CS-107 : Mini-projet 2
Jeux d'énigmes sur « Grille »

B. Châtelain, J. Sam, B. Jobstmann

Version 1.5

Table des matières

1 Présentation

Ce projet a pour objectif de vous faire programmer un petit moteur de jeux vous permettant de créer des jeux sur grille[Lien] en deux dimensions de type RPG et dont existe pléthore de déclinaisons célèbres :

Pokémon Emeraude [Lien]

Stardew Valley [Lien]

et tant d'autres . . . mais nous nous bornerons, au vu des temps impartis, à des versions très 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 ce type 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 de conception 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 cinq étapes :

- Étape 1 (« Brique de base ») : au terme de cette étape vous disposerez d'un outil basique permettant de dessiner dans une fenêtre des entités élémentaires (préfiguration de la notion d'acteur d'un jeu), d'en simuler le mouvement et de les contrôler de façon simple.
- Étape 2 (« Jeux de grille ») : cette étape permettra de mettre en place la notion de *grille* et d'*aires de jeu*. Ceci vous permettra de simuler des mondes dépassant les limites de la fenêtre d'affichage. Vous serez alors en mesure de simuler des entités (dont la notion de personnage principal, le « joueur ») placées sur une grille.
- Étape 3 (« Interactions sur grille ») : cette étape permettra aux entités de la grille d'interagir entre elles ou d'être réceptives à la nature d'une case de la grille (par exemple au fait que la case fasse partie d'une porte ou d'un mur). Vous aurez alors

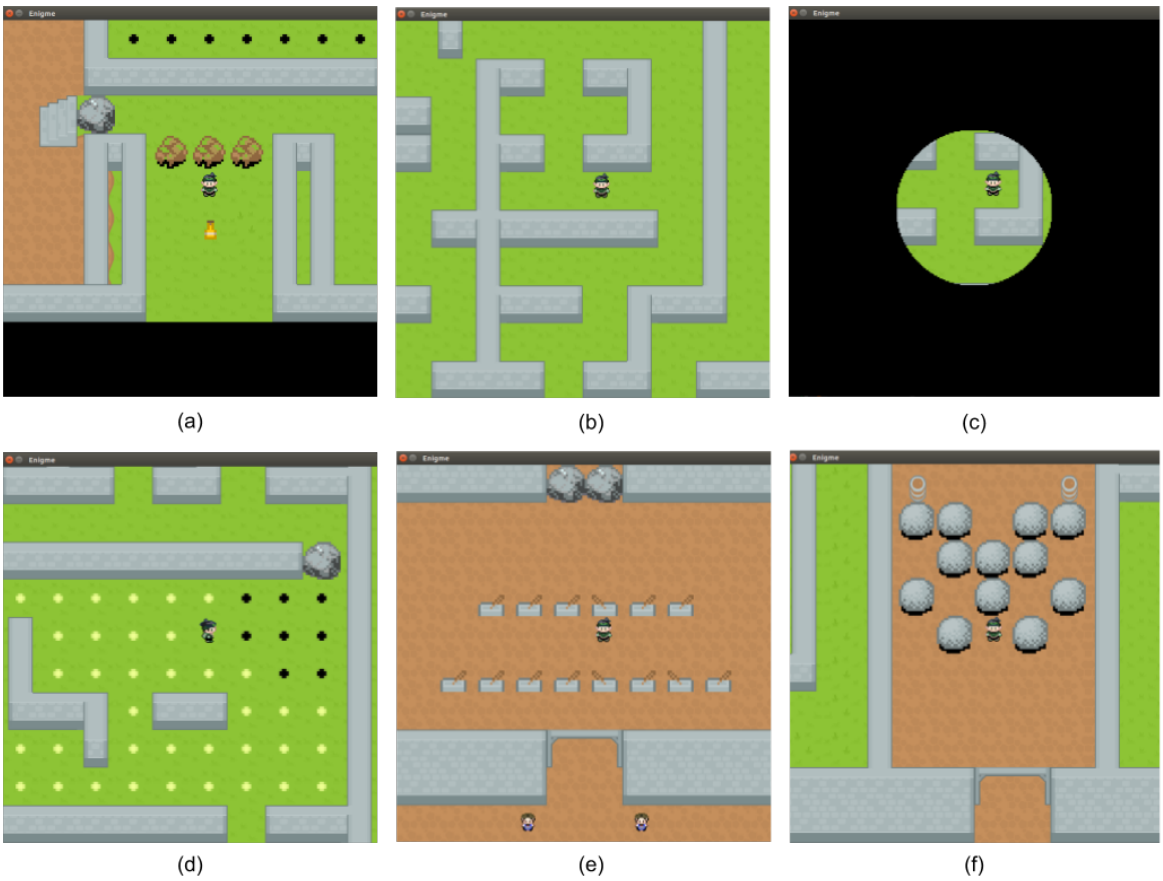


FIG. 1 : Le joueur devra résoudre des énigmes par exemple : (a) casser un rocher avec un objet à trouver, (b) et (c) trouver son chemin dans un labyrinthe avec ou sans champ de vision restreint, (d) activer tous les signaux en marchant dessus, (e) trouver la bonne combinaison de leviers ou (f) se frayer un passage en poussant les rochers pour accéder à des ressources utiles ou à d'autres niveaux de jeux.

mis en place l'essentiel de l'architecture de votre projet. Cette dernière vous permettant de dériver toutes sortes de jeux sur grilles.

- Étape 4 (« Jeux d'énigmes » sur grille) : durant cette étape le moteur sera enrichi de composants aux interactions plus complexes, ce qui permettra de créer des petits jeux d'énigmes : le joueur doit par exemple ouvrir des portes, activer des leviers, utiliser des plaques de pression etc. pour parvenir à un but souhaité.
- Étape 5 (Extensions) : Durant cette étape, diverses extensions plus libres vous seront proposées et vous pourrez créer un jeu de votre propre invention.

Coder quelques extensions (à choix) fait partie des objectifs du projets.

Les trois premières étapes sont volontairement très guidées. Il s'agira essentiellement de

prendre en main l'architecture de base suggérée, les outils fournis, de bien comprendre les problématiques soulevées à chaque fois et comment nous vous proposons d'y répondre[1].

Une partie du matériel sera évidemment fournie.

Voici les consignes/indications principales à observer pour le codage du projet :

1. Les situations d'erreurs sur les paramètres des méthodes (objets nécessaires valant indûment `null`, dimensions invalides, fichiers non trouvés etc.) seront considérées comme des erreurs irrécupérables causant l'arrêt des jeux lancés.

2. Le projet sera codé avec les outils Java standard (import commençant par `java.` ou `javax.`). Si vous avez des doutes sur l'utilisation de telle ou telle librairie, posez-nous la question et surtout faites attention aux alternatives que Eclipse vous propose d'importer sur votre machine. Le projet utilise notamment la classe `Color`. Il faut utiliser la version `java.awt.Color` et non pas d'autres implémentations provenant de divers packages alternatifs.

3. Vos méthodes seront documentées selon les standard javadoc (inspirez-vous du code fourni).

4. Votre code devra respecter les conventions usuelles de nommage et être bien modularisé et encapsulé.

5. Les indications peuvent être parfois très détaillées. **Cela ne veut pas dire pour autant qu'elles soient exhaustives**. Les méthodes et attributs nécessaires à la réalisation des traitements voulus ne sont évidemment pas tous décrits et ce sera à vous de les introduire selon ce qui vous semble pertinent et en respectant une bonne encapsulation.

[1] 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 programme sont hors de portée de ce projet. De plus un des objectifs et de vous apprendre à composer avec du code existant et d'en tirer parti.

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

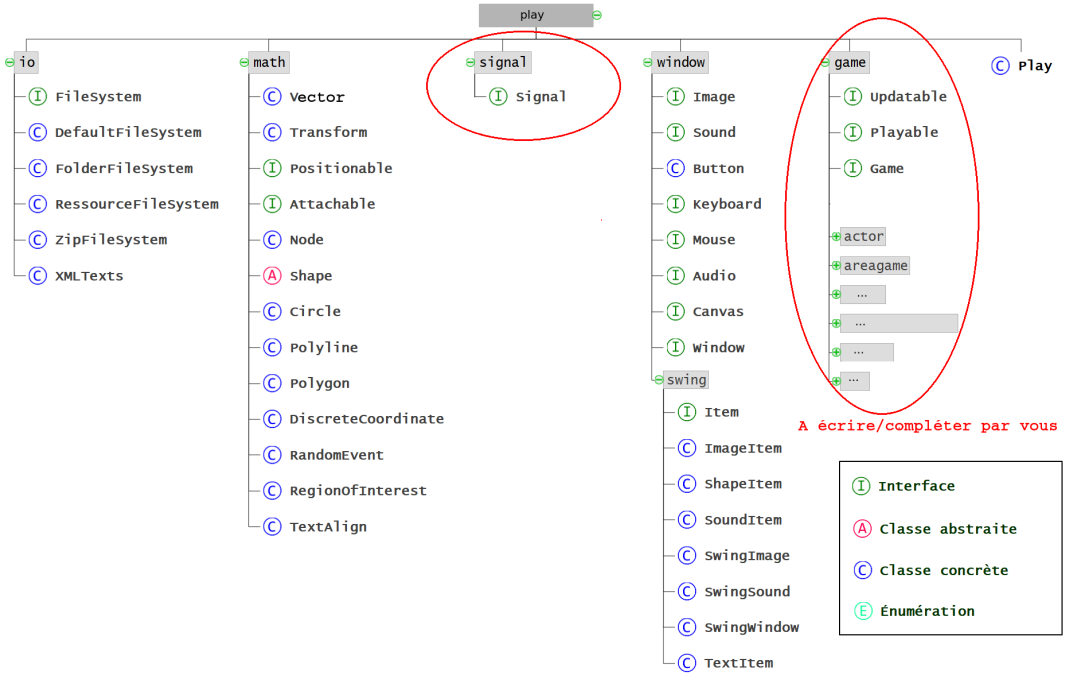


FIG. 2 : Paquetages principaux du projet. Vous interviendrez essentiellement dans le paquetage `game` et le paquetage `signal`

Un petit descriptif des paquets fournis est donné ci-dessous. Le code et sa documentation devraient répondre aux détails, vous donnant ainsi l'occasion d'accéder à un code émanant de programmeurs plus expérimentés[2].

Il ne vous est pas demandé de consulter ce matériel dans le détail, mais d'y revenir quand vous coderez, au gré de vos besoins (et la plupart du temps en fonction de nos indications). Vous trouverez également dans les annexes 9 quelques compléments d'information utiles.

- Le paquetage `io` contient divers utilitaires permettant de gérer des entrées-sorties sur fichiers. Typiquement, les images qui serviront à représenter les entités peuplant nos jeux sont stockées dans des fichiers et ces utilitaires permettront de lire ces fichiers et de les exploiter.

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

- Le paquetage `math` permet de modéliser des concepts mathématiques tels que les vecteurs, les transformations affines et les variables aléatoires. Ce paquet inclut des notions de géométrie du plan (soit à deux dimensions). Il développe par exemple ce qu'est une forme et plus précisément une ligne, un cercle ou un polygone, que vous pourrez utiliser ensuite lors de calculs mathématiques ou lors de la représentation d'éléments graphiques. Les points et tailles sont toujours donnés en valeurs flottantes pour se rapprocher un maximum du plan géométrique continu que nous essayons de simuler. Pour garder une certaine cohérence avec les notions géométriques de base, l'axe vertical *oy* y sera par définition orienté vers le haut et l'axe horizontal *ox* vers la droite.

- Le paquetage `window` : fournit les abstractions `Window` (fenêtre), `Canvas` (zone de dessin), `Mouse` (souris), `Keyboard` (clavier) etc. modélisant les éléments de base liés à l'interface graphique. La classe `SwingWindow` du répertoire `swing` est une réalisation concrète de la notion de fenêtre basée sur les composants Java Swing. Les objets ayant accès au canevas peuvent demander le dessin d'une image, d'une forme ou d'un texte. Une demande ajoute un item graphique du type correspondant dans une liste qui sera triée puis rendue à la prochaine mise à jour de la fenêtre. La liste est ensuite vidée en attendant les demandes pour l'actualisation suivante. De manière analogue, les objets ayant accès au contexte audio peuvent demander qu'un son soit joué. Cet outillage permet de représenter une fenêtre de jeu vidéo dynamique qui sera probablement (pour la plupart des jeux du moins) rafraîchie à relativement haute fréquence (de 20 à 60 fois par seconde). Ces demandes de dessins pour les items graphiques devront donc être réitérées régulièrement, idéalement une fois avant chaque rafraîchissement tant qu'ils doivent être rendus. Le résultat graphique obtenu à la fin de chaque mise à jour est appelé *frame*. Nous parlerons donc d'avoir entre 20 et 60 frames par seconde.

- Le paquetage `game` est celui qui va vous occuper tout au long de ce projet. C'est en effet ce dernier que vous serez amené à compléter selon les directives de l'énoncé. Un certain nombre d'interfaces et d'ébauches de classes y sont déjà présentes. Ces éléments vous seront expliqués au fur et à mesure que vous avancerez. Les petits rectangles gris avec des pointillés représentent des instances de jeux que vous aurez à créer.

- Le paquetage `Signal` permettra d'inclure des composants liés aux jeux d'énigme. Vous aurez à le compléter.

Vous trouverez sous ce lien, la javadoc du code fourni.

3 Briques de base (étape 1)

Cette partie du projet vise à commencer à prendre en main l'architecture fournie et à l'enrichir en y insérant vos premiers éléments de 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 Notion de « jeu »

Comme point de départ plutôt évident, intéressons nous à la modélisation de la notion de « jeu ». Nous allons partir d'une abstraction d'assez haut niveau, qui consiste à dire :

1. qu'un jeu est forcément quelque chose qui *évolue* au cours du temps ;
2. et que pour être *jouable* il doit pouvoir
   - *commencer* proprement (c'est-à-dire s'initialiser notamment en incorporant toutes les entités qui sont amenées à y évoluer) ; lancer un jeu nécessitera vraisemblablement l'accès à un contexte graphique (pour indiquer sur quoi faire les rendus graphiques) et à un système de fichiers (pour permettre d'accéder à des fichiers de ressources utiles) ;
   - et *se terminer* proprement (mettre en oeuvre un certain nombre d'actions qui caractérisent sa fin, cela peut être un message de fin apparaissant à l'écran ou tout autre action pertinente).

Pour partir de ce modèle, les entités suivantes sont fournies dans le paquetage `game` :

- l'interface `Updatable` qui modélise toute entité évoluant au cours du temps et qui exige de ce fait l'implémentation d'une méthode d'évolution appelée `update` ;
- l'interface `Playable` qui hérite de `Updatable` et qui impose en plus l'implémentation d'une méthode `begin` (qui gère le commencement d'une entité « jouable ») et `end` qui en gère la fin ;
- et enfin l'interface `Game`, qui est un `Playable` et qui impose en plus la définition d'une méthode `getFrameRate` qui dicte la fréquence de rafraîchissement voulue pour la partie graphique du jeu.

Quelqu'un souhaitant lancer un jeu devrait alors s'y prendre comme dans le programme principal du fichier `Play` :

- Le programme commence par créer une instance de jeu (ligne 31), un système de fichier (ligne 28) et une fenêtre (ligne 35).
- Ensuite, il lance le jeu avec `begin` en lui passant en paramètres le système de fichier pour le connecter au monde extérieur et la fenêtre pour lui donner accès à un contexte graphique (et audio).

- Une fois le jeu lancé, et en fonction de la fréquence de rafraîchissement demandé, le jeu et la fenêtre seront l'un après l'autre mis à jour (lignes 65 et 68). L'actualisation de la fenêtre consiste à redessiner son contenu depuis une liste d'items graphiques vidée après chaque itération. C'est le rôle du jeu, lors de sa propre mise à jour, de faire les demandes de dessin (et de son) pour approvisionner cette liste en prévision de la frame suivante. Pour le jeu, les mises à jour consistent d'abord à actualiser tous ses composants (par exemple les repositionner) en fonction du temps écoulé depuis le dernier appel, puis d'exécuter les demandes pour les redessiner (et lancer des sons si nécessaire).

3.1.1 Premier « jeu »

Pour voir concrètement à quoi correspondent les explications ci-dessus, créez un paquetage `demo1` dans le paquetage `game`. Ajoutez à ce paquetage une classe `Demo1` destinée à contenir votre tout premier « jeu ». `Demo1` va implémenter `Game` et donc devoir redéfinir les méthodes nécessaires :

- La méthode `getTitle` retournera la chaîne de caractères *`"Demo1"`*.
- La méthode `getFrameRate` retournera un entier raisonnable (prenez 24).
- La méthode `begin` retournera simplement `true` (le jeu peut toujours être lançé proprement à ce stade).

Le corps des autres méthodes sera laissé vide.

Faites ensuite en sorte que le programme principal `Play`, lance votre jeu `Demo1` (utilisez `` `Ctrl-Shift-O` `` dans Eclipse pour réactualiser les importations) et exécutez `Play`.

Vous devriez voir se lancer le jeu ... du vide inter-sidéral, c'est à dire une fenêtre noire. Un besoin impérieux d'y placer des « acteurs » y jouant un rôle devrait en principe se saisir de vous.

3.2 Acteurs génériques

Nous allons considérer que tous les jeux que nous souhaitons programmer mettront en scènes des *acteurs* agissant selon certaines modalités. Ces derniers pourront avoir toutes sortes de déclinaisons, allant de la simple pièce géométrique (comme dans un Tetris®) à un personnage complexe de RPG.

Le matériel fourni offre déjà un certain nombre de classes et d'interfaces permettant de modéliser la notion d'acteurs de façon générique (voir le répertoire `game.actor` dans le code fourni ainsi que ce schéma de classe[Lien]).

