Damage.FIRE, 1.0, getPosition()))
      // faire en sorte ici que la boule feu disparaisse
      // une fois qu'elle a infligé un dommage.
  }
```

`Box.isColliding` est équivalent à appliquer `Box.getCollision` et vérifier que le résultat est non nul. Ce code prévoit donc que la boule de feu ne disparaît que si la cible a bel et bien été brûlée.

Pour l'instant, le joueur ne réagit pas au feu et ne sera donc pas victime de ses propres projectiles. Toutefois, prévoyons le cas particulier où la boule touche son lanceur. **Ajoutez** un attribut `owner` de type `Actor`, passé en argument dans le constructeur. **Modifiez** `interact` pour ignorer le cas où `other` est `owner`.

Premier exemple complet d'interaction Nous n'avons vu jusqu'ici qu'une facette de l'interaction (comment un acteur inflige des dégâts). Examinons sur un autre exemple, comment un acteur peut y réagir concrètement.

Nous allons pour cela introduire comme nouvel acteur les plates-formes de sauts (`Jumper`) dont le rôle est de propulser dans les airs tout acteur entrant en contact avec. En clair, le `Jumper` inflige un dégât de type `AIR`.

Pour éviter de propulser à tout-va, un mécanisme doit être introduit permettant à la plate-forme de ne propulser que quand elle est « à froid » (c'est à dire, pas en train de propulser). Introduisez pour cela un attribut `cooldown` de type `double`, permettant de faire des comptes à rebours.

Surchargez ensuite les méthodes `update` et `interact` de `Jumper` comme suit :

```
@Override
public void update(Input input) {
  super.update(input);
  cooldown -= input.getDeltaTime();
}

@Override
public void interact(Actor other) {
  super.interact(other);
  if (cooldown <= 0 && getBox().isColliding(other.getBox())) {
    Vector below = new Vector(position.getX(),
       position.getY() - 1.0);
    if (other.hurt(this, Damage.AIR, 10.0, below))
      cooldown = 0.5;
  }
}
```

Les dégâts de type `AIR` sont localisés autour d'un point, définissant le centre d'action de cette sorte d'effets.

Surchargez ensuite la méthode `hurt` de `Player` comme suit :

```
@Override
public boolean hurt(Actor instigator, Damage type, double
   amount, Vector location) {
  switch (type) {
    case AIR:
      velocity = getPosition().sub(location).resized(amount);
      return true;
    default:
      return super.hurt(instigator, type, amount, location);
  }
}
```

Remarquez que le centre `below` du `Jumper` a été placé en dessous de ce dernier, afin que les objets qui le touchent soient propulsés vers le haut[17].

Lorsque le joueur se fait propulser, son vecteur vitesse est en effet modifié, en fonction de l'épicentre de l'attaque.

Finalement, pour améliorer l'aspect visuel, faites en sorte que si `cooldown` est supérieur à zéro, l'image associée au `Jumper` soit *`"jumper.extended"`*, et sinon que ce soit *`"jumper.normal"`*.

Notez aussi qu'il ne faut pas oublier d'appeler `super.hurt`, pour préserver les aspect du comportement hérités de la super-classe.

[17] Ceci est fait ici de façon assez ad hoc et pourrait donner lieu à des améliorations si nous avions plus de temps.

Ajoutez enfin un `Jumper` à `BasicLevel` et testez en le comportement : le joueur lorsqu'il arrive dessus devrait être propulsés dans les airs.

4.3.3 Les zones d'activation

Il est parfois souhaitable d'infliger des dégâts à tous les acteurs situés dans une zone, plutôt qu'à un acteur en particulier. Imaginez par exemple un acteur de type « bombe » qui explose au bout d'un temps donné et dont les effets doivent se faire sentir sur tous les objets du voisinage. Un acteur n'a en effet aucun moyen de connaître les autres objets qui l'entourent. La mécanique de gestion des collisions dont nous disposons à ce stade est trop limitée pour répondre à cette problématique. Il nous faut donc un nouvel outil, à l'échelle du monde, pour endommager tout les acteurs d'une zone. **Ajoutez** la méthode suivante à `World` et `Simulator` :

```
@Override
public int hurt(Box area, Actor instigator, Damage type,
    double amount, Vector location) {
  int victims = 0;
  for (Actor actor : actors)
    if (area.isColliding(actor.getBox()))
      if (actor.hurt(instigator, type, amount, location))
        ++victims;
  return victims;
}
```

Les arguments sont similaires à l'événement `hurt` défini précédemment, avec la région d'intérêt en plus. Tout acteur touchant cette région rectangulaire reçoit les dégâts. Notez que la valeur de retour indique le nombre d'acteurs effectivement blessés.

Vous pourrez tester ce nouvel outillage, lorsque vous aurez introduits les acteurs décrits dans les deux paragraphes suivants.

4.3.4 Relancement

Si vous lancez le programme et que vous jouez aussi mal que la préposée principale à ce cours, vous tomberez rapidement dans un vide sans fond. Il faut maintenant permettre au joueur de disparaître et au jeu de se relancer lorsque le joueur atteint des profondeurs abyssales.

Commencez par modéliser le fait que le joueur n'est pas éternel. Dotez le d'un attribut `health` de type `double` permettant de modéliser son état de santé et initialisé à la construction. Il faut aussi modéliser le fait qu'il y a un plafond à son espérance de vie afin qu'il ne puisse pas devenir éternel par divers artifices, comme collecter des points de vie. Le niveau de santé maximal sera donc aussi un attribut qu'il faut initialiser à la construction. Il faudra après cela adapter un peu le code existant.

Lorsque l'état de santé atteint zéro le joueur doit mourir, ce qui signifie qu'il doit se désenregistrer du monde et indiquer qu'une transition de niveau doit avoir lieu (`nextLevel`). On

se contentera ici de redémarrer le niveau par défaut (celui généré par `Level.createDefaultLevel()`)[18].

Vous prendrez soin de faire cela proprement, en introduisant un méthode dédiée à gérer la fin de vie du joueur.

L'idée est ensuite d'introduire un nouvel acteur, `Limits`, modélisant les limites du monde à ne pas dépasser par le joueur. `Limits` a pour rôle d'infliger des dégât irréparables aux acteurs qu'il rencontre

```
other.hurt(this, Damage.VOID, Double.POSITIVE_INFINITY,
   Vector.ZERO);
```

Le joueur devra y réagir de façon adaptée : recevoir un dommage de type `VOID` doit réduire à néant son niveau de santé.

Vous pourrez enregistrer `Limits` dans votre niveau de jeu selon l'exemple suivant

