CS-107 : Mini-projet 2
Jeux sur grille : « Super Pacman »

Q. Juppet, J. Sam, B. Jobstmann

Version 1.6

Table des matières

1 Présentation

> Ce document utilise des couleurs et contient des liens cliquables (textes soulignés). Il est préférable de le visualiser en format numérique.

Vous vous êtes familiarisés ces dernières semaines avec les fondamentaux d'un petit moteur de jeux adhoc (voir tutoriel) vous permettant de créer des jeux sur grille en deux dimensions. L'ébauche simple obtenue s'apparente à ce que l'on peut trouver dans un jeu de type RPG. Le but de ce mini-projet est d'en tirer parti pour créer des déclinaisons concrètes d'un autre type de jeu. Le but étant, entre autres, de démontrer qu'un outillage pensé au bon niveau d'abstraction peut être réutilisé dans des contextes très divers. Le jeu de base qu'il vous sera demandé de créer sera une variation du célèbre Pacman incluant des composants de jeux d'énigmes[1]. La figure 1 montre un exemple de l'ébauche de base[2] que vous pourrez enrichir ensuite à votre guise, au gré de votre fantaisie et imagination.

Outre son aspect ludique, ce mini-projet vous permettra de mettre en pratique de façon naturelle les concepts fondamentaux de l'orienté-objet. Il vous permettra d'expérimenter le fait qu'une conception située à un niveau d'abstraction adéquat permet de produire des programmes facilement extensibles et adaptables à différents contextes.

Vous aurez concrètement à complexifier, étape par étape, les fonctionnalités souhaitées ainsi que les interactions entre composants.

Le projet comporte quatre étapes :

- Étape 1 (« Pacman de base ») : au terme de cette étape vous aurez créé, en utilisant les outils du moteur de jeu fourni, une instance basique de Pacman avec un acteur évitant les murs d'un labyrinthe en se déplaçant et capable de transiter d'un niveau à l'autre en passant par des portes. Dans ce qui suit, nous appellerons « acteur principal » le pacman contrôlé par le joueur.
- Étape 2 (« Points de vie et ressources ») : lors de cette étape il vous sera demandé d'enrichir le moteur de jeu de fonctionnalités permettant l'affichage de graphismes statiques, comme le décompte des points de vie de l'acteur principal. Ce dernier pourra commencer à collecter des objets.
- Étape 3 (« Fantômes ») : cette étape permettra à votre acteur d'affronter des ennemis qui le traquerons dans le labyrinthe selon divers stratégies.
- Étape 4 (Extensions) : durant cette étape, diverses extensions plus libres vous seront proposées et vous pourrez enrichir à votre façon le jeu créé à l'étape précédente ou en créer d'autres.

1. de ce fait l'idée s'apparente à la variante Super Pacman : https://en.wikipedia.org/wiki/Super_Pac-Man
2. voir la vidéo de démonstration

FIG. 1 : Exemple d'un niveau du jeu où le joueur (pastille jaune) se déplace en collectant des objets, résolvant des énigmes (par exemple trouver quelles clés ramasser pour ouvrir des portes et passer à un niveau supérieur) et en se faisant pourchasser par des fantômes hostiles pouvant le manger.

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

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

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

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

3. Votre code devra respecter les conventions usuelles de nommage et être bien **modularisé et encapsulé**. En particulier, les getters intrusifs, publiquement accessibles, sur des objets modifiables seront à éviter.

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

5. Votre projet **ne doit pas être stocké sur un dépôt public** (de type github). Pour ceux d'entre vous familier avec git, nous recommandons l'utilisation de GitLab : https ://gitlab.epfl.ch/, mais tout type de dépôt est acceptable pour peu qu'il soit privé.

La première étape est volontairement guidée. Il s'agira essentiellement de compléter votre compréhension de la maquette fournie et de commencer à en tirer parti concrètement.

2 SuperPacman de base (étape 1)

Le but de cette étape est de commencer à créer votre propre petit jeu sur grille qui prendra la forme d'une variation du célèbre Pacman®. La jeu à produire est inspiré de « super pacman » https://en.wikipedia.org/wiki/Super_Pac-Man, ce qui permettra d'intégrer des éléments plus complexes, comme des énigmes à résoudre pour passer d'un niveau à l'autre.

La version de base à produire lors de cette étape contiendra un personnage principal, la fameuse pastille jaune, capable de passer des portes lui permettant d'accéder à des nouveaux niveaux de jeux. Le passage des portes se fera selon le mécanisme plus général des *interactions entre acteurs*, tel que décrit dans le tutoriel 3. Ce jeu fera donc intervenir :

- un personnage principal :
- des acteurs « porte » qui permettront au personnage de les traverser. Il s'agira d'une interaction de contact (le personnage doit être dans une cellule contenant un acteur « porte » pour pouvoir traverser cette « porte »).

Pour le moment nous n'inclurons pas d'interactions à distance, mais nous y reviendrons plus tard.

