CS-107 : Eléments de moteur de jeux

J. Berdat, B. Châtelain, Q. Juppet et J. Sam Version 1.0.0

Table des matières

1 Introduction

Pour mettre en œuvre le second mini-projet, le cours met à votre disposition une maquette contenant un certain nombre d'outils. Cette maquette est un moteur de jeux basique permettant de créer des jeux en 2D et notamment des jeux sur grille[Lien] pouvant se décliner en de nombreuses variantes, inspirés d'exemples tels que :

Pokémon Emeraude [Lien]Super Pacman [Lien]

Le temps et les connaissances nécessaires pour implémenter l'entièreté d'un jeu de ce type sont en effet hors de portée de notre cours. De plus, apprendre à composer avec du code existant et en tirer parti de façon adéquate est un objectif à part entière dans l'apprentissage de la programmation orientée-objet.

Le but de ce document est de vous permettre de comprendre le contenu de la maquette fournie. Cette dernière vous servira donc de base pour l'implémentation de votre second mini-projet.

2 Schéma général de la maquette

L'architecture de la maquette est schématisée dans les grandes lignes par le diagramme de la Figure 1.

Un petit descriptif des paquets fournis est donné ci-dessous. Le code et sa documentation ainsi que ce tutoriel devraient répondre aux détails, vous donnant ainsi l'occasion d'accéder à un code émanant de programmeurs plus expérimentés [1]

- 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.

- 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

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

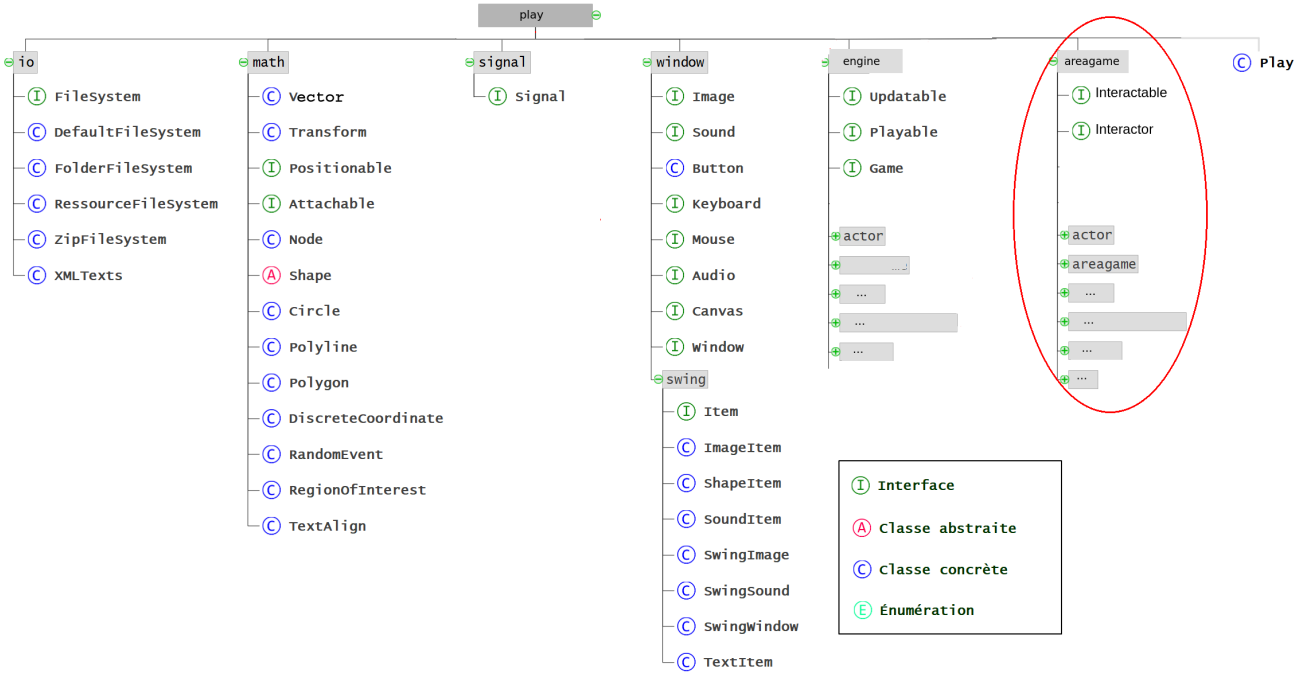


FIG. 1 : Paquetages principaux du projet. Ce tutoriel vous présentera pour l'essentiel les paquetages de `engine` et `areagame`

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/dessinée 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 ainsi d'avoir entre 20 et 60 frames par seconde.

- Le paquetage `engine` contient des éléments abstraits fondamentaux du moteur de jeu (principalement des modélisations abstraites d'« acteurs »).
- Le paquetage `areagame` bâtit sur les éléments de `engine` pour modéliser des jeux se déroulant sur des grilles. C'est qui va vous occuper principalement tout au long de ce tutoriel. Le travail à réaliser dans le cadre du mini-projet consiste à utiliser le contenu du paquetage `areagame` et à l'étendre.
- Le paquetage `signal` permettra d'inclure des composants liés à des signaux, essentiellement logiques.

Il ne vous est pas demandé de consulter ce matériel dès maintenant, mais d'y revenir durant la lecture de ce tutoriel, et quand vous coderez le mini-projet, au gré de vos besoins et en fonction de nos indications. Vous trouverez également dans les annexes (voir section 7) quelques compléments d'information utiles.

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

Ce tutoriel est structuré en 3 parties qu'il faudra lire selon le calendrier suggéré dans les énoncés d'exercices :

- la première partie décrit la modélisation de la notion de jeu sur grille qui implique de définir ce qu'est une *aire de jeu* et la *grille* associée ; elle décrit également le concept d'*acteur* à un niveau *abstrait* ;
- la seconde partie explique comment le concept d'*interface* est exploité afin de modéliser ces éléments à un niveau suffisamment abstrait pour les rendre utilisables dans différents contextes ; elle décrit aussi plus précisément la notion de grille et les spécificités des acteurs évoluant sur des grilles ;
- la dernière partie décrit le schéma exploité pour mettre en œuvre des interactions entre acteurs ; toujours à un niveau suffisamment abstrait ; elle illustre aussi comment la notion de classes imbriquées est utilisée pour une meilleure encapsulation dans certaines parties du code.

3 Mise en place du mini-projet 2

Pour pouvoir suivre ce tutoriel, il est nécessaire d'installer le code fourni dans un projet IntelliJ (ou Eclipse). Vous trouverez ci-dessous les indications nécessaires.

> Pour la réalisation du tutoriel et du projet ainsi que pour éviter des problèmes de rendu, veillez à configurer Intellij en UTF-8 (voir pour cela "Réglage de l'encodage des caractères" dans le Tutoriel 2 d'IntelliJ ou dans la contre-partie Eclipse)

Le matériel fourni se trouve dans l'archive : tutoriel-2025.zip
Pour installer le projer sur IntelliJ :

1. dezzipez cette archive dans un répertoire de votre choix ;
2. ouvrez dans IntelliJ le répertoire `TUTO-2025` ;
3. supprimez l'archive `tutoriel-2025.zip`.

Pour Eclipse, une fois l'archive dezippée créez le projet en utilisant l'option `"From existing sources"` et en indiquant le répertoire hôte comme racine du projet.

- Le matériel fourni comporte deux dossiers : `game-engine` qui contient les outils offerts par la maquette et `tutos` qui est le dossier où vous allez faire les exercices.
- **Il est normal que le projet ne s'exécute pas** : le jeu à lancer vaut `null` dans le fichier `Play.java` du dossier `tutos`. Vous allez corriger cela en progressant dans la mise en œuvre du tutoriel.
- **Vous ne modifierez pas le contenu du dossier `game-engine`.**
- Profitez des fonctions de recherche de `IntelliJ` (par exemple « Clic-droit » sur un identificateur puis `Go to > Declaration or usage` peut s'avérer très profitable).
- Règle simple à retenir pour résumer :
  - Le contenu de la maquette est à consulter dans les paquetages de `ch.epfl.cs107.play` **du dossier `game-engine`.**
  - les ressources graphiques fournies sont dans le dossier `src/main/resources` **du dossier `tutos`.**
  - Les exercices sont à faire dans des sous-paquetages de `ch.epfl.cs107.play` **du dossier `tutos`.**
  - Apprenez à explorer les facilités de votre outil de développement.

Les 3 volets du tutoriel présentés dans ce qui suit, ainsi que l'énoncé du mini-projet 2 vous indiqueront ce qu'il faut consulter et compléter.

4 Tutoriel I

Ce premier tutoriel a pour but de commencer à vous faire investiguer les concepts fournis par la maquette mise à votre disposition. Il explique dans les grandes lignes ce qu'est un « jeu » dans la conception choisie et comment son évolution au cours du temps est réalisée par la boucle de simulation du programme principal fourni, `Play`. Il explique également comment est modélisée une *aire de jeu* ainsi que des *acteurs*, très rudimentaires pour le moment, qui peuvent y prendre place. Il est conseillé d'ouvrir le code concerné, essentiellement dans les paquetages `engine` `areagame`, et de l'examiner en parallèle de la lecture des explications données. Une fois les concepts présentés, vous aurez à coder en guise d'exercice, une ébauche rudimentaire de jeu.

4.1 `Playable`

La maquette fournie (`game-engine`) utilise le concept abstrait d'« élément jouable » (`Playable` du sous-paquetage `engine`). Cela peut être un jeu complet, une aire dans le jeu etc. Il s'agit d'une abstraction d'assez haut niveau qui consiste à dire que pour qu'un élément du programme soit « jouable » il faut qu'il puisse :

1. *évoluer* au cours du temps (disposer d'une méthode de mise à jour :
   `void update(float time)`) ;

2. *commencer* proprement (c'est-à-dire s'initialiser notamment en incorporant toutes les entités qui sont amenées à y évoluer) ; ceci nécessite l'accès à un contexte graphique/fenêtre et à un système de fichiers pour aller chercher des ressources, comme des images par exemple (présence d'une méthode :
   `begin(Window window, FileSystem filesystem)`) ;

3. et *se terminer* proprement ; c'est-à-dire mettre en œuvre 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 (présence d'une méthode `end`).

Un « élément jouable » sera en outre caractérisé par un *nom*, une chaîne de caractères retournée par une méthode `getTitle()`.

Le concept de `Playable` est codé au moyen de la notion d'*interface* Java et nous y reviendrons lorsque cela aura été abordé en cours. Pour le moment c'est uniquement le concept en tant que tel qui nous intéresse et non sa mise en œuvre concrète. Voyez-le pour le moment un peu comme une classe abstraite incorporant les éléments cités plus haut. Notez que `Game`, qui modélise la notion de « jeu » est une sorte de `Playable` avec en plus une fréquence de rafraîchissement.

4.2 Boucle de simulation

Quelqu'un souhaitant « lancer » un « jeu », doit alors s'y prendre comme dans l'exemple de programme principal fourni dans `Play.java` du dossier `tutos` :

- Créer une instance de jeu (ligne 31), un système de fichier (ligne 28) et une fenêtre/-contexte graphique (ligne 34).