```
world.register(new Limits(new Box(Vector.ZERO, 40, 30)));
```

Vous pouvez dès maintenant relancer `Program.java` et vérifier si la chute du joueur provoque le relancement de `BasicLevel`.

Le relancement est un peu abrupt mais nous vous donnerons en section 6 des indications sur comment opérer des transitions de niveaux plus douces.

4.3.5 Ajout de nouveaux acteurs

Vous disposez maintenant d'outils suffisants pour créer des acteurs supplémentaires.

Introduisez les acteurs décrits ci-dessous :

L'acteur `Overlay` : Cet acteur permet de nous informer sur le niveau de vie du joueur. Il aura comme attribut un objet de type `Player`. Il dessinera des petits cœurs au dessus du joueur en fonction de son état de santé selon les modalités suggérées ci-dessous :

```
double health = 5.0 * player.getHealth() /
   player.getHealthMax();
       for (int i = 1; i <= 5; ++i) {
           String name;
           if (health >= i)
               name = "heart.full";
           else if (health >= i - 0.5)
               name = "heart.half";
           else
               name = "heart.empty";
           // trouver le Sprite associé à name
           // dessiner ce Sprite en desssus de Player.
         }
```

[18] On pourrait bien sûr imaginer de faire passer le joueur par le purgatoire.. mais laissons cela à plus tard éventuellement.

`player` représente le nom de l'attribut, à vous d'adapter le nom des méthodes suggérées en fonction de votre propre code. Notez que `Overlay` doit disparaître si l'acteur n'existe plus dans le monde (c'est à dire si l'attribut `world` de ce dernier vaut `null`).

L'acteur `Heart` : cet acteur inflige un dégât de type `HEAL` à tout acteur qu'il touche. Le `Heart` doit disparaître une fois touché et réapparaître au bout de 10 secondes. Vous introduirez pour cela une attribut `cooldown` initialisé à la construction et correspondant au temps devant s'écouler avant la réapparition de l'objet. Vous introduirez aussi un compteur décrémenté de `input.getDeltaTime();` à chaque appel à `update`. Le compteur est initialisé à 0.0 et mis à `cooldown` en cas de contact avec le joueur. Le `Heart` ne s'affiche que si le compteur est inférieur ou égal à zéro. Lorsque le joueur reçoit des dégât de type `HEAL`, son état de santé est augmenté du montant de l'attaque (plafonné au niveau maximal de santé) Le `Sprite` associé à *`"heart.full"`* est à disposition.

L'acteur `Spike` : il s'agit de pics qui infligent des dégâts de type `PHYSICAL` d'une grande ampleur à tout objet tombant dessus (composante `y` de la vitesse négative, par exemple $<$ -1). Lorsque le joueur reçoit des dégât de type `PHYSICAL`, son état de santé est diminué du montant de l'attaque. Les boules de feu n'y réagissent pas. Le `Sprite` associé à *`"spikes"`* est à disposition.

L'acteur `Torch` : il s'agit d'une torche qui s'allume sous l'effet des dégâts `FIRE` et qui s'éteint lorsqu'elle reçoit un dégât `AIR`. Une torche est donc capable de réagir à un dégât lancé par une `Fireball`. Pour voir le dégât `AIR` à l'œuvre, vous permettrez au joueur de "souffler" (notamment dans une région où la torche se trouve). Pour cela vous programmerez la touche `B` en ajoutant le code suivant à `Player.update` :

```
if (input.getKeyboardButton(KeyEvent.VK_B).isPressed())
   getWorld().hurt(getBox(), this, Damage.AIR, 1.0,
      getPosition());
```

La torche sera de priorité moyenne (par exemple 34) afin qu'elle soit dessinée après les blocks (priorité 0), mais avant les personnages (priorité 42) et projectiles (priorité 666). Elle n'est pas solide et est caractérisée par un `Box`, comme les blocs.

Elle se dessinera soit avec *`"torch.lit.1"`* si elle est allumée, sinon au moyen de *`"torch"`*. Un attribut booléen `lit` permettra d'indiquer si la torche est allumée.

Il est à noter un point important, concernant la facilité d'utilisation de vos classes. Bien que la torche soit caractérisée par un rectangle, il n'est ni logique, ni pratique, de demander un `Box` dans le constructeur. En effet, si les proportions ne sont pas respectées, la torche sera déformée. Nous vous conseillons donc de ne demander que le centre de la torche comme argument du constructeur. Le constructeur prendra aussi un booléen indiquant si la torche doit être allumée au départ ou non.

la méthode `getBox` retournera une `Box` centrée sur la position de la torche et de hauteur et largeur 0.8 (notez que le constructeur de `Box` est surchargé).

Une petite note maintenant sur l'amélioration du visuel. Jusqu'à présent, nous avons généralement dessiné une seule image par acteur, avec pour seule subtilité la rotation de

la boule de feu.

Une solution pour ajouter un peu de vie à nos dessin est de faire varier l'image au cours du temps. Pour commencer simple, nous allons utiliser une séquence d'image, chacune durant quelques dixièmes de seconde.

Ajoutez à la torche un attribut `variation` de type `double`, initialisé à zéro qui sera utilisé pour choisir l'image de la flamme dans la méthode `draw` de la torche

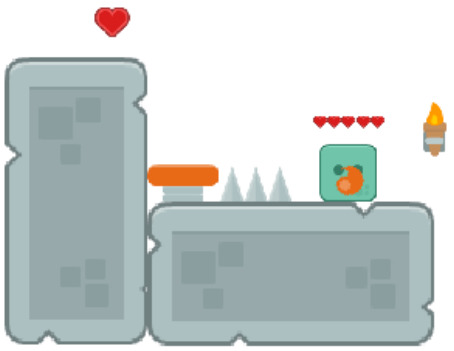
```
    // dans la partie où l'on dessine une torche allumée
    String name = "torch.lit.1";
    if (variation < 0.3)
      name = "torch.lit.2";
  }
```

Pour que l'animation totale dure 0.6 secondes, équitablement répartie entre les deux images. variation doit être mis à jour en fonction du temps, se répétant à l'infini dans la méthode `update` de la torche :