2.1 Préparation du jeu `SuperPacman`

Tout d'abord, en utilisant une logistique analogue à ceux de nos précédentes ébauches de jeu, nous allons changer complètement de décor !

Préparez un jeu `SuperPacman` en vous inspirant de `Tuto2`. Ce dernier sera constitué pour commencer des composants suivants :

- La classe `SuperPacmanPlayer` qui modélise un personnage principal, à placer dans `game.superpacman.actor`; laissez cette classe vide pour le moment ; nous y reviendrons un peu plus bas.
- La classe `SuperPacman`, équivalente à `Tuto2` à placer dans le paquetage `game.superpacman`; cette classe héritera de `RPG` et n'aura donc, pour le moment, plus besoin d'attribut spécifique pour le personnage, ce dernier étant hérité de la classe `RPG` . Étudiez ce que vous offre cette classe pour en faire un usage approprié, notamment la méthode `initPlayer` . Prenez note du fait que les mécanismes de transition du personnage principal d'une aire à l'autre y sont aussi déjà fournis ; la méthode `update` de `SuperPacman` peut donc pour le moment se résumer à appeler celle de sa super-classe. N'oubliez pas non plus d'adapter la méthode `getTitle()` qui retournera un nom de votre choix (par exemple *`"Super Pac-Man"`* ;-)) ; Les aires spécifiques du jeu `SuperPacman` sont décrites un peu plus bas.
- La classe `SuperPacmanArea` équivalente à `Tuto2Area`, à placer dans un sous-paquetage `game.superpacman.area`. Vous associerez un facteur d'échelle commun à toutes les

aires de ce type (`15.f` par exemple ; augmentez cette valeur pour voir de plus grandes parties des aires).

- Les classes `Level0`, `Level1` et `Level2` héritant de `SuperPacmanArea`, à placer dans le paquetage `game.superpacman.area` (elles sont équivalentes aux classes `Ferme`, et `Village` de `game.tutos.area.tuto2`) ; les intitulés associés seront respectivement *`"superpacman/Level0"`*, *`"superpacman/Level1 "`* et *`"superpacman/Level2"`* (le corps de `createArea` sera laissé vide pour commencer) ;
- de la classe `SuperPacmanBehavior` analogue à `Tuto2Behavior` à placer dans `game.superpacman.area` et qui contiendra une classe publique `SuperPacmanCell` équivalente de `Tuto2Cell` (les adaptations nécessaires sont décrites ci-dessous).

2.1.1 Adaptation de `SuperPacmanBehavior`

Dans l'esprit, `SuperPacmanBehavior` et `SuperPacmanCell` sont équivalentes à `Tuto2Behavior` et `Tuto2Cell`. Néanmoins, contrairement à l'ébauche de jeu du tutoriel, nous n'avons pas ici de « décor » conditionnant le déplacement à proprement parler ; par exemple, une zone « lac » sur laquelle on ne pourrait pas marcher. Nous préférons en effet considérer que les murs du labyrinthes sont des acteurs plutôt que des éléments de décor, ce qui permet potentiellement beaucoup de flexibilité dans le jeu comme expliqué plus bas.

Nous partons donc de l'idée ici que la seule chose qui peut entraver le déplacement sur la grille est un acteur.

La méthode `canEnter` des `SuperPacmanCell` retournera donc vrai seulement si la cellule ne contient que des acteurs traversables (souvenez-vous de la méthode `takeCellSpace` des acteurs). Considérer les murs de labyrinthes comme des acteurs nous donnera, si on le souhaite, la possibilité d'en avoir de plusieurs types, avec des comportements spécifiques ; ce qui nous permettra de complexifier le jeu à souhait.

Pour notre jeu, le type d'une cellule va donc plutôt décrire un labyrinthe et son contenu typique :

```
NONE(0),                  // never used as real content
WALL(-16777216),          //black
FREE_WITH_DIAMOND(-1),    //white
FREE_WITH_BLINKY(-65536), //red
FREE_WITH_PINKY(-157237), //pink
FREE_WITH_INKY(-16724737), //cyan
FREE_WITH_CHERRY(-36752), //light red
FREE_WITH_BONUS(-16478723), //light blue
FREE_EMPTY(-6118750);     // sort of gray
```

Vous coderez donc dans `SuperPacmanBehavior` une méthode protégée :

```
void registerActors(Area area)
```

permettant de créer automatiquement des acteurs en fonction du type des cellules et de les enregistrer dans l'aire `area`.

Pour le moment, cette méthode n'enregistrera que des acteurs « mur ». Le codage de ce type d'acteur est un peu laborieux et pas très intéressant en tant que tel. Il vous est donc donné directement dans `superpacman.actor`. Très concrètement, la méthode `registerActors` va, pour toute cellule `[x][y]` dont le type est `WALL`, créer un acteur de type `Wall` à la position `[x][y]` et l'enregistrer dans l'aire `area`. Le constructeur d'un `Wall` a besoin qu'on lui donne un tableau 3x3 de booléens indiquant si le voisinage autour de sa position contient un mur ou pas (`true` indiquant la présence d'un mur). Vous veillerez à modulariser le code et à introduire, pour ce faire, les méthodes utilitaires nécessaires.