La classe `Entity` est une implémentation abstraite de l'interface `Actor`, elle représente une entité dotée d'une position, d'une vitesse et d'un référentiel qui lui est propre (accessible au moyen de `getTransform`). Un petit complément d'explication sur la notion de transformée

et de référentiel est donné dans l'annexe 9. Il n'y a en principe pas besoin de comprendre cette notion en profondeur pour ce projet.

3.2.1 Premier acteur

Reprenez votre classe `Demo1` et ajoutez lui un premier acteur en guise d'attribut :

```
private Actor actor1;
```

Et, soyons fou, faisons de cet « acteur » un cercle rouge.

Dans la méthode `begin`, initialisez pour cela `actor1` au moyen de la ligne suivante :

```
new GraphicsEntity(Vector.ZERO,
                   new ShapeGraphics(new Circle(radius), null,
                                     Color.RED, 0.005f));
```

où `radius` représente le rayon de notre acteur « cercle rouge » et vaut la valeur `0.2f`.

`actor1` est donc une entité graphique positionnée en `(0,0)` et associé à l'image d'un cercle de diamètre `.2f`, sans couleur de remplissage associée, avec un pourtour rouge d'épaisseur `0.005f`.

La méthode `update` de `Demo1` aura maintenant pour vocation de

- mettre à jour notre acteur : ici notre cercle rouge ne fait rien donc un simple commentaire suffit :

  ```
  // ici donner un peu de vie au premier acteur si
     nécessaire
  ```

- puis de le dessiner, ce qui doit nécessairement se faire dans la fenêtre associée au jeu

  ```
  actor1.draw(window);
  ```

On voit qu'il est nécessaire que la méthode `update` ait accès à la fenêtre mise à disposition lors de l'appel à `begin` (il en sera de même pour l'accès au système de fichiers !). Il devient donc nécessaire d'ajouter ces deux attributs à `Demo1` :

```
private Window window;
private FileSystem fileSystem;
```

et de les initialiser dans la méthode `begin`. Une fois ces modifications faites, relancez `Play`. Vous devriez voir se dessiner un cercle rouge centré au milieu de la fenêtre.

Parlons un peu de ce qui se passe au niveau du choix des valeurs et des positions. Par défaut (si l'on ne fait rien), la fenêtre de dessin est centrée à l'origine et est considérée comme une vue 1x1 du monde simulé. Ainsi, si l'on change la taille du cercle dessiné à 0.5, il va occuper toute la fenêtre, puisqu'il sera de taille 1x1. Si l'on veut visualiser des mondes à un autre échelle, il suffit d'appliquer un changement d'échelle à la fenêtre. Par exemple, l'ajout des instructions suivante après l'initialisation de l'attribut `window` dans `begin` :

```
Transform viewTransform =
   Transform.I.scaled(10).translated(Vector.ZERO);
window.setRelativeTransform(viewTransform);
```

permettrait de représenter un monde 10x10 (et notre cercle serait pour le coup 10 fois plus petit !). De façon analogue, si l'on voulait décaler la vue vers la droite, on pourrait faire une translation de la fenêtre vers la gauche :

```
Transform viewTransform =
   Transform.I.scaled(1).translated(new Vector(-0.2f, 0.0f));
window.setRelativeTransform(viewTransform);
```

L'annexe 9 vous donne un petit complément d'explication sur les transformées.

3.3 Premier jeu où il se passe des choses..

Le premier acteur est codé en « dur » dans le jeu. Cela reste admissible car il est extrêmement basique. Intéressons nous maintenant à coder un acteur plus complexe auquel nous allons dédier une classe à part entière. Notre imagination fertile étant sans limites, notre nouvel acteur sera un rocher qui se déplace.

3.3.1 Un second acteur plus complexe

Créer un sous-paquetage `demo1.actor` dans lequel vous coderez une nouvelle classe d'acteurs appelé `MovingRock`. Il s'agira d'un acteur doté d'une représentation graphique, c'est à dire une sous-classe de `GraphicsEntity` et auquel sera associé un petit texte. Il aura donc pour attribut :

```
private final TextGraphics text;
```

Son constructeur aura pour entête :

```
public MovingRock(Vector position, String text)
```

Il invoquera l'un des constructeurs de sa super-classe avec les arguments suivants :

```
position, new ImageGraphics(ResourcePath.getSprite("rock.3"),
    0.1f,0.1f, null, Vector.ZERO, 1.0f, -Float.MAX_VALUE)
```

L'image associée sera ainsi recherchée dans le répertoire `res/images/sprites/rock.3.png`. Le constructeur initialisera aussi l'attribut spécifique au moyen de la tournure :

```
new TextGraphics(text, 0.04f, Color.BLUE);
```

Pour faire en sorte que le texte soit associé au rocher et donc se déplace plus tard avec lui, il faut l'y attacher :

```
text.setParent(this);
```

Le point d'ancrage du texte peu être décalé par ce genre de tournure :

```
this.text.setAnchor(new Vector(-0.3f, 0.1f));
```

Faites en sorte que la méthode `draw` dessine l'objet et le texte associé, vous devriez voir s'afficher ceci :

FIG. 3 : `MovingRock` est un acteur graphique (rocher) auquel est associé un texte

> Remarque : dans tous les exemples d'affichage donnés, le fond de la fenêtre est blanc. Il devrait être noir lorsque vous exécutez le code avec la matériel fourni (vous pouvez changer cela à la ligne 163 de `window/swing/SwingWindow.java`).

3.3.2 Contrôles

Pour donner un peu de vie à tout cela, faites en sorte que la méthode `update` de `Demo1` fasse bouger le rocher en le décalant d'un pas fixe de `-(0.005f, 0.005f))` lorsque la flèche du bas est appuyée. C'est la méthode update de `MovingRock` qui procédera au décalage :

```
@Override
    public void update(float deltaTime){
      // (here compute displacement in function of deltaTime
         for example)
      // for simplification, deltaTime ignored :
      setCurrentPosition(getPosition().sub(0.005f, 0.005f));
    }
```

Pour tester que la flèche du bas est appuyée :

```
Keyboard keyboard = window.getKeyboard();
Button downArrow = keyboard.get(Keyboard.DOWN);
```

```
if(downArrow.isDown())
{
  //...
}
```

Observer ce qui résulte de mettre en commentaire l'appel à `setParent` qui attache le texte au rocher : le rocher et le texte devraient se désolidariser lors du déplacement !.

Enfin complétez le code de sorte à ce que lorsque la distance entre la position du cercle rouge et celle du rocher est inférieure au rayon du cercle, le texte "BOUM !!!" s'affiche en rouge :

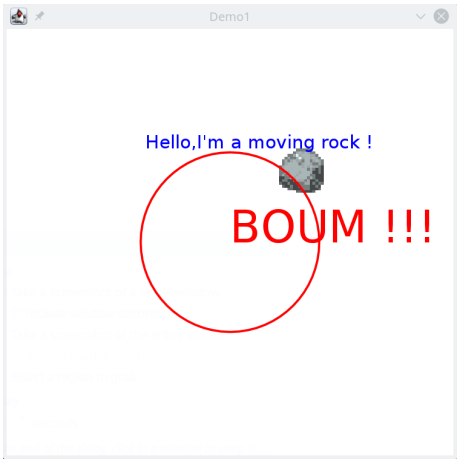


FIG. 4 : Le texte "BOUM !!!" doit cesser de s'afficher lorsque la distance entre le rocher et le cercle redevient trop grande.

Notez que tous les jeux à venir seront codés selon cette structure : le paquetage dédié au jeu sera un sous-paquetage de `game`. Il contiendra lui-même un sous paquetage pour ses acteurs spécifiques.

3.4 Validation de l'étape 1

Pour valider cette étape vous vérifierez que le rocher se déplace en diagonale vers le bas et à gauche lorsque l'on appuie sur la flèche du bas, que le texte se déplace de façon solidaire avec lui, que le texte `"Boum !!!"` s'affiche bien lorsque le rocher devient suffisamment proche du cercle et que ce message disparaît dès qu'il s'en éloigne. Le rocher peut poursuivre sa course en dehors de la fenêtre et disparaître.

Le jeu `Demo1` dont le comportement est décrit ci-dessus est à rendre à la fin du projet.

4 Jeux sur grilles (étape 2)

Le jeu `Demo1` préfigure de façon simple, l'esprit dans lequel vont se coder l'ensemble des jeux à venir. Bien entendu cette façon élémentaire de procéder ne va pas nous suffire :

- il y aura rapidement besoin de manipuler un nombre quelconque d'acteurs, donc un *ensemble* d'acteurs plutôt que deux acteurs codés en dur dans le jeu ;
- cantonner l'aire de déroulement du jeu à la seule fenêtre physique est indubitablement trop frustrant ;
- et la façon dont vont interagir physiquement les acteurs a besoin d'être mieux formalisée (va-t-on gérer des collisions physiques ? comment caractériser les zones d'interaction entre acteurs ? etc.)

La réponse au troisième point va en fait conditionner la façon de mettre en place le reste : nous allons nous orienter vers le codage de jeux se déroulant sur des *grilles*.

Plus concrètement les jeux de ce type (`AreaGame`) seront composés d'un répertoire de zones (ou niveaux) que nous appellerons `Area` (section 4.1). Chaque `Area` définit une grille à deux dimensions composée de cellules (`Cell`) dans lesquelles évolueront et interagiront des composants (des `Actor`). Dans l'optique de ressembler visuellement à certains jeux de type *Game Boy*, cette grille est introduite notamment pour simplifier la gestion des interactions et de déplacement des acteurs : c'est notamment la cellule et son contenu qui conditionneront les interactions (par opposition à une approche gérant les interactions en détectant des collisions « physiques » entre acteurs dans un modèle complètement continu du monde par exemple).

Une `Area` est donc en quelque sorte un jeu plus ou moins indépendant, avec un ensemble d'acteurs, un peu à l'image de ce qu'était notre `Demo1`. Elle sera en charge de gérer les interactions entre ses acteurs.

Pour ne pas surcharger la classe `Area`, une zone n'a pas connaissance directe de la grille qui la définit. Elle délègue cette connaissance ainsi que toutes ses fonctionnalités à une classe `AreaBehavior` (section 4.2). Par conséquent, chaque `Area` possède une `AreaBehavior` qui gèrera le comportement et les mécaniques de la zone avec sa grille, ses cellules et de leur contenu.

Encore une fois, beaucoup de code clé en main vous sera fourni pour le codage des ces classes essentielles dans l'architecture de base fournie. L'idée est que vous partiez d'un socle commun pour vos propres jeux. Construire ces classes par vous même, petit bout par petit bout, vous permet en principe de mieux en comprendre le fonctionnement. Les classes `AreaGame`, `Area` et `AreaBehavior` seront amenées à évoluer encore dans l'étape à venir. Ces classes sont à compléter dans le paquetage `game.areagame`

4.1 Aires de jeu

Il s'agit ici de commencer à compléter la coquille de classe abstraite `Area`, fournie dans le paquetage `game.areagame`.

La classe `Area` modélise donc une zone dans un jeu sur grille. Elle ressemble beaucoup à ce que vous avez déjà codé dans `Demo1`, à la différence près qu'il n'y aura plus deux acteurs spécifiques qui y interviendront, mais une *liste* d'acteurs quelconques. Les attributs de `Area`, ressembleront donc à ceci :

```
// Context objects
private Window window;
private FileSystem fileSystem;

/// List of Actors inside the area
private List<Actor> actors;
```

Elle sera aussi pourvue de méthodes `begin` et `end` codées de façon analogue à ce que vous avez fait dans `Demo1`. La liste des acteurs sera initialisée au moyen d'une liste chaînée (voir l'annexe 9) :

```
actors = new LinkedList<>();
```

La classe `Area` sera aussi dotée d'une méthode `update` faisant évoluer tous ses acteurs. Le rôle de cette méthode est, pour l'instant, de faire évoluer tous les acteurs de la liste `actors` (au moyen de leur propre méthode `update`) puisse de les dessiner dans `window`.

Bien entendu, une aire de jeu devra permettre l'incorporation ou la suppression d'acteurs dans sa liste.

Vous y compléterez donc les méthodes d'ajout et de retrait d'acteurs. Nous vous proposons d'ébaucher la méthode `addActor` comme suit :

```
private void addActor(Actor a, boolean forced) {

  // Here decisions at the area level to decide if an actor
  // must be added or not
  boolean errorOccured = !actors.add(a);

  if(errorOccured && !forced) {
        System.out.println("Actor " + a + " cannot be
           completely added, so remove it from where it
           was");
        removeActor(a, true);
    }
}
```

La méthode `removeActor` sera codée de façon analogue. Quelques explications s'imposent sur cette proposition de codage. En fait, nous faisons le choix de conception que l'aire doit avoir autorité sur son contenu. Même si elle est amenée à déléguer une partie des traitements à son gestionnaire de grille (`AreaBehavior`, voir plus bas section 4.2), elle doit garder en tout temps un droit de *veto* sur les opérations d'ajout et suppression d'acteurs, ainsi que sur les déplacements dans la grille. La proposition de codage de `addActor` permettra d'intégrer plus tard une stratégie éventuellement restrictive de la part de l'aire comme :

```
private void addActor(Actor a, boolean forced){
  boolean errorOccured = !agreeToAdd(a);
  errorOccured = errorOccured || vetoFromGrid();
  //...
  }
```

La grille pourra refuser (`vetoFromGrid()`)[3] un ajout si par exemple une cellule est déjà occupée par une autre entité non-traversable ou alors que la cellule n'est pas adaptée à l'entité proposée. Si la grille ou l'aire s'opposent à l'ajout (ou au retrait), alors l'opération est annulée. Le paramètre `forced` lorsqu'il est à `false` indique que l'on veut tenir compte du *veto* de l'aire ou d'autres composants et ne pas forcer l'ajout ou le retrait.

Vous vous posez maintenant sans doute la question de pourquoi donner un droit d'accès privé à ces méthodes. C'est ce que nous allons voir ci-dessous.

4.1.1 Modification d'une liste en cours de parcours

La méthode `update` de `Area` va faire évoluer chacun des acteurs au moyen de sa méthode `update` spécifique. Il se peut qu'en évoluant, un acteur cause la création d'un nouvel acteur ou la suppression d'un acteur existant (exemple : en ouvrant un coffre, il libère un mauvais génie qui deviendra un actif sur l'aire de jeu au même titre que les autres). Or, modifier le contenu d'une collection pendant son parcours au moyen d'une itération sur ensemble de valeurs n'est pas possible[Lien]. Ceci provoque dans le cas général une `ConcurrentModificationException`. Pour éviter ce problème, nous recourons à un schéma classique. L'idée consiste à enregistrer les nouveaux venus, ou ceux à disparaître, dans des listes d'attente, et de mettre à jour `actors` après que tous ses éléments ont reçu les événements `update` et `draw`.

Ajoutez ces listes d'attente comme nouveaux attributs :

```
private List<Actor> registeredActors;
private List<Actor> unregisteredActors;
```

Codez ensuite les méthodes permettant de gérer ces listes :

- `registerActor(Actor a)` ajoutera simplement l'acteur `a` à la liste `registeredActors` ;
- `unregisterActor(Actor a)` ajoutera l'acteur `a` à la liste `unregisteredActors` ;

Enfin, modifiez votre méthode `update` de sorte à ce qu'elle fasse appel à une méthode `final void purgeRegistration()` :

```
@Override
  public void update(float deltaTime) {
    purgeRegistration();
    // suite comme avant
    }
```

[3] `agreeToAdd()` et `vetoFromGrid()` sont des noms donnés à titre d'exemple, libre à vous d'implémenter cela à votre guise le moment venu

La méthode `purgeRegistration` va :

- ajouter à la liste d'acteurs de l'aire de jeu tous les acteurs de `registeredActors` (en tenant compte du *veto* éventuel de l'aire, paramètre `forced` à `false` donc) ;
- supprimer de la liste d'acteurs tous les acteurs de `unregisteredActors` (toujours en tenant compte du *veto* éventuel de l'aire) ;
- vider les listes `registeredActors` et `unregisteredActors`

Ainsi, si un acteur cause la création d'un nouvel acteur, il ne pourra que l'enregistrer dans la liste des acteurs à venir (la méthode `addActor` ne lui étant pas accessible). Le nouvel acteur sera pris en compte lors du pas de simulation suivant. La méthode `purgeRegistration` sera évidemment privée.

4.1.2 Transition d'une aire à l'autre

Nos jeux sur grilles seront composés de plusieurs aires dont une seule (l'aire courante) sera jouée à la fois. Lorsque l'on passe d'une aire à l'autre, plusieurs stratégies peuvent être envisagées : si l'on revient sur une aire déjà jouée auparavant, on peut par exemple soit reprendre le jeu sur cette aire depuis le début ou alors dans l'état où on l'avait laissée. Pour cela, il est raisonnable d'anticiper dans la classe `Area` les méthodes suivantes :

- `public void suspend()` qui par défaut ne fait rien d'autre qu'invoquer `purgeRegistration` mais qui, une fois redéfinie, mettra en oeuvre toutes les autres actions nécessaires lorsque l'on quitte une aire de jeu pour passer à une autre (comme la sauvegarde éventuelle d'informations sur l'état de l'aire de jeu) ;
- `public boolean resume(Window window, FileSystem fileSystem)` qui retourne `true` par défaut mais qui, une fois redéfinie, permettra de reprendre le jeu sur une aire à partir d'un état intermédiaire éventuel où on l'aurait laissé. Le booléen de retour indique si la reprise du jeu sur l'aire a été possible ou pas.