```
    variation -= input.getDeltaTime();
    if (variation < 0.0)
      variation = 0.6;
```

4.4 Validation de l'étape 2

Créez un objet `BasicInteract` de type `Level` ressemblant à ceci :



et permettant de tester que l'ensemble des acteurs créés jusqu'ici ont le comportement voulu :

- le ressort doit projeter le joueur dans les airs ;
- le joueur peut allumer les torches en lançant des boules de feu et les éteindre en leur envoyant un dégât de type air ;
- les pics diminuent le niveau de santé du joueur et le cœur les augmente
- le joueur qui tombe dans le vide perd la vie et le niveau `BasicInteract` est rechargé dans ce cas (le niveau `BasicInteract` devient le nouveau niveau par défaut de `Level`). .

5 « Puzzles et énigmes » (étape 3)

Cette partie du projet a pour but d'enrichir notre petit moteur de jeux de sorte à pouvoir créer des puzzles ou jeux d'énigmes. Ce type de jeux nécessite la mise en place de mécanismes de connexions entre les acteurs, rappelant les systèmes logiques que vous aborderez lors du cours *conception de systèmes numériques*.

Par exemple, le joueur doit pouvoir allumer des torches et activer des leviers dans un certain ordre pour pouvoir prendre un ascenseur ; lequel lui permettra d'atteindre une clé ouvrant une porte de sortie vers un autre niveau de jeu.

5.1 Les signaux

Imaginons qu'une porte dépende de l'obtention d'une clé. Il est envisageable d'indiquer à chaque clé la porte qui lui est associée. L'inverse est aussi possible, demander à la porte de vérifier si sa clé a été collectée. Quelle que soit la solution choisie, clé et porte sont liées. Requérir de multiples clés, ou carrément les remplacer par un levier ou un ennemi à tuer, demande de modifier le code de la porte, ce qui n'est pas souhaitable.

Nous allons introduire un nouveau concept, les signaux. Un signal est une valeur logique, c'est-à-dire un booléen indiquant si la condition est remplie ou non. Tout objet, acteur ou non, implémentant l'interface `Signal` représente un signal qui pourra être utilisé pour prendre des décisions.

```
public interface Signal {
  public boolean isActive();
}
```

Commençons par un exemple simple, en reprenant notre torche.

Supposons que nous souhaitions utiliser le fait qu'une torche soit allumée comme un signal qui conditionne le comportement d'un autre objet. Il suffit alors de faire en sorte que `Torch` puisse être perçue comme un `Signal` (en implémentant l'interface `Signal` pour signifier une torche se comporte comme un signal). Il suffit alors de redéfinir proprement `isActive` dans `Torch` comme ceci par exemple.

```
@Override
public boolean isActive() {
  return lit;
}
```

Maintenant `Torch` peut non seulement être un composant du jeu mais aussi servir de signal utile à d'autres composants.

Pour illustrer cela concrètement, nous allons en effet créer plus tard un acteur `Mover` (ascenseur), dont l'activation, c'est à dire le fait de commencer à monter, dépend d'un signal. Si l'on veut conditionner l'activation de l'ascenseur au fait qu'une torche donnée soit allumée, on créera les objets comme suit :

```
Torch torch = new Torch(new Vector(3, 4), true);
Mover mover = new Mover(..., torch);
```

Le `update` de `Mover` devra bien évidemment prendre en compte l'état du signal de la torche (actif ou pas) pour réagir de façon appropriée.

5.2 Combinaison de signaux

Un signal simple représente une valeur logique (vrai ou faux, actif ou pas).

Produire des signaux n'est pas suffisant pour définir des logiques intéressantes. Il est aussi nécessaire de les manipuler et les combiner. Pour cela, nous allons créer plusieurs classes représentant diverses opérations de base. Comme vous le verrez au prochain semestre, il s'agit de portes logiques.

5.2.1 Négation

L'opération la plus simple est probablement la négation, inversant le signal reçu. Pour cela, nous allons créer la classe `Not`.

```
public class Not implements Signal {

  private final Signal signal;

  public Not(Signal signal) {
    if (signal == null)
      throw new NullPointerException();
    this.signal = signal;
  }

  @Override
  public boolean isActive() {
    return !signal.isActive();
  }

}
```

Ainsi, si une porte requiert que la torche soit éteinte, et non allumée, il suffira d'inverser le signal.

Vous noterez que `Not` n'est pas un acteur. Il s'agit en effet d'un signal pur, qui ne peut être perçu que comme un `Signal` (contrairement à la torche de tout à l'heure, qui peut être perçu comme un `Signal` mais peut aussi exister en tant que telle).

Allons un peu plus loin.

5.2.2 Conjonction/disjonction

Supposons que l'ouverture d'une porte nécessitele fait que plusieurs torches soit allumées pour s'ouvrir.

Nous allons créer un signal dépendant de deux autres signaux (les entrées) et qui est actif uniquement si toutes les entrées sont actives.

Nous allons définir la classe `And`, avec deux signaux en attributs.

```
@Override
public boolean isActive() {
  return left.isActive() && right.isActive();
}
```

Ainsi, si trois torches sont requises, il suffira de les combiner avec deux portes.

```
// signal = t1 && t2 && t3
Signal signal = new And(new And(t1, t2), t3);
```

Créez le signal `Or` sur le même modèle. Ce dernier doit s'activer lorsqu'au moins un des signaux est actif.

À noter qu'il peut être intéressant de définir une classe `Constant`, permettant de définir manuellement un signal. Par exemple on peut vouloir faire en sorte qu'une porte de sortie soit toujours ouverte et donc associée à un signal constant toujours activé.

Pour mettre en pratique tout cela concrètement et diversifier un peu notre univers il est temps de définir des acteurs concrets dépendant de signaux.

5.3 Composants dépendant de signaux

5.3.1 Portes et clés

Définissez une classe `Key`, représentant une clé qu'il faut ramasser. Outre sa position, elle est caractérisée par un attribut `taken` de type `boolean`. Tant que la clé n'a pas été collectée, elle est visible et réagit au monde. Et à l'instant où elle interagit avec le joueur, elle disparaît et se dés-enregistre du monde.

La clé jouera le rôle d'un signal auquel peuvent réagir d'autre composant (le signal est actif lorsque la clé est collectée et inactif sinon).

Combinée à cela, nous allons définir une porte simpliste. L'idée est de sous-classer `Block` afin de lui changer son comportement en fonction d'un signal, en attribut.

Cette classe `Door` sera tel un bloc, solide, lorsque le signal est faux.

Dans le cas contraire, il ne sera ni dessiné, ni solide, et `getBox` retournera `null`. Vous remarquerez que `Door` a-un `Signal` et est-un `Signal`!

Vous disposez avec cela du moyen de créer des portes dont l'ouverture est conditionnée par des clés :