- Ensuite, lancer le jeu avec `begin` en lui passant en paramètres le système de fichier

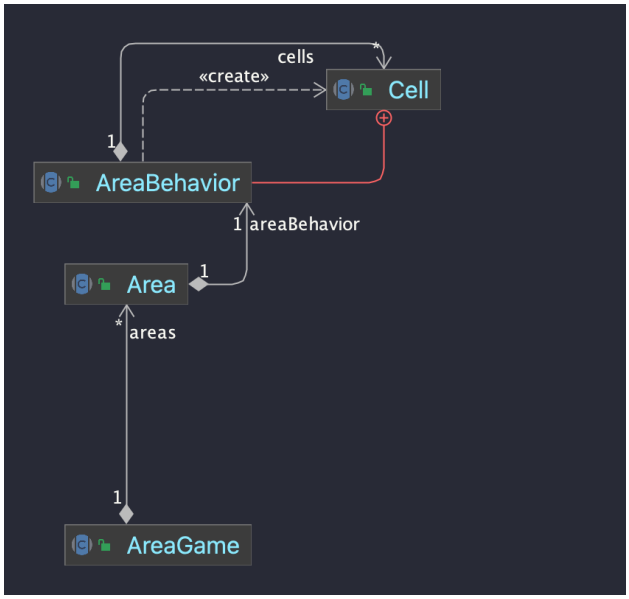


FIG. 2 : Un `AreaGame` *a-un* ensemble de `Area`. Chaque `Area` *a-une* grille (`AreaBehavior`). Chaque grille est constituées de cellules (`Cell`) dont elle a la charge de la création.

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 demandée, le jeu et la fenêtre seront l'un après l'autre mis à jour (lignes 66 à 71). L'actualisation de la fenêtre consiste à redessiner son contenu depuis une liste d'items graphiques (automatiquement) vidée après chaque itération. C'est le rôle du jeu, dans sa propre méthode de rendu `draw`, 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 à actualiser tous ses composants (par exemple les repositionner) en fonction du temps écoulé depuis le dernier appel.

Lorsque le jeu doit cesser de s'exécuter sa méthode `end` doit être invoquée (ligne 74).

4.3 Jeux sur grille

Les outils fournis dans le paquetage `engine` permettent de mettre en œuvre des jeux où l'on veut :

- que puisse intervenir un nombre indéterminé d'acteurs (des personnages, des objets caractéristiques etc.) ;
- que l'aire de déroulement du jeu ne se cantonne pas à la seule fenêtre physique ;
- et que les acteurs puissent interagir physiquement de façon simple (utilisation de concepts simples pour gérer les collisions physiques et caractériser les zones d'interaction entre acteurs).

Les jeux qui se déroulent sur une grille (package `areagame`) offrent l'avantage de simplifier sensiblement le dernier point (gestion des collisions et interactions) et c'est sur cette base

que nous allons partir pour le codage du mini-projet de cette année[2].

Ce tutoriel va donc essentiellement vous décrire les classes mises à disposition dans la maquette pour modéliser des « jeux sur grille ».

Pour résumer :

- `AreaGame` : concept de « jeu sur grille ».
- `Area` : zone de `AreaGame`. Un `AreaGame` comporte potentiellement plusieurs `Area`.
- `AreaBehavior` : grille associée à une `Area` (chaque zone du jeu possède une grille caractéristique).

Concrètement, comme le montre la figure 2, le code de la maquette permet de modéliser des jeux sur grille (`AreaGame`) composés d'un ensemble de zones appelées `Area` (section 4.4). A Chaque `Area` correspond une grille à deux dimensions composée de cellules (`Cell`) dans lesquelles évolueront et interagiront des acteurs/composants (des `Actor`).

Dans l'optique de ressembler visuellement à certains jeux de type *Game Boy*, et comme dit plus haut, cette grille est introduite notamment pour simplifier la gestion des interactions entre acteurs ainsi que leurs déplacements : c'est la cellule et son contenu qui conditionnent 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, c'est à dire un « élément jouable » aussi, avec un ensemble d'acteurs. Pour ne pas surcharger cette classe, elle 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 5.2). Par conséquent, chaque `Area` possède une `AreaBehavior` qui gère le comportement et les mécaniques de la zone avec sa grille, ses cellules et de leur contenu.

Vous trouverez dans ce qui suit quelques explications concernant les classes `Area` et `AreaGame`. Les explications sur `AreaBehavior` vous seront données dans le tutoriel II (section 5).

4.4 Aires de jeu : classe `Area`

La classe abstraite `Area` fournie dans le sous paquetage `areagame.area` modélise une zone (jouable) dans un jeu sur grille. Ouvrez-la Javadoc et cherchez-y cette classe (petite fenêtre de recherche en haut à droite)[3] pour l'examiner dans les grandes lignes. N'hésitez pas non plus à ouvrir le code lui-même tel que fourni.

Une `Area`, est un élément « jouable », et va de ce fait naturellement disposer des méthodes `begin`, `end` et `update` évoquées plus haut.

Une aire a une liste d'acteurs (ceux qui y évoluent, des `Actor`). Elle a une grille associée (`AreaBehavior`) qui quadrille la zone de jeu et sur laquelle les acteurs vont évoluer.

[2] notez que la maquette permet aussi de programmer des « jeux » détachés de la notion de grille

[3] n'hésitez pas à recourir à la Javadoc pour toute la suite du mini-projet 2

Parmi les méthodes fondamentales à retenir :

- `begin` qui réalise toutes les initialisations nécessaires au démarrage d'une zone jouable ;
- `update` qui fait évoluer les acteurs (en invoquant leur propre méthode `update`) ;
- `draw` qui en fait le rendu ; c'est à dire les dessine et fait jouer leurs éventuels effets sonores (méthodes `bip` et `draw` des acteurs) ;
- `registerActor` et `unregisterActor` qui permettent d'ajouter/supprimer un acteur dans la liste ;
- `end` qui réalise toutes les actions à effectuer lorsque le jeu sur l'aire courante prend fin.

4.4.1 Transition d'une aire à l'autre

Les jeux qui nous intérèssent sont censés être 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, la classe `Area` offre les méthodes suivantes :

- `void suspend()` qui par défaut ne fait rien mais qui, une fois redéfinie, peut implémenter toute stratégie spécifique à mettre en œuvre 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) ;
- `boolean resume(Window window, FileSystem fileSystem)` qui retourne `true` par défaut mais qui, une fois redéfinie, peut permettre 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.

4.4.2 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 et selon un facteur d'échelle donné. Parmi les méthodes importantes :

- `getCameraScaleFactor()` une méthode abstraite qui peut être redéfinie dans les sous classes pour retourner le facteur d'échelle voulu (en fonction de sa valeur la vue est plus ou moins "zoomée") ;
- `setViewCandidate` qui permet de centrer la vue sur un acteur. C'est cela qui va permettre à la caméra de suivre un personnage dans de nombreuses aires, tel que vous allez pouvoir l'expérimenter dans vos premières ébauches de jeu.

4.5 Jeux avec aires : classe `AreaGame`

La classe abstraite `AreaGame` du sous-paquetage `areagame` de la maquette modélise simplement le concept de « jeu avec plusieurs aires ». Un jeu avec plusieurs aires est aussi avant tout un élément « jouable » et vous y retrouverez les attributs et méthode spécifique à ce concept (attributs de type `Window` et `FileSystem`, et méthodes `update`, `begin` et `end`).

Vous y trouverez 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) :

```
/// 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 a été choisi le type `Map` (table associative clé–valeur) qui permettra de retrouver une aire de jeu à partir de son nom (`getTitle`) (voir l'annexe 7).

La méthode `getTitle` des aires est très importante car c'est par le biais de ce "titre" qu'une aire est identifiée (et c'est par ce biais ausi que se fera aussi le lien de l'aire avec certaines des ressources qui lui sont utiles).

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

Parmi les méthodes importantes :

- `addArea()` qui permet l'ajout dynamique d'aires au jeu ;
- `setCurrentArea` qui permet de choisir l'aire courante (simulée) parmi toutes les aires ; son second paramètre permet d'indiquer si au moment du passage à cette aire on souhaite la redémarrer (paramètre `forceBegin` à `true`) ou la continuer à l'endroit où on l'avait laissée lors d'un éventuel précédent passage).

4.6 Acteurs génériques

La maquette proposée permet de programmer de jeux mettant 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 ».

Vous trouverez dans les sous-paquetages `engine.actor` et `areagame.actor` de la maquette un certain nombre de classes (et d'interfaces) permettant de modéliser la notion d'acteurs de façon générique (voir ce schéma de classe[Lien]). Le concept d'« acteur » est modélisé par l'entité `engine.actor.Actor` (il s'agit d'une interface, mais voyez-le pour le moment comme une classe abstraite). Pour ce modèle très abstrait, un acteur est simplement une entité qui évolue au cours du temps (méthode `update`) et qui peut émettre des sons.

La classe `Entity` est une mise en oeuvre particulière et basique d'`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 7. Il n'y a en principe pas besoin de comprendre cette notion en profondeur pour démarrer le projet.

Le premier acteur que vous allez utiliser est très rudimentaire. Il sera codé comme sous-classe de `Entity`.

4.7 Exercice 1 : premier « jeu avec aires »

Il est temps maintenant de commencer à essayer d'exploiter par vous-même les quelques éléments de la maquette présentés jusqu'ici. Vous allez coder une ébauche de jeu avec aires.

Les EDI comme Eclipse ou IntelliJ sont très pratiques pour ajouter automatiquement les directives d'importations manquantes dans une classe. Cet usage est recommandé, mais faites bien attention, lorsqu'il y a plusieurs choix, à inclure l'option qui correspond à vos besoins et à ne pas inclure d'éléments non standards (il ne faut garder que les import commençant par `java.` ou `javax.`). La maquette fournie utilise notamment la classe `Color`. Il faut utiliser la version `java.awt.Color` et non pas d'autres implémentations provenant de divers packages alternatifs.

4.7.1 Premier acteur concret

Dans le dossier `tutos`, créez un sous-paquetage `actor` du paquetage fourni `ch.epfl.cs107.play.tuto1` dans lequel vous coderez une nouvelle classe d'acteurs appelée `SimpleGhost`. Ces acteurs dérivent de la classe `Entity`[4]. Il s'agira d'un acteur doté d'une représentation graphique, c'est-à-dire d'un attribut de type `Sprite` (type fourni dans le sous-paquetage `engine.actor` de la maquette).

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).

Un `SimpleGhost` sera également caractérisé par un *niveau d'énergie* (codé comme une `float`). Vous le doterez des méthodes :

- `boolean isWeak()` retournant le booléen `true` si le niveau d'énergie du fantôme est inférieur ou égal à zéro ;
- `void strengthen()` remettant le niveau d'énergie à une valeur positive donnée (toujours la même (choisissez 10 par exemple) ;

Son constructeur aura pour entête :

```
public SimpleGhost(Vector position, String spriteName)
```

`spriteName` est le nom de l'image associée au fantôme lors de sa construction (cette image sera recherchée dans le dossier `src/main/resources/` [5] par le code de `Sprite`). Le `Sprite` associé à `SimpleGhost` peut être créé au moyen de la tournure :

```
new Sprite(spriteName, 1f, 1f, this);
```

[4] nous le nommons `SimpleGhost` car il est dérivé de `Entity` qui est une classe d'« acteurs » de très bas niveau, il s'agit en effet simplement d'un objet positionnable dans l'espace

[5] les ressources sont dans le dossier de la maquette (`game-engine`)

dans le constructeur de `SimpleGhost`. Le paramètre `this` permet au constructeur du `Sprite` de le rattacher à l'objet courant.
Le constructeur de `SimpleGhost` initialisera le niveau d'énergie avec une valeur par défaut (choisissez une valeur pas trop élevée comme `10` ... vous verrez pourquoi un peu plus loin).
Nous souhaitons par ailleurs afficher le niveau d'énergie à côté du fantôme.
Pour cela, il faudra déclarer un attribut :