4.1.3 Gestion de la caméra

Une aire de jeu peut être vaste et dépasser ce qui est visible dans la fenêtre. Il est donc nécessaire de permettre à la vue de se placer à un endroit précis d'une aire donnée. Nous avons déjà vu plus haut comment agir sur la vue :

```
// Compute new viewport
Transform viewTransform =
   Transform.I.scaled(scaleFactor).translated(viewCenter);
  window.setRelativeTransform(viewTransform);
```

Examinons comment peuvent être spécifiés le facteur d'échelle (`scaleFactor`) et le centre de la vue. Le facteur d'échelle est dépendant du jeu concret implémenté. Il sera retourné par une méthode abstraite :

```
    /** @return (float) : camera scale factor, assume it is
       the same in x and y direction */
    public abstract float getCameraScaleFactor();
```

Dans de nombreuses situations, il est judicieux de penser que la vue sera centrée sur un acteur. Nous vous proposons d'ajouter à la classe `Area` les attributs suivants :

```
    // Camera Parameter
    // actor on which the view is centered
    private Actor viewCandidate;
    // effective center of the view
    private Vector viewCenter;
```

`viewCenter` sera initialisé à `Vector.ZERO` dans la méthode `begin` et `viewCandidate` à `null`.

`viewCandidate` pourra être modifié depuis l'extérieur :

```
    public final void setViewCandidate(Actor a){
        this.viewCandidate = a;
    }
```

Vous doterez `Area` d'une méthode privée permettant de positionner la caméra `void updateCamera()` qui met en oeuvre l'algorithme suivant :

- s'il y a un acteur sur lequel centrer la vue (`viewCandidate` différent de `null`) affecter la position de l'acteur à `viewCenter` ;
- positionner la vue comme indiqué précédemment en lui appliquant le facteur d'échelle voulu et en lui appliquant la translation dictée par `viewCenter`.

Bien entendu, `updateCamera` devra être invoquée avant le dessin des acteurs dans la méthode `update`, afin de régler les paramètres de la vue comme souhaité avant le rendu. Complétez la classe `Area` fournie dans le paquetage `game.areagame` en tenant compte de la description ci-dessus.

4.2 Grille

On souhaite maintenant modéliser le fait que l'aire de jeu prend place sur une grille qui va conditionner les comportements de tout ce qui y prend place. La classe abstraite `AreaBehavior` permet de modéliser une telle grille. Cette classe a pour attribut un tableau de cellules qu'elle initialise depuis une image couleur comme celle de la figure Figure 5 où chaque pixel représente une cellule et chaque couleur une nature ou un type différent de cellule. Nous verrons un peu plus loin comment sera mise en place cette correspondance et comment elle sera utilisée.

Il s'agit donc ici de produire une première version simple de la classe abstraite `AreaBehavior` du paquetage `game.areagame`. Cette version sera amenée à évoluer lors de la prochaine étape.

Commencez par introduire les attributs caractéristiques de cette classe :

FIG. 5 : Exemple d'une image de comportement avec la correspondance couleur-type

```
/// The behavior is an Image of size height x width
 private final Image behaviorMap;
 private final int width, height;
 /// We will convert the image into an array of cells
 private final Cell[][] cells;
```

La notion de cellule est spécifique à nos grilles de jeux et il est de ce fait plus pertinent de faire du type `Cell`, une classe imbriquée :

```
public abstract class AreaBehavior {
  //...
   /**
   * Each game will have its own Cell extension.
   */
  public abstract class Cell {
   //...
  }
}
```

`Cell` représente une cellule générique et chaque extension de `AreaBehavior`, qui sera spécifique à un jeu donné, devra bien sûr aussi redéfinir des extensions spécifiques de la classe `Cell`.

Par ailleurs, la gestion des interactions nécessitera l'accès à la cellule d'une grille. En effet, pour gérer les interactions que les acteurs pourront avoir avec les cellules il devront pouvoir y accéder. C'est la raison pour laquelle la classe `Cell` est prévue en accès `public`.

Une `Cell` est destinée à avoir un contenu, mais pour l'heure nous ne nous y intéressons pas encore. Elle sera donc à ce stade simplement caractérisée par ses coordonnées sur la grille (de type `DiscreteCoordinates`). Son constructeur prendra en paramètre deux entiers `x` et `y` permettant d'initialiser ses coordonnées.

Dotez la classe `AreaBehavior` d'un constructeur prenant en paramètre un contexte graphique (de type `Window`) et une chaîne de caractères représentant le nom, `fileName`, du

fichier contenant l'image à associer à l'`AreaBehavior`. Ce constructeur initialisera :

- l'attribut `behaviorMap` au moyen du contenu du fichier `fileName`, c'est à dire ce qui est retourné par la tournure suivante :

  ```
  window.getImage(ResourcePath.getBehaviors(fileName), null,
     false);
  ```

- les attributs `width` et `height` à respectivement `behaviorMap.getWidth()` et `behaviorMap.getHeight()` ;
- et le tableau `cells` comme un tableau à `width` lignes et `height` colonnes sans contenu.

4.2.1 Aire de jeu sur grille

Maintenant que nous savons modéliser de façon basique une grille au travers de la classe `AreaBehavior`, il est temps de l'associer à nos aires de jeux, qui rappelons-le, sont toutes destinées à se jouer sur des grilles. Ajoutez pour cela un nouvel attribut à la classe `Area`

```
/// The behavior Map
  private AreaBehavior areaBehavior;
```

Chaque aire de jeu, sera bien entendu associée à une sous-classe concrète de `AreaBehavior`. Cette association ne sera pas forcément unique et nous faisons donc le choix de permettre la modification de l'`AreaBehavior` associée à une `Area`. Prévoyez pour cela la méthode protégée et finale `setBehavior(AreaBehavior ab)`.

4.3 Jeux avec aires

A ce stade, nous savons donc représenter une aire de jeu (`Area`) et lui associer une grille qui définit son comportement (`AreaBehavior`). Pour compléter notre conception de base des jeux sur grilles, il ne reste plus qu'à définir les jeux composés de *plusieurs aires*.

Complétez pour cela la classe abstraite `AreaGame` du paquetage `game.areagame`.

Vous la doterez des attributs propres à un jeu tel que nous avons désormais pris l'habitude de le faire, avec en plus un attribut permettant de représenter l'*ensemble* des aires qui constituent le jeu et un attribut représentant l'aire de jeu courante (qui sera la seule à être simulée) :

```
// Context objects
  private Window window;
  private FileSystem fileSystem;
  /// A map containing all the Area of the Game
  private Map<String, Area> areas;
  /// The current area the game is in
  private Area currentArea;
```

Comme structure de données pour l'ensemble des aires nous avons choisi le type `Map` (table associative clé–valeur) qui nous permettra de retrouver une aire de jeu à partir de son nom (`getTitle`) (voir l'annexe 9).

Vous coderez les méthodes `begin` et `end` de la classe `AreaGame` de façon analogue à ce que vous avez fait pour coder votre jeu `Demo1`. L'ensemble des aires y sera initialisé à un ensemble *vide* comme suit :

```
areas = new HashMap<>();
```

La méthode `update` se contentera de mettre à jour l'aire courante.

L'ensemble des aires de jeux n'est pas figé au démarrage. Nous faisons en effet le choix de permettre l'ajout dynamique d'aires de jeu (pendant le déroulement du jeu).

Pour cela nous devons définir les méthodes suivantes :

- ```
  protected final void addArea(Area a) {
      areas.put(a.getTitle(), a);
  }
  ```

- ```
  protected final Area setCurrentArea(String key, boolean
     forceBegin){
      // algorithme expliqué plus bas
  }
  ```

Ces méthodes étant potentiellement sensibles pour l'encapsulation, nous avons fait le choix de les définir comme protégées et final : il n'y a qu'une façon définitive d'ajouter une aire de jeu ou de cibler l'aire courante et seules les sous-classes de `AreaGame` et celles du paquetage `area.areagame` (un certain nombre restreint de classes en charge de la logistique des jeux avec aires sur grille) seront habilitées à les employer.

La méthode `setCurrentArea` mettra en oeuvre l'algorithme suivant :

1. si l'aire courante ne vaut pas `null`, la suspendre et y purger tous les éléments en attente d'enregistrement ;
2. affecter l'aire à laquelle on veut passer (`areas.get(key)`) à l'aire courante ;
3. si cette aire existe bel et bien (ne vaut pas `null`) :
   - si elle n'a jamais été abordée en cours de jeu ou si `forceBegin` vaut `true` on la (re)commence depuis zéro (appel de sa méthode `begin`)
   - sinon, on la poursuit (méthode `resume`) ;
4. si elle n'existe pas, l'aire courante redevient celle qu'elle était précédemment et si cette dernière vaut `null` on lance une exception (choisissez celle vous semblant la mieux appropriée).

La mise en oeuvre de l'algorithme ci-dessus, nécessite de savoir si une aire a déjà été abordée en cours de jeu. Ajoutez ce qui vous semble nécessaire au code pour vérifier cette condition, sans nuire à l'encapsulation.

Vous noterez que les interfaces sont un puissant outil d'encapsulation : `Area` et les acteurs auront besoin de se connaître mutuellement, ce qui implique de leur part d'ouvrir l'accès à certaines informations (failles d'encapsulations potentielles). Cependant, si en tant qu'utilisateur, on s'astreint à la discipline de ne voir d'un jeu que sa logique d'utilisation édictée par `Game` (comme c'est le cas de `Play` par exemple), alors les accès sensibles ne sont plus exposés.

4.4 Acteur pour le « fond d'écran »

Chaque aire de jeu sera assortie d'un « fond d'écran » qui définira son aspect visuel. On peut imaginer qu'un tel visuel peut changer au cours du temps (par exemple des visuels différents pour un cycle nuit-jour). Au lieu de coder le fond d'écran associé à une aire comme un attribut figé, il est possible de définir ce dernier comme ... un acteur. Cet acteur ressemble beaucoup plus dans l'esprit à celui que vous avez codé dans `Demo1` : il s'agit simplement d'un dérivé de `GraphicEntity` tout comme `MovingRock`. Le code vous est fourni dans la classe `Background` du paquetage `areagame.actor`. Commencez par y jeter un oeil. Vous constaterez que par défaut l'image associée à cet acteur graphique est celle dont le nom de fichier correspond au nom de son aire tel que retourné par la méthode `getTitle()`. Ainsi, il s'agira par exemple de `Enigme1.png` du dossier `res/images/background` si le `getTitle()` retourne *`"Enigme1"`*.

Pour pouvoir ajuster sa taille à celui de l'aire sur laquelle il s'applique comme fond, l'acteur `Background` a besoin de connaître les dimensions de cette aire. Ajoutez pour cela les méthodes `getWidth()` et `getHeight()` manquants à la classe `Area`. La largeur de l'aire est celle de sa grille (`AreaBehavior`) associée ; c'est à dire le nombre de lignes du tableau de cellules associé. Un raisonnement analogue sera tenu pour la hauteur. Comme vous pouvez vous en douter, il y a un lien entre l'aspect du fond d'écran et l'image qui décrit le comportement de la grille :

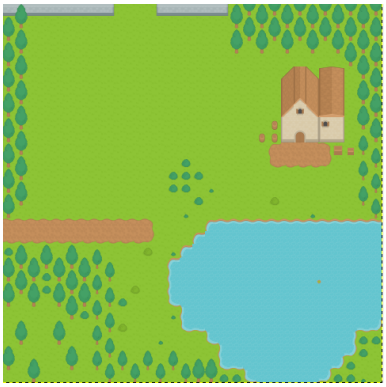
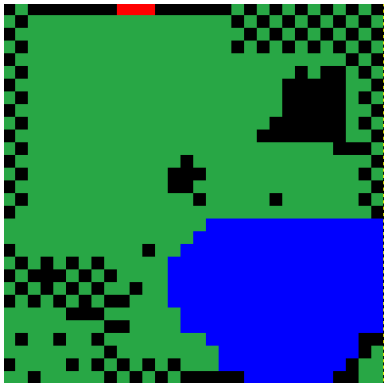


image de fond



« comportement » correspondant

Le répertoire `res/` fournit quelques fonds d'écran dans le répertoire `res/images/background` et les images de « comportement » associées dans `res/images/behavior` (la correspondance

s'établit au travers du nom). L'annexe 9 fournit un outil permettant de créer des images `background` et `behavior` associées[4].

Voilà, une partie importante de la modélisation du support de nos futurs jeux est maintenant en place. Il est temps de commencer à en écrire une instance concrète.

4.5 Premier jeu sur grille

Le but est maintenant de créer un tout premier jeu de grille : `Demo2`. Il s'agit d'une ébauche de nos futurs jeux d'énigme où un personnage (le personnage principal ou « player ») pourra circuler sur une grille, en permettant à cette dernière de lui dicter où il ne peut pas aller, et pourra passer des portes. Vous travaillerez dans le paquetage `game.enigme`. Pour le moment nous allons uniquement mettre en place le support sur lequel ce jeu va se dérouler (le personnage viendra un peu plus tard).

4.5.1 Grilles spécifiques

La classe `AreaBehavior` permet de modéliser de façon très générale et abstraite la grille attachée à une aire de jeu. Il s'agit maintenant d'en coder une version spécialisée, permettant une gestion spécifiques des cellules. Il vous est demandé pour cela de coder dans le paquetage `game.enigme`, une sous-classe `Demo2Behavior` héritant de `AreaBehavior`.

Cette sous-classe aura pour spécificité de donner une interprétation particulières aux cellules de la grille en fonction de la couleur qui leur est associée dans la `behaviorMap` correspondante.

Pour cela définissez dans `Demo2Behavior` le type énuméré :

```
public enum Demo2CellType {
        NULL(0),
        WALL(-16777216),  // RGB code of black
        DOOR(-65536),     // RGB code of red
        WATER(-16776961), // RGB code of blue
        INDOOR_WALKABLE(-1),
        OUTDOOR_WALKABLE(-14112955);

        final int type;

        Demo2CellType(int type){
            this.type = type;
        }
    }
```

Ajoutez à ce type énuméré la méthode `static Demo2CellType toType(int type)` retournant la valeur du type énuméré correspondant à l'entier `type`. Par exemple, `toType(-65536)`

[4] mais il ne vous est pas demandé de l'utiliser forcément

retournera la valeur `DOOR`. La valeur `NULL` sera retournée si `type` ne correspond à aucune valeur prévue pour le type énuméré.

Le type `Demo2CellType` nous permettra d'interpréter la couleur rouge[5] comme une porte, la noire comme un mur et la bleue comme de l'eau.

L'idée est que si l'on associe à `Demo2Behavior` une `imageBehavior` telle que celle de la figure 5 alors les cellules correspondant aux pixels rouges pourront être interprétées commes des portes ce qui lui permettra dans notre cas de passer d'une aire de jeu à l'autre, les cellules correspondant aux pixels noir comme des murs sur lesquels il ne faudra pas marcher.

Il est donc nécessaire de définir de façon adaptée les cellules assorties à `Demo2behavior` de sorte à permettre à cette grille de dicter des contraintes spécifiques en fonction de leur nature. Pour cela, définissez dans `Demo2Behavior` la sous-classe imbriquée publique `Demo2Cell` héritant de `Cell`. Une `Demo2Cell` sera caractérisée par sa nature (de type `Demo2CellType`). Vous doterez `Demo2Cell` d'un constructeur privé d'entête

```
Demo2Cell(int x, int y, Demo2CellType type)
```

Dotez enfin `Demo2Behavior` d'un constructeur permettant d'initialiser la grille en la remplissant de `Demo2Cell`. Pour trouver le type à associer à la `Demo2Cell` de coordonnées `[x][y]` lors de sa construction, vous pourrez utiliser la tournure suivante :

```
Demo2CellType cellType =
   Demo2CellType.toType(getBehaviorMap().getRGB(height-1-y,
   x));
```

4.5.2 Aires de jeu spécifiques

Créez ensuite dans un paquetage `game.enigme.area.demo2`, deux aires spécifiques `Room1` et `Room0` qui auront pour titre respectif *`"Level1"`* et *`"LevelSelector"`*.

Leur méthode `begin` devra en plus leur associer une grille de type `Demo2Behavior` :

```
setBehavior(new Demo2Behavior(window, getTitle()));
```

et y enregistrer un acteur unique de type `Background` :

```
registerActor(new Background(this));
```

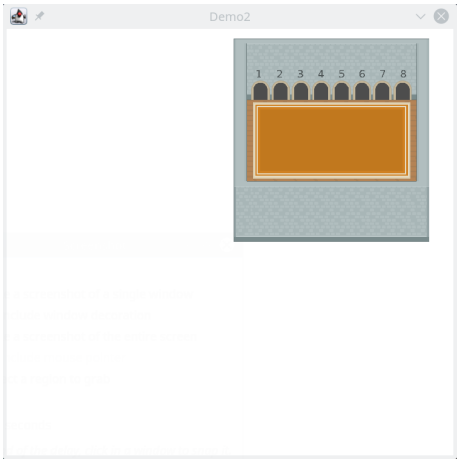
ce qui permettra d'associer à l'aire `Room1` le fond associé à l'image `res/images/backgrounds/Level1.png` (et avec un raisonnement analogue pour `Room0`). Revoyez le code de la classe `Background` si cela n'est pas clair.

[5] https ://stackoverflow.com/questions/25761438/understanding-bufferedimage-getrgb-output-values

Indication : le code des classes `Room1` et `Room0` étant très proches, il est sans doute judicieux de créer une super-classe commune `Demo2Area` dans le paquetage `enigme.area.demo2`. Il est alors important que dans l'instruction `registerActor(new Background(this)) ;` le `this` référence bien la bonne aire (car du titre de cet aire dépend le choix de la bonne image !).

4.5.3 `Demo2`

Créez enfin dans le paquetage `game.enigme` un jeu `Demo2` spécialisant `AreaGame`. Ce jeu sera constitué des deux aires `Room0` et `Room1`. Par défaut, l'aire courante sera `Room0`. Lancez le jeu `Demo2`. Si tout se passe bien, vous devriez voir s'afficher l'aire `Room0` :



Vous allez maintenant oeuvrer à placer un personnage dans ce décor.

4.6 Acteurs de jeux de grille

Nous disposons déjà d'une façon très générique de modéliser des acteurs intervenant dans un jeu au moyen de l'interface `Actor` et de la classe abstraite `Entity`. Nous allons maintenant étendre cette modélisation pour incorporer les spécificités des acteurs évoluant sur une grille. Les classes décrites ci-dessous sont à compléter dans le paquetage `areagame.actor`.