```
Key blue = new Key(...);
...
Key red = new Key(...);
..
```

```
world.register(new Door(..., new And(blue,red),...));
```

Les `Sprite` associés aux chaînes commençant par *`"lock"`* ou *`"key"`* peuvent être utilisées (faites votre "marché" dans le répertoire `/res`!).

Notez que même si une clé n'est plus gérée par le simulateur, elle continue de renvoyer `true` via `isActive`.

5.3.2 Leviers

Définissez une classe `Lever` représentant un levier. Un levier sera considéré comme non-solide (on ne se pose pas dessus) et peut agir comme un signal.

Pour seul attribut, `value` de type `boolean`, indiquant si le levier est activé.

Les images *`"lever.left"`* et *`"lever.right"`* doivent être dessinées en fonction de cette valeur.

Un levier doit être sensible au dégât `ACTIVATION` : s'il reçoit un dégât `ACTIVATION` d'une ampleur positive, son attribut `value` sera mise à son opposé.

Il faut aussi que notre joueur soit capable de lui envoyer des dégâts `ACTIVATION`. Programmez la touche `E` est à cet effet.

Notez qu'il est possible d'ajouter un chronomètre pour réinitialiser le levier après une certaine durée. Pour cela, **ajoutez** les attributs `duration` et `time`. Lorsque `value` est mise à `true`, `time` prend la valeur `duration` et sera décrémentée à chaque mise à jour. Une fois redescendue à zéro, le levier se remet à `false`. Remarquez que le comportement de base, sans chronomètre, peut être conservé en utilisant `Double.POSITIVE_INFINITY` comme durée.

5.3.3 Ascenseurs

En plus de la porte, d'autres éléments peuvent utiliser un signal pour guider leur comportement. Ainsi, des plate-forme animées peuvent être déplacées en fonction d'une valeur logique.

Pour cela, **définissez** la classe `Mover`, une sous-classe de `Block` (il s'agit donc d'un objet solide sur lequel on peut se poser) . La nouveauté sera deux attributs de type `Vector`, représentant les deux positions au repos, que nous nommerons `on` et `off`. Un attribut `signal` sera utilisé pour choisir entre les deux.

Suivant l'application désirée, la transition n'est pas forcément la même. Le plus simple étant que la plate-forme se téléporte sur l'une des deux options, dès que le signal change.

Proposer une transition souple et continue est toutefois plus agréable. Ainsi, nous allons ajouter un attribut `current` de type `double`. Sa valeur sera modifiée en fonction du signal, fluctuant à vitesse constante entre 0 et 1.

```
@Override
public void update(Input input) {
  super.update(input);
  if (signal.isActive()) {
    current += input.getDeltaTime();
    if (current > 1.0)
      current = 1.0;
  } else {
    current -= input.getDeltaTime();
    if (current < 0.0)
      current = 0.0;
  }
}
```

Surchargez la méthode `getBox` pour interpoler entre `off` et `on` en fonction de `current`. Évidemment, il serait judicieux de fournir des paramètres pour configurer la vitesse de déplacement.

Notez aussi que la plate-forme se déplace à vitesse constante. Pour donner un sentiment d'accélération, il serait intéressant d'utiliser d'autres types d'interpolation. Par exemple $f(x) = -2x^3 + 3x^2$, dont l'origine est laissée en exercice[19].

5.3.4 Porte de sortie

Une porte non solide, `Exit`, permettant au joueur de transiter vers un autre niveau. Le constructeur de `Exit` pourra prendre en paramètre une position, le niveau vers lequel transiter et le signal conditionnant son ouverture. Par défaut le signal sera constant (et toujours actif).

Supposons qu'il existe un `Level` nommé `Arena`. Pour enregistrer dans le monde une porte de sortie dépendante d'un signal `s` et permettant d'accéder à `Arena`, on pourrait écrire quelque chose comme :

