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

B. Châtelain, J. Sam, B. Jobstmann

Version 1.0

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

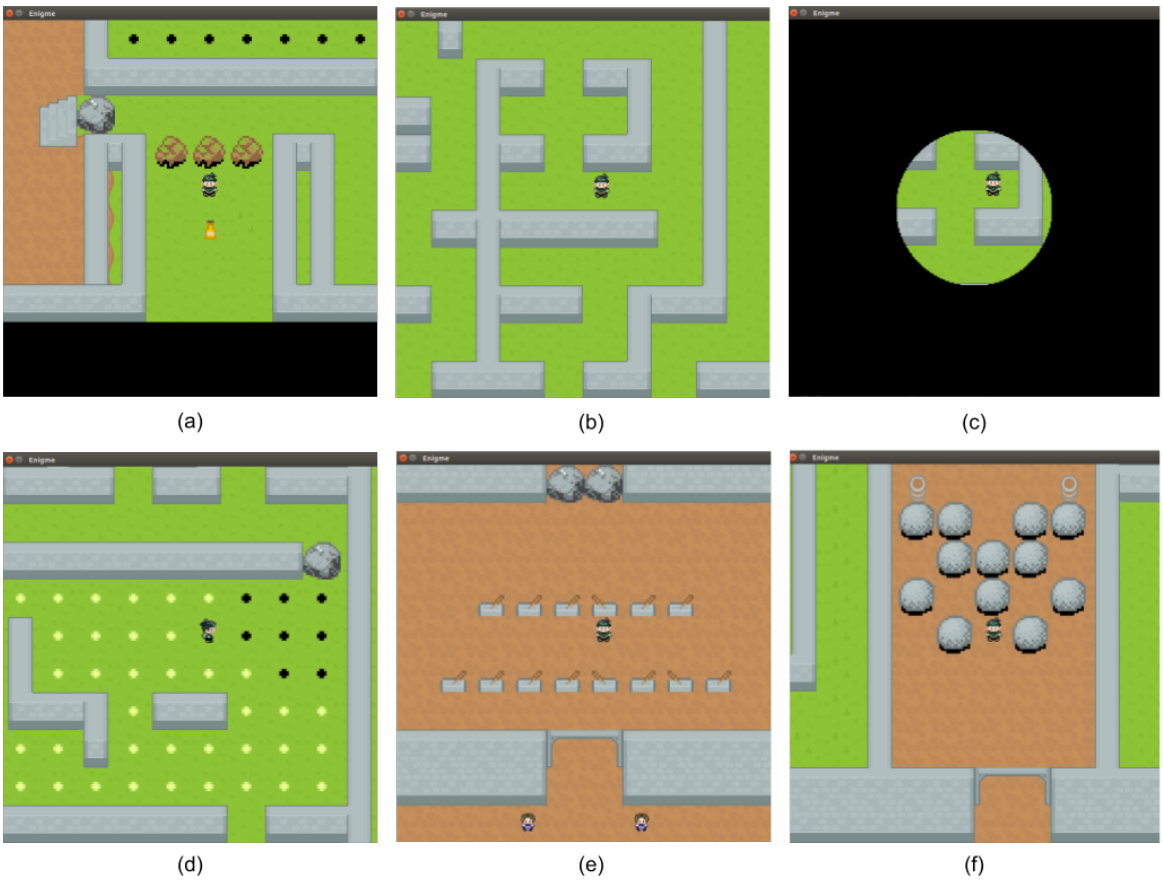


FIG. 1 : Le joueur devra résoudre des énigmes par exemple : (a) casser un rocher avec un objet à trouver, (b) et (c) trouver son chemin dans un labyrinthe avec ou sans champ de vision restreint, (d) activer tous les signaux en marchant dessus, (e) trouver la bonne combinaison de leviers ou (f) se frayer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

Dans cette première version de l'énoncé, seule l'étape 1 (sur 5) est présente. Les parties suivantes seront publiées le 22.11.

[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.