```
private  TextGraphics hpText;
```

qui sera initialisé dans le constructeur au moyen de la tournure :

```
new TextGraphics(Integer.toString((int)hp), 0.4f, Color.BLUE);
```

où `hp` représente l'attribut « niveau d'énergie ».
Pour faire en sorte que ce texte soit lié au fantôme, et donc se déplace avec lui, il faut l'y attacher :

```
hpText.setParent(this);
```

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

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

Ces deux instructions sont à placer, une fois pour toute, dans le constructeur.
Notre acteur `SimpleGhost` hérite d'une méthode `void draw(Canvas canvas)` héritée de `Entity`. Cette dernière permet d'afficher sur un support de type `Canvas`[6], l'image associée à notre objet. Notez que `Sprite` et `TextGraphics` disposent de méthodes `void draw(Canvas canvas)`.
Redéfinissez la méthode `draw` héritée de `Entity` pour que le texte du niveau d'énergie s'affiche aussi. Redéfinissez également la méthode `void update(float deltaTime)`. Son rôle sera de décrémenter de `deltaTime` le niveau d'énergie du fantôme ; le fantôme ne pourra cependant pas avoir un niveau d'énergie plus bas que zéro. Il faudra penser à mettre à jour en conséquence le texte `hpText`.

4.7.2 Premières aires de jeu concrètes

Dans le paquetage `ch.epfl.cs107.play.tuto1.area` du dossier `tutos`, vous trouverez une sous-classe `SimpleArea` de `Area`. Cette classe impose à ses sous-classes concrètes la définition d'une méthode `createArea` permettant de créer le contenu d'une aire de jeu spécifique. Les méthodes `getWidth` et `getHeight` vous seront expliquées dans le tutoriel suivant. Elles ne sont pas utiles à ce stade. Vous pouvez redéfinir `getCameraScaleFactor` dans la classe `SimpleArea`. Faites-lui retourner la valeur `10f` par exemple. Ceci permet de définir un facteur d'échelle par défaut à utiliser pour toutes les aires de type `SimpleArea`.
Créez un sous-paquetage `ch.epfl.cs107.tuto1.area.maps` et créez-y les aires spécifiques `Village` et `Ferme` héritant de `SimpleArea`.
Vous donnerez des définitions concrètes à la méthode `createArea` dans chacune de ces classes. Cette méthode doit :

- ne rien faire pour le moment dans `Ferme` ;

[6] Window dérive de Canvas

- créer un acteur `SimpleGhost` dans `Village` et l'y enregistrer (souvenez-vous de la méthode `registerActor` ... et avez vous une idée de pourquoi on n'utilise pas ici plutôt `addActor` ? Vous utiliserez un `Vector(20,10)` en guise de position et l'image nommée *"ghost.2"*.

Tout « jeu » spécifique doit spécifier le nom qui le caractérise. Vous ajouterez pour cela à la classe `Village`, la redéfinition :

```
@Override
public String getTitle() {
  return "zelda/Village";
}
```

et vous procéderez de façon analogue pour `Ferme` :

```
@Override
public String getTitle() {
  return "zelda/Ferme";
}
```

Enfin, vous noterez que les constructeurs « par défaut, par défaut » sont suffisants pour ces deux classes.

4.7.3 Premier jeu avec aires concrètes

Nous disposons à ce stade de deux aires dont l'une est amenée à contenir un acteur concret. Il nous reste à définir un jeu constitué de ces deux aires. Rappelons que `AreaGame` permet précisément de modéliser un jeu avec plusieurs aires. Définissez donc dans le paquetage `ch.epfl.cs107.play.tuto1`, la classe `Tuto1` héritant de `AreaGame`. Dotez là d'une méthode privée `createAreas()` qui a pour vocation d'ajouter les aires voulues au jeu. Cette méthode se contente donc d'appeler la méthode `addArea` de `AreaGame`, par exemple comme ceci pour ajouter l'aire `Ferme` :

```
addArea(new Ferme());
```

Comme tout élément jouable, `Tuto1` doit définir les méthodes `begin`, `end` , `update` et `getTitle`. Vous les coderez comme suit :

- la méthode `end` ne fait rien de particulier ;
- la méthode `update` se contente d'invoquer celle de la super-classe ;
- la méthode `getTitle` retourne un intitulé associé au jeu, comme *"Tuto1"* ;
- la méthode `begin` utilisera la tournure suivante :

  ```
  if (super.begin(window, fileSystem)) {
    // traitement spécifiques à Tuto1
      return true;
  }
  else return false;
  ```

  Les traitement spécifiques à `Tuto1` consisteront à :

  – créer les aires (au moyen de `createAreas`) ;

– et indiquer que l'aire courante est l'aire intitulée *"zelda/Ferme"* (on souhaite que lors du passage à cette aire, elle soit redémarée, le paramètre `forceBegin` vaudra donc `true`)

Question 1

Que se passe-t-il si l'on oublie d'invoquer le `update` de la super-classe dans la méthode `update` de `Tuto1` ?

Il ne reste plus qu'à indiquer dans le programme principal `Play`, que l'on souhaite lancer l'élément jouable `Tuto1`.

Pour cela, commentez simplement la ligne :

```
//final Game game = new Tuto1();
```

et ajoutez la ligne suivante juste après :

```
final AreaGame game = new Tuto1();
```

Rappelez-vous que le programme principal appelle en boucle la méthode `update` du jeu simulé (ici `Tuto1`), lequel appelle la méthode `update` de son aire courante, qui a son tour appelle la méthode `update` de chacun de ses acteurs. C'est ainsi que la simulation peut évoluer au cours du temps.

Prêts pour le grand saut ? lancez votre programme `Play`.

Si tout se passe bien ... une fenêtre toute vide s'affiche. Mais où est donc passé notre fantôme ?

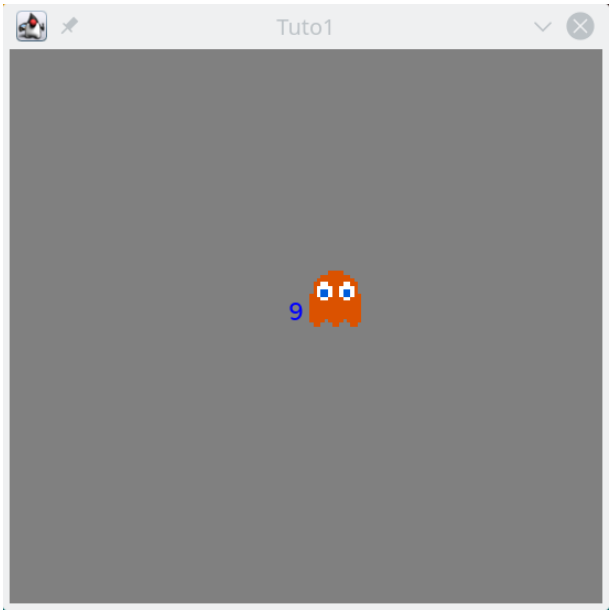
Réponse : il est dans l'aire qui ne s'affiche pas. En effet nous avons indiqué que notre aire courante (la seule simulée) était "`zelda/Ferme`" et notre fantôme se trouve dans l'aire nommée *"zelda/Village"*. Vous pouvez vous en assurez en changeant l'aire courante dans `Tuto1`. Vous devriez alors voir le niveau d'énergie du fantôme décroître au cours du temps pour se stabiliser à zéro.

4.7.4 Personnage principal

Nous nous intéressons à coder des jeux où l'on contrôle (via le clavier) un *personnage principal* au travers de ses pérégrinations sur des aires du jeu. Cet acteur aura un statut tout à fait particulier et l'on va considérer qu'il fait partie des éléments caractéristiques du jeu. Ajoutez à votre jeu `Tuto1`, un attribut

```
private SimpleGhost player;
```

qui remplira ce rôle. Dans la méthode `begin` de `Tuto1`, commencez par donner une valeur au personnage principal en créant un `SimpleGhost` positionné en `(18, 7)` et avec pour nom d'image *"ghost.1"*. Enregistrez ensuite le personnage principal dans l'aire courante (une fois qu'elle aura été désignée) et indiquez que la caméra doit désormais le suivre (souvenez vous que `setCurrentArea` retourne l'aire courante et que la méthode `setViewCandidate` permet de centrer la caméra sur un acteur particulier). Lorsque *"zelda/Ferme"* est l'aire de démarrage du jeu (celle choisie dans `begin`), vous devriez obtenir ceci au lancement du jeu :



Note : La couleur de fond de la fenêtre peut être changée dans le fichier `SwingWindow` du paquetage `window.swing` (dans `game-engine`), ligne 182.

4.7.5 Contrôles

Nous n'avons pour le moment guère de prise sur le personnage principal. Il est temps d'y remédier. Pour cela programmez dans votre classe `SimpleGhost` des méthodes `moveUp`, `moveDown`, `moveLeft` et `moveRight` permettant de le faire bouger d'une petite distance fixe (par exemple `0.05`), dans une direction donnée. Voici comment peut être codée `moveUp` par exemple :

```
setCurrentPosition(getPosition().add(0f, delta));
```

nous utilisons les méthodes `setCurrentPosition` et `getPosition` propres aux `Entity` et dont hérite `SimpleGhost` ainsi que la méthode `add` propre à la classe utilitaire fournie, `Vector`. Vous procéderez de façon analgue pour les autres méthodes.

Modifiez ensuite la méthode `update` de votre jeu `Tuto1`, de sorte à ce qu'elle soit réceptive à des événements clavier. Si l'utilisateur appuie sur « flèche-haut » du clavier, c'est la méthode `moveUp` du personnage principal qui est appelée. S'il appuie sur « flèche-bas » c'est `moveDown`, sur « flèche-gauche » c'est `moveLeft` et sur « flèche-droite » c'est `moveRight`.

Voici comment l'API fournie vous permet dans ce contexte de tester que la flèche-haut a été appuyée :