```
world.register(new Exit(new Vector(0.0, 3.0), new Arena(), s));
```

(l'ordre des paramètres n'est pas imposé).

Les `Sprite` suivants sont à disposition : *`"door.open"`*, *`"door.closed"`*.

[19] Indice : interpolation cubique, dont les dérivées valent zéro en $x = 0$ et $x = 1$, pour simuler l'arrêt.

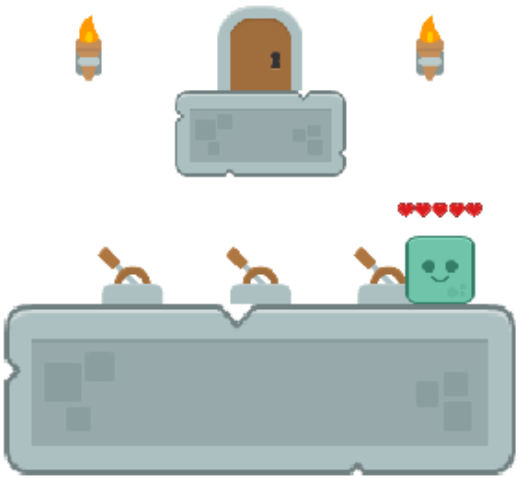
5.4 Validation de l'étape 3 et résultat du projet

Comme résultat final du projet, créez deux niveaux de jeu impliquant l'ensemble des composants codés jusqu'ici (joueur avec points de vie, coeur, torches, jumper, portes et clés, ascenceurs, leviers, pics, boules de feu et porte de sortie) et permettant de les tester. Le second niveau de jeu sera accessible via une porte de sortie depuis le premier. Une (petite) partie de la note sera liée à l'inventivité dont vous ferez preuve dans la conception des niveaux.

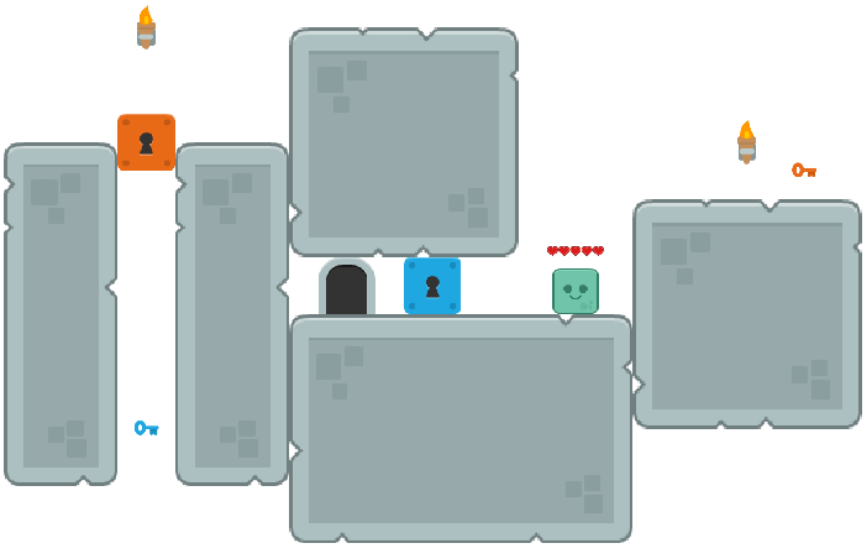
Vous prendrez soin de commenter soigneusement dans votre README, le nom de votre niveau et les modalités de jeu qu'il implique. Nous devons notamment savoir quels signaux activer et pourquoi sans avoir à lire votre code.

Voici des exemples dont vous pourrez vous inspirer.

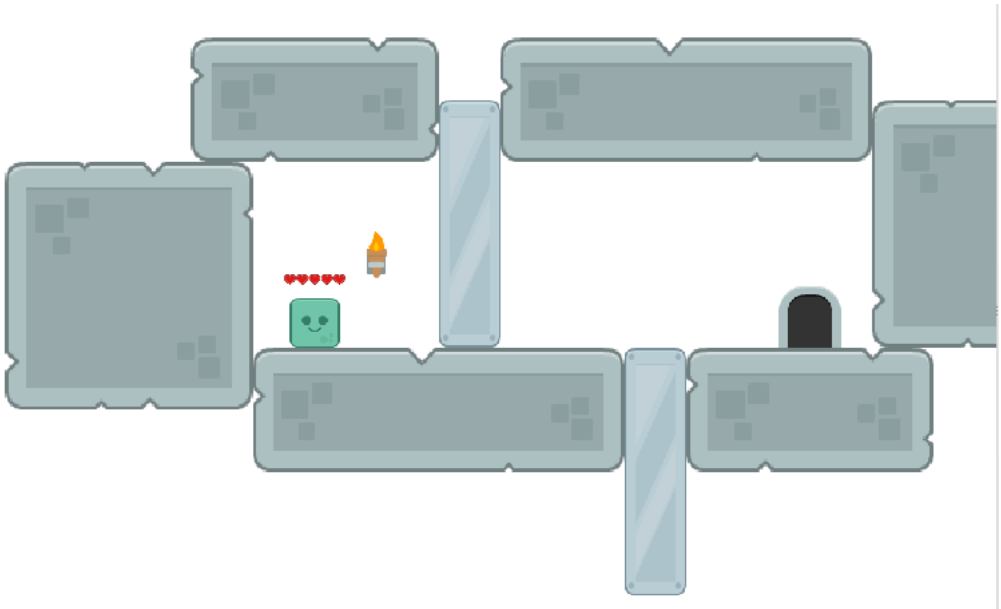
- Un niveau `Levers` comportant trois leviers, deux torches et une porte de sortie. L'ouverture de la porte de sortie permet de transiter vers le niveau suivant `Doors` décrit ci-dessous. Cette ouverture est conditionnée par un signal complexe combinant les signaux des torches et des leviers au moyen de `Or`, `And` et `Not`.



- Un niveau `Doors` comportant deux portes dont chacune s'ouvre avec sa clé respective (la clé rouge ouvre la porte rouge et la clé bleue ouvre la porte bleue). Le but est d'ouvrir les portes pour atteindre une porte de sortie qui permet de transiter au niveau `Movers` décrit ci-dessous.

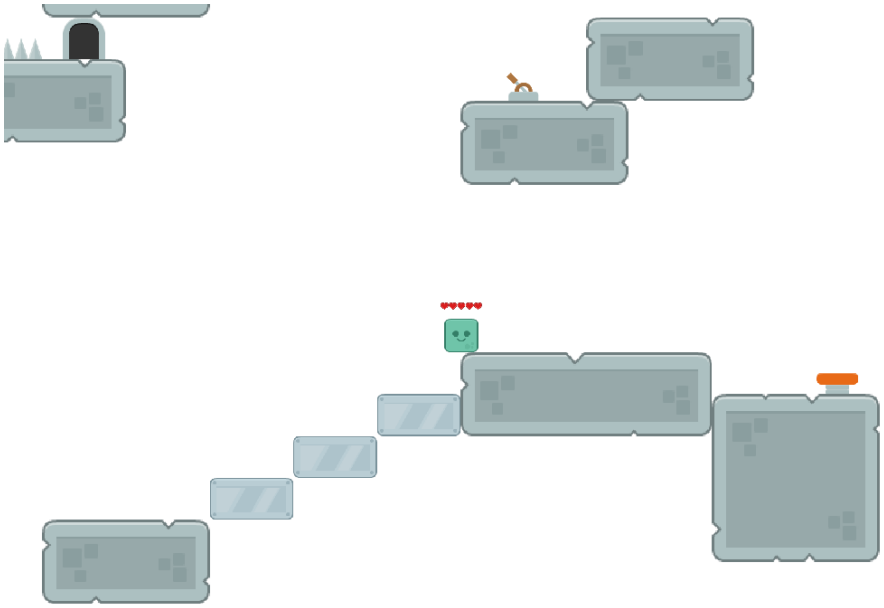


- Un niveau `Movers` comportant deux ascenceurs dont l'activation dépend de l'extinction d'une torche. Le joueur doit activer les ascenceurs pour atteindre une porte de sortie lui permettant d'atteindre le niveau `Jumps` décrit ci-dessous.



(l'extinction de la flamme va faire monter les deux ascenseurs. Si le joueur n'est pas placé au bon endroit au bon moment le deuxième ascenseur lui bloquera l'accès à la sortie)

- Un niveau `Jumps` avec un ressort, un levier, trois ascenseurs, des pics et une porte de sortie ressemblant à ceci : Le joueur doit atteindre la porte de sortie sans se faire toucher par les pics. Les ascenseurs sont activables grâce au levier.



- etc.

Plus d'images Nous vous avons fournis un ensemble d'images, conçues et aimablement mises à disposition par le studio Kenney. Leur site propose de nombreuses autres images dans le même style, garantissant une certaine unité pour le jeu. Toutefois, libre à vous d'utiliser d'autres images, qu'elles soient de votre création ou collectées sur la toile. Pensez cependant à indiquer leur origine.

6 Aller plus loin

La base que vous avez codée jusqu'ici peut être enrichie à l'envi. Si vous êtes motivés, laisser maintenant parler votre imagination, et essayez vos propres idées. Des suggestions vous sont données plus bas.

Les personnes qui atteignent cette partie facultative peuvent concourir au prix du « meilleur jeu d'énigme du CS107 ».[20]

Si vous souhaitez concourir, vous devrez nous envoyer d'ici au **12.12 à midi** un petit "dossier de candidature" par mail à l'adresse **cs107@epfl.ch**. Il s'agira d'une description de votre jeu et des extensions que vous y avez incorporées (sur 2 à 3 pages en format .pdf avec quelques copies d'écran mettant en valeur vos ajouts).

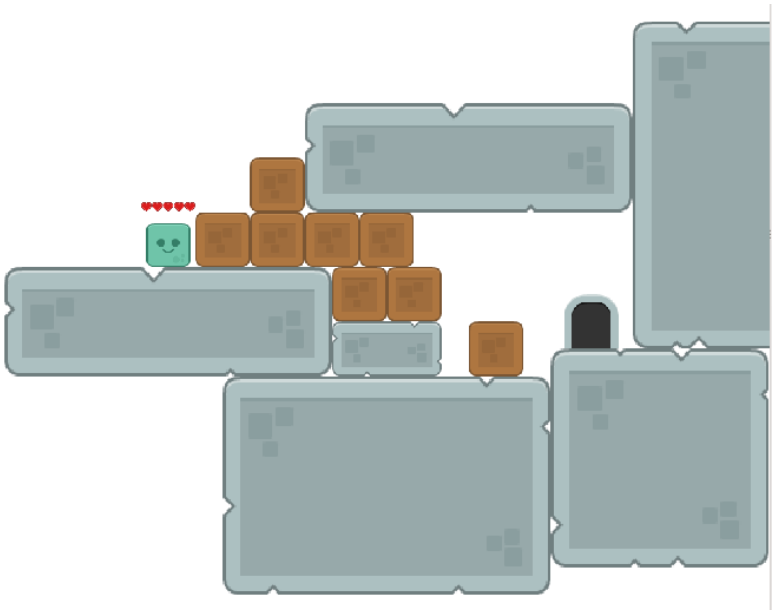
Vous trouverez ci-dessous quelques idées en vrac pour pour aller plus loin.

Une description plus détaillée de comment mettre en oeuvre ces idées est donnée pour l'extension des jeux de puzzle et des projectiles. L'idée est que nous restons ici dans l'esprit du jeu de puzzle qui était notre objectif principal.

6.1 Variation autour des composants existants

Au gré de votre imagination, toutes sortes de variations peuvent être envisagées :

- Des blocs destructibles, détruits sous le coup des boules de feu (ou autres projectiles). La destruction des blocs pourra conditionner l'ouverture de passages comme dans cet exemple :



- Des blocs qui crachent une flèche quand on passe devant. La flèche, pour le coup, peut se planter dans le premier objet qu'elle blesse, et s'y attacher. Le joueur peut se balader avec une flèche dans la tête ;-).

[20] il faudra cependant qu'on trouve autre chose que du chocolat à vous donner !

- Des plaques de pressions. Similaires aux leviers, ils s'agira d'un autre type d'interrupteur qui peut réagir à la présence du joueur, ou de tout autre acteur (donc une classe `Plate` qui s'active lorsqu'elle entre en collision avec un autre acteur).
- etc.

6.2 Enrichissement du visuel

Vous avez un peu enrichi le visuel en introduisant l'animation de la flamme de la torche.

On peut évidemment aller beaucoup plus loin.

Jusqu'à présent, chaque acteur a été responsable de son propre dessin, ce qui est une bonne chose pour son aspect physique.

Néanmoins, ce n'est pas idéal pour les effets associés à l'objet, telles de la fumée ou des étincelles. En effet, à sa disparition, l'acteur cesse instantanément de dessiner le nuage, produisant une transition tout sauf agréable.

Un autre aspect concerne aussi les restes d'un acteur. Qu'il s'agisse de l'os du jambon destiné à restaurer le joueur, ou du corps de ce dernier en cas de malheur, la nature de ces reliques ne devrait pas être associée à l'acteur lui-même. Un joueur mort n'a pas le même comportement que lorsqu'il était vivant. Ainsi, il n'est pas forcément logique de représenter un cadavre avec la même classe.

Une solution consiste à séparer ces effets dans un acteur dédié, ce qui permet aussi une meilleure réutilisation du code. Nous allons donc **créer** la classe `Particle`, se chargeant uniquement de dessiner une image pendant quelques instants. Une particule est représentée par un `Sprite`, une position, une taille, un angle (en radians), une durée de vie et le temps écoulé depuis sa création (en secondes), et sera de priorité haute (par exemple 1337) afin d'être dessinée avant tous les autres `Actor`s. Notez que `getBox` doit être défini en fonction de la position et de la taille, une particule étant carrée. La transparence de l'image se chargera de faire les motifs nécessaires. Une fois la durée de vie atteinte, la particule se retire de ce bas monde :

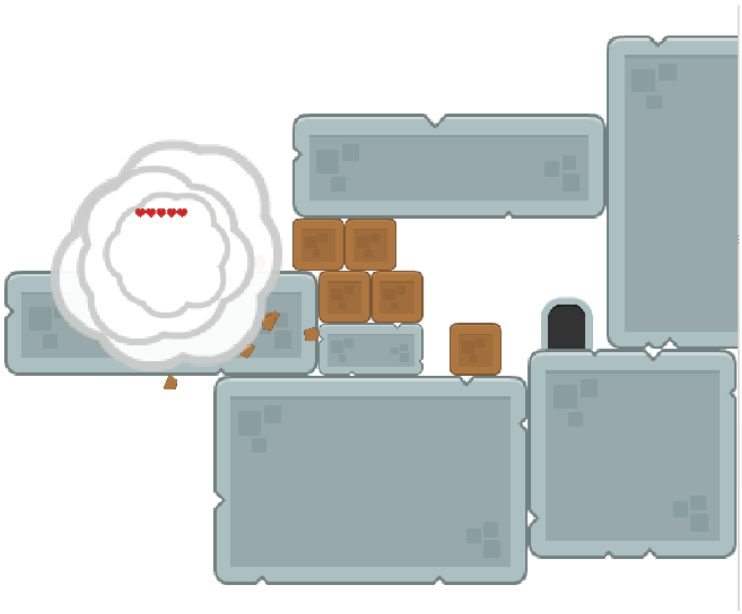
```
  @Override
  public void update(Input input) {
    super.update(input);
    time += input.getDeltaTime();
    if (time >= duration)
      getWorld().unregister(this);
}
```

L'utilisation d'une particule se veut simple et générale. Pensez donc à ajouter des accesseurs pour chacun des attributs, pour permettre une utilisation de ce style :