Une fois la méthode `registerActors` créée, complétez les classes `Level0`, `Level1` et `Level2` de sorte à ce que les murs de leur labyrinthes respectifs soient enregistrés (méthode `createArea`). Il ne s'agit en principe que d'une seule ligne de code (à ne pas dupliquer si possible dans chaque sous-classe) !

2.2 Ébauche du personnage principal

Pour le moment codez `SuperPacmanPlayer` dans le même esprit que `GhostPlayer`. Il héritera cependant de la classe `Player` décrite dans le tutoriel 3. Etudiez plus en détail cette classe pour comprendre les fonctionnalités dont va bénéficier notre personnage. Le constructeur de `SuperPacmanPlayer` prendra en paramètre l'aire à laquelle il appartient et sa position de départ sous la forme d'une `DiscreteCoordinates`. L'image associée sera donnée directement dans le corps du constructeur. Pour commencer, choisissez-en une simple quelconque comme *`"superpacman/bonus"`* : c'est uniquement pour que vous voyiez rapidement quelque chose s'afficher en guise de test. Nous vous donnerons en effet des indications plus bas pour coder proprement la représentation graphique. Il sera orienté par défaut vers la droite.

La position initiale du `SuperPacmanPlayer` devra être dictée par l'aire qu'il occupe. Une constante `PLAYER_SPAWN_POSITION` spécifique à chaque aire est donc un choix judicieux ici. Donnez lui la valeur `(10, 1)` dans `Level0`, `(15,6)` dans `Level1` et `(15,29)` dans `Level2` par exemple. Il faudra bien entendu tenir compte de ces valeurs lors de la création du `SuperPacmanPlayer` dans le jeu.

2.2.1 Méthode `update`

La méthode `update` de `SuperPacmanPlayer` ressemblera beaucoup à celle de `GhostPlayer`. La différence principale réside dans le fait que le personnage se déplacera tout seul. Les flèches directionnelles ne serviront qu'à indiquer un souhait de changement d'orientation. La vitesse de déplacement est une constante spécifique à notre personnage, valant par example 6. Plus concrètement, la méthode `update` de `SuperPacmanPlayer` va implémenter l'algorithme suivant :

1. identifier l'orientation voulue par le joueur pour le personnage (selon les flèches directionnelles appuyées) ; le personnage ne sera pas encore orienté à proprement parler dans cette direction (l'orientation souhaitée peut donc simplement être consignée comme telle dans un attribut `desiredOrientation` par exemple) ;

2. si aucun déplacement n'est en cours (méthode `isDisplacementOccurs`) :

- si l'aire le laisse entrer dans la cellule en face de lui (méthode `canEnterAreaCells` de l'aire), alors orienter le personnage vers l'orientation voulue (méthode `orientate` des `MovableAreaEntity`) ;
- le faire se déplacer (méthode `move`) en utilisant un nombre de « frames » choisi (par exemple une constante `SPEED` valant 6).

3. invoquer la méthode `update` de la super-classe pour finaliser proprement le déplacement.

Vous noterez donc que la direction souhaitée peut être choisie pendant un déplacement mais que celle-ci ne prendra effet que quand ce sera possible (plus de déplacement en cours).

La cellule en face du personnage pour une orientation voulue `desiredOrientation` est donnée par :

```
Collections.singletonList(getCurrentMainCellCoordinates()
                                .jump(desiredOrientation.toVector()));
```

En tant que `Interactor`, `SuperPacmanPlayer` doit définir les méthodes :

- `getCurrentCells` : ses cellules courantes (qui se réduiront à l'ensemble contenant uniquement sa cellule principale, comme vous avez déjà eu l'occasion de l'exprimer) ;
- `getFieldOfViewCells()`: pour le moment notre personnage n'a pas de champs de vision et vous pouvez retourner `null`

En tant que `Interactor`, il voudra systématiquement toutes les interactions de contact et, pour le moment, aucune interaction à distance (ce que vous pourrez changer si vous complexifiez le jeu plus tard).

En tant que `Interactable`, le personnage `SuperPacmanPlayer` sera l'objet d'interactions à distance uniquement (nous verrons cela un peu plus loin, lorsque nous lui coderons des ennemis !). Il sera traversable (on peut lui marcher dessus !).

Intéressons-nous maintenant à la gestion concrète des interactions.

2.3 Interaction avec les portes

Il vous est demandé d'appliquer le schéma suggéré par le tutoriel 3[3] pour mettre en place les interactions de `SuperPacmanPlayer` avec l'acteur `Door` (cet acteur est également fourni et décrit dans le tutoriel 3).