```
Keyboard keyboard = getWindow().getKeyboard() ;
Button key = keyboard.get(Keyboard.UP) ;
if(key.isDown())
{
  //...
}
```

(et l'on procède de façon analogue avec les valeurs `Keyboard.DOWN`, `Keyboard.LEFT` et `Keyboard.RIGHT`).

Si vous lancez à nouveau le jeu, vous aurez cependant l'impression que ces touches restent sans effet. C'est normal, car la caméra se recentre sur le personnage principal à chacun de ses déplacements et comme le fond d'écran est uni, vous n'avez pas l'impression de le voir bouger.

Pour rendre le déplacement perceptible, il nous faudrait par exemple un fond d'écran avec un décor et l'on verrait alors le personnage se mouvoir, par référence à ce décor. Une autre façon de faire est de le placer dans une aire où il y a un élément qui ne bouge pas. C'est le cas de notre aire `"zelda/Village"` où le second fantôme n'est pas contrôlable via le clavier et ne se déplace pas.

Pour finaliser l'exercice, programmer dans `Tuto1`, une méthode `void switchArea()` qui permet au personnage principal de transiter d'une aire à l'autre : s'il est dans l'aire `"zelda/Ferme"` il doit passer à celle intitulée `"zelda/Village"` et vice-versa. A chaque fois qu'il quitte une aire il doit y être désenregistré. Lorsqu'il rentre dans une aire, cette dernière doit devenir l'aire courante, le personnage doit y être enregistré et la caméra centrée sur lui. Lors de son passage dans une autre aire, le personnage principal verra son niveau d'énergie renforcé (méthode `strengthen()`). La transition d'une aire à l'autre devra se faire automatiquement dès que le fantôme devient faible (méthode `isWeak`).

Si vous avez fait les choses correctement, vous devriez voir le personnage principal (fantôme orange) s'afficher tout seul au lancement du jeu, puis son niveau d'énergie décroître progressivement jusqu'à atteindre zéro. A ce moment, il doit transiter vers l'aire où il y a le fantôme bleu. Dans cette aire, les flèches peuvent le faire bouger de façon visible (vous aurez l'impression que c'est le fantôme bleu qui bouge car le référentiel est toujours centré sur le fantôme orange).

Note : dans votre codage de la méthode `switchArea`, vous avez vraisemblablement appelé `setCurrentArea`. Expérimentez le fait de l'appeler avec `false` ou `true` comme second argument (vous devriez voir une incidence sur le niveau d'énergie du fantôme bleu). Avec `false` on retrouve l'aire dans l'état où on l'avait laissée et dans le second on la recrée depuis zéro. Vous pouvez enfin jouer avec le facteur d'échelle pour voir son incidence sur l'affichage du jeu.

Ce petit exercice clôt ce premier tutoriel. Il vous a, en principe, permis de vous familiarise avec la maquette pour la modélisation d'un jeu constitué de plusieurs aires ainsi que pour la modélisation d'acteurs simples. L'objectif du second tutoriel est de commencer à exploiter les grilles associées aux aires.

5 Tutoriel II

Ce second tutoriel revient sur l'usage des interfaces dans la maquette fournie. Il présente également plus en détail, la notion de grille associée à une aire de jeu et les acteurs spécifiques qui peuvent y prendre place. Comme pour le premier tutoriel, de petits exercices vous permettront d'utiliser par vous-même les concepts présentés.

5.1 Parlons un peu des interfaces

Maintenant que la notion d'interface a été introduite en cours, vous pouvez commencer à vous intéresser de plus près à leur usage dans la maquette fournie. L'interface `Game` modélise la notion abstraite de « jeu ». Elle est mise en œuvre plus finement au moyen des interfaces `Playable`, `Updatable` et `Drawable` que nous vous invitons à examiner (elles se trouvent dans le paquetage `engine`).

Question 2

Pourquoi à votre avis est-il préférable de déclarer la variable `game` du programme `Play`, comme une `Game` plutôt que comme un `AreaGame` ?

Elément de réponse : `Game` représente le concept de « jeu » du point de vue fonctionnel et abstrait. `AreaGame` n'est qu'une implémentation possible de ce concept. Si l'on déclare `game` comme une `AreaGame` (jeu sur grille), le programme principal `Play` voit de cet objet beaucoup plus que sa représentation fonctionnelle abstraite, « jeu ». Il voit tous ses détails d'*implémentation* en tant que jeu sur grille. Ce programme peut alors utiliser ces détails à loisir (et à mauvais escient) dans sa propre implémentation. Cela induit potentiellement des failles d'encapsulation fâcheuses. Qu'en est-il par exemple si `Play` décide de lancer un jeu qui n'est pas un jeu avec aires et qu'il a utilisé des méthodes spécifiques à `AreaGame` dans sa méthode `main` ?

Vous noterez donc que les interfaces sont un puissant outil d'encapsulation : l'aire, la grille et les acteurs ont besoin de se connaître mutuellement, ce qui implique de leur part d'ouvrir l'accès à certaines informations. Typiquement une aire de jeu doit accéder aux acteurs et les acteurs doivent savoir dans quel jeu ils évoluent (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.

Dans le même esprit, l'interface `Actor` permet de modéliser de façon minimaliste et abstraite les aspects fonctionnels d'un acteur dans un jeu, sans avoir à exposer forcément l'API de ses implémentations possibles. Examinez pour cela le contenu de `Actor` dans `engine.actor`. `Actor` modélise un acteur abstrait très simple. La classe `Entity` du même paquetage en est une implémentation possible dont vont dériver toutes sortes d'autres implémentations spécifiques. Pour protéger les codes d'éventuelles modifications dans les implémentations spécifiques il faut éviter d'exposer ces dernières. Manipuler tout acteur sous l'étiquette de `Actor` plutôt que sous celle d'une implémentation spécifique nous permet d'atteindre cet objectif.

FIG. 3 : Exemple d'une image de comportement avec la correspondance couleur-type

Pour compléter votre connaissance de la maquette, vous pouvez donc maintenant, revenir sur les liens `implements` des classes déjà présentées, comme `Area` ou `AreaGame` par exemple.
Poursuivons après cela l'exploration des abstractions fournies. Les aires dans notre premier jeu sont un peu ... sombres. Pour y remédier, il s'agit, dans ce qui suit, de lever le voile sur la notion de grille associée à une aire (`AreaBehavior`) ainsi que sur la classe `Background`. Cette dernière permettra d'apposer un visuel un peu moins sidéral sur les grilles en question.

5.2 Grille et cellules

Nous avons vu dans le tutoriel précédent que la classe `Area` a un attribut de type `AreaBehavior`. Cet attribut modélise une *grille* qui va conditionner les comportements de tout ce qui y prend place. Ouvrez le code de la classe `AreaBehavior`. Vous constaterez qu'elle a pour attribut un tableau de cellules (`Cell`), ainsi qu'une image (l'attribut `behaviorMap`). Le constructeur de `AreaBehavior` (et des ses sous-classes concrètes) est en effet amené à initialiser le tableau de cellules depuis une image couleur comme celle de la figure Figure 3, où chaque pixel représente une cellule et chaque couleur une nature ou un type différent de cellule. Nous expliquerons un peu plus loin comment est mise en place cette correspondance et comment elle doit être utilisée.
La classe `Cell` représente une cellule générique et chaque extension de `AreaBehavior`, qui sera spécifique à un jeu donné, devra redéfinir des extensions spécifiques de la classe `Cell`.

Note : Pour le moment, la grille et les cellules sont deux entités autonomes. Ce n'est pas la meilleure conception possible et nous y reviendrons dans le dernier volet de ce tutoriel.
Une `Cell` a un contenu (l'ensemble des entités du jeu occupant la cellule), mais pour l'heure nous ne nous y intéressons pas encore. Il suffit de noter qu'elle est caractérisée par ses coordonnées sur la grille (de type `DiscreteCoordinates`).

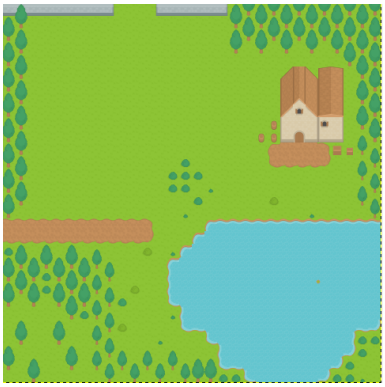
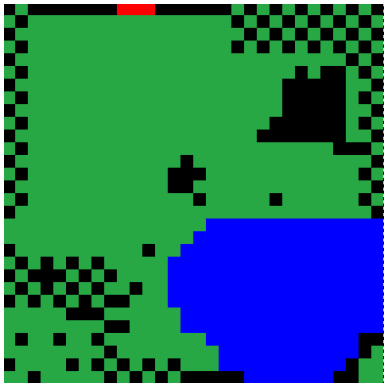


image de fond



« comportement » correspondant

Dans le constructeur de `AreaBehavior`, la ligne :

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

permet simplement d'aller lire une image depuis un nom de fichier donné.

A chaque aire de jeu concrète, sera bien entendu associée à une sous-classe concrète de `AreaBehavior`. Cette association ne sera pas forcément unique (il est imaginable qu'une aire puisse changer de grille associée en cours d'existence) et la maquette permet donc la modification de l'`AreaBehavior` associée à une `Area`. C'est ce qui explique la présence de la méthode *protégée* et finale `setBehavior(AreaBehavior ab)` dans la classe `Area`.

5.3 Acteur pour le « fond d'écran »

A chaque aire de jeu il est prévu d'assortir 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 de la classe `Area`, vous aurez, en utilisant les éléments de la maquette, à le coder comme ... un acteur. Cet acteur ressemble beaucoup plus dans l'esprit à celui que vous avez codé dans `Tuto1` : il s'agit simplement d'un dérivé de `Entity` tout comme `SimpleGhost`. Le code vous est fourni dans la classe `Background` du paquetage `engine.actor`. Commencez par y jeter un œil. 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 `Village.png` du dossier `src/main/resources/images/background/zelda` si le `getTitle()` retourne *`"zelda/Village"`*.

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. C'est la raison d'être des méthodes `getWidth()` et `getHeight()` de 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, comme le montre la figure 5.3.

Le répertoire `src/main/resources` du dossier `game-engine` fournit quelques fonds d'écran

dans les répertoires `src/main/resources/images/background` et les images de « comportement » associées dans `src/main/resources/images/behavior` (la correspondance s'établit au travers du nom). L'annexe 7 fournit également un outil permettant de créer des images `background` et `behavior` associées[7].

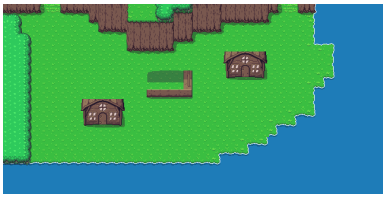
5.3.1 Exercice 1 (suite : ajout d'un fond d'écran)

`SimpleArea` est une sorte d'aire de jeu qui n'exploite pas vraiment la grille associée mais qui permet uniquement de tenir compte de sa taille. Cela va vous permettre d'introduire vos premiers acteurs « fond d'écran ».

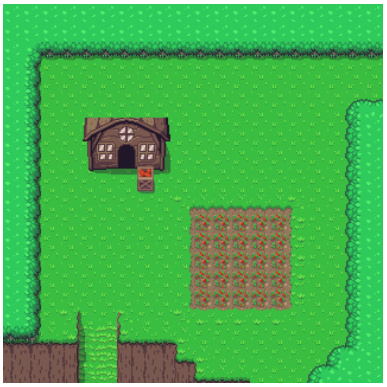
Complétez les aires spécifiques `Village` et `Ferme` du paquetage `ch.epfl.cs107.play.area.tuto1` pour faire en sorte que leur méthode `createArea` ajoute l'acteur « fond d'écran » qui leur est associé. Ceci se fait typiquement au moyen de l'instruction :

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

Si vous lancez à nouveau le jeu `Tuto1` au moyen de `Play`, vous devriez voir les fonds d'écran suivants s'afficher (en partie seulement en raison du facteur d'échelle) :



fond d'écran « village »



fond d'écran « ferme »

L'usage des flèches donne désormais l'impression de voir se déplacer notre fantôme rouge (comme la vue se recentre sur lui, le fond d'écran ne s'affiche pas toujours au même endroit). Notez que vous pouvez aussi enregistrer un acteur `Foreground` selon le même principe (testez l'effet d'un tel ajout pour comprendre ce qui peut le motiver).

C'est un bon début, mais il y a encore du travail ! On aimerait en effet que notre fantôme ne puisse plus se perdre dans le néant en sortant d'une aire. Il est souhaitable aussi que le visuel graphique de l'aire ait un impact : par exemple que le changement d'aire se fasse lorsque le fantôme transite par une zone dont le visuel est celui d'une porte/passage ou encore que l'on soit capable d'empêcher le fantôme de marcher sur un visuel de plan d'eau. Il s'agit maintenant pour vous d'apprendre à le faire au travers d'un nouvel exercice.

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

5.4 Exercice 2 : premier jeu sur grille

Le but de cet exercice est de créer une variante `Tuto2` de `Tuto1` où des grilles concrètes sont associées aux aires. Il s'agit d'une ébauche de RPG où notre fantôme circulera sur une grille qui lui dictera où il peut et ne peut pas aller. Vous travaillerez dans le paquetage `ch.epfl.cs107.play.tuto2` du dossier `tutos`.

En guise de mise en place, créez le jeu `Tuto2` dont le contenu sera pour le moment quasiment identique à celui de `Tuto1` (n'oubliez pas cependant d'adapter la méthode `getTitle` qui doit retourner *`"Tuto2"`*).

5.4.1 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 spécifique des cellules. Il vous est demandé pour cela de coder dans le paquetage `ch.epfl.cs107.play.tuto2.area`, une sous-classe `Tuto2Behavior` héritant de `AreaBehavior`.

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

Pour cela définissez dans `Tuto2Behavior` le type énuméré :