```
  Particle smoke = new Particle();
  String name = "smoke.white.1";
  // retrouver le Sprite associé à name
  smoke.setPosition(getPosition());
  smoke.setSize(2.0);
```

```
smoke.setDuration(5.0);
// enregistrer smoke dans le monde
```

Voici un exemple d'utilisation de particules pour simuler un dégagement de fumée après l'explosion d'une bombe et le fait que cela fasse voler en éclat des blocs destructibles :



6.3 Composants avancés pour puzzle

Jusqu'à présent, les signaux ont toujours été directement connecté de la source à l'objectif, formant des systèmes simples. Cependant, pour concevoir des énigmes plus complexes, impliquant par exemple une notion de temps, de nouveaux composants sont nécessaires.

Mais avant de se lancer dans des explications avancées, il est conseillé d' ajouter un outil de diagnostique. Vous pouvez ajouter la classe `Led`, sous-classe de `Block` et implémentant `Signal`. Ce nouveau bloc aura comme spécificité d'afficher l'état du signal associé, stocké en attribut, et transmis sans changement par `isActive`. Outre cette dernière, la seule méthode à surcharger concerne le dessin, ignorant l'appel au parent :

```
@Override
public void draw(Input input, Output output) {
  String name = "box.3.disabled";
  if (isActive())
    name = "box.3.enabled";
  Sprite sprite = getWorld().getLoader().getSprite(name);
  output.drawSprite(sprite, box);
}
```

Il devient ainsi possible d'ajouter des informations visuelles utiles pour la suite.

Il est en effet possible d'introduire des `Oscillator`, dont le signal varie au cours du temps, à intervalles réguliers. Pour cela, inspirez vous de ce qui a été fait pour les animations de la torche.

Une remarque importante peut être faite concernant les paramètres de nos composants. On pourrait juger utile de pouvoir désactiver l'oscillateur, par exemple en utilisant un attribut `enabled` de type `boolean`. Une approche plus souple serait d'utiliser le type `Signal`, permettant de lier cet état à n'importe quel autre composant. Toutefois, aucune de ces solutions n'est conseillée, car il est possible d'arriver à un tel résultat sans ajouter de complexité à notre classe `Oscillator`. En effet, il suffit d'ajouter une porte `And` à sa sortie, combinant l'oscillation et le signal d'activation.