Pour cela, créez dans un sous-paquetage `game.superpacman.handler`, l'interface `SuperPacmanInteractionVisitor` héritant de `RPGInteractionVisitor`, et qui fournit une définition par défaut des méthodes d'interaction du jeu de `SuperPacman` avec tous les acteurs spécifiques du jeu. Pour le moment le seul acteur spécifique déjà codé est un personnage

3. Un complément vidéo est aussi disponible pour expliquer la mise en oeuvre des interactions : https://proginsc.epfl.ch/wwwhiver/mini-projet2/mp2-interactions.mp4

de type (`SuperPacmanPlayer`) ! Vous allez donc pour le moment vous contenter de décrire l'interaction par défaut entre deux `SuperPacmanPlayer`.

Toutes les définitions (par défaut) de `SuperPacmanInteractionVisitor` auront un corps vide pour exprimer le fait que par défaut, l'interaction consiste à ne rien faire. Libre à vous de modifier ce point plus tard si vous l'estimez pertinent.

Notez qu'il n'est pas nécessaire de définir l'interaction par défaut avec l'acteur « porte », `Door` du paquetage `game.rpg.actor`, qui est déjà prévue dans `RPGInteractionVisitor`.

`SuperPacmanPlayer` en tant que `Interactor` du jeu `SuperPacman`, doit fournir le cas échéant une définition plus spécifique de ces méthodes.

Pour cela, définissez dans la classe `SuperPacmanPlayer`, une classe imbriquée privée `SuperPacmanPlayerHandler` implémentant `SuperPacmanInteractionVisitor`. Ajoutez-y les définitions nécessaires pour gérer plus spécifiquement l'interaction avec une porte (une seule ligne de code suffit, examinez bien le contenu des classe `Player` et `SuperPacmanPlayer` pour comprendre comment coder cette ligne).

Indication : conformément au tutoriel 3, pour que cela fonctionne, il faut indiquer que `SuperPacmanPlayer` accepte de voir ses interactions avec les autres acteurs gérées par un gestionnaire de type `SuperPacmanInteractionVisitor`. La méthode `acceptInteraction` qui ne faisait rien dans `GhostPlayer`, doit être en charge de ce traitement dans `SuperPacmanPlayer`

```
@Override
  public void acceptInteraction(AreaInteractionVisitor v) {
    ((SuperPacmanInteractionVisitor)v).interactWith(this);
  }
```

2.3.1 Enregistrement des acteurs dans les aires

Le jeu `SuperPacman` aura *`"superpacman/Level0"`* comme aire de démarrage. On considérera que chaque aire a une position de placement de départ du personnage : `(10, 1)` pour `Level0`, `(15,6)` pour `Level1` et `(15,29)` pour `Level2` (une constante `PLAYER_SPAWN_POSITION` est tout indiquée).

Au lancement du jeu, un personnage principal de type `SuperPacmanPlayer` sera créé à la position qui lui est destinée dans l'aire de démarrage. Souvenez-vous que la classe `RPG` fournit une méthode `initPlayer`.

Nous avons vu plus haut que les acteurs caractéristiques des labyrinthes (les murs pour le moment) sont automatiquement enregistrés lors de la création des aires. Les autres acteurs, telles les portes doivent aussi être enregistrés. Il s'agit là d'un type d'acteurs dont le placement peut varier en fonction de la logique du jeu. Nous faisons donc le choix de ne pas en faire des données caractéristiques des labyrinthes (au contraire des murs) et donc de les découpler de la grille (`SuperPacmanBehaviour`).

La méthode `createArea` de `Level0` dans `game.superpacman.area` enregistrera donc une

porte permettant de passer au niveau supérieur (et conformément à l'image du « behavior » associé). En voici les caractéristiques :

porte	destination	coord. arrivée	orientation	cellule principale	autres coord.
1	"superpacman/Level1"	*pos*	haut	(5,9)	(6,9)

où *pos* est la position de placement de départ du personnage dans l'aire d'accueil (ici `Level1`).

De même, la méthode `createArea` de `Level1` dans `game.superpacman.area` enregistrera une porte vers le niveau suivant dont voici les caractéristiques :

porte	destination	coord. arrivée	orientation	cellule principale	autres coord.
1	"superpacman/Level2"	*pos*	bas	(14,0)	(15,0)

où *pos* est la position de placement de départ du personnage dans l'aire d'accueil (ici `Level2`). Enfin, la méthode `createArea` de `Level2` dans `game.superpacman.area` n'enregistra aucune porte (mais libre à vous par la suite de permettre de transiter vers un niveau supérieur).

Vous noterez que dans le jeu de base, les portes servent de jonction au niveau suivant et n'ont pas pour vocation de faire revenir le personnage dans un niveau déjà fait.

Toutes les portes seront associées à des signaux en permanence « allumés » (`Logic.TRUE`).

Lancez votre jeu `SuperPacman`. Vous devriez voir votre personnage transiter d'une aire à l'autre par le biais des portes, sans qu'aucun test explicite sur la nature d'une cellule ne soit nécessaire. Passer une porte doit lui permettre d'être dans l'aire destination de cette dernière (de l'autre côté de la porte). En guise de test, faites également en sorte que l'une des portes soit créée avec un signal toujours fermé (`Logic.FALSE`), le personnage ne devrait plus être capable de la traverser.