4.6.1 La classe `AreaEntity`

La classe abstraite `AreaEntity` permet de modéliser les acteurs appartenant à une aire de jeu grillagée. Leur principale spécificité pour le moment est qu'ils occuperont des cellules de cette grille. Ils peuvent en toute généralité en occuper plusieurs, mais une seule cellule servira à les localiser, ce que nous appellerons la *cellule principale*. Les acteurs d'une aire de jeu grillagée auront également une *orientation*, qui permettra de les dessiner différemment selon vers où il se dirigent en se déplaçant. Nous partons enfin de l'hypothèse de conception

assez naturelle qu'un tel acteur peut « voir » son voisinage et par conséquent a connaissance de l'aire à laquelle il appartient.

Au vu de cette description, complétez les attributs de `AreaEntity` comme suit :

```
/// an AreaEntity knows its own Area
private Area ownerArea;
/// Orientation of the AreaEntity in the Area
private Orientation orientation;
/// Coordinate of the main Cell linked to the entity
private DiscreteCoordinates currentMainCellCoordinates;
```

Le type `Orientation` est fourni dans le paquetage `game.areagame.actor`.

Complétez également les coquilles de méthodes suggérées en vous basant sur les commentaires javadoc donnés.

La méthode `void setCurrentPosition(Vector v)` sera redéfinie de sorte à s'adapter à fait que l'on travaille désormais sur une grille :

- Si les coordonnées de la position sont suffisamment proches d'une coordonnée discrète (utiliser la méthode `DiscreteCoordinates.isCoordinates`), les coordonnées de la position sont arrondies à l'entier le plus proche (`v.round()`) avant d'être affectée à la position. Les coordonnées principales (`currentMainCellCoordinates`) prennent également cette valeur arrondie dans ce cas.
- Autrement, le comportement de cette méthode est le même que celui de la super-classe.

Nous n'avons pas codé l'acteur `Background` comme une `AreaEntity` car il s'agit d'un acteur qui n'est pas censé « habiter » des cellules de la grille. Ceci démontre qu'un jeu de grille peut parfaitement faire intervenir d'autres types d'acteurs que ceux spécifiquement dédiés à occuper des cellules.

Des getters-setters sont autorisés (et utiles) pour l'orientation, mais devront rester en accès protégés.

4.6.2 Interfaces `Interactor` et `Interactable`

La grille a pour vocation de gérer le contenu de ses cellules et ce qu'il s'y passe, comme par exemple autoriser ou interdire le passage d'un acteur d'une cellule à l'autre et gérer les interactions entre acteurs occupant des cellules identiques ou voisines. Pour ce faire, il est naturel de penser à doter la classe `Cell` de `AreaBehavior` d'un attribut supplémentaire représentant ce contenu avec pour type `Set<Actor>` par exemple (voir l'annexe 9 sur les ensembles).

En fait, les acteurs n'intéressent la grille qu'en tant qu'entités réceptives aux interactions. On peut en effet très bien imaginer que certains acteurs (par exemple un fond d'écran)

soient hermétiques à toute interaction et que de ce fait, ils n'aient pas besoin d'être pris en compte par la grille. De plus, une entité réceptive aux interactions n'est pas forcément un acteur. Ce peut être simplement une cellule. Le type `Set<Actor>` n'est donc pas tout à fait adapté pour modéliser le contenu d'une cellule et il est préférable d'introduire des nouvelles abstractions, représentant des entités capables d'interagir.

Nous vous proposons de le faire au travers des interfaces suivantes, que vous placerez dans le paquetage `game.areagame.actor` :

- `Interactable` : cette interface permet de modéliser toute entité réceptive à une demande d'interaction ;
- `Interactor` : cette interface permet de modéliser toute entité pouvant interagir avec une `Interactable`.

Comme leur nom l'indique, ces deux interfaces sont destinées à fonctionner en symbiose, les `Interactor` étant prévus pour interagir avec les `Interactable`.

Nous partirons de l'hypothèse que toute entité de grille (`AreaEntity`) doit se définir en tant qu'objet sujet à des interactions. Vous ferez donc en sorte que `AreaEntity` du paquetage `area.areagame.actor` implémente l'interface `Interactable`.

Le contenu d'une cellule ne sera donc pas un ensemble d'`Actor`, mais un ensemble de `Interactable`. Ajoutez un attribut de type `Set<Interactable>` dans la classe `Cell` de `AreaBehavior` (nous y reviendrons un peu plus tard).

Par ailleurs, nous ferons la différence entre deux types d'interactions :

- les *interactions de contact* : ont lieu entre un `Interactor` et les `Interactable` se trouvant dans les mêmes cellules que lui.
- les *interactions distantes* : ont lieu cette fois entre un `Interactor` et les `Interactable` se trouvant dans les cellules de son champs de vision.

Pour illustrer cette différence, prenons un exemple. Imaginons une situation avec trois acteurs : deux personnages et une plaque de glace. Les deux personnages peuvent interagir de façon *distante* par exemple pour dialoguer ; ils n'ont pas besoin d'être dans la même cellule pour se parler. Par contre les personnages n'interagiront avec la plaque de glace que par *contact* : en entrant dans la cellule contenant la plaque de glace, ils pourront glisser.

Pour le moment, nous ne préoccuperons pas de définir le contenu de l'interface `Interactor` (uniquement nécessaire pour prendre en charge les interactions entre acteurs, ce que nous ferons à l'étape suivante). Codez l'interface `Interactable` permettant de modéliser une entité réceptives aux interactions. Un `Interactable` :

- occupe une liste de cellules : méthode `List<DiscreteCoordinates> getCurrentCells()` ;
- lorsqu'il occupe une cellule peut la rendre non traversable par d'autres (il peut empêcher d'autres `Interactable` d'investir la cellule qu'il occupe) : méthode `boolean takeCellSpace()`. Un `Interactable` pour lequel `boolean takeCellSpace()` retourne `true` sera dit « non traversable » dans la suite de l'énoncé (bien entendu le fait qu'il soit traversable ou pas peut dépendre de divers conditions et n'a pas besoin d'être

toujours vrai ou faux) ;

- indique avec une méthode booléenne s'il accepte les interactions *distantes* : méthode `boolean isViewInteractable()` ;
- et indique avec une méthode booléenne s'il accepte les interactions de *contact* : méthode `boolean isCellInteractable()`.

Vous considérerez qu'au niveau d'abstraction d'une `AreaEntity`, il n'est pas possible de définir concrètement les méthodes dictées par l'interface `Interactable`.

Par ailleurs, faites en sorte que `AreaBehavior.Cell` implémente `Interactable` pour indiquer que les cellules sont aussi réceptives aux interactions. Au niveau d'abstraction de `Cell`, seule la méthode `getCurrentCells` peut être redéfinie. Elle retournera une liste dont l'unique élément sera les coordonnées de la cellule.

L'interface `Interactable` est amenée à être un peu enrichie par la suite.

4.6.3 La classe `MovableAreaEntity`

Certains des nos acteurs sur grilles seront naturellement en mouvement (ils pourront se déplacer sur la grille). La classe abstraite `MovableAreaEntity`, dérivant de `AreaEntity` permet de modéliser ce type d'acteurs. Ses caractéristiques spécifiques seront principalement :

- le fait d'avoir une cellule cible vers laquelle se déplacer ;
- et la présence d'une méthode `move` permettant à l'acteur d'initier un déplacement linéaire depuis sa cellule principale vers une cellule cible.

Pour pouvoir avoir lieu, le déplacement doit être autorisé par l'`Area` et par chacune des cellules qui seraient quittées ou investies par l'entité lors de ce déplacement. Les méthodes permettant de savoir quelles cellules seront quittées ou atteintes par l'acteur s'il se déplace seront implémentées comme des méthodes protégées et finales :

```
List<DiscreteCoordinates> getLeavingCells();

List<DiscreteCoordinates> getEnteringCells();
```

Vous pourrez leur donner la définition simple suivante :

- les cellules quittées seront les cellules courantes (`getCurrentCells()`) ;
- les cellules investies seront toutes celles qui parmi les projections des cellules courantes dans la direction de l'acteur font partie de la grille. La projection d'une coordonnée dans la direction de l'acteur peut simplement se calculer comme suit : `coord.jump(getOrientation().toVector())`.

Ces implémentations simples peuvent s'avérer inadaptées lorsque plusieurs acteurs voisins se déplacent (chevauchement des cellules quittées/investies). Libre à vous par la suite d'affiner ces méthodes pour mieux répondre à vos besoins.

Par définition, le déplacement aura toujours lieu depuis la cellule principale courante jusqu'à une cellule adjacente à cette dernière, définie par l'orientation courante de l'entité. L'acteur mobile se déplace d'une case à la fois et pour s'assurer de ne jamais se retrouver entre deux cases, un déplacement attendra toujours la fin du précédent avant de commencer.

La méthode `update` d'une `MovableAreaEntity` appliquera donc l'algorithme simple suivant :

1. Si l'acteur est en cours de déplacement et si la cellule cible n'est pas encore atteinte alors progresser d'une étape vers la cellule cible, ce qui se traduit par une instruction telle que :

   ```
   Vector distance = getOrientation().toVector();
   distance = distance.mul(1.0f / framesForCurrentMove);
   setCurrentPosition(getPosition().add(distance);
   ```

2. sinon, réinitialiser les attributs relatifs aux déplacement (méthode `resetMotion` décrite plus bas).

`framesForCurrentMove` est le nombre de frames (étapes) choisie pour mettre en oeuvre la continuité du déplacement. Nous introduirons plus tard la possibilité d'assortir à chaque étape (frame) une représentation graphique différente, ce qui permettra d'animer le déplacement.

Les attributs d'une `MovableAreaEntity` (relatifs au déplacement) qui se dégagent de cette description sont donc naturellement :

```
/// Indicate if the actor is currently moving
  private boolean isMoving;
  /// Indicate how many frames the current move is supposed
     to take
  private int framesForCurrentMove;
  /// The target cell (i.e. where the mainCell will be after
     the motion)
  private DiscreteCoordinates targetMainCellCoordinates;
```

Le rôle de la méthode `protected boolean move(int framesForMove)` qu'il vous est demandé de compléter est de décider si un déplacement peut avoir lieu et si oui de l'initier. L'algorithme à appliquer est le suivant :

1. Si l'acteur n'est pas en cours de déplacement ou s'il a atteint sa cellule cible (un nouveau déplacement est alors possible), demander à son aire si elle permet de quitter les cellules données par `getLeavingCells()` et d'entrer dans les cellules données par `getEnteringCells()` ;

2. si l'aire ne le permet pas, le déplacement n'est pas possible et `move` retourne `false` ;

3. sinon, le déplacement est initié, ce qui concrètement veut dire :
   (a) attribuer la valeur `framesForMove` à `framesForCurrentMove` (cette valeur sera ramenée à 1 si `framesForMove` est en dessous de 1)
   (b) choisir la cellule cible (celle vers laquelle l'acteur regarde) :

```
Vector orientation = getOrientation().toVector();
targetMainCellCoordinates =
   getCurrentMainCellCoordinates().jump(orientation);
```

(c) et indiquer qu'un déplacement est désormais en cours. La méthode `move` retournera alors `true`.

Pour le moment il n'y a pas de conditions imposées par l'aire. Ajouter simplement un commentaire :

```
// TODO : add area conditions here
```

à l'endroit où il est prévu de le faire nous y reviendrons un peu plus tard.

Complétez le code de `MovableAreaEntity` conformément à la description qui vient de vous en être faite et à la coquille de code fourni. Les contraintes suivantes seront appliquées :

- Le changement d'orientation d'une `MovableAreaEntity` ne devra être possible que si elle n'est pas en cours de déplacement.
- La méthode `resetMotion` réinitialisera les attributs relatifs aux déplacement : `isMoving` devient `false`, `framesForCurrentMove` prend la valeur zéro et `targetMainCellCoordinates` prend la valeur de `getCurrentMainCellCoordinates()`.
- La méthode `resetMotion` sera naturellement invoquée à la construction d'une `MovableAreaEntity`.

4.7 L'aire et sa grille dictent leurs conditions

Il est temps maintenant de permettre à la grille et à l'aire d'imposer leurs conditions sur le placement et le déplacement des entités qui y prennent place. Pour le moment, l'ajout ou le retrait d'acteurs dans l'aire de jeu (`addActor`/`removeActor`) ne tient pas compte du *veto* potentiel de la grille. Or celle-ci doit normalement pouvoir s'opposer à l'ajout d'un acteur dans une cellule donnée. Par exemple, un acteur qui aurait une taille trop grande en nombre de cellules pour être placé à une position voulue (débordement de la grille) doit pouvoir être refusé par la grille. De même, pour le moment, la méthode `move` des `MovableAreaEntity` ne permet pas à l'aire ou à la grille d'exprimer des contraintes sur le déplacement (nous n'avons fait que le prévoir au travers d'un commentaire). Elle devrait typiquement au moins s'opposer à ce qu'une entité sorte de la grille. Il faut que nous commencions à nous intéresser de plus près au contenu des cellules et que nous retouchions un peu le code existant.

4.7.1 Cellules avec un contenu

Nous avons prévu de doter les cellules de `AreaBehavior` d'un attribut de type `Set<Interactable>`[6]. L'initialisation de cet attribut à un ensemble vide, dans le constructeur de la cellule, se fera au moyen de la tournure :

[6] le type `Set` garantit que l'ajout d'un élément déjà existant ne se fera pas à double, et ce, sans avoir besoin de prendre des mesures particulières

```
new HashSet<>()
```

La classe `Cell` sera enrichie d'une méthode privée `enter` permettant l'ajout d'un `Interactable` donné à cet ensemble et d'une méthode privée `leave` permettant le retrait d'un `Interactable` de cet ensemble. Afin de permettre à la grille d'exprimer un droit de regard sur les déplacements, vous doterez la classe `Cell` des méthodes protégées :

- `boolean canEnter(Interactable entity)` : retournant `true` si `entity` a le droit de s'ajouter au contenu de la cellule et `false` sinon ;
- `boolean canLeave(Interactable entity)` retournant `true` si `entity` a le droit de se soustraire au contenu de la cellule et `false` sinon.

Vous considérerez que ces méthodes ne peuvent être définies à ce niveau d'abstraction.

4.7.2 `Interactable` à l'écoute de la grille

Afin de permettre à la grille (`AreaBehavior`) de dicter ses décisions en matière de déplacement ainsi qu'en matière d'ajout ou de retrait dans la grille de jeu, dotez-la des méthodes supplémentaires suivantes :

- `public boolean canLeave(Interactable entity, List<DiscreteCoordinates> coordinates)` : retourne `true` si `entity` peut quitter les cellules de l'ensemble `coordinates` (chacune de ses cellules donne l'autorisation de le faire) et `false` sinon ;
- `public boolean canEnter(Interactable entity, List<DiscreteCoordinates> coordinates)` : retourne `true` si `entity` peut entrer dans les cellules de l'ensemble `coordinates` (chacune de ses cellules donne l'autorisation de le faire) et si chacune des coordonnées de coordinates est dans les limites de la grille. La valeur `false` sera retournée sinon ;
- `protected void leave(Interactable entity, List<DiscreteCoordinates> coordinates)` permet de supprimer `entity` de toutes les cellules de coordonnées `coordinates` ;
- `protected void enter(Interactable entity, List<DiscreteCoordinates> coordinates)` permet d'ajouter `entity` à toutes les cellules de coordonnées `coordinates`.

Nous mettons un droit d'accès protégé aux deux dernières méthodes afin que les accès intrusifs à la grille soient strictement limités aux classes de `game.areagame`. Les quatre méthodes ci-dessus définissent les traitements permettant à un `Interactable` d'être à l'écoute de la grille sur laquelle il se trouve.

4.7.3 Adaptation de `Area`

Au moment de l'ajout ou du retrait d'un acteur à un aire de jeu, la grille associée a désormais son mot à dire. Dotez la classe `Area` de deux nouvelles méthodes :

- ```
  public final boolean leaveAreaCells(Interactable entity,
      List<DiscreteCoordinates> coordinates)
  ```
  ```

qui teste si la grille associée à l'aire permet à `entity` de quitter les cellules de coordonnées de `coordinates`. Si oui, elle enregistre que ce retrait doit être fait et retourne `true`. Sinon, elle retournera `false`.

- ```
  public final boolean enterAreaCells(Interactable entity,
      List<DiscreteCoordinates> coordinates)
  ```

  qui teste si la grille associée à l'aire permet à `entity d'investir` les cellules de coordonnées de `coordinates`. Si oui, elle enregistre que cet ajout doit être fait et retourne `true`. Sinon, elle retournera false.

Nous verrons un peu plus bas ce que signifie *"enregistrer qu'un ajout ou retrait doit être fait"*.

Modifiez `addActor` de sorte à ce que les acteurs de type `Interactable` puissent investir les cellules qui leur ont été attribuées à la construction (ce seront celles retournées par `getCurrentCells()`). Une tentative échouée d'investir la grille (cette dernière posant son *veto*) sera considérée comme une situation d'erreur :

```
if(a instanceof Interactable)
errorOccured = errorOccured || !enterAreaCells(((Interactable)
   a), ((Interactable) a).getCurrentCells());
```

Nous admettons ici, par simplification, le test sur la catégorie `Interactable` sachant que dans notre conception tout acteur fera partie de trois catégories au plus pour `Area` : `Actor`, `Interactable` et `Interactor`. Nous ne sommes pas ici en train de faire un test de type sur une classe spécifique[7].

Modifiez `removeActor` de façon analogue.

Comme vu précédemment, l'`update` d'un acteur dans la boucle `update` de `Area` peut causer l'ajout ou la suppression d'un acteur. Ce dernier devra s'ajouter ou se supprimer dans les cellules de la grille associée. Or, l'ajout ou le retrait effectif de l'acteur en question ne se fera qu'au prochain `update` (lors de l'appel à `purgeRegistration`). Par conséquent, l'ajout ou le retrait effectif de l'acteur des cellules de la grille ne doit aussi se faire qu'à ce moment là. C'est pourquoi `leaveAreaCells` et `enterAreaCells` ne procèdent pas aux ajouts et retraits directement dans la grille mais doivent simplement mémoriser que ces opérations doivent être réalisées. Elles doivent l'être effectivement avant d'entamer le prochain `update` (ce qui se fera naturellement aussi dans `purgeRegistration`). Dans le même esprit que ce que nous avons fait pour l'enregistrement des acteurs, il vous est donc suggéré d'utiliser deux nouveaux attributs :

```
private Map<Interactable, List<DiscreteCoordinates>>
   interactablesToEnter;
private Map<Interactable, List<DiscreteCoordinates>>
   interactablesToLeave;
```

Pour ajouter une entrée à `interactablesToEnter` il suffit d'écrire :

```
interactablesToEnter.put(entity, coordinates);
```

[7] il y a moyen de faire mieux, mais vous conviendrez que c'est déjà assez compliqué comme ça !
```

(idem pour `interactableToLeave`)

Enfin la méthode `purgeRegistration` procédera aux opérations effectives :

- pour toute entrée `entry` de `interactablesToEnter`, faire en sorte que l'interactable lié à la clé de `entry`, entre dans les cellules données par la valeur de `entry` (voir l'annexe 9 sur les `Map`), puis vider `interactablesToEnter` ;
- procéder de façon analogue pour les cellules à quitter.