```
public enum Tuto2CellType {
  NULL(0, false),
  WALL(-16777216, false),       // #000000 RGB code of black
  IMPASSABLE(-8750470, false), // #7A7A7A, RGB color of gray
  INTERACT(-256, true),         // #FFFF00, RGB color of yellow
  DOOR(-195580, true),          // #FD0404, RGB color of red
  WALKABLE(-1, true),;          // #FFFFFF, RGB color of white

  final int type;
  final boolean isWalkable;

  Tuto2CellType(int type, boolean isWalkable){
    this.type = type;
    this.isWalkable = isWalkable;
  }
}
```

Ajoutez à ce type énuméré la méthode `static Tuto2CellType toType(int type)` retournant la valeur du type énuméré correspondant à l'entier `type`. Par exemple, `toType(-195580)` retournera la valeur `DOOR`. La valeur `NULL` sera retournée si `type` ne correspond à aucune valeur prévue pour le type énuméré.

Le type `Tuto2CellType` nous permettra d'interpréter la couleur rouge[8] comme une porte, la noire comme un mur, la grise comme une zone infranchissable (comme de l'eau par exemple) etc. Si vous ouvrez d'ailleurs le fichier `src/main/resources/behavior/zelda/Village.png`, l'image de « comportement » associée à l'aire *`"zelda/Village"`*, vous pourrez constater

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

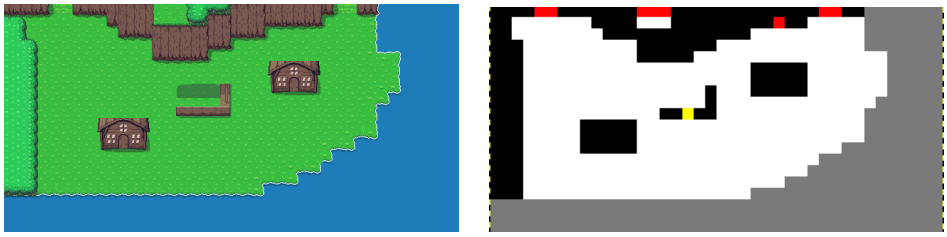


image de fond « comportement » correspondant

FIG. 4 : Aire « Village » et « behavior » correspondant

comment ce type énuméré permet de codifier le rôle de chacune des cellules de la grille (voir la figure 4).

L'idée est donc que si l'on associe à `Tuto2Behavior` une `imageBehavior` telle que celle de la partie droite de la figure 4, alors les cellules correspondant aux pixels noirs pourront être vues comme des murs sur lesquels il ne faut pas marcher, les cellules correspondant aux pixels rouges comme des points de passage exploitables pour transiter d'une aire à l'autre etc.

Il est donc nécessaire de définir de façon adaptée les cellules assorties à `Tuto2behavior` de sorte à permettre à cette grille de dicter des contraintes spécifiques en fonction de leur nature. Pour cela, au même niveau que `Tuto2Behavior`, définissez la sous-classe `Tuto2Cell` héritant de `Cell`. Une `Tuto2Cell` sera caractérisée par sa nature (de type `Tuto2CellType`).

Vous doterez `Tuto2Cell` d'un constructeur d'entête

```
Tuto2Cell(int x, int y, Tuto2CellType type)
```

(Note : ce constructeur pourra devenir privé lorque vous utiliserez les classes imbriquées).

Dotez enfin `Tuto2Behavior` d'un constructeur permettant d'initialiser la grille en la remplissant de `Tuto2Cell`. Pour trouver le type à associer à la `Tuto2Cell` de coordonnées `[x][y]` lors de sa construction, vous pourrez utiliser la tournure suivante :

```
Tuto2CellType cellType =
   Tuto2CellType.toType(getRGB(height-1-y, x));
```

Indication : Les valeurs d'un type énuméré sont retournés par la méthode `values()` (ici `Tuto2CellType.values()`) et il est bien sûr possible d'itérer dessus avec une itérations sur ensemble de valeur (`for (Type val : setOfType)`).

Les `Tuto2Cell` héritent de `Cell` mais doivent être concrètement instantiables. Vous considérerez qu'il est toujours possible d'en sortir. Pour le moment codez le fait qu'il est toujours possible d'y entrer (nous y reviendrons plus loin de sorte à ce que les conditions dépendent de la nature de la cellule).

Comme `Cell` implémente l'interface `Interactable`, le compilateur exigera de vous la définition des méthodes `isCellInteractable()` et `isViewInteractable()` vous pouvez leur faire retourner respectivement `true` et `false` (mais ça n'a pas de réelle importance à ce stade et nous y reviendrons). Pour la méthode `void acceptInteraction(AreaInteractionVisitor v, bool isCellInteraction)` également exigée par l'interface `Interactable`, laissez simplement un corps vide pour le moment.

5.4.2 Aires de jeu spécifiques

Dans le paquetage `ch.epfl.cs107.play.tuto2.area`, créez une classe `Tuto2Area` représentant nos premières aires de jeu associées à des grilles spécifiques. Cette classe sera similaire à la classe fournie `SimpleArea` aux différences près suivantes :

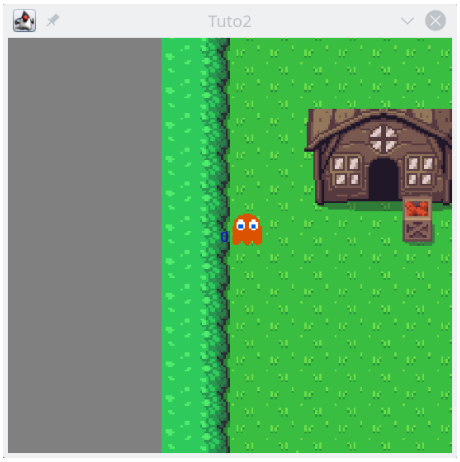
- elle n'a pas besoin de redéfinir les méthodes `getWidth` et `getHeight` car celles héritées de `Area` lui conviennent bien (ses largeur et hauteur sont celles de la grille associée) ;
- sa méthode `begin` doit lui associer une grille de type `Tuto2Behavior` :