2.4 Animation du personnage

En utilisant le concept d'animation présenté dans le tutoriel 3, faites en sorte que le visuel graphique de `SuperPacmanPlayer` devienne animé.

Voici quelques indications pour y parvenir :

- comme suggéré 4 animations remplaceront l'image fixe de pastille jaune utilisée jusqu'ici ; ces dernières seront créées à la construction du personnage. Parmi ces animations seule l'animation en cours sera dessinée et mise à jour (vous ferez en sorte qu'au départ l'animation en cours soit celle animant le personnage vers le haut). Les 4 animations seront extraites de l'image *`"superpacman/pacman"`*
- la méthode `update` devra sélectionner laquelle des animations est l'animation en cours, en fonction de l'orientation du personnage ;
- l'animation en cours ne doit être mise à jour par la méthode `update` que si un déplacement est en cours (`isDisplacementOccurs`) (sinon elle sera réinitialisée au moyen de sa méthode `reset`)

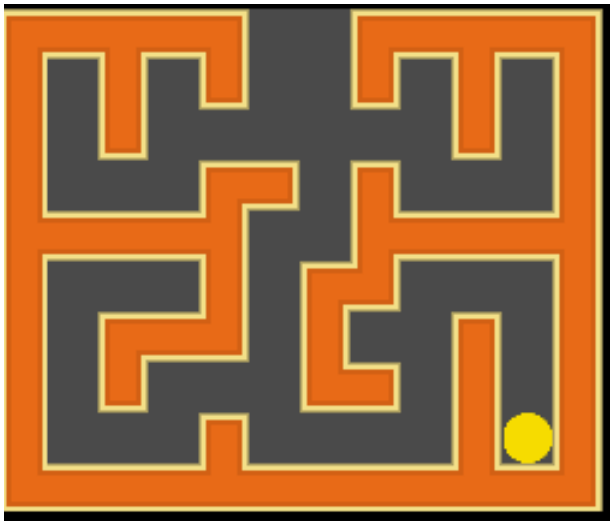

FIG. 2 : Placement attendu des murs pour l'aire `Level0`

Essayez de tirer parti de la méthode `ordinal()` des énumérations pour coder la sélection de l'animation en cours de façon concise.

Question 1

La logistique mise en place, telles qu'exposée dans les tutoriels et exploitée concrètement dans cette première partie du projet, peut sembler *a priori* inutilement complexe. L'avantage qu'elle offre est qu'elle modélise de façon très générale et abstraite, les besoins inhérents à de nombreux jeux où des acteurs se déplacent sur une grille et interagissent soit entre eux soit avec le contenu de la grille. Comment pourriez-vous en tirer parti pour mettre en oeuvre un vrai jeu de type RPG, à l'image de celui ébauché dans le tutoriel, mais où le personnage aurait à interagir avec toutes sorte d'autres personnages par exemple ? Que suffirait-il de définir ?

Vous aurez dans la suite du projet à coder de nombreuses autres interactions entre acteurs ou avec les cellules. Toutes les interactions à venir devront impérativement être codées selon le schéma mis en place lors de cette partie et ne devront pas nécessiter de tests de types sur les objets.

2.5 Validation de l'étape 1

Pour valider cette étape, vous vérifierez :

1. que les murs des labyrinthes sont placés conformément aux « behavior » associé à leurs niveaux respectifs (voir les figures 2 et 3) ;
2. que `SuperPacmanPlayer` ne peut pas marcher que dans les corridors du labyrinthe ;