4.7.4 Adaptation de `MovableAreaEntity`

Pour permettre à la méthode `move` de tenir compte du *veto* potentiel de la grille, il vous est demandé de coder les conditions selon lesquelles l'aire permet au déplacement d'avoir lieu (vous avez en principe mis un commentaire à l'endroit où il était prévu de le faire). La condition devra s'exprimer comme suit : si l'aire permet à l'acteur d'investir les `getEnteringCells()` et de sortir des `getLeavingCells()`, alors le déplacement peut avoir lieu. Les méthodes `enterAreaCells()` et `leaveAreaCells()` de `Area` vous permettent maintenant naturellement de réaliser ces traitements. Procédez aux modifications ainsi suggérées.

4.8 Premier jeu de grille avec un personnage

Vous disposez maintenant de (presque) toute la logistique de base permettant de coder des jeux sur grille, acteurs compris (Ouf !). Pour voir ceci concrètement à l'oeuvre, vous allez placer un personnage dans `Demo2`.

Afin que la grille `Demo2Behavior` puis dicter ses conditions aux acteurs qui s'y trouvent, les cellules assorties à `Demo2Cell` seront caractérisées par le fait qu'elles :

- ne permettent pas d'entrer dans une cellule dont le type est `NULL` ou correspond à un mur ;
- acceptent les interactions de contact
- n'acceptent pas les interaction à distance
- peuvent toujours être quittées.

4.8.1 Acteurs spécifique

Nous souhaitons maintenant coder un acteur spécifique capable de transiter de l'aire `Room0` à l'aire `Room1` en passant des portes. Codez dans le paquetage `game.enigme.actor.demo2`, la classe `Demo2Player` héritant de `MovableAreaEntity`. Cet acteur acceptera tout type d'interaction, il sera non traversable et capable de passer des portes.

Il sera à ce titre doté d'un attribut spécifique indiquant s'il est en train de passer une porte (ce qui va conditionner son comportement).

Il sera aussi doté de méthodes lui permettant de :

- rentrer dans une aire donnée en s'y plaçant à une position donnée :

  ```
  public void enterArea(Area area, DiscreteCoordinates
     position)
  ```

  L'algorithme consistera à :

  1. s'y enregistrer comme acteur (en prenant les dispositions nécessaires pour indiquer son aire d'appartenance) ;
  2. mettre à jour sa position ;
  3. et se mettre en situation d'immobilité (`resetMotion`).

- de quitter l'aire à laquelle il appartient (s'y désenregistrer)
- et d'agir sur son attribut spécifique (en le mettant à vrai ou faux).

Le constructeur de `Demo2Player` aura l'entête suivante :

```
public Demo2Player(Area area, Orientation orientation,
   DiscreteCoordinates coordinates)
```

Comme représentation graphique, il utilisera une `Sprite`.

Les jeux de type *Game Boy* simulent souvent une vue aérienne dite en vue du dessus. Pour respecter l'effet désiré qui dicte qu'être en dessous implique d'être devant, il faut que les images soient dessinées de haut en bas pour ne pas créer de contradiction. Les `Sprite` sont de simples images graphiques dont la profondeur dépend de la coordonnée `y` de l'entité à laquelle ils se rapportent. Les `Sprite` permettent aussi de préciser dans leur constructeur à quels objets ils s'attachent (voir au besoin le code de cette classe).

La manipulation d'un `Sprite` est autrement analogue à celle d'une `ImageGraphics`. Le `Sprite` associé à `Demo2Player` peut être créé au moyen de la tournure :

```
new Sprite("ghost.1", 1, 1.f, this);
```

Pour pouvoir être instancié, un `Demo2Player` devra contenir des définitions concrètes des méthodes imposées par `Interactable` et `MovingAreaEntity`.

```
@Override
public List<DiscreteCoordinates> getCurrentCells() {
    return
       Collections.singletonList(getCurrentMainCellCoordinates());
}
```

Par simplification on considère ici que l'acteur n'occupe que sa cellule principale.

La méthode `update` de `Demo2Player` implémente l'algorithme suivant :

- si le bouton correspondant au bouton `Keyboard.LEFT` est enfoncé (`isDown`) alors si l'acteur est orienté à gauche, on initie le déplacement vers la gauche (appel à `move`).
- Sinon, on oriente l'acteur vers la gauche.

Le nombre de « frames » utilisées par `move` pourra être défini comme une constante statique :

```
/// Animation duration in frame number
private final static int ANIMATION_DURATION = 8;
```

On procédera de façon analogue pour toutes les autres orientations.

Le constructeur de `Demo2Player` :

```
public Demo2Player(Area area, DiscreteCoordinates coordinates)
```

initialisera l'orientation à `Orientation.DOWN`.

`Demo2Player` devra évidemment avoir une méthode de dessin spécifique, laquelle se contentera de dessiner le `Sprite` associé.

Enfin, la méthode `move` sera redéfinie de sorte à ce que si une cellule entrante est une porte, il soit indiqué que l'acteur est en train de passer une porte.

Vous veillerez à garantir que seuls les acteurs des jeux de grilles ait accès à l'aire à laquelle ils appartiennent.

4.8.2 `Demo2` avec un personnage

Compléter `Demo2` de sorte à ce que ce jeu soit caractérisé par un personnage de type `Demo2Player`. Le personnage sera créé au démarrage du jeu, il sera enregistré dans l'aire courante et c'est sur lui que sera centrée la caméra. Sa méthode `update` implémentera le fait que si le joueur passe une porte alors il bascule vers l'autre aire possible (s'il était dans `Room0` il passe dans `Room1` et vice-versa).

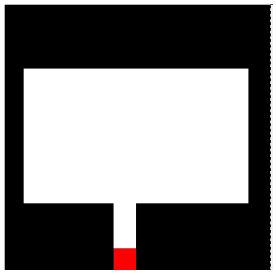
Vous utiliserez `(5,5)` comme coordonnées de départ dans `Room0` et `(5,2)` dans `Room1` et ce sont ces mêmes coordonnées qui seront utilisées comme coordonnées de départ à chaque fois que l'acteur rebascule vers ces aires. Vous pourrez utiliser 22 comme facteur d'échelle et il fait sens que cette valeur soit une constante statique finales spécifique au jeu, c'est à dire `Demo2`.

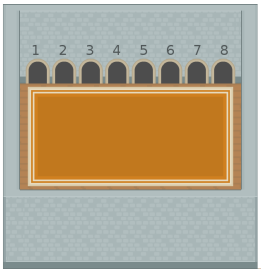
4.9 Validation de l'étape 2

Pour valider cette étape, vous vérifierez que `Demo2Player` :

1. peut se déplacer sur toute la surface des aires de jeu sans sortir de la grille ;
2. ne peut pas marcher sur les zones d'obstacles (mur ou `NULL`) ;


Room0 : 8 cellules « porte »


Room1 : 1 cellule « porte »



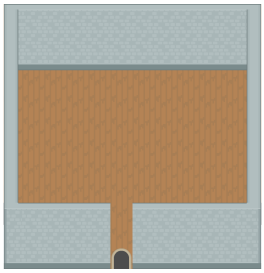


FIG. 6 : Grille de comportement des deux aires de `Demo2`

3. est bien suivi par la caméra lors de ses déplacements ;
4. peut marcher sur les portes ;
5. peut transiter correctement de l'aire `Room0` à l'aire `Room1` et vice-versa en passant par des cellules répertoriées comme des portes (voir la figure 6).

Le jeu `Demo2` dont le comportement est décrit ci-dessus est à rendre à la fin du projet.

5 Interactions entre acteurs (étape 3)

Dans notre jeu précédent, la couleur des pixels de l'image associée à la grille indique les endroits correspondant à un visuel de porte. Il n'est pas certain que nous souhaitions forcément interpréter toutes les cellules correspondant à un pixel rouge comme des portes dans nos jeux. De ce fait, il serait préférable de plutôt créer un acteur `Door` à placer (en général) sur les zones rouges (mais pas forcément toutes). De plus, le test indirect sur la couleur du pixel dans l'image associée à la grille n'est pas satisfaisant (car trop spécifique). Il s'agit donc maintenant d'enrichir notre architecture pour faire en sorte que le personnage puisse *interagir avec un autre acteur* (ici l'acteur `Door`). C'est l'objet de cette étape du projet. Nous allons créer une variante `Enigme` du jeu précédent où interviendront deux nouveaux acteurs réceptifs aux interactions :

- des portes qui permettront au personnage de les traverser et qui indiqueront vers quelle aire elle permettent de transiter. Il s'agira d'une interaction de contact (le personnage

doit être dans une cellule contenant un acteur « porte » pour pouvoir traverser cette « porte »).

- des pommes qui « permettront » au personnage de les ramasser. Il s'agira d'une interaction à distance (le personnage peut attraper la pomme se trouvant sur une case voisine).

Le personnage devra jouer un rôle plus actif en exprimant s'il souhaite une interaction ou pas (il n'est par exemple pas forcé de ramasser la pomme). Il s'agira donc d'une entité qui *demande* une interaction, c'est à dire un `Interactor`.

5.1 Préparation du jeu `Enigme`

Préparez un jeu `Enigme` en vous inspirant de `Demo2`. Ce dernier sera constitué (au moins) :

- de la classe `Enigme`, équivalente à `Demo2` et que vous placerez dans le paquetage `game.enigme` ;
- de la classe `EnigmeArea`, équivalente à `Demo2Area`, à placer dans le paquetage `game.enigme.area` ;
- des classes `LevelSelector`, `Level1`, `Level2` et `Level3` héritant de `EnigmeArea`, à placer dans le paquetage `game.enigme.area` (elles auront pour titre la chaîne de caractère homonyme, c'est à dire "LevelSelector" pour l'aire `LevelSelector` par exemple) ;
- de la classe `EnigmeBehavior` analogue à `Demo2Behavior` à placer dans `game.enigme` et qui contiendrait une classe publique `EnigmeCell` équivalente de `Demo2Cell` ;
- et de la classe `EnigmePlayer` quasi identique à `Demo2Player` mais que vous placerez dans `game.enigme.actor`.

Le jeu `Enigme` sera constitué des aires `LevelSelector`, `Level1`, `Level2` et `Level3`. Il démarrera sur l'aire `LevelSelector` en y enregistrant un `EnigmePlayer` sur lequel sera centrée la vue tout au long du jeu.

La méthode `update` de `Enigme` se contentera pour le moment d'invoquer le `update` de la super-classe (il faudra la réécrire pour qu'elle s'adapte aux nouveaux contenus).

La méthode `move` n'aura plus besoin d'être redéfinie spécifiquement dans `EnigmePlayer`. **Vous la supprimerez donc de cette classe**.

Codez ensuite les deux nouveaux acteurs décrits ci-dessous. Ces acteurs seront communs à tous les jeux d'énigmes à venir, vous pourrez donc les coder dans le paquetage `game.enigme.actor`.

5.1.1 l'acteur `Apple`

Il s'agira d'un acteur de type `AreaEntity` se dessinant sous la forme d'une pomme. Prenez *`"apple.1"`* par exemple comme nom du `Sprite` associé. Il aura la particularité de pouvoir être *collecté*. Dès qu'il l'est, il disparaît de l'aire de jeu. Il s'agit d'un acteur non traversable qui n'accepte que les interactions à distance. Ses cellules courantes (retour de

`getCurrentCells()`) se réduiront à l'ensemble contenant uniquement sa cellule principale (pareil que pour `EnigmePlayer`). Son constructeur sera également analogue à celui de `EnigmePlayer`.

5.1.2 l'acteur `Door`

Une porte est un acteur de type `AreaEntity` qui permet de transiter vers une aire de destination.

Cet acteur sera caractérisé par :

- le nom de l'aire vers laquelle il permet de transiter (une chaîne de caractères) ;
- les coordonnées d'arrivée dans l'aire de destination ;

Elle occupera à la construction *un ensemble de cellules*. Le constructeur d'une `Door` prendra en paramètre : l'aire à laquelle elle appartient, le nom de l'aire de destination, les coordonnées d'arrivée dans l'aire de destination, une orientation, la position de sa cellule principale et la liste des coordonnées des cellules qu'elle occupe en plus de sa cellule principale (vous pouvez utiliser une ellipse pour exprimer ce dernier paramètre). Il s'agit d'un acteur traversable qui pour le moment n'accepte que les interactions de contact.

La méthode `update` de `Enigme` peut maintenant être mieux décrite. Elle devra implémenter l'algorithme suivant :

Si le personnage est en train de passer une porte il doit quitter l'aire courante et entrer dans l'aire de destination de la porte passée qui devient alors l'aire courante du jeu.

Indication : dotez le personnage `EnigmePlayer` d'une méthode void `setIsPassingDoor(Door door)` lui permettant d'indiquer qu'il est en train de passer une porte et de mémoriser la porte qu'il vient de passer ainsi que d'une méthode `Door passedDoor()` retournant la dernière porte passée. Dotez également la classe `Door` des méthodes vous semblant nécessaires à la mise en oeuvre de la méthode `update` de `Enigme`.

Il est temps maintenant de s'intéresser à la manière dont le personnage entrera en interaction avec les pommes et les portes.

5.2 Les Interactor entrent en scène

Complétez l'interface `Interactor` permettant de modéliser un objet :

- qui occupe une liste de cellules et est donc doté d'une méthode `List<DiscreteCoordinates> getCurrentCells()` retournant les coordonnées de ces cellules ;
- qui a un certain nombre de cellules dans son champs de vision et est donc doté d'une méthode `List<DiscreteCoordinates> getFieldOfViewCells()` retournant les coordonnées des cellules de son champs de vision ;

- qui indique avec une méthode booléenne `boolean wantsCellInteraction()` s'il demande une interaction de *contact* ;
- et qui indique avec une autre méthode `boolean wantsViewInteraction()` s'il demande une interaction *distante*.

Jusqu'ici la méthode `update` d'une aire de jeu (`Area`), ne se préoccupe pas de mettre en place les interactions entre les acteurs qui y sont présents. L'idée est maintenant de compléter cette méthode `update` de sorte à ce qu'après la boucle des mises à jour des acteurs :

```
for (Actor actor : actors) {
  actor.update(deltaTime);
}
```

ait lieu une gestion des interactions ressemblant à ceci :

```
for (Interactor interactor : interactors) {
   if (interactor.wantsCellInteraction()) {
     // demander au gestionnaire de la grille (AreaBehavior)
     //de mettre en place les interactions de contact
   }
   if (interactor.wantsViewInteraction()) {
     // demander au gestionnaire de la grille de mettre en place
     // les interactions distantes
   }
}
```

La grille `AreaBehavior` étant le gestionnaire de tous les mécanismes qui y prennent place, c'est en effet à elle de fournir les méthodes gérant l'interaction à proprement parler.

Ceci soulève deux nouvelles problématiques : comment se définit/construit l'ensemble des interacteurs ? (la variable `interactors` dans le code ci-dessus) et comment la grille intervient pour gérer les interactions ?

5.2.1 Ensemble d'`Interactors`

Tout acteur de type `AreaEntity` est susceptible d'être réceptif à une interaction. C'est pourquoi la classe `AreaEntity` implémente déjà l'interface `Interactable`. Par contre, les classes qui implémenteront `Interactor` seront plutôt proches des objets concrets (le fait de décider si un objet est désireux d'entrer en interaction se fait plutôt de façon spécifique). Par exemple le `EnigmePlayer` est un candidat naturel pour être un `Interactor`.

Les acteurs jouant le rôle d'`Interactor` ont un rôle spécial à remplir. Il faut donc être capable de les distinguer des autres. Il fait donc sens de prévoir dans `Area` un attribut `interactors` consignant tous les acteurs de types `Interactor`. Une liste chaînée est une option raisonnable pour le type de cet attribut. `addActor` aura pour rôle d'alimenter l'attribut interactors (et donc de catégoriser les acteurs selon qu'ils soient `Interactor` ou pas). Un acteur de type `Interactor` sera consigné aussi bien dans la liste des `actors` que dans la liste `interactors`.

Il n'est pas rare en programmation de référencer le même objet depuis plusieurs endroits. Ceci permet de manipuler les objets en question selon différents points de vue : un `Interactor` doit pouvoir être vu comme un `Actor` pour qu'on puisse lui appliquer sa méthode `update` ou comme un `Interactor` pour qu'on puisse le faire interagir avec les autres acteurs.

Par simplification, vous procéderez comme pour les `Interactable` :

```
if(a instanceof Interactor)
    errorOccured = errorOccured ||
       !interactors.add((Interactor) a);
```

La méthode `removeActor` sera également adaptée en conséquence.