```
setBehavior(new Tuto2Behavior(window, getTitle()));
```

Créez enfin dans un paquetage `ch.epfl.cs107.play.tuto2.area.maps`, les aires spécifiques `Village` et `Ferme`, quasiment identiques à leurs versions dans `ch.epfl.cs107.tuto1.area.maps` mais héritant cette fois de `Tuto2Area`. Il s'agit là de deux aires spécifiques auxquelles nous savons associer une grille spécifique de type `Tuto2Behavior`.

5.4.3 `Tuto2`

Reprenez votre ébauche de jeu `Tuto2` et faites en sorte que les deux aires qui le constituent soient les aires `Village` et `Ferme` de `ch.epfl.cs107.play.tuto2.area.maps` (et non pas celles de `ch.epfl.cs107.play.tuto1.area.maps` !). Par défaut, l'aire courante est donc `Ferme` de `ch.epfl.cs107.play.tuto2.area.maps`. Lancez le jeu `Tuto2`. Si tout se passe bien, vous devriez voir s'afficher (partiellement) l'aire `Ferme` :



Commentez la création et l'enregistrement de l'acteur dans la méthode `begin` et faites en sorte que la méthode `update` ne contienne que l'appel à la méthode `update` de la super-classe (nous allons en effet devoir changer le type d'acteur ainsi que sa façon d'évoluer dans la suite de cet exercice).

Le petit fantôme qui nous a servi d'acteur dans le jeu `Tuto1` est en réalité uniquement une image dotée d'une position. Afin qu'il puisse tenir compte de la présence de la grille pour se déplacer correctement, il est nécessaire de le sophistiquer un peu. Il faut pour cela avoir recours à des types plus évolués d'acteurs offerts par la maquette qui vous sont présentés maintenant.

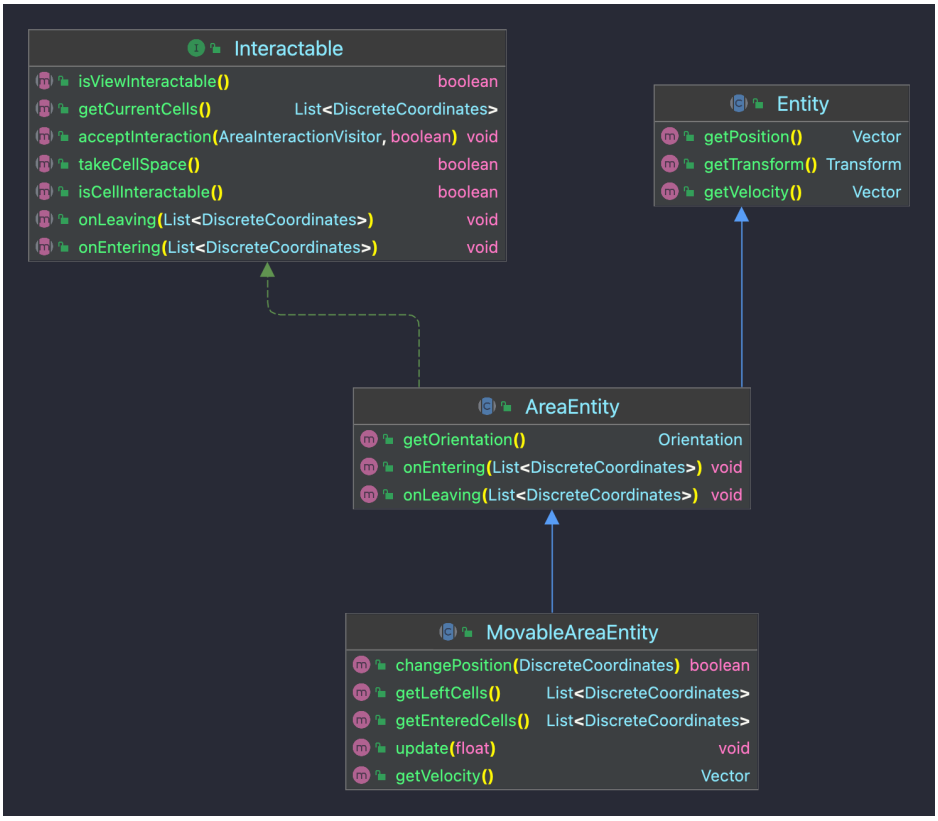


FIG. 5 : Hiérarchie d'acteurs de jeux sur grille

5.5 Acteurs de jeux sur 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 voir comment cette modélisation est étendue dans la maquette pour incorporer les spécificités des acteurs *évoluant sur une grille*. Les classes décrites ci-dessous sont dans le paquetage `ch.epfl.cs107.play.areagame.actor`.

5.5.1 La classe `AreaEntity`

La classe abstraite `AreaEntity` permet de modéliser les acteurs appartenant à une aire de jeu grillagée. Leur principale spécificité est qu'*ils occupent 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 ont également une *orientation*, qui permettra de les dessiner différemment selon vers où ils 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.
Le type `Orientation` est fourni dans le paquetage `math` du dossier `game-engine`.
Vous noterez que la méthode `void setCurrentPosition(Vector v)` héritée de `Entity` est redéfinie de sorte à s'adapter au fait que l'on travaille désormais sur une grille (pour permettre aussi de mettre à jour sa cellule principale) 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.

Enfin, quelques « getters-setters » utiles sont également fournis. Vous remarquerez les accès protégés choisis pour les méthodes présentant un danger potentiel pour l'encapsulation.

Une méthode importante des `AreaEntity` est `setOwnerArea` qui permet d'indiquer à un acteur quelle est son aire d'appartenance.

5.5.2 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. C'est le rôle de l'attribut `entities` de la classe `Cell` (codé au moyen du type prédifini `Set`, voir l'annexe 7 sur les ensembles). Cet ensemble n'est pas modélisé comme un ensemble d' `Actor` et nous allons voir pourquoi.

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 le codage de cet ensemble fait appel à de nouvelles abstractions. Il s'agit concrètement d'entités capables d'interagir.

La maquette offre pour cela les interfaces suivantes, placées dans le sous-paquetage `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 partons de l'hypothèse que toute entité de grille (`AreaEntity`) doit se définir en tant qu'objet sujet à des interactions. C'est pour cette raison que `AreaEntity` implémente l'interface `Interactable` !

L'attribut `entities` d'une `Cell` n'est donc pas un ensemble d'`Actor`, mais un ensemble de `Interactable`. Elle contient notamment une méthode protégée `enter` permettant l'ajout d'un `Interactable` donné à cet ensemble et d'une méthode protégée `leave` permettant le retrait d'un `Interactable` de cet ensemble.

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 d'examiner le contenu de l'interface `Interactor` (uniquement nécessaire pour prendre en charge les interactions entre acteurs, ce que nous ferons à l'étape suivante). Examinez le code de l'interface `Interactable` permettant de modéliser une entité réceptives aux interactions.

La conception proposée modélise le fait qu'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()`.

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

Notez enfin que `Cell` implémente également `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 retourne une liste dont l'unique élément est consitité des coordonnées de la cellule.

Nous reviendrons aux autres méthodes de l'interface `Interactable` un peu plus loin.

5.5.3 La classe `MovableAreaEntity`

Certains de nos acteurs sur grilles seront naturellement en mouvement. A la différence de notre acteur `SimpleGhost`, ils devront se déplacer en tenant compte de la présence de la grille.

La classe abstraite `MovableAreaEntity` du sous-paquetage `areagame.actor`, dérivant de `AreaEntity` permet de modéliser ce type d'acteurs. Sa caractéristique principale est la présence d'une méthode `move` permettant à l'acteur d'effectuer un déplacement continu.

Le paramètre `framesForMove` est le nombre de frames (étapes) choisie pour mettre en oeuvre la continuité du déplacement. Dans les implémentations concrètes de jeux, nous introduirons la possibilité d'assortir à chaque étape (frame) une représentation graphique différente, ce qui permettra d'animer le déplacement.

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.

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.

Le rôle de la méthode `protected boolean move(int framesForMove, int frame)` est de décider si un déplacement peut avoir lieu et si oui de l'initier. Elle retourne true pour indiquer qu'un déplacement est en cours.

5.6 L'aire et sa grille dictent leurs conditions

Il est temps maintenant de se pencher sur les modalités permettant à la grille et à l'aire d'imposer leurs conditions sur le placement et le déplacement des entités qui y prennent place. Rappelons que chaque aire est dotée d'une grille. Pour bien faire, l'ajout ou le retrait d'un acteur de l'aire doit tenir compte du *veto* potentiel de la grille. Par exemple, cette dernière 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 et donc ne pas pouvoir être ajouté comme nouvel acteur dans l'aire. De même, la méthode `move` des `MovableAreaEntity` doit permettre à l'aire ou à la grille d'exprimer des contraintes sur le déplacement. Elle devrait typiquement au moins s'opposer à ce qu'une entité sorte de la grille. Pour dicter ces conditions, nous partons de l'idée que la cellule peut dicter ses conditions, lesquelles impacteront les décisions de la grille à laquelle elle appartient et qui à leur tour pourront impacter les décisions de l'aire associée à la grille.

5.6.1 Conditions dictées par les cellules et la grille

Afin de permettre à une cellule d'exprimer un droit de regard sur les placements/déplacements, la classe `Cell` contient les méthodes abstraites 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.

Ces méthodes ainsi que la connaissance qu'a la grille de ses dimensions, lui permettront de conditionner les déplacements et positionnement des `Interactable` qui peuvent habiter ses cellules.

5.7 Exercice 2 (suite : ajout d'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'œuvre, vous allez compléter le codage du jeu `Tuto2`.

Afin que la grille `Tuto2Behavior` puisse dicter ses conditions aux acteurs qui s'y trouvent, les cellules assorties à `Tuto2Cell` seront caractérisées par le fait qu'elles :

- ne permettent d'entrer dans une cellule que si son attribut `isWalkable` vaut true (définir `canEnter()`en conséquence) ;
- acceptent les interactions de contact (définir `isCellInteractable()` proprement) ;
- n'acceptent pas les interactions à distance (définir `isViewInteractable()` proprement) ;
- peuvent toujours être quittées (définir `canLeave()` proprement).

5.7.1 Acteurs spécifiques

Il s'agit maintenant de créer un acteur de jeu sur grille, `GhostPlayer` dans `ch.epfl.cs107.play.tuto2.actor`. Ce type d'acteurs héritera donc de `MovableAreaEntity`. Il acceptera tout type d'interaction et sera non traversable.Il aura par ailleurs le même comportement que l'acteur `SimpleGhost` (points de vie, méthode `isWeak`, et transition d'une aire à l'autre quand le nombre de points de vie devient nul)[9].

Il sera aussi doté de méthodes lui permettant de :

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

  ```
  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 absolue : setCurrentPosition(position.toVector()) ;
  3. et se mettre en situation d'immobilité (`resetMotion`).

- et de quitter l'aire à laquelle il appartient (s'y désenregistrer).

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

```
public GhostPlayer(Area owner, Orientation orientation,
   DiscreteCoordinates coordinates, String sprite)
```

`coordinates` est la case occupée par le fantôme à sa création.

Pour pouvoir être instancié, un `GhostPlayer` 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 `GhostPlayer` implémente l'algorithme suivant :

1. démarrage du déplacement ou orientation en fonction des touches enfoncées par l'utilisateur ;
2. appel à la méthode `update` de la super-classe (pour effectivement effectuer le déplacement initié s'il y a lieu).

Pour l'étape 1 de l'algorithme ci-dessus, l'algorithme est le suivant :

- si le bouton correspondant au bouton `Keyboard.LEFT` est enfoncé (`isPressed`) alors si l'acteur est orienté à gauche, on initie le déplacement vers la gauche (appel à `move`). Gérer cette interaction directement dans la méthode `update` de l'acteur (ceci est maintenant possible car une `AreaEntity` a connaisance de l'aire à laquelle elle appartient et à donc accès à sa méthode `getKeyboard()`).

[9] reprenez directement ce qu vous avez fait à ce sujet dans la classe `SimpleGhost`

- 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.

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

Enfin, comme `Interactable`, `GhostPlayer` devra aussi fournir une implémentation vide de `acceptInteraction` pour le moment (comme `Tuto2Cell`).

> Vous noterez que seuls les acteurs des jeux de grilles ont accès à l'aire à laquelle ils appartiennent.

5.7.2 Placement du personnage

Compléter `Tuto2` de sorte à ce que ce jeu soit caractérisé par un personnage de type `GhostPlayer`. Le personnage sera créé au démarrage du jeu, avec pour orientation `Orientation.DOWN`. Il sera enregistré dans l'aire courante et c'est sur lui que sera centrée la caméra. Sa méthode `update` implémentera simplement le fait que si le personnage est trop faible, il transitera à l'aire suivante. Ceci ce fera exactement à l'image de que qui était fait dans `Tuto1` (s'il était dans `Village` il passe dans `Ferme` et vice-versa).

> `update` n'a plus besoin de gérer les interactions clavier, qui sont gérée directement dans le `update` du personnage.

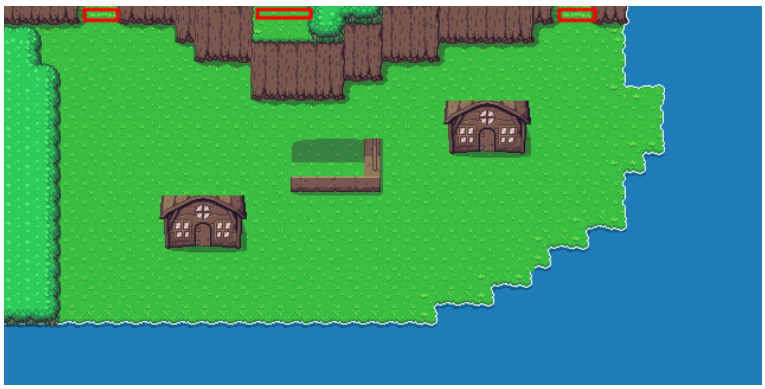
Vous utiliserez `(2,10)` comme coordonnées de départ dans `Ferme` et `(5,15)` dans `Village` et ce sont ces mêmes coordonnées qui seront utilisées comme coordonnées de départ à chaque fois que l'acteur rebascule vers ces aires. Vous pourrez utiliser `13.f` comme facteur d'échelle et il fait sens que cette valeur soit une constante statique finale spécifique au jeu, c'est à dire `Tuto2`.

5.7.3 Validation de la solution codée

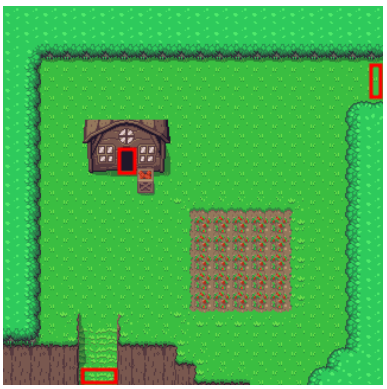
Vous vérifierez que `GhostPlayer` :

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 (toutes les zones correspondant à du noir ou gris dans l'image de comportement associée, typiquement l'eau ou les barrières ne peut pas être franchies)
3. est bien suivi par la caméra lors de ses déplacements ;
4. peut transiter correctement de l'aire `Village` à l'aire `Fermes` et vice-versa. Pour le moment il le fera uniquement en fonction de ses points de vie.

Il serait naturel que le personnage transite d'une zone à l'autre plutôt en passant par des zones correspondant à des portes (voir la figure 6).Pour ce faire, et pour compléter l'outillage nécessaire à la création de jeux, il faut pouvoir modéliser proprement les interactions qui peuvent avoir lieu entre acteurs. C'est le thème du dernier tutoriel.



portes dans ”zelda/Village”



portes dans ”zelda/Ferme”

Fig. 6 : Description des portes

6 Tutoriel III

Ce troisième et dernier tutoriel présente les outils de la maquette dédiés à coder des *interactions entre acteurs*. Au préalable, vous pourrez revisiter votre conception en faisant bon usage des classes imbriquées. Quelques classes utilitaires, qui vous seront utiles pour aborder le mini-projet, sont également présentées en fin de tutoriel.

Il n'y aura pas d'exercice à proprement parler, les concepts présentés seront directement exercés dans la première partie du projet.