```
Oscillator oscillator = new Oscillator(0.5);
Torch torch = new Torch(new Vector(2.0, 3.0));
Signal and = new And(oscillator, torch);
Led led = new Led(and, new Vector(1.0, 1.0));
```

Un autre concept vital pour définir une séquence d'objectifs est la notion de mémoire. Vous verrez ce composant au semestre prochain sous le nom de *flip-flop*, que nous **ajoutons** sous le nom de `Memory`. Il prend en entrée deux signaux, que nous représenterons par deux attributs `input` et `load`. À chaque mise à jour, si `load` est actif, la valeur d'`input` est mémorisée dans un attribut `value`. Dans le cas contraire, la mémoire n'est pas affectée :

```
@Override
public void update(Input input) {
  super.update(input);
  if (load.isActive())
    value = input.isActive();
}
```

Pour mieux comprendre le principe, prenons l'exemple suivant. Une porte doit s'ouvrir une fois qu'une plaque de pression a été activée. Pour éviter qu'elle ne se referme dès que le joueur quitte la plaque, il faut mémoriser cette information. Autrement dit, stocker `true` lorsque le joueur l'active :

```
Signal input = new Constant(true);
Signal load = new Plate(new Vector(5.0, 0.0));
Signal memory = new Memory(input, load);
Door door = new Door(memory, new Box(0.0, 0.5), 1.0);
```

Enfin, nous vous proposons un dernier exemple, le retardateur de signal. Que ce soit pour limiter le temps du joueur ou retarder l'ouverture d'une porte, ajouter un délai offre de nouvelles possibilités. Une remarque fondamentale doit être faite, afin de choisir l'implémentation de cette classe `Delay`. Retarder un signal d'une seconde implique de mémoriser la séquence complète durant une seconde. Comme vous avez pu le constater, cela fait beaucoup d'information, notre boucle de jeu étant suffisamment rapide.

Le première solution serait donc de mémoriser, par exemple, les mille valeurs précédentes et de les restituer en décalé. Autrement dit, à chaque mise à jour, `isActive` prend la valeur la plus ancienne, qui est enlevée du début de la liste. Et à la fin de la liste s'ajoute la valeur actuelle du signal en entrée. Il s'agit du comportement typique d'une file d'attente, le dernier arrivé est le dernier sorti, représenté par l'interface `java.util.Queue`, et notamment implémenté par `java.util.LinkedList`.

Cependant, la durée d'une mise à jour n'est pas constante, cette solution pouvant entraîner quelques incohérences. Si ce phénomène pose des problèmes de précision que vous ne

pouvez pas vous permettre, alors il faudra trouver une implémentation différente. Par exemple, mémoriser pour chaque valeur en entrée l'instant d'arrivée, ce qui pourrait permettre de corriger le problème. Notez que cela commence à gagner en complexité, et qu'il serait judicieux de ne pas mémoriser tous les états, mais seulement les instants où le signal a changé.

6.4 Divers projectiles

Il y a des myriades de projectiles envisageables, allant de la boule de feu à la flèche, en passant par des bombes. Une super-classe `Projectile`, regroupant les fonctionnalités de base s'impose laquelle regroupera des attributs tels que la position, la vitesse, la taille, le lanceur. Vous pouvez utiliser `Fireball` comme modèle, et progressivement définir toutes les propriétés d'un projectile. Le comportement des projectiles doit être modifiable selon la nature de l'objet. Par exemple, le rebond parfait de la boule de feu n'est pas applicable à une boule de bowling. Ainsi, les propriétés suivantes peuvent être à envisager :

- Un attribut `bounciness` de type `double`, dont la valeur est comprise entre 0 et 1, indique l'énergie conservée lors d'un rebond. Ainsi à 1, la vitesse sera calculée comme la boule de feu, avec `Vector.mirror`. Alors que dans l'absence totale de rebond, la vitesse sera mise à zéro dans la direction du mur. Utiliser des valeurs intermédiaires devrait donc amortir le rebond de l'objet.

- Comme nous l'avons fait pour les personnage, appliquer des frottements est une bonne idée. Le coefficient utilisé dans `Player.update` était 0.001. Il s'agit de la vitesse conservée sur une seconde. Autrement dit, un projectile avec 0.5 de frottement divisera par 2 sa vitesse chaque seconde, tandis qu'un frottement de 0 arrêtera instantanément l'objet.

- Un projectile devrait pouvoir s'arrêter net au contact d'un objet, par exemple s'il s'agit d'une grenade adhésive. Toutefois, mettre la vitesse à zéro ne suffit pas à représenter cela. En effet, si l'objet auquel notre grenade est attaché se déplace, il faut que le projectile suive. Une solution est donc de mémoriser l'acteur sur lequel nous sommes fixés, ainsi que le vecteur allant de sa position à notre position. Ainsi, on obtiendrait un code pour `update` du genre suivant :

  ```
  if (support != null)
    position = support.getPosition() + difference;
  else {
    // ...
  }
  ```

- Pour aller plus loin dans cette idée, si l'objet sur lequel nous sommes fixé disparaît, nous devons nous détacher et recommencer à simuler notre propre physique. Cependant l'objet en question ne sait pas que nous sommes attaché à lui, il ne peut donc pas nous avertir de sa mort. Une manière de tester cela est de regarder si `support.getWorld()` est `null`, à chaque mise à jour.

6.4.1 Bombe et missile

Avant d'aller plus loin dans nos armes et objets, prenons deux exemples concrets de projectiles, couvrant la plupart des fonctionnalités possibles. Créez la classe `Bomb`, héritant de `Projectile` et utilisant l'image *`"bomb"`*. Définissez aussi un frottement de 0.001 et un rebond de 0.5, tel que suggéré dans la section précédente. Nous allons construire point par point notre bombe :

- Utilisez un attribut `timer` de type `double` initialisé à 4 secondes, décrémenté à chaque mise à jour, pour déterminer l'instant de l'explosion. L'explosion provoque, dans un rayon de 2, 50 dégâts de feu et 5 dégâts d'air.
- Pour montrer le rayon de l'explosion, la bombe crée 3 particules avant de disparaître. Utilisez par exemple les images *`"smoke.gray.1"`*, *`"smoke.yellow.2"`* et *`"smoke.gray.3"`*. Afin d'avoir un effet de fumée qui se dissipe, si vous l'avez implémenté, commencez avec une taille de 2 et une vitesse d'agrandissement de 0.5. De plus, la particule disparaît au bout de 5 secondes, en devenant de plus en plus transparente.
- Surchargez la méthode `hurt` pour qu'elle explose lorsque la bombe reçoit des dégâts de feu. Cela implique de placer le code qui provoque l'explosion dans une méthode dédiée `explode`. Réagissez aussi aux dégâts d'air pour projeter la bombe, similairement au joueur.
- Un aspect important de ce genre d'objet est de donner un retour visuel au joueur, indiquant le compte à rebours. Vous pouvez par exemple faire clignoter la bombe en alternant avec l'image *`"bomb.white"`*, ou faire varier la taille de l'image dessinée. Augmentez la rapidité en fonction du temps restant, pour que cela s'accélère à mesure que le timer se rapproche de zéro.

Un autre exemple typique peut prendre la forme d'une classe `Missile`, équivalent à une bombe explosant à l'impact, propulsée par un réacteur. Il s'agira donc d'une sous-classe de `Bomb`, utilisant l'image *`"missile"`*.

- Afin d'ajouter une force de poussée de 0.5, à chaque mise à jour, juste avant d'appeler `super.update`, appliquez une accélération dans la direction du projectile :