5.2.2 `Interactor` à l'écoute de la grille

Pour reprendre l'ébauche faite précédemment, nous nous intéressons maintenant à compléter le code de la méthode `update` de `Area` de sorte à ce que la boucle des mises à jour soit suivie par :

```
for (Interactor interactor  : interactors) {
      if (interactor.wantsCellInteraction()) {
          // demander à la grille de mettre en place les
             interactions de contact
      }
      if (interactor.wantsViewInteraction()) {
         // demander à la grille de mettre en place les
            interaction distantes
      }
}
```

Les commentaires vont clairement se traduire par des méthodes. Comme l'idée est que ce soit la grille qui soit en charge de mettre en place les mécanismes d'intraction, il s'agira de deux nouvelles méthodes de `AreaBehavior` :

- `public void cellInteractionOf(Interactor interactor)` : qui gérera toutes les interactions de contact entre `interactor` et les `Interactable` aux mêmes positions que celles qu'il occupe. Cette méthode va parcourir toutes les cellules aux positions `interactor.getCurrentCells()` et leur appliquer une méthode `cellInteractionOf(interactor)` spécifique aux `Cell`.
- `public void viewInteractionOf(Interactor interactor)` : qui gérera toutes les interactions à distance entre `interactor` et les `Interactable` de son champs de vision. Cette méthode va parcourir toutes les cellules aux positions `interactor.getFieldOfViewsCells()` et leur appliquer une méthode `viewInteractionOf(interactor)`spécifique aux `Cell`.

Ces deux méthodes permettent à l'`Interactor` d'être à l'écoute de la grille. Elles exigent l'ajout des méthodes suivantes à `AreaBehavior.Cell` :

- `private void cellInteractionOf(Interactor interactor)`
- `private void viewInteractionOf(Interactor interactor)`

Voici comment nous vous proposons de coder la première de ces méthodes pour commencer :

```
private void cellInteractionOf(Interactor interactor){
            for(Interactable interactable  : entities){
                if(interactable.isCellInteractable())
                    interactor.interactWith(interactable);
            }
}
```

où `entities` représente l'ensemble de `Interactable` répertoriés dans la cellule. Vous procéderez de façon analogue pour la seconde méthode.

5.2.3 Interactions génériques

Nous voici au coeur du sujet, l'interface `Interactor` doit être dotée d'une méthode

```
void interactWith(Interactable other);
```

et la question se pose de comment la coder concrètement.

Notre personnage dans `Enigme` va typiquement être un `Interactor` ; c'est-à-dire une entité qui invite à des interactions. Comment pourrait-on a priori définir la méthode `void interactWith(Interactable other)` dans la classe de `EnigmePlayer` de sorte à lui permettre d'interagir avec les acteurs `Door` et `Apple` ?

La façon triviale de procéder serait de recourir à des *tests de type* :

```
void interactWith(Interactable other){
  if (other instanceof Apple()) //...
  if (other instanceof Door) //..
}
```

ce qui est très *ad hoc* et peu extensible. En fait lorsque l'on programme un jeu, tout `Interactor` peut potentiellement interagir avec tous les autres acteurs possibles et tous les cas doivent être envisagés. Un schéma de conception est utilisé de façon classique dans ce genre de situations où il y a des actions à effectuer sur toutes sortes d'objets qui n'ont pas forcément de liens entre eux. Il consiste à déléguer la gestion de ces actions à une classe externe qu'on appellerait ici le gestionnaire d'interaction du personnage[8] :

```
class EnigmePlayerHandler {
  public void interactWith(Door door) {
   // fait en sorte que la porte soit passée par l'acteur
```

[8] Communément appelé le patron de conception « visiteur » (« visitor pattern »)

```
  }
  public void interactWith(Apple apple){
    // fait en sorte que la pomme soit ramassée
  }
}
```

Ce gestionnaire est spécifique à `EnigmePlayer`, il serait dans notre cas codé comme classe privée interne de cette classe.

Le classe `EnigmePlayer` aurait comme attribut son gestionnaire d'interaction :

```
private final EnigmePlayerHandler handler;
```

et une méthode générique :

```
public void interactWith(Interactable other) {
        other.acceptInteraction(handler);
}
```

Chaque `Interactable` devrait alors offrir une méthode indiquant qu'il accepte de faire partie d'une interaction gérée par le gestionnaire du personnage. Par exemple dans `Apple` on aurait :

```
public void acceptInteraction(EnigmePlayerHandler v) {
  // fait en sorte que le gestionnaire d'interaction du
     personnage gère l'interaction avec Apple
       v.interactWith(this);
}
```

Cette solution offre l'avantage de pouvoir coder une méthode unique très générale dans les `Interactor`, la méthode `interactWith(Interactable)`.

Un seul bémol encore, l'argument de `acceptInteraction` dans `Apple` est encore trop spécifique : il faudrait ajouter une méthode `acceptInteraction` avec les gestionnaires de chaque `Interactor` possible (ici nous n'avons qu'un seul `Interactor`, mais rien n'empêche d'en introduire d'autres).

L'idée est donc de plutôt de recourir au schéma suivant :

On fait donc hériter `EnigmePlayerHandler` de gestionnaires d'interactions plus généraux. De cette façon, l'interface `Interactable` doit offrir comme unique méthode supplémentaire :

```
public void acceptInteraction(AreaInteractionVisitor v) {
        // avec une définition par défaut simple
}
```

La méthode `acceptInteraction` de `Apple` (ou `Door`) s'écrirait alors simplement :

```
public void acceptInteraction(AreaInteractionVisitor v) {
        ((EnigmeInteractionVisitor)v).interactWith(this);
}
```

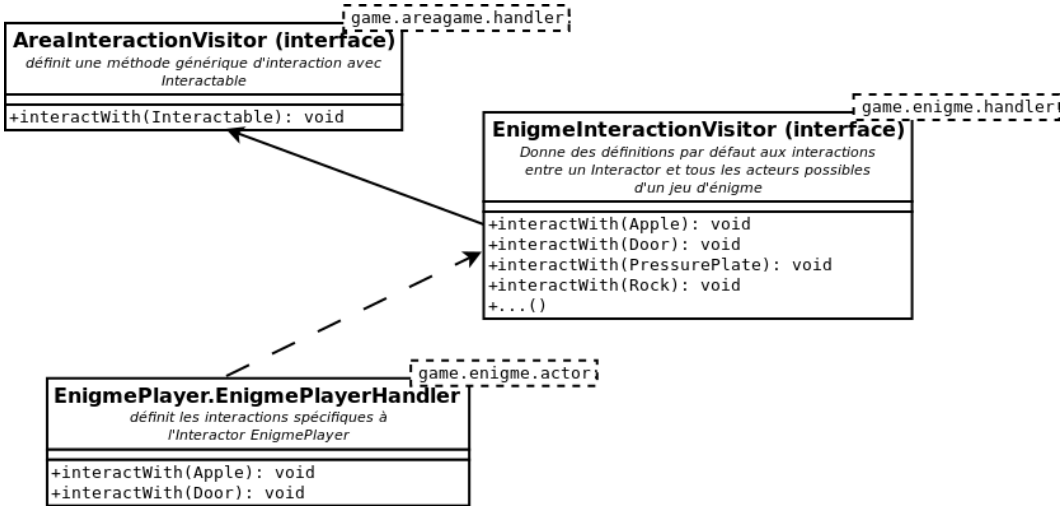


FIG. 7 : Schéma de classes pour la mise en place des interactions

Il y a certes une conversion à effectuer, mais une seule. Cette conversion permet de déléguer la gestion des interactions au gestionnaire spécifique au jeu auquel `Apple` participe.

On indique ainsi que la pomme accepte de voir ses interactions gérées par le gestionnaire d'interaction spécifique aux jeux d'énigme. Ce gestionnaire prévoit que tout `Interactor` peut avoir des interactions avec chaque acteur du paquetage `enigme`. L'ajout d'un nouvel acteur dans les jeux d'énigme implique de retoucher le gestionnaire `EnigmeInteractionVisitor` et uniquement les `Interactor` qui souhaiteraient une interaction avec ce nouvel acteur. Les autres acteurs ne subissent par contre aucune modification dûe à l'introduction de ce nouvel acteur (ce qui n'était pas le cas avec les autres tentatives vers la solution évoquées plus haut).

Pour mettre tout ceci en place, commencez par ajouter à l'interface `Interactable` la méthode :

```
void acceptInteraction(AreaInteractionVisitor v);
```

L'interface `AreaInteractionVisitor` modélisant un gestionnaire d'interaction générique et pour lequel on peut imaginer une implémentation par défaut est fournie dans le paquetage `areagame.handler`. Dans le paquetage `game.enigme.handler` codez une interface plus spécifique
`EnigmeInteractionVisitor` héritant de `AreaInteractionVisitor` et donnant une définition par défaut à *toutes* les méthodes d'interaction possibles dans un jeu d'énigme, par exemple pour l'interaction avec la pomme :

```
/**
  * Simulates an interaction between Interactors and Apple
     in enigme
  * @param apple (Apple), not null
  */
 default void interactWith(Apple apple){
     // by default the interaction is empty
 }
```

ou avec une `EnigmeCell` :

```
default void interactWith(EnigmeBehavior.EnigmeCell cell){
    // by default the interaction is empty
}
```

Par défaut, ces interactions ne feront donc rien et devront en principe être codée pour tout acteur ou cellule intervenant dans un jeu d'énigme.

Question 1

Que se passe-t-il si on oublie de donner une définition par défaut à l'interaction avec des `Interactable` impliqués dans un jeu (par exemple l'interaction de `EnigmePlayer` avec lui même en tant que `Interactable` ?

Codez enfin dans `EnigmePlayer` la classe imbriquée `EnigmePlayerHandler` spécialisant `EnigmeInteractionVisitor` et y redéfinissant *seulement* les méthodes pour lesquelles il se passe vraiment quelque chose durant les interactions, typiquement :

```
/**
 * Specific interaction handler for an EnigmePlayer
 */
private class EnigmePlayerHandler implements
   EnigmeInteractionVisitor {

    @Override
    public void interactWith(Door door) {
      // gère ce qui se passe lorsque le personnage passe
         les porte
    }

    @Override
    public void interactWith(Apple apple){
       //gère ce qui se passe lorsque le personnage
          interagit avec une pomme
    }
}
```

Indications :

- pensez à la méthode `setIsPassingDoor` de `EnigmePlayer` pour gérer ce qui se passe lorsque le personnage passe la porte ;
- la classe `Apple` préfigure tout sorte d'objets qui peuvent être ramassés ; il peut donc être une bonne idée de regrouper les éléments communs à ce type d'objets dans une super-classe (`Collectable` ou `Pickup` par exemple) ;
- les cellules ont été conçues comme des `Interactable` il est donc nécessaire de compléter la méthode désormais manquante dans `EnigmeBehavior.EnigmeCell`.

Question 2

Que faire si on veut coder un autre `Interactor` qui a un autre rapport avec la pomme (par exemple il ne la ramasse pas mais l'empoisonne) ?

5.3 Enigme

Complétez et modifiez le jeu `Enigme` de sorte à ce que :

- le joueur puisse indiquer au moyen de la touche 'L' qu'il veut une interaction à distance ;
- il puisse ramasser une pomme dans son champ de vision (en actionnant la touche 'L')
- il puisse à nouveau passer les portes mais sans faire de tests spécifique sur la couleur des cellules de la grille (c'est-à-dire uniquement en interagissant avec les acteurs `Door`. Passer une porte lui permettra d'être dans l'aire destination de cette dernière (de l'autre côté de la porte).

Le joueur voudra systématiquement toutes les interactions de contact. Il acceptera d'être lui même l'objet d'interaction par contact ou a distance et il sera non traversable (on ne peut pas lui marcher dessus !). Vous aurez pour cela à coder le gestionnaire d'interaction `EnigmePlayerHandler` conformément à la description ci-dessus et à modifier la méthode `update` de `Enigme`. Le champs de vision du joueur sera la cellule en face de lui.

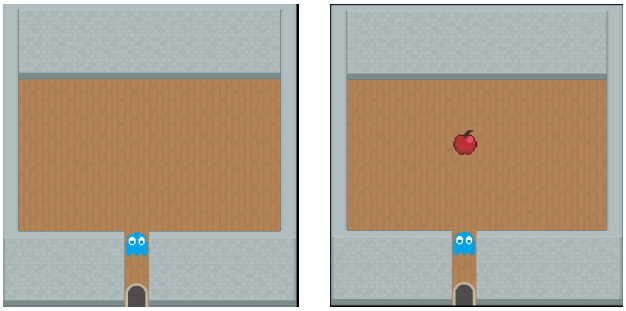
Vous aurez également à compléter les sous-classes de `EnigmeArea` :

- dans `LevelSelector` : vous placerez des portes en (i,7) (i allant de 1 à 8). chaque porte i occupera uniquement sa position principale (i,7). La porte 1 permet d'accéder à `Level1` et la porte 2 à `Level2`. Les autres portes ne donnent accès à aucune aire pour le moment (indiquez simplement la chaîne vide en guise d'identifiant de l'aire destination). La coordonnées d'arrivée pour les portes 1 et 2 sera (5,1).
- dans `Level1` : placez une porte en (5,0) qui permet de retourner sur `LevelSelector` en position (1,6) ;
- dans `Level2` : placez une porte en (5,0) qui permet de retourner sur `LevelSelector` en position (2,6) et placez une pomme en (5, 6) ;

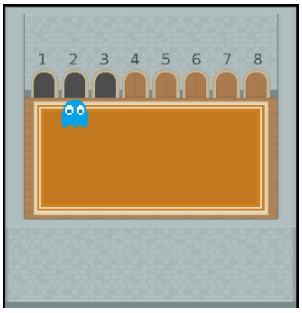
5.4 Validation de l'étape 3

Pour valider cette étape, vous vérifierez que `EnigmePlayer` :

1. ne peut pas marcher sur la pomme ;
2. peut marcher sur les portes ;
3. peut interagir à distance avec la pomme au moyen de la touche 'L' lorsqu'il est dans une case immédiatement voisine seulement. Il ramasse alors la pomme qui doit alors disparaître de la scène.



Aire Level1 Aire Level2



Aire LevelSelector

FIG. 8 : Emprunter la porte 1 permet de se rendre dans `Level1`, la porte 2 dans `Level2`. Sortir de `Level1` ou `Level2` nous ramène dans `LevelSelector`
.

4. peut franchir la porte 1 dans `LevelSelector` pour arriver dans la chambre vide `Level1` et retourner dans `LevelSelector` en empruntant la porte dans l'autre sens ;
5. peut franchir la porte 2 dans `LevelSelector` pour arriver dans la chambre `Level2` contenant une pomme et retourner dans `LevelSelector` en empruntant la porte dans l'autre sens ;
6. que les autres portes le "ramènent" instantanément au `LevelSelector`.

Attention : le fantôme doit toujours s'orienter proprement pour pouvoir interagir. N'oubliez pas de le faire avant de demander une interaction (même si le visuel ne suit pas).

Le jeu `Enigme` dont le comportement est décrit ci-dessus est à rendre à la fin du projet.

Question 3

La logistique que vous venez de mettre en place peut sembler *a priori* inutilement complexe. L'avantage qu'elle offre est qu'elle modélise de façon très générale les besoins inhérents à de nombreux jeux où des acteurs se déplaçent sur une grille et interagissent soit entre eux soit avec le contenu de la grille. Comment pourriez-vous en tirer parti pour mettre en oeuvre un jeu de Pacman par exemple ?

6 « Puzzles et énigmes » (étape 4)

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 ramasser des clés, allumer des torches et activer des leviers dans un certain ordre pour pouvoir ouvrir des portes ou libérer des passages obstrués par des obstacles, en vue de passer à un niveau de jeu supérieur.

6.1 Les signaux

Imaginons que l'ouverture d'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.

L'interface `Signal` est fournie dans le paquetage `signal`. Elle modélise très simplement un signal comme une entité dotée d'une intensité (une valeur de type `float` comprise entre 0.0 et 1.0).

Tout objet, acteur ou non, implémentant l'interface `Signal` représente un signal dont la valeur de l'intensité pourra être utilisée, de diverses manières, pour prendre des décisions. Consultez le code de cette interface telle que fournie dans le paquetage `signal`.

Afin que l'ouverture d'une porte soit conditionnée par l'obtention d'une clé, il est possible de procéder comme suit :

- Coder un acteur `Key` (clé) qui se *comporte comme* un `Signal`.
- Lier l'intensité du signal de `Key` au fait que la clé soit ramassée ou pas (par exemple 0.0 pour exprimer que la clé n'est pas ramassée et 1.0 pour le fait qu'elle l'est).
- Enfin, créer une catégorie de porte sensibles aux signaux et qui serait dotée d'un attribut `signal` de type `Signal`. Pour signifier que la porte n'est sensible aux interactions que si le signal associé est actif on aurait alors dans cette nouvelle catégorie de porte quelque chose comme :

```
@Override
    public boolean isCellInteractable() {
      // return true if signal has 1.0 as intensity
      // return false if signal has 0.0 as intensity
    }
```

Voilà pour l'idée générale. Cela soulève maintenant quelques questions pour la mise en oeuvre concrète : comment jouer sur l'intensité du signal de la clé ? comment intervient le temps dans le calcul de l'intensité ? est-il pertinent que ce soit dans ce cas un signal continu (que l'intensité puisse théoriquement avoir toute valeur comprise ente 0.0 et 1.0) ? et que faire si l'on veut que le signal associé à la porte soit plus complexe (par exemple que l'ouverture de la porte soit conditionnée par le fait que plusieurs clés soient ramassées et une torche allumée) ?

Le but de cette partie du projet est de modéliser plus finement les signaux (pour notamment avoir :

- des signaux logiques de type activé/désactivé, ce qui serait plus adapté pour modéliser le fait qu'une clé soit ramassée ou pas ;
- des combinaisons de signaux (dont on créera plusieurs déclinaisons concrètes) ;
- et d'implémenter des entités dépendantes de signaux.