6.1 Un peu de « refactoring » à l'aide des classes imbriquées (facultatif)

Maintenant que les classes imbriquées ont été présentées en cours, vous pouvez, si vous le souhaitez, améliorer la conception de la maquette fournie en faisant en sorte que la notion de cellule soit indissociable de celle de grille. La classe `Cell` deviendrait donc une classe publique imbriquée dans la classe `AreaBehavior`. Il faudrait alors aussi faire en sorte que `Tuto1Cell` et `Tuto2Cell` deviennent des sous-classes imbriquées de `Tuto1Behavior` et `Tuto2Behavior`. Ceci permettra d'améliorer l'encapsulation de la classe `Cell` : ses méthodes `cellInteractionOf` et `viewInteractionOf` peuvent en effet ainsi devenir privées, car elles ne sont en principe pas utiles en dehors des grilles. Les constructeurs peuvent devenir protégés.

Les modèles de cellules en elles-mêmes restent des classes *publiques*, parce que pour gérer les interactions que les acteurs pourront avoir avec les cellules, ils devront pouvoir y accéder.

6.2 Corrigé du tutoriel

Pour accéder au corrigé du tutoriel il vous suffit d'installer un nouveau projet IntelliJ avec le matériel fourni dans l'archive : tuto-solution-2025.zip Pour installer le projet sur IntelliJ :

1. dezzipez cette archive dans un répertoire de votre choix ;
2. ouvrez le répertoire `TUTO-SOL-2025` ;
3. supprimez l'archive `solution.zip`.

Pour Eclipse, une fois l'archive dezippée créez le projet en utilisant l'option `"From existing sources"` et en indiquant le répertoire hôte comme racine du projet.
La solution des tutoriels se trouve dans le paquetage `ch.epfl.cs107.play` du dossier `tutos`
La nouvelle version de `AreaBehavior` imbrique la classe `Cell`. La solution fournie se base sur cette conception.

Pour démarrer le projet, vous vous inspirerez de la correction du tutoriel 2. Prenez-en connaissance.

6.3 Interactions entre acteurs

Dans notre jeu précédent, il aurait été naturel de permettre à notre personnage principal de transiter d'une aire à l'autre via les endroits avec un visuel de « passage » (pixel rouges). Une façon simple de faire aurait été justement d'utiliser la couleur des pixels de l'image associée à la grille pour donner un comportement spécifique au personnage en fonction de cette couleur. Il n'est cependant pas très bon de procéder ainsi pour plusieurs raisons :

- cela implique que la grille doit communiquer aux personnages des informations spécifiques (par exemple fournir une méthode `boolean isDoor(int i, int j)` permettant de savoir si une cellule donnée correspond à un pixel rouge (pas suffisamment général : que se passe t-il si la couleur rouge doit être interprétée autrement à un autre niveau du jeu ?) ;
- il n'est pas certain que nous souhaitions forcément exploiter toutes les cellules correspondant à un pixel rouge comme des portes dans nos jeux ;
- un endroit avec un visuel de « passage » peut correspondre à différents types de portes (on peut par exemple imaginer avoir des portes s'ouvrant avec une clés, d'autres que l'on peut passer sans conditions etc.).

De ce fait, il est préférable de plutôt créer un *acteur* `Door` à placer (en général) sur les zones rouges (mais pas forcément toutes). L'interaction doit alors se faire plutôt entre deux acteurs (un acteur « porte/passage » et un acteur « personnage »). Il s'agit donc maintenant d'étudier les composants de la maquette permettant de gérer des *interactions entre acteurs*.

6.3.1 Les Interactor

Supposons donc que l'on souhaite créer un jeu où un personnage peut interagir avec un acteur « porte » et un acteur « touffe d'herbe » au sens où il pourra traverser la porte et couper la touffe d'herbe. Le personnage doit jouer un rôle plus actif en exprimant s'il souhaite une interaction ou pas (il n'est par exemple pas forcé de couper l'herbe). Il s'agira donc d'une entité qui *demande* une interaction. Cette catégorie particulière d'acteurs, demandeur d'interaction, est modélisable dans la maquette au moyen de l'interface `Interactor`. Ouvrez l'interface `Interactor`. Vous constaterez que cette dernière permet 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* ;
- qui indique avec une autre méthode `boolean wantsViewInteraction()` s'il demande une interaction *distante* ;
- et qui permet d'interagir avec un `Interactable` au moyen de la méthode `void interactWith(Interactable, boolean isCellInteraction)`. Le second paramètre permet de préciser le mode d'interaction voulu : par contact (paramètre valant `true`) ou à distance `false`).

Voyons maintenant comment ce type particulier d'acteurs intervient dans la simulation. Jusqu'ici, nous nous sommes préoccupés que de quelques lignes dans la méthode `update` d'une aire de jeu (`Area`). Consultez à nouveau le code de cette méthode et observez sa méthode `update`. Vous y verrez qu'après la boucle des mises à jour des acteurs :

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

a lieu la gestion des interactions à proprement parler :

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

6.3.2 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 personnage d'un jeu 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. C'est la raison pour laquelle la classe `Area` a un attribut `interactors` consignant tous les acteurs de types `Interactor`. Si vous examinez de plus près sa méthode `addActor`, vous verrez qu'elle a aussi 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` est 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.

6.3.3 Gestion des interactions au niveau de la grille

L'idée est donc qu'il incombe au final à la grille de mettre en place les mécanismes d'interaction. C'est la raison pour laquelle la classe `AreaBehavior` est dotée des méthodes :

- `public void cellInteractionOf(Interactor interactor)` : qui gère toutes les interactions de contact entre `interactor` et les `Interactable` aux mêmes positions que celles qu'il occupe. Cette méthode parcourt toutes les cellules aux positions `interactor.getCurrentCells()` et leur appliquer une méthode `cellInteractionOf(interactor)` spécifique aux `Cell`.

- `public void viewInteractionOf(Interactor interactor)` : qui gère toutes les interactions à distance entre `interactor` et les `Interactable` de son champs de vision. Cette méthode parcourt toutes les cellules aux positions `interactor.getFieldOfViewsCells()` et leur applique une méthode `viewInteractionOf(interactor)`spécifique aux `Cell`.

Ces deux méthodes permettent à l'`Interactor` (en paramètre de ces deux méthodes) d'être à l'écoute de la grille. Elles exigent la présence des méthodes suivantes dans `Cell` :

- `private void cellInteractionOf(Interactor interactor)`

- `private void viewInteractionOf(Interactor interactor)`

Voici comment se présente le code de la première de ces méthodes :

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

où `entities` représente l'ensemble de `Interactable` répertoriés dans la cellule. La seconde méthode est codée dans le même esprit.

6.4 Interactions génériques

Nous voici au coeur du sujet, comment coder concrètement la méthode

```
void interactWith(Interactable other, boolean
   isCellInteraction);
```

pour un `Interactor` donné ?
Plaçons nous dans un contexte plus général que celui des ébauches de jeux déjà codée et supposons que nous ayons à coder un personnage principal interagissant avec d'autres acteurs. Appelons-le `MyPlayer`. Ce dernier va typiquement être un `Interactor` ; c'est-à-dire une entité qui invite à des interactions. Comment pourrait-on a priori définir sa méthode spécifique `void interactWith(Interactable other, boolean isCellInteraction)` de sorte à lui permettre d'interagir avec des acteurs `Door` (porte) et `Grass` (touffe d'herbe)?
La façon triviale de procéder serait de recourir à des *tests de type* :

```
void interactWith(Interactable other, boolean isCellInteraction){
  if (other instanceof Grass && !isCellInteraction) // interaction à
     distance avec l'herbe ...
  if (other instanceof Door && isCelInteraction) //interaction de contact
     avec la porte ...
}
```

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 du jeu 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[10] :

```
  /* gère les interaction de MyPlayer avec tous les acteurs */
  class MyPlayerHandler {
  public void interactWith(Door door, boolean isCellInteraction) {
    // fait en sorte que la porte soit passée par l'acteur
  }
  public void interactWith(Grass grass, boolean isCellInteraction){
    // fait en sorte que l'herbe soit coupée
  }
}
```

Ce gestionnaire est spécifique à `MyPlayer`, il serait dans notre cas codé comme classe privée interne de cette classe.
Le classe `MyPlayer` aurait comme attribut son gestionnaire d'interaction :

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

```
private final MyGamePlayerHandler handler;
```

et une méthode générique :

```
  /* demande à other d'accepter d'avoir ses interactions
      avec MyPlayer gérées par handler
  */
 public void interactWith(Interactable other, boolean isCellInteraction) {
        other.acceptInteraction(handler, isCellInteraction);
 }
```

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

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

Cette solution offre l'avantage de pouvoir coder une méthode unique très générale dans les `Interactor`, la méthode `interactWith(Interactable, boolean isCellInteraction)`.
Un seul bémol encore, l'argument de `acceptInteraction` dans `Grass` 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 de la figure 7.

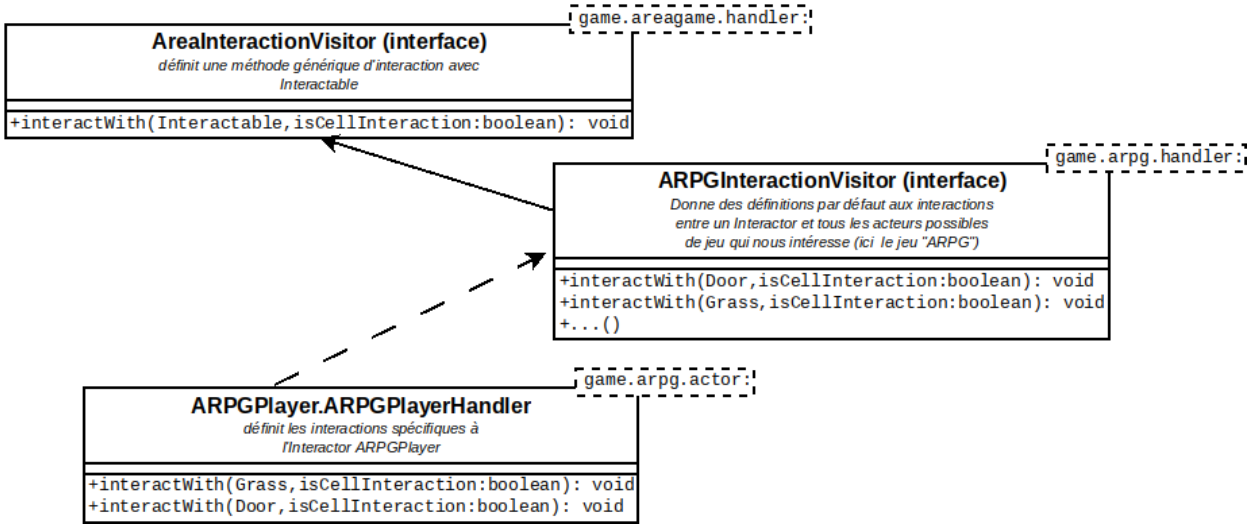


FIG. 7 : Schéma de classes pour la mise en place des interactions

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

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

L'interface `AreaInteractionVisitor` modélise un gestionnaire d'interaction générique et pour lequel on peut imaginer une implémentation par défaut qui est fournie dans le paquetage `areagame.handler`. La méthode `acceptInteraction` de `Grass` (ou `Door`) s'écrirait alors simplement :

```
 public void acceptInteraction(AreaInteractionVisitor v,
                               boolean isCellInteraction) {
        ((MyGameInteractionVisitor)v).interactWith(this, isCellInteraction);
}
```

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 `Grass` participe.