```
Vector acceleration =
   Vector.X.rotated(getAngle()).resized(0.5);
setVelocity(getVelocity().add(acceleration.mul(input.getDeltaTime())))
super.update(input);
```

- De plus, faites apparaître à intervalles réguliers des particules de fumée.
- Enfin, lorsque le missile entre en contact avec un objet solide, il explose. Une question intéressante se pose pour les objets non solides. Invariablement exploser au contact d'un acteur quelconque n'est pas viable, car la moindre particule de fumée ferait détoner le projectile. Une solution consiste à infliger de faibles dégâts physiques lors d'une collision. Si la cible est effectivement blessée, alors la détonation s'active.

Notez que si vous souhaitez utiliser la souris pour viser et orienter le tir, il suffit d'utiliser `Input.getMouseLocation` et `Input.getMouseButton`, comme nous l'avions fait pour déplacer la vue.

6.5 Gestion de la transition de niveau

Une transition moins brutale d'un niveau à l'autre peut se faire par le biais d'un acteur `End` qui retarde la nécessité de passer au niveau suivant au moyen d'un compteur et qui affiche un écran s'estompant en attendant :

```
public void draw(Input input, Output output) {
        Sprite sprite = getSprite("pixel.black");
        double transparency = Math.max(0.0, time - duration +
            1.0);
        output.drawSprite(sprite, output.getBox(), 0.0,
            transparency);
    }
```

Le joueur en mourant peut enregistrer un tel acteur au lieu de directement invoquer `nextLevel`.

Il est aussi possible de coder un `Level` "sélection", rempli de portes qui mènent à divers mondes (divers `Level`) :

On pourrait sélectionner ces portes en cliquant dessus.

6.6 Bonus et doses de soins

Jusqu'ici, le joueur peut se ressourcer en "collectant" des `Heart`. On peut imaginer toute sorte d'objets avec des effets différents (par exemples des pièces de monnaie qui rendent le joueur plus riche, la richesse conditionnant certains comportement ou des malus tels des poches de poison). Il convient alors de trouver une bonne conception permettant de regrouper les caractéristiques communes à ce types d'objets collectés (super-classe `Pickup` par exemple).

6.7 Ennemis et alliés

On peut bien sûr imaginer d'introduire toute sortes de personnages avec des modalités de déplacement et de comportement spécifiques ; pouvant être hostiles ou amicaux à l'égard du joueur (voir la figure 3 pour un petir exemple). Il convient alors de trouver une bonne conception permettant de distinguer la notion de "personnage" (dont fera partie le joueur) du reste des acteurs "non vivants" codés jusqu'ici.

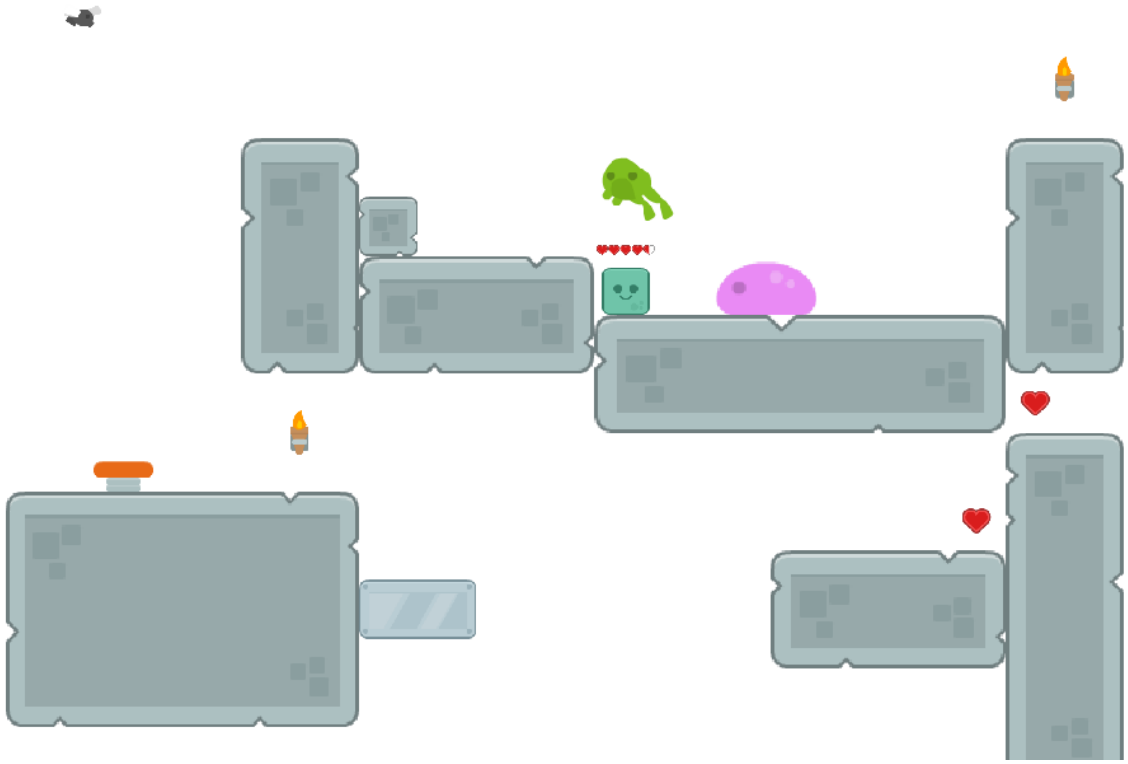


FIG. 3 : Exemple où le joueur doit faire face à des mouches, grenouilles et autres « slimes » hostiles qu'il combattra courageusement à coup de boules de feu.