Ceci permettra notamment de répondre aux questions soulevées par notre exemple.

Commencez par affiner la modélisation des signaux selon ce qui est décrit ci-dessous.

6.2 Signal logique

Créez un sous paquetage `logic` dans le paquetage `signal`. Implémentez-y une nouvelle une nouvelle interface `Logic` représentant des signaux binaire (on/off ou activé/désactivé) et indépendants du temps. `Logic` dérivera de `Signal` et contiendra les méthodes suivantes :

- `boolean isOn()` dont la définition concrète dépend du type spécifique de signal logique implémenté ;
- une surcharge `float getIntensity()` qui a par défaut la définition par défaut suivante : elle retourne `1.0f` si `isOn()` retourne `true` et `0.0f` sinon ;
- une redéfinition de la méthode héritée de `Signal` avec pour définition par défaut :

```
@Override
   default float getIntensity(float t) {
       return getIntensity();
   }
```

L'interface `Logic` offrira aussi deux constantes de type `Logic` (oui Java permet les définitions récursives !). La constante `TRUE` et la constante `FALSE`. Voici comme il faudrait définir la constante `TRUE` :

```
Logic TRUE = new Logic() {
     @Override
     public boolean isOn() {
         return true;
     }
};
```

Que veut dire cette tournure ? et a t'on le droit d'instancier une interface ??

Ce code signifie que l'on crée une classe anonyme (sans nom) où est redéfinie la méthode `isOn`. `TRUE` est une instance de cette classe (et non pas de l'interface !). Procédez de façon analogue pour définir la constante `FALSE`. Ainsi `Logic.TRUE` représente un signal toujours activé (qui peut être affecté à une variable de type `Logic`) et `Logic.FALSE` représente une signal toujours désactivé.

Implémentez ensuite dans le paquetage `logic` une classe abstraite `LogicSignal` implémentant `Logic`. Cette classe redéfinira simplement les deux méthodes `getIntensity`. La classe `LogicSignal` offrira les mêmes implémentations que celles par défaut mais en les finalisant.

Question 4

En quoi la fait de finaliser ces méthodes est-il utile ?

6.3 Combinaison de signaux

Un signal `LogicSignal` est un signal binaire simple (activé/désactivé). Produire des signaux simples n'est cependant pas suffisant pour définir des logiques intéressantes. Il est aussi nécessaire de les combiner. Pour cela, vous allez créer plusieurs classes représentant diverses opérations de base. Comme vous le verrez au prochain semestre, il s'agit de portes (gate) logiques (d'où le nom de la super classe !).

6.3.1 Négation

L'opération la plus simple est probablement la négation, qui inverse un signal reçu. Codez le signal `Not` qui *est un* `LogicSignal` caractérisé par le signal `s` dont il est la négation (un `Logic`). Dotez-le d'un constructeur prenant en paramètre ce signal. `Not` redéfinira la méthode `boolean isOn()` de sorte à ce qu'elle retourne `true` si `s` est différent de `null` et est désactivé (à tester avec la méthode `isOn()` de `s` bien sûr) et `false` sinon.

6.3.2 Combinaison de signaux

Supposons que l'ouverture d'une porte nécessite le fait que plusieurs clés soit ramassées. Il nous faut pour modéliser cette situation être capable de mettre en oeuvre des conjonctions de signaux.

Codez pour cela le signal `And`, avec deux signaux (`Logic`) en attributs. `And` redéfinit `boolean isOn()` de sorte à ce que cette méthode retourne `true` si les deux signaux sont actifs différent de null et tous les deux activés, et `false` sinon.

Ainsi, si trois clés `c1`, `c2` et `c3` sont requises, il suffira de les combiner avec deux portes `And`

```
// signal = c1 && c2 && c3
Logic signal = new And(new And(t1, t2), t3);
```

Créez ensuite selon le même modèle :

- Le signal `Or` qui dépend de deux autres signaux. `Or` est activé si les deux signaux qu'il reçoit en entrée sont différents de `null` et si au moins un des deux signaux est activé. Il est déactivé autrement.
- Le signal `MultipleAnd` qui généralise `And` à un nombre quelconque de signaux d'entrée.
- le signal `LogicNumber` qui permet d'interpréter un ensemble de signaux comme une puissance de 2. `LogicNumber` est caractérisé par un nombre, nb (float) et un ensemble, e, de signaux(`Logic`). La méthode `isOn()` de `LogicNumber` retourne `true` si l'évaluation de e comme un nombre vaut nb. Supposons que l'ensemble `e` soit constitué des signaux s_0, s_1, ... s_n. Soit v_i la valeur numérique associée au signal s_i (1 s'il est activé et 0 sinon), alors il est possible de calculer un nombre $signalNumber$ comme valant :$\sum_{i=0}^{n}(2^i * v_i)$ si $nbSignal$ vaut nb alors la méthode `isOn()` de `LogicNumber` retournera `true`. Sinon elle retournera `false`. Si le nombre de signaux dépasse 12 ou si nb est négatif ou supérieur à $2^{e.length}$, `isOn()` retournera également `false`.

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

> Dans ce qui suit, il vous est demandé de coder un certains nombre d'acteurs liés aux signaux. Prenez le soin de lire les spécifications de tous ces acteurs et de réfléchir à une conception avant de vous lancer dans le codage.

6.4 Acteurs « signaux »

Pour commencez, codez les acteurs suivants dans le paquetage `enigme.actor`.

6.4.1 Clé

Définissez une classe `Key`, représentant une clé qui peut être ramassée (par une demande d'interaction). Tant que la clé n'a pas été ramassée, elle est visible et réagit au monde. La clé sera associée à une aire et dotée d'une position à sa création. La clé jouera le rôle d'un signal auquel peuvent réagir d'autre composants : le signal est activé lorsque la clé est collectée et désactivé sinon. Vous pourrez la dessiner au moyen du `Sprite` associé à *`"key.1"`*. Une clé sera un acteur non traversable acceptant uniquement les interactions à distance. A sa création une clé est nécessairement désactivée.

6.4.2 Torche

Définissez une classe `Torch`, représentant une torche qui peut être allumée ou éteinte (par une demande d'interaction). La torche sera associée à une aire et dotée d'une position à sa

création. La torche jouera le rôle d'un signal auquel peuvent réagir d'autre composants : le signal est activé lorsque la torche est allumée et désactivé sinon. Vous pourrez la dessiner au moyen des `Sprite` associés à *`"torch.ground.off"`* et *`"torch.ground.on.1"`*. Une torche sera un acteur non traversable acceptant uniquement les interactions à distance. L'état de la torche (allumée ou éteinte) sera spécifié à sa création.

6.4.3 Bouton « pression »

Définissez une classe `PressureSwitch`, représentant un bouton s'activant/désactivant si on lui passe dessus (cas (d) de la figure 1). Le bouton-pression sera associé à une aire et doté d'une position à sa création. Vous pourrez le dessiner au moyen des `Sprite` associés à *`"GroundLightOn"`* et *`"GroundLightOff"`*. Une `PressureSwitch` sera un acteur traversable acceptant uniquement les interactions de contact. Par défaut, le bouton-pression sera désactivé à sa création.

6.4.4 Levier

Définissez une classe `Lever` représentant un levier qui peut être poussé soit à droite soit à gauche (par une demande d'interaction). Le levier sera associé à une aire et doté d'une position à sa création. Il jouera le rôle d'un signal : activé lorsqu'il est poussé à gauche et désactivé sinon. Vous pourrez le dessiner au moyen des `Sprite` associés à *`"lever.big.left"`* et *`"lever.big.right"`*. Un levier sera un acteur non-traversable acceptant uniquement les interactions à distance. Le levier sera désactivé par défaut à sa création.

Remarque : les 3 derniers acteurs ci-dessus ont plusieurs points communs en tant que signaux : ils peuvent notamment « switcher » du mode activé au mode désactivé et vice-versa (sur une demande d'interaction). Par ailleurs, la clé représente un objet ramassable de plus (en plus de `Apple` déjà codé) et il pourrait y en avoir d'autres . Factorisez les points communs en utilisant des classes intermédiaires. Documentez vos choix de conception dans votre fichier `CONCEPTION.md`. Si vous apportez de modifications au code existant, assurez-vous que le jeu `Enigme` reste fonctionnel.

6.4.5 Plaque de pression

Définissez une classe `PressurePlate`, représentant une plaque de pression s'activant si on lui passe dessus. La plaque de pression sera associée à une aire et dotée d'une position à sa création. Contrairement au bouton pression qui se désactive/active lorsqu'on lui passe dessus, la plaque de pression se désactive seule. Elle s'active lorsqu'on lui passe dessus et reste active pendant un petit moment après qu'on l'a quittée. Une fois ce temps écoulé, elle se désactive. Ainsi si par exemple l'ouverture d'une porte dépend d'une plaque de pression, le joueur aura (peut-être) le temps d'atteindre le porte avant que le signal ne se désactive à nouveau. Vous pourrez définir un temps d'activation par défaut (par exemple 0.3f). Le temps d'activation d'une plaque donnée doit rester inchangé après son initialisation. Vous pourrez dessiner une plaque de pression au moyen des `Sprite` associés à *`"GroundPlateOff"`* et

"GroundLightOn". Une `PressurePlate` sera un acteur traversable acceptant uniquement les interactions de contact. Par défaut, une plaque de pression est désactivée à sa création.

6.5 Acteurs interagissant avec les signaux

Pour le moment seul le personnage `EnigmePlayer` sera capable d'interagir avec les acteurs signaux. Pour toutes les interactions à distance vous appliquerez les mêmes caractéristiques que pour la collecte de `Apple` précédemment codée : `EnigmePlayer` voit juste la cellule droit devant lui et interagit à distance avec son contenu au moyen de la touche 'L'. Le personnage `EnigmePlayer` doit interagir :

- avec la clé en la ramassant ;
- avec la torche et le levier en « switchant » leur état : la torche s'allume si elle était éteinte et s'éteint si elle était allumée. Le levier passe à gauche s'il était à droite et passe à droite s'il était à gauche ;
- avec le bouton pression en le faisant « switcher » d'état lorsqu'il lui passe dessus. Une fois activé/désactivé, le bouton-pression doit rester dans cet état tant que le personnage ne repasse pas dessus une seconde fois. Le changement d'état ne doit donc pas se faire inconditionnellement dès que `EnigmePlayer` occupe la cellule du bouton. L'état du bouton doit être switché lorsque l'acteur a initié un déplacement dans sa cellule et qu'il a atteint sa destination. Ainsi, le bouton pression ne se ré-eteint pas immédiatement dès que le personnage a quitté sa cellule ;
- avec la plaque de pression en l'activant lorsqu'il passe dessus.

6.6 Acteurs dépendants de signaux

Maintenant que vous disposez d'acteurs se comportant comme des signaux, il vous est demandé de coder deux acteurs dépendant de signaux.

Codez une variante de `SignalDoor` de `Door` qui représente une porte dépendante d'un `Logic` donné à la construction. La porte `SignalDoor` n'est sensible aux interactions que si le signal associé est activé. Adoptez un visuel différent selon que le signal associé à la `SignalDoor` est actif ou pas (`Sprite` associé à *"door.close.1"* pour un signal désactivé et à *"door.open.1"* pour un signal activé).

Créez aussi un acteur `SignalRock` qui se dessine comme *"rock.3"* par exemple. Cet acteur devient traversable et n'est plus affiché lorsque le signal qui lui est associé est activé. Il redevient visible et non traversable si le signal qui lui est associé est désactivé.

6.7 Premier jeu d'énigme

Pour vérifier vos derniers développement, débloquez la porte 3 en l'associant à l'aire/niveau `Level3`. Ce niveau sera doté :

- d'une clé en (1,3)
- d'une torche en (7,5)
- d'une plaque de pression en (9,8)
- de boutons-pression en (4,4), (5,4), (6,4), (5,5),(4,6), (5,6) et (6,6)
- de leviers en (10,5), (9,5) et (8,5)
- d'une `SignalDoor` en (5,9) dépendant de l'obtention de la clé (elle n'est sensible aux interactions et donc ne pourra être traversée que si la clé est ramassée). Lorsque franchie, cette porte nous ramène au niveau `LevelSelector` en position (3,6) ;
- de `SignalRock` en (6,8), (5,8) et (4,8) : le premier disparaît si la plaque de pression est activée, le second si tous les boutons-pression sont activés et le 3ème si les leviers sont levés/baissés de sorte à former le chiffre 5 (activé, désactivé, activé) ou si la torche est allumée.

Comme le `SignalRock` en position (5,8) obstrue la porte, pour passer à `E` il faudra ramasser la clé et faire disparaître ce rocher.

Modifiez aussi `LevelSelector` de sorte à ce que les portes associées deviennent des `SignalDoor`. Les 3 premières ont un signal inconditionnellement activé et les autres un signal inconditionnement désactivé. Les 3 premières portes doient ainsi pouvoir être franchies et les autres non. Vous pourrez débloquer certaines de ces portes en codant des extensions.

6.8 Validation de l'étape 4

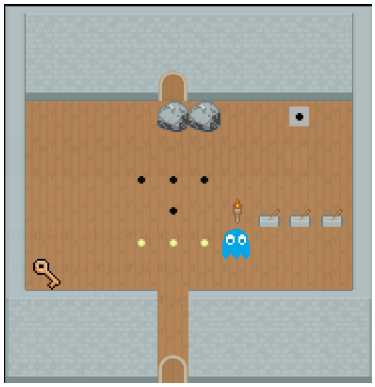


FIG. 9 : Aire `Level3` correspondant à un premier jeu d'énigme

Pour valider cette étape, vous vérifierez que :

- `EnigmePlayer` peut interagir à distance avec la torche en l'allumant et l'éteignant et qu'il ne peut pas marcher dessus ;
- qu'il ne peut marcher sur une `SignalDoor` que si son signal associé est activé ;

- qu'il peut interagir à distance avec les leviers en les levant baissant et qu'il ne peut pas marcher dessus ;
- qu'il peut interagir à distance avec la clé en la ramassant (elle disparaît alors), il ne peut pas marcher dessus s'il ne la pas ramassée ;
- qu'il active les boutons-pression désactivés en marchant dessus et qu'il les désactive au contraire s'ils étaient activés ;
- que le signal de la plaque de pression n'est activé que pendant un petit moment (le rocher associé doit réapparaître après avoir disparu) ;
- que les leviers doivent être dans cette configuration (nombre 5) ou que la torche doit être allumée pour que le rocher associé disparaisse
- que la transition vers `LevelSelector` se passe bien une fois la clé ramassée et les bouton-pressions allumés ;
- que la transition vers `LevelSelector` n'est pas possible si l'une ou l'autre de ces conditions n'est pas vérifiée.
- que le reste des niveaux de `Enigme` reste fonctionnel comme il l'était auparavant.

Le jeu `Enigme` avec le niveau supplémentaire `Level3` et dont le comportement est décrit ci-dessus est à rendre à la fin du projet.

7 Extensions (étape 5)

Pour atteindre la note maximale, il vous est demandé de coder quelques extensions librement choisies parmi celles suggérées ci-dessous. Vous devrez cumuler 10 points pour atteindre le 6. Vous pouvez coder plus que 10 points d'extensions mais au plus 20 points seront comptabilisés (coder beaucoup d'extensions pour compenser les faiblesses des parties antérieures n'est dont donc pas une option possible).

La mise en oeuvre est libre et très peu guidée. Seules quelques suggestions et indications vous sont données ci-dessous. Une estimation de barème pour les extensions suggérées vous sera donnée en temps voulu. Un petit bonus sera attribué si vous faites preuve d'inventivité dans la conception du jeu.

Vous coderez ces extensions dans le cadre de nouveaux niveaux du jeu `Enigme`. Les ressources `Enigme0`, `Enigme1` et `Enigme2` sont à disposition dans le répertoire `res/backgrounds` et `res/behaviors`.

Vous prendrez soin de **commenter soigneusement** dans votre `README.txt`, les niveaux de votre jeu ainsi que les modalités qu'ils impliquent. Nous devons notamment savoir quels contrôles utiliser et avec quels effets sans aller lire votre code.

Il est attendu de vous que vous choisissiez quelques extensions et les codiez jusqu'au bout (ou presque). L'idée n'est pas de commencer à coder plein de petits bouts d'extensions disparates et non aboutis pour collectionner les point nécessaires ;-).

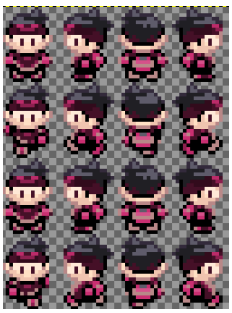
7.1 Dialogues et animation

Les jeux d'énigme peuvent rapidement devenir injouables sans quelques indications dispensées à bon escient. Introduire la possibilité de recourir à quelques dialogues peut donc être un plus fort agréable. Par ailleurs le visuel peut être amélioré en animant un peu certains acteurs. Quelques indications vous sont fournies ci-dessous pour aller dans ce sens.

7.1.1 Animations (~2 à 4pts)

(nombre de points dépendant de comment c'est réalisé : animation codée en dur ou classe dédiée).

Au lieu de représenter un acteur au moyen d'un `Sprite` unique, il est possible de plutôt lui associer une animation qui serait une séquence de `Sprite` affichés tour à tour pour donner une illusion de mouvement. Un `Sprite` complexe, tel que celui associé à `"max.new.2.png"` :



est constitué de 4x4 petits `Sprite` de taille 16x21. Il peut être découpé en ensembles permettant des animations. Par exemple la colonne de gauche permet d'animer des déplacements vers le bas et peut s'obtenir ainsi :

```
Vector anchor = new Vector(0.25f, 0.32f);
  //...
spritesDOWN[i] = new Sprite("max.new.1", 0.5f, 0.65625f, this,
   new RegionOfInterest(0, i * 21, 16, 21), anchor);
```

pour `i` allant de 0 à 3.