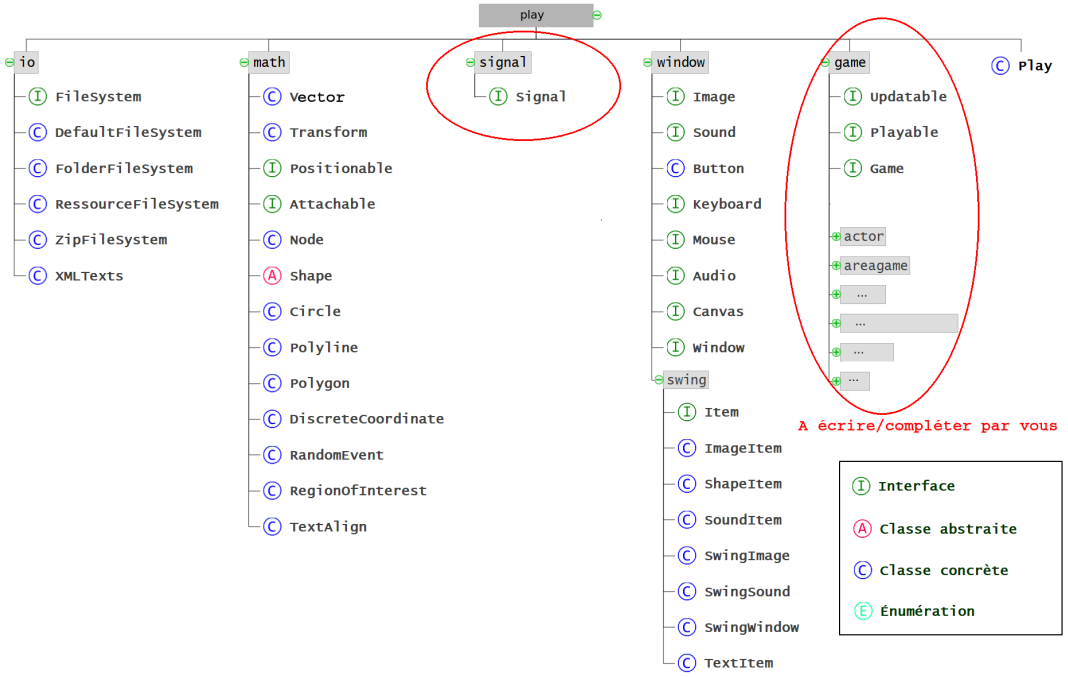


FIG. 2 : Paquetages principaux du projet. Vous interviendrez essentiellement dans le paquetage `game` et le paquetage `signal`

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 4 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.

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 4. 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 4 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 à au rayon du cercle, le texte "BOUM !!!" s'affiche en rouge :

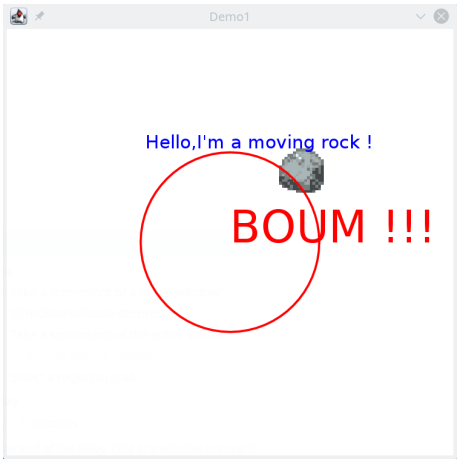


FIG. 4 : Le texte "BOUM !!!" doit cesser de s'afficher lorsque la distance entre le rocher et le cercle redevient trop grande.

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.

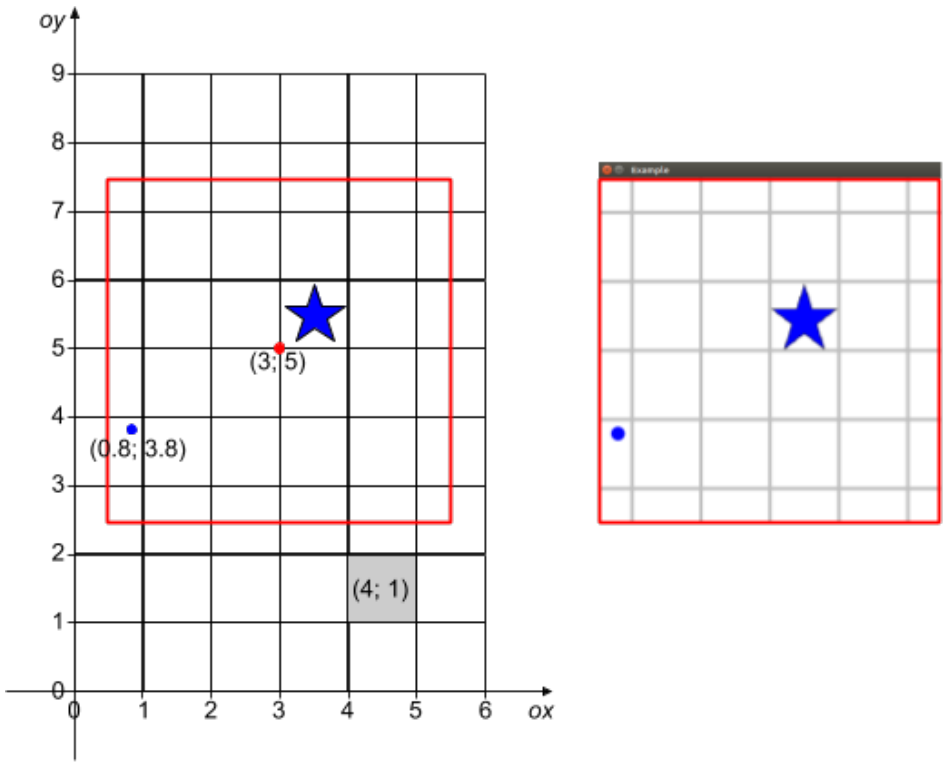


FIG. 5 : La vue sur une partie ciblée de la grille s'obtient par une transformation affine de la fenêtre (ici une simple translation)

4 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 5).

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) :

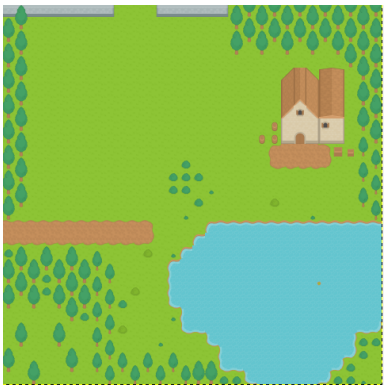
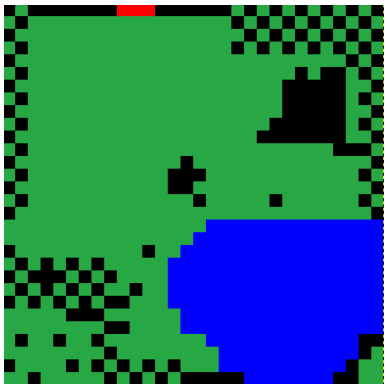


image de fond



« 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]