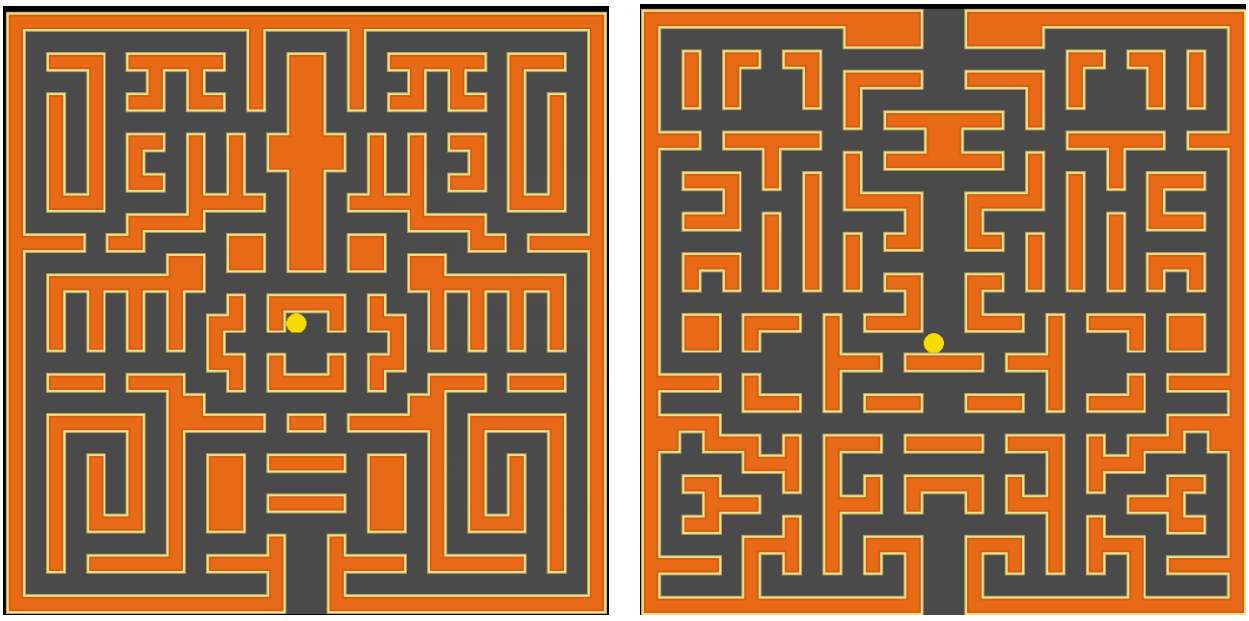

FIG. 3 : Placement attendu des murs pour les aires `Level1` et `Level2`.

3. qu'il peut passer au niveau supérieur en passant par une porte mais ne peut pas revenir en arrière ;
4. que les animations du personnage sont conformes à son orientation (voir la vidéo de démonstration) ;
5. et qu'il répond correctement aux contrôles lui permettant de changer d'orientation.

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

FIG. 4 : Affichage graphique du statut du personnage principal

3 Collecte d'objets (étape 2)

La vocation fondamentale de notre personnage principal est d'« avaler » des objets sur lesquels il passe. Ceci lui permet d'augmenter son score ; le but étant évidemment de terminer la partie avec le plus grand score possible. Dans cette seconde partie du projet, il s'agit de doter notre « pastille jaune » de la capacité de collecter différents types d'objets et de lui associer des points de vie. Dans le monde hostile que nous nous apprêtons à lui créer, notre personnage pourra en effet être sérieusement affaibli par de mauvaises rencontres. S'il n'a plus de force , il sera amené à recommencer inexorablement le niveau en cours. Les points de vie sont à la base de la mise en oeuvre de ce mécanisme. Les graphismes seront donc élaborés de sorte à permettre de visualiser les informations caractéristiques que sont les points de vie et le score du joueur (voir la figure 4). Dans la foulée, nous introduirons un acteur dépendant de la collecte d'objets, ce qui pimentera un peu le jeu.

3.1 Points de vie et score

Le personnage principal (`SuperPacmanPlayer`) sera doté de points de vie et d'un score (des `int`). Le nombre de points de vie est plafonné à une valeur maximale (par exemple 5), identique et constant pour tous les personnages de ce type. On supposera aussi que le nombre de points de vie à la création du personnage est fixé à une valeur constante identique pour toutes les instances, par exemple 3. Le score est évidemment initialisé à zéro.

3.2 Graphismes statiques

Avant d'aborder cette section, il est conseillé de jeter à nouveau un petit oeil à l'annexe 1 du tutoriel. Les points principaux qui nous intéressent sont :

- qu'à la fenêtre de dessin s'attache un référentiel donné par la méthode `getTransform`;
- que `getTransform().getOrigin()` retourne les coordonnées de l'origine de ce référentiel ;
- que les méthodes `getScaledWidth` et `getScaledHeight` retournent la taille de la fenêtre à l'échelle du monde simulé (c'est-à-dire après application du facteur d'échelle) ;
- et que l'ancre (`anchor`) d'une image dans un référentiel n'est autre que les coordonnées de son coin supérieur gauche dans ce référentiel.

Afin de pouvoir visualiser les différents éléments mis en place, il vous est demandé de mettre en place leur représentation graphique selon l'exemple de la figure 4. Ces informations seront encapsulées dans un seul concept, celui d'« objet graphique décrivant le statut d'un personnage principal » (`SuperPacmanPlayerStatusGUI`). Un tel objet implémentera naturellement un `Graphics`.