Une bonne modélisation, offrira un concept d'`Animation` caractérisé par un ensemble de `Sprite` à jouer tour à tour. Au personnage serait alors associées des animations permettant de l'animer lorsqu'il se déplace vers le haut, le bas, la gauche ou la droite.

Des animations peuvent bien entendu être associées à n'importe quel acteur. Une torche peut par exemple offrir un visuel animé donnant l'impression que sa flamme bouge.

A défaut d'animer les acteurs on peut tout au moins les orienter visuellement en choisissant des `Sprite` spécifiques à l'orientation.

7.1.2 Dialogues (~3 à 5 points)

(nombre de points dépendant de l'effort réalisé pour mettre en place les dialogues).

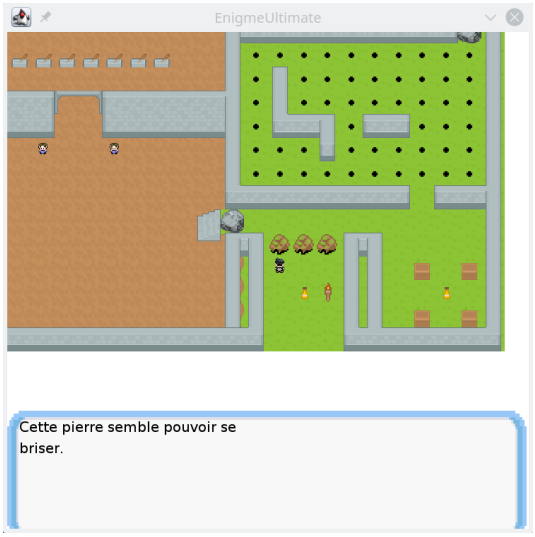
Il est possible d'attacher des textes aux acteurs (un peu comme nous l'avons fait avec `MovingRock`). Vous pouvez initier des dialogues dans certaines situations. Par exemple, lorsque le personnage demande une interaction avec un rocher, une indication peut alors s'afficher pour mettre le joueur sur la piste de ce qu'il faut faire pour déplacer ou briser ce rocher.

Des textes prédéfinis peuvent être stockés dans des fichiers en format `.xml` comme celui fourni dans `res/strings/enigme_fr.xml`. A titre d'exemple :

```
import ch.epfl.cs107.play.io.XMLTexts;
...
XMLTexts.getText("use_eclatrock");
```

retournera la chaîne de caractères `"Allez ! On utilise EclatRock !"`. Le fichier `xml` associé au jeu est défini dans `Play`.

Alternativement, et de façon un peu plus proche de ce qui se fait dans les jeux de type *GameBoy*, vous pouvez exploiter la classe fournie `Dialog` (de `enigme.actor`) et obtenir des visualisations de dialogues ressemblant à ceci :



7.1.3 Pause du jeu(~2 pts)

La notion d'aire peut-être exploitée pour introduire la mise en pause des jeux. Sur requête du joueur, le jeu peut basculer en mode pause puis rebasculer en mode jeu.

7.2 Nouveaux acteurs ou extensions du joueur

Toutes sortes d'acteurs peuvent être envisagés. Une liste (non exhaustive) de suggestions est données ci-dessous.

- rochers que l'on peut pousser et qui peuvent servir à appuyer sur une plaque de pression (pour maintenir une porte ouverte) ; (~4pts)
- signaux avancés pour puzzle (oscillateurs , signaux avec retardateur) : un oscillateur est un signal dont l'intensité varie au cours du temps ; ((~4pts/signal)
- toute sortes de personnages avec des modalités de déplacement et de comportement spécifiques ; pouvant être hostiles ou amicaux à l'égard du joueur ; (~4pts/personnage)
- faire en sorte que le joueur puisse subir de dommages (point de vie) ou recevoir des doses de soin (~4pts)
- passages permettant de se téléporter vers certains niveaux/endroits du jeu ; (~2pts)
- modélisation d'un système de ressources (or, argent, bois, nourriture, etc.) ;(~4pts)
- créer un nouveau mode de déplacement pour votre personnage (à vélo, en courant) avec une adaptation adéquate des sprites/animations ; (~3pts)
- créer un personnage suiveur à l'image du Pikachu de Red dans pokémon jaune ; (~5pts)
- créer un ou plusieurs événements de scénario se déclenchant avec des signaux. Par exemple un personnage qui arrive dans l'aire pour donner un objet ou donner une consigne. (~5pts)
- implémenter une amélioration de la classe `Background` qui serait animée ;(~2pts)
- implémenter une classe `Foreground` pour permettre un rendu plus réaliste de la profondeur (~2pts)
- ajouter de nouveaux type de cellules avec des comportements appropriés (eau, glace, feu, etc.) ; (~2pts/cellules)
- implementer un cycle jour/nuit qui pourrait servir de signal ou qui conditionnerait le comportement du personnage (par exemple il ne peut plus avancer s'il fait trop noir et il devrait se munir d'une lampe de poche) ; (~5pts)
- ajouter une ombre ou un reflet au joueur et à certains acteurs ; (~5pts)
- ajouter de nouveaux contrôles avancés (interactions, actions, déplacements, etc.) ; (~2pts/control)

- ajouter des événements aléatoires (décors, sigaux, etc.) ; (~4pts)
- ajouter des dialogues à choix multiples ;(~4pts)
- ajouter un objet `Box` ou `Safe`, dont l'ouverture serait dirigée par un signal aurait pour contenu un ou plusieurs objets ; (~3pts)
- créer une implémentation d'inventaire simple avec interface qui permettrait la gestion des objets collectés ainsi que leur utilisation/activation. (~15pts) (attention, l'implémentation complète de cette extension sera certainement sous évaluée due à sa complexité)
- etc.

En réalité, la base que vous avez codée peut être enrichie à l'envi. Vous pouvez aussi laisser parler votre imagination, et essayer vos propres idées. S'il vous vient une idée originale qui vous semble différer dans l'esprit de ce qui est suggéré et que vous souhaitez l'implémenter pour le rendu ou le concours (voir ci-dessous), il faut la faire valider avant de continuer (en envoyant un mail à CS107@epfl.ch).

L'annexe 9 vous donne des indications pour enrichir les ressources graphiques.

> Attention cependant à ne pas passer trop de temps sur le projet au détriment d'autres branches !

7.3 Validation de l'étape 5

Comme résultat final du projet, créez deux niveaux de jeu (ou plus) impliquant l'ensemble des composants codés. Une (petite) partie de la note sera liée à l'inventivité et l'originalité dont vous ferez preuve dans la conception des énigmes

8 Concours

Les personnes qui ont terminé le projet avec un effort particulier sur le résultat final (gameplay intéressant, richesse de niveaux de jeu, effets visuels, extensions intéressantes/originales etc.) peuvent concourir au prix du « meilleur jeu du CS107 ».[9]

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).

[9] Nous avons prévu un petit « Wall of Fame » sur la page web du cours et une petite récompense symbolique :-)

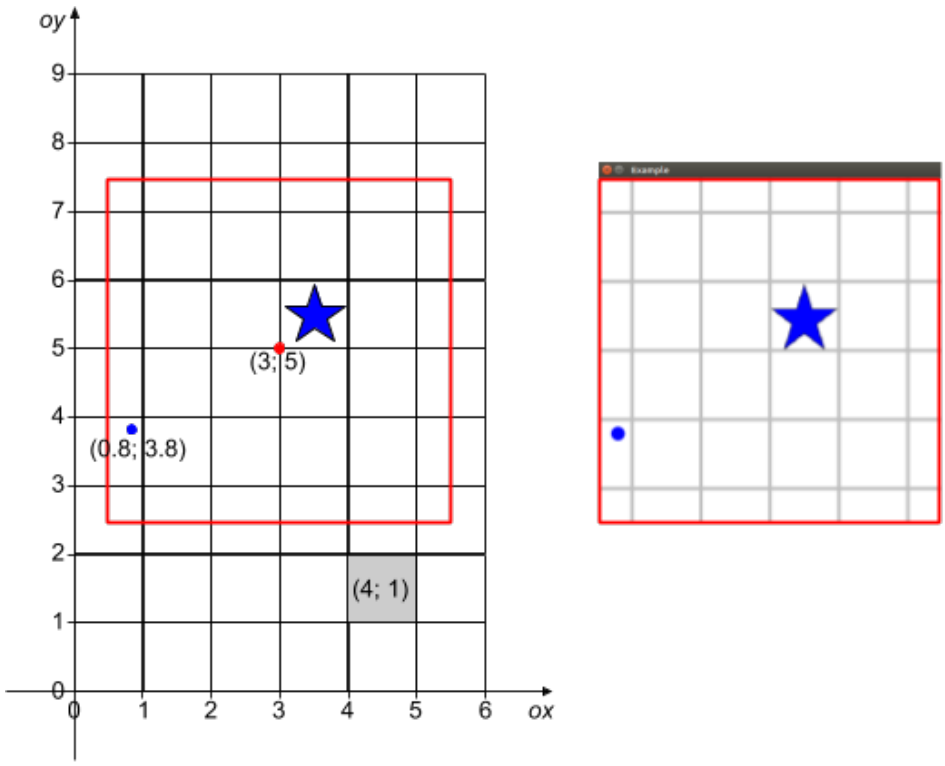


FIG. 10 : La vue sur une partie ciblée de la grille s'obtient par une transformation affine de la fenêtre (ici une simple translation)

9 Annexes

Annexe 1 : Objets « positionnables », transformées et objets graphiques

Le positionnement et l'affichage des éléments simulés dans la fenêtre de simulation sont évidemment des points fondamentaux.

La première remarque à faire à ce propos est que pour positionner les objets simulés il n'est pas commode de raisonner en pixels : cela nous rend dépendant de la taille de la fenêtre ce qui est contre-intuitif ; nos univers simulés seront probablement plus grands que ce que l'on souhaite afficher.

Nous allons donc exprimer toutes nos grandeurs relatives aux positions, dimensions etc. dans les échelles de grandeurs de la grille simulée et non pas en terme de pixels dans la fenêtre. Comme la grille peut être plus grande que la fenêtre d'affichage, nous allons faire subir à cette dernière des transformations affines (translation, zoom etc.) pour nous permettre de nous focaliser sur une partie spécifique du monde (voir la Figure 10).

La fenêtre d'affichage est un exemple typique d'élément nécessitant d'être placé/modifié dans le repère absolu par le biais de transformations. En fait tous les éléments à positionner dans le repère absolu, peuvent l'être selon le même procédé (par exemple les formes ou les

images à dessiner).

En raison de ce besoin, l'API fournie met à disposition les éléments suivants :

- l'interface `Positionable` qui décrit un objet dont on peut obtenir la position absolue par le biais d'une transformation affine (classe `Transform`). Une `Entity` est typiquement un `Positionable`.
- l'interface `Attachable` qui décrit un `Positionable` que l'on peut attacher à un autre (son parent). Ceci se fait au moyen de la méthode `setParent`. Il est caractérisé par une transformée relative, qui indique comment l'objet sera positionné dans le référentiel de son parent (ou dans l'absolu si elle n'a pas de parent).
- la classe `Node` qui est une implémentation concrète simple de l'interface `Attachable`.

La méthode `getTransform()` appliquée à un `Positionable` permet en fait de se situer dans son référentiel local/relatif.

Par ailleurs, l'API fournie met à disposition dans `game.actor` des classes telles que `TextGraphics`, `ImageGraphics` et `ShapeGraphics` qui implémentent la notion d'objets « dessinables » (`Graphics`). Un `Graphics` peut être attaché à une `Entity` par le biais de la méthode `setParent`. Le dessin peut alors se faire de façon simple sans référence explicite aux transformées employées : si un objet graphique est attaché à un entité son dessin se fera nécessairement dans le référentiel de cette entité sans qu'il soit nécessaire de l'y placer explicitement au moyen d'une transformation (vous en avez un exemple avec le texte attaché au rocher dans le premier « jeu » à créer, `Demo1`).

Il est toutefois nécessaire parfois de préciser le point d'ancrage de l'objet gaphique par rapport à l'entité qui lui sert de parent (c'est à dire de combien l'image doit être décalée de l'origine pour se superposer proprement à l'entité). Jetez un oeil à l'API concernée pour voir comment se concrétise cette notion de point d'ancrage.

Annexe 2 : Structures de données utiles

Il existe de nombreuses structures de données. Par exemple, dans le cadre de ce cours, vous avez appris à utiliser les tableaux dynamiques par le biais de la classe `ArrayList`. En réalité, `ArrayList` est une implémentation particulière de la structure de données abstraite *liste*.

Les structures de données sont fournies en Java sous la forme :

- D'une interface qui décrit les fonctionnalités usuellement admises pour la structure de données en question ; par exemple, le fait de pouvoir ajouter un élément en fin de liste pour les listes. Pour les listes justement, l'interface qui en donne les fonctionnalités est `List`.
- D'une implémentation de base très générale de cette interface sous la forme d'une classe abstraite : `AbstractList` pour les listes.
- De (généralement) plusieurs implémentations spécifiques dérivant de la classe abstraite, par exemple `ArrayList` ou `LinkedList` pour les listes. Ces implémentations spécifiques ont chacune des particularités qui font que l'on préférera utiliser l'une plutôt que l'autre en fonction du contexte. Par exemple les `LinkedList` offrent des opérations d'ajout ou de suppression après un élément donné en temps constant (O(1)), mais n'offrent pas la possibilité d'accéder à un élément à une position donnée en temps constant. Pour les `ArrayList` (« tableau liste ») c'est l'inverse. On aura donc tendance à privilégier les `LinkedList` (« liste chaînée ») si les opérations d'ajout ou de suppression sont plus nombreuses que celles nécessitant un accès direct.

Certaines structures de données s'avèrent plus appropriées que d'autres selon les situations. Nous vous en décrivons brièvement deux supplémentaires qui vont s'avérer utiles dans le cadre de ce mini-projet (une présentation plus en profondeur de ces structures de données et de leur caractéristiques sera faite au second semestre).

Les tables associatives

Les tables associatives (« map ») permettent de généraliser la notion d'indice à des types autres que des entiers. Elles permettent d'associer des *valeurs* à des *clés*.

Par exemple :

```
import java.util.Map;
import java.util.HashMap;
import java.util.Map.Entry;

//...
        // String est le type de la clé et Double le type de la
           valeur
        Map<String, Double> grades = new HashMap<>();
        grades.put("CS107", 6.0); // associe la clé "CS107" à
           la valeur (note ici) 6.0
        grades.put("CS119", 5.5);
```

```
        // … idem pour les autres cours auxquels on aimerait
           associer sa note

        // Trois façon d'itérer sur le contenu de la map
        for (String key  : grades.keySet()) {
            //itérer sur les clés
            System.out.println(key+ " " +grades.get(key));
        }

        for (Double value  : grades.values()) {
            //itérer sur les valeurs
            System.out.println(value);
        }

        for (Entry<String,Double> pair  : grades.entrySet()) {
            //itérer sur les paires clé-valeur
            System.out.println(pair.getKey() + " " +
               pair.getValue());
        }
```

La clé d'une Map peut dont être vue comme la généralisation de la notion d'indice. L'interface Java qui décrit les fonctionnalités de base des tables associatives est `Map`, l'implémentation concrète que nous utiliserons est `HashMap`.

Les ensembles

Il est parfois nécessaire de manipuler une collection de données comme un *ensemble* au sens mathématique ; c'est-à-dire où *chaque élément est unique*. Par exemple si nous souhaitons modéliser l'ensemble des voyelles, il n'y a pas de raison que la lettre `'a'` y apparaisse deux fois. La méthode d'ajout d'un élément dans un ensemble garantit que l'élément n'y est pas ajouté s'il y était déjà :

```
import java.util.Set;
import java.util.HashSet;

//...
        Set<Character> voyels = new HashSet<>();
        voyels.add('a'); // voyels -> {'a'}
        voyels.add('u'); // voyels -> {'a', 'u'}
        voyels.add('a'); // voyels -> {'a', 'u'}

        // affiche : a u
        for(Character letter  : voyels) {
            System.out.print(letter + " ");
        }
```

L'interface Java qui décrit les fonctionnalités de base des ensembles est `Set`, l'implémentation

concrète que nous utiliserons est `HashSet`.

Annexe 3 : Ressources graphiques et éditeur de niveaux

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. Il est alors indispensable d'en citer l'origine !

Éditeur de niveaux Les aires de jeu ont une image de fond qui se superpose à une image dictant son comportement (couleur des pixels) :

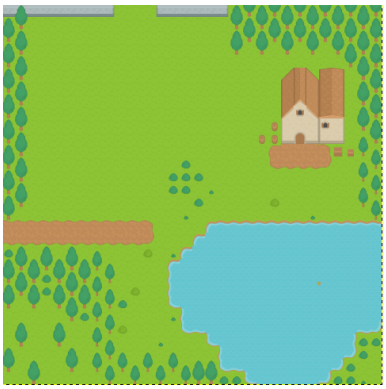
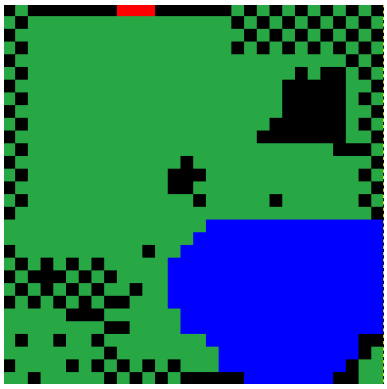


image de fond



« comportement » correspondant

Nous vous fournissons quelques exemple dans le fichiers de ressources `res` où le répertoire `images/background/` contient des images de fond et à chacune de ces images correspond une image de « comportement » possible dans le dossier `behaviors/`.

Il est évidemment intéressant de pouvoir créer de nouvelles images. Si vous le souhaitez (ça n'est pas demandé dans le cadre du projet), vous pouvez utiliser l'éditeur de niveau simple proposé ici par Bastien Chatelain : https ://github.com/blchatel/LevelEditor [Lien]