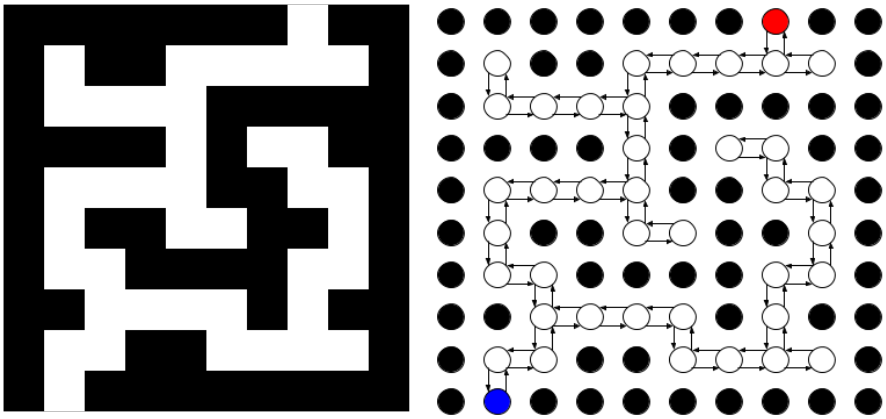


FIG. 8 : Exemple d'une image de comportement représentant un labyrinthe (parois en noir) et un graphe représentant les connexions entre les cellules praticables (en blanc)

On indique ainsi que la touffe d'herbe accepte de voir ses interactions gérées par le gestionnaire d'interaction spécifique `MyGameInteractionVisitor` et dont `MyPlayerHandler` est une réalisation concrète. Ce gestionnaire prévoit que tout `Interactor` peut avoir des interactions avec chaque acteur du jeu concerné. L'ajout d'un nouvel acteur implique de retoucher le gestionnaire `MyGameInteractionVisitor` 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).

La figure 7 résume graphiquement le schéma de conception suggéré pour gérer les interactions entre acteurs. La seconde partie du projet vous permettra de mettre en oeuvre concrètement ce schéma. Il vous sera notamment donné des indications précises sur où placer les classes évoquées dans le schéma présenté et comment les coder.

Les composants de la maquette ci-dessous sont utiles pour le codage du projet ou d'extensions. Survolez ce matériel rapidement pour le moment pour prendre connaissance de son existence et revenez-y en fonction de vos besoins pour le projet.

6.5 Classe `AreaGraph`

La classe `AreaGraph` du paquetage `ch.epfl.cs107.play.areagame` permet d'associer un graphe connecté à une grille de jeu. Ceci peut être utilisé pour simuler un début d'intelligence artificielle pour le déplacement d'acteurs (acteurs qui se déplacent en suivant un chemin).

La classe `AreaGraph` offre notamment la méthode :

```
Queue<Orientation> shortestPath(DiscreteCoordinates from,
   DiscreteCoordinates to)
```

qui permet de trouver le plus court chemin entre un point de départ et un point de destination dans le graphe associé à une grille. Ce chemin est décrit comme une « file » https://fr.wikipedia.org/wiki/File_(structure_de_donn%C3%A9es) d'`Orientation`. Il s'agit de la séquence d'orientations à adopter pour arriver à la case `to` en partant de la case `from`. Les files sont implémentées en Java au moyen du type `Queue` (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Queue.html).

Supposons qu'un acteur se déplace sur une grille associée à un graphe `graph`. Si cet acteur occupe la cellule de coordonnées `start` et qu'il veut se rendre à la cellule de coordonnées `stop`, l'orientation `nextOrientation` qu'il doit adopter est donnée par :

```
Queue<Orientation> path = graph.shortestPath(start, stop);
Orientation nextOrientation = path.poll();
```

La méthode `poll` des `Queue` permet d'extraire et de supprimer la tête de file.

6.6 Classes `RPGSprite` et `Animation`

Vous avez utilisé la classe `Sprite` de manière simple pour représenter graphiquement le personnage principal du jeu. D'autres classes sont mises à disposition pour affiner la représentation graphique des objets.

Il est en effet possible d'utiliser la classe `RPGSprite` dérivant de `Sprite` et qui associe une profondeur à l'image. Il peut être intéressant de jouer de la profondeur pour par exemple placer le personnage en avant ou en arrière d'un objet par exemple.

Par ailleurs un `Sprite` ou `RPGSprite` ne correspond pas forcément une représentation unique.

Une image complexe, peut être constituées de plusieurs `Sprite`. Par exemple, celle de la figure 9, est constituée de 4x4 petits `Sprite` de taille 16x32. Elle peut être découpée en ensembles permettant des vues différentes sur le personnages ou des animations

Fig. 9 : Exemple d'image servant de base à des animations

Par exemple la ligne du haut qui permet d'animer des déplacements vers le bas peut s'obtenir ainsi :

```
spritesDOWN[i] = new Sprite("zelda/player", 1, 2, this, new
   RegionOfInterest(i*16, 0, 16, 32));
```

pour `i` allant de 0 à 3.

Le concept d'animation, offert par la classe `Animation` du paquetage `engine.actor` de `game-engine`, est caractérisé par un ensemble de `Sprite` à afficher tour à tour. Sa méthode `update` permet de choisir quel élément/« frame » de l'ensemble est l'élément courant. C'est

celui qui sera affiché si l'on appelle la méthode `draw` de l'animation. Il est possible d'agir sur la vitesse à laquelle on change de frame lors des appel à `update`, au moyen des attributs `frameDuration` et `speedfactor` (et des méthodes associées).

Au personnage principal par exemple, peuvent donc êtres associés 4 animations permettant de l'animer lorsqu'il se déplace vers le haut, le bas, la gauche ou la droite. Dessiner le personnage revient donc à dessiner les 4 animations qui lui sont associées.

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.

Vous noterez que la classe `RPGSprite` offre quelques méthodes utiles pour l'extraction de `Sprite` d'une image complexe ou la création d'animations à partir de tableaux de `Sprite`, en particulier, `extractSprites()` et `createAnimations()`. Ainsi si l'on veut créer des animations correspondant aux orientations haut, bas, gauche droit d'un personnage, on pourra faire quelque chose comme ceci :

```
Sprite[][] sprites = RPGSprite.extractSprites("zelda/player",
   4, 1, 2,
this, 16, 32, new Orientation[] {Orientation.DOWN,
   Orientation.RIGHT, Orientation.UP, Orientation.LEFT});
// crée un tableau de 4 animation
  Animation[] animations =
     Animation.createAnimations(ANIMATION_DURATION/2, sprites);
```

où `ANIMATION_DURATION` est le nombre de frames utilisé pour le déplacement (ici on passe d'une animation à l'autre tous les deux pas de déplacement).

Vous noterez que la maquette fournit une abstraction de plus haut niveau qui simplifie cependant l'utilisation des animations attachées à une entité orientée, à savoir la classe `OrientedAnimation`.

6.7 Les signaux

L'interface `Signal` est fournie dans le paquetage `ch.epfl.cs107.play.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 `ch.epfl.cs107.play.signal`. Nous vous invitons également à examiner dans les grandes lignes les différents types de signaux fournis dans le même paquetage. Vous noterez que l'interface `Logic` offre en particulier deux constantes de type `Logic` (oui Java permet les définitions récursives !) : la constante `TRUE` et la constante `FALSE`. Voici quelques explications sur la tournure :

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

```
};
```

(en particulier a t'on le droit d'instancier une interface ?)

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

Il est possible par exemple de coder des acteurs dont le comportement dépend de signaux : par exemple un acteur « porte » qui serait ouvert ou fermé selon qu'un acteur "clé" a été ramassé ou pas. L'acteur « clé » serait un signal (ON lorsque ramassé par le joueur et OFF sinon) et la porte aurait comme attribut la clé conditionnant son ouverture.

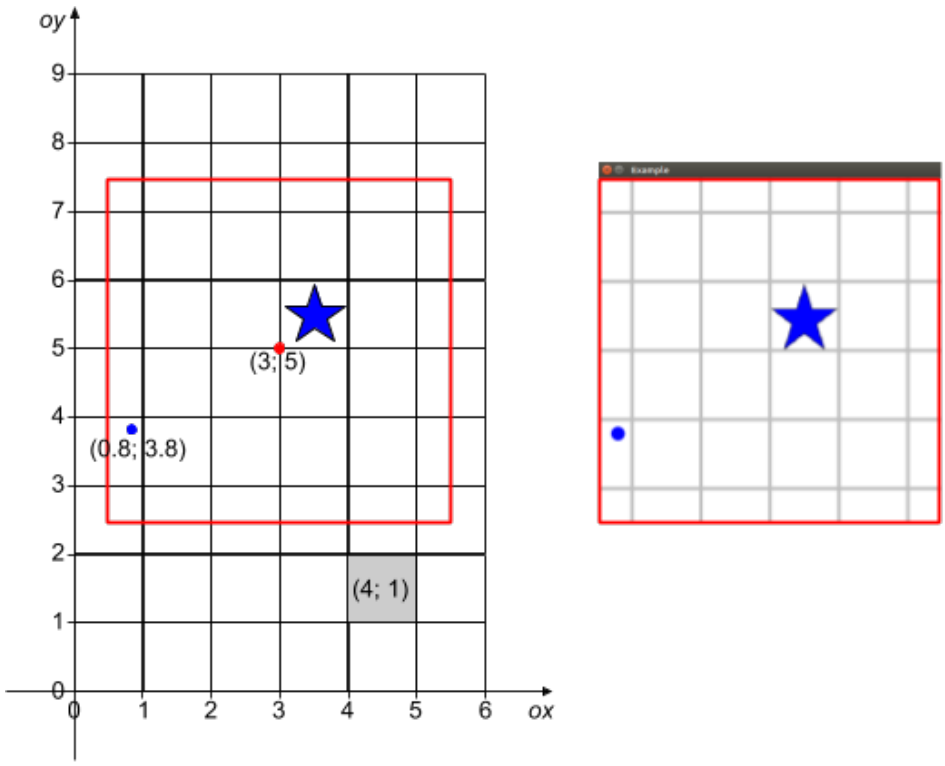


FIG. 10 : La vue sur une partie ciblée de la grille s'obtient par une transformation affine de la fenêtre (ici une simple translation)

7 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 `ch.epfl.cs107.play.engine.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 œil 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 leurs 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 Libre à vous d'utiliser d'autres images, qu'elles soient de votre création ou collectées sur la toile. Il est alors nécessaire 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) :

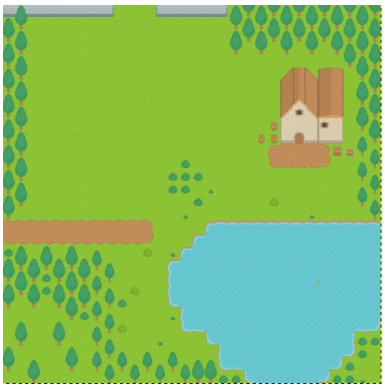
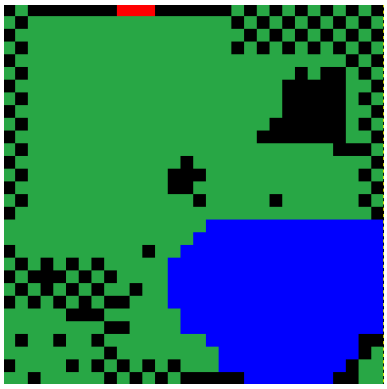


image de fond



« comportement » correspondant

Nous vous fournissons quelques exemples dans le fichier de ressources `src/main/resources` 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 par Bastien Chatelain et complété par Sami Abuzakuk (assistants du cours) : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/LevelEditor.zip [Lien sur une archive de projet eclipse]