Indications :

- Les points de vie peuvent être dessinés par `SuperPacmanPlayerStatusGUI` comme des `ImageGraphics` du répertoire `game.actor`.
  Les sprites « pacman » de *"superpacman/lifeDisplay"* peuvent être utilisés. Voici un exemple de comment placer le référentiel :

  ```
  float width = canvas.getScaledWidth();
  float height = canvas.getScaledHeight();
  Vector anchor = canvas.getTransform().getOrigin()
                    .sub(new Vector(width/2, height/2));
  ```

  Puis dessiner un point de vie :

  ```
  ImageGraphics life = new
     ImageGraphics(ResourcePath.getSprite("superpacman/lifeDisplay"),
     1.f, 1.f, new RegionOfInterest(m, 0, 64, 64),
     anchor.add(new Vector(n, height - 1.375f)), 1, DEPTH);
  life.draw(canvas);
  ```

  où `m` vaut 0 si l'on extrait le premier sprite (jaune) et 64 si l'on extrait le second (gris). `n` est le décalage horizontal éventuellement souhaité. Il y aura autant de « pacman » dessinés que le nombre maximal possible de points de vie. Le pacman à la position `i` en partant de la gauche sera dessiné en jaune si le nombre de points de vie est supérieur ou égal à `i` et en gris sinon.

- le score quand à lui peut être affiché comme `TextGraphics` du répertoire `game.actor` :

```
TextGraphics score = new TextGraphics(...); // prendre un
   constructeur vous permettant d'ancrer le texte au bon
   endroit (par exemple celui de la ligne 78)
score.draw(canvas);
```

`Color.YELLOW` peut être utilisée pour la couleur de remplissage des lettres et `Color.BLACK` pour leur contour (il s'agit du type `java.awt.Color`).

Il est attendu de vous un code concis et modulaire pour cette partie : une dizaine de ligne suffit pour coder la méthode de dessin de `SuperPacmanPlayerStatusGUI`.

Voici enfin les contraintes de codage à respecter :

- La façon d'afficher le statut du personnage principal peut être amenée à changer (même si dans la version de base du projet, on n'utilisera que celle décrite ci-dessus). Par conséquent, `SuperPacmanPlayerStatusGUI` sera plutôt codé dans une classe indépendante qu'une classe imbriquée.
- La description graphique du statut est largement liée aux acteurs. Vous ferez en sorte que seuls les acteurs de jeux SuperPacman ou les sous-classes de `SuperPacmanPlayerStatusGUI` puissent construire un objet de ce type.

3.2.1 Tâche

Il vous est demandé de :

- coder les concepts décrits précédemment conformément aux spécifications et contraintes données.
- tester vos développements de façon « ad'hoc » en attribuant « en dur » dans le code différents points de vie de départ pour votre personnage. `SuperPacman` vit en effet pour l'heure des moments bien paisibles ou aucun autre personnage ne lui fait perdre de points de vie ! Modifiez aussi le score initial « en dur » dans le code (ne pas oublier de remettre à zéro sa valeur de départ pour la suite).

Vous vérifierez alors :

1. que le score est correctement affiché ;
2. que les points de vie s'affichent correctement ;
3. que la description graphique du statut du personnage reste placée au bon endroit y compris quand le personnage se déplace.
4. et que cette description continue à s'afficher proprement lorsque l'on passe d'une aire à l'autre.

Pour le moment, le fait que le personnage n'ait plus de points de vie n'a pas d'incidence sur le déroulement du jeu.

3.3 Collecte d'objets

Afin que le personnage principal puisse augmenter son score, il vous est maintenant demandé d'introduire dans votre mini-projet, le concept d'objet « ramassable ». Ce concept est *a priori* intéressant pour tout jeu sur grille et il est raisonnable d'introduire le concept général de `CollectableAreaEntity` (que vous pourrez coder dans le paquetage `areagame.actor` typiquement). Un objet « ramassable » a pour caractéristique fondamentale de pouvoir être ramassé et de disparaître de l'aire simulée au moment où il l'est.

Dans votre jeu `SuperPacman`, un objet ramassable aura une déclinaison encore plus spécifique : il sera automatiquement ramassé lorsqu'on lui marche dessus. Ce point doit être explicitement modélisé dans votre conception.

Vous introduirez trois types d'objets de ce type, à coder dans le paquetage `superpacman.actor` :

1. les bonus (`Bonus`) : ils n'augmentent pas le score, mais ils seront destinés plus tard à rendre notre personnage moins vulnérable face à ses ennemis. Un bonus est représentable graphiquement au moyen de la ressource `superpacman/coin`.

2. les cerises (`Cherry`) : vous considérerez que ce type d'objet augmente le score d'un montant fixe lorsque ramassé (200 par exemple). Les cerises sont représentables graphiquement au moyen de la ressource `superpacman/cherry`.

3. les diamants (`Diamond`) : vous considérerez que chaque diamant ramassé augmente le score d'un montant fixe (10 par exemple). Les diamants sont représentables graphiquement au moyen de la ressource `superpacman/diamond`.

Pour le moment, les interactions attendues sont que le personnage principal augmente son score du montant attendu pour chaque type d'objets ramassé. Vous ferez en sorte que les `Bonus` soient animés (tournent autour d'eux même).

Les sprites associés à `superpacman/coin` peuvent être extraits comme suit :

```
Sprite[] sprites =
   RPGSprite.extractSprites("superpacman/coin", 4, 1, 1,
   parent, 16, 16);
```

où `parent` est le `Positionnable` auquel attacher cette représentation graphique. Une animation peut être construite à partir d'un tel tableau de `Sprite`.

> Vous éviterez les duplications inutiles de code. L'évaluation du score rapportés par chaque type d'objets devrait pouvoir se faire de façon polymorphique.

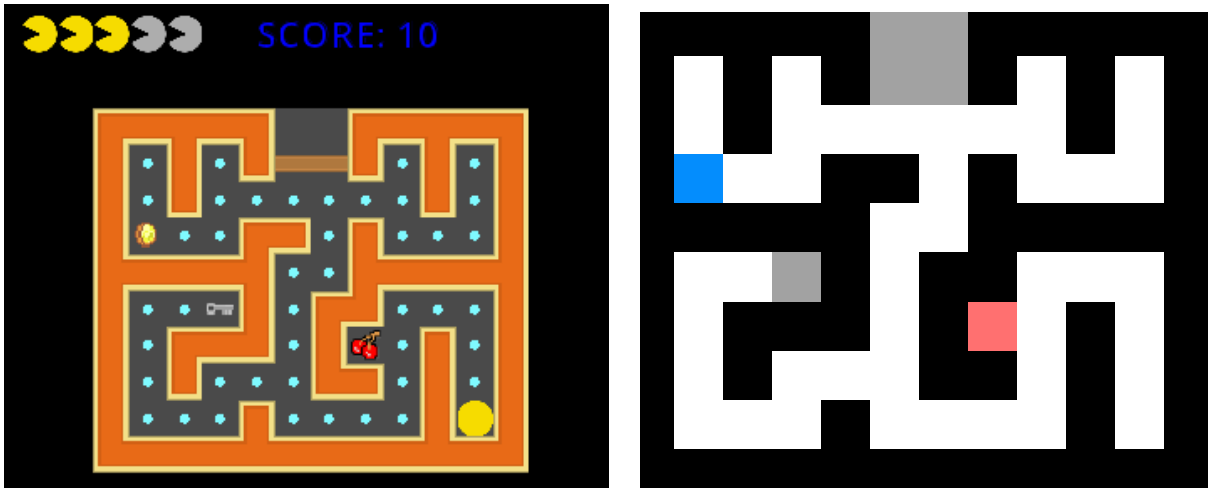

FIG. 5 : Placement attendus pour les acteurs de l'aire `Level0` (à gauche), conforme à l'image `res/images/behaviors/superpacman/Level0.png` (à droite)

3.3.1 Création automatique d'objets à collecter

Usuellement, il y a un grand nombre d'objets à collecter dans les jeux de types « pacman ». Ajouter manuellement ces objets dans chaque niveau peut donc s'avérer bien fastidieux. De plus, la plupart des objets à collecter ainsi que leur position dans le labyrinthe sont généralement spécifiques à chaque niveau.

Il vous est maintenant demander de modifier la méthode `registerActors()` de `superpacman/SuperPacmanBehavior`, de sorte à ce qu'en plus de créer les acteurs « mur », elle crée également les acteurs de type `Bonus`, `Cherry` et `Diamond` en fonction du type des cellules.

L'acteur `Key` joue un rôle un peu différent, il est censé pouvoir être placé dynamiquement dans le labyrinthe en cours de jeu. Il fait donc exception à ce placement automatique.

3.3.2 Tâche

Il vous est demandé de :

- coder les éléments suggérés ci-dessus conformément aux spécifications et contraintes décrites ;
- les tester au moyen des niveaux fournis.

Les clés seront crées et enregistrées « manuellement » comme acteurs des différentes aires, lors de leur création. Créez-en une en `(3,4)` pour `Level0`.

Vous vérifierez :

1. que les objets `Key`, `Bonus`, `Cherry` et `Diamond` se créent bien et aux endroits prévus en fonction du type des cellules pour les trois derniers : voir la figure 5 pour ce qui doit être obtenu pour le niveau zéro par exemple (voir aussi plus loin la figure 7 pour les

niveaux supérieurs) ;

2. que le fait de marcher sur un objet ramassable le fait disparaître de l'aire de jeu ;
3. que le fait de ramasser des objets se répercute proprement sur l'affichage du statut (score) ;
4. que l'objet `Bonus` tourne sur lui-même ;
5. et que les points vérifiés dans la sections 3.2.1 sont toujours fonctionnels.

3.4 Objets dépendant de signaux

Le tutoriel a introduit la notion de signal qui peut être exploitée pour rendre le jeu plus attractif en le faisant dépendre de la résolution d'énigmes. On peut imaginer par exemple que pour obtenir un bonus avec lequel se protéger de ses ennemis, le personnage principal ait à ouvrir, au moyen d'une clé, une barrière qui en bloque l'accès. L'outillage fourni, permet typiquement de répertorier une clé comme un signal attaché à la barrière. Pour commencer à travailler avec la notion de signal, il vous est demandé de mettre en oeuvre un nouvel acteur nommé `Gate` (barrière), dérivant de l'acteur fourni `AreaEntity`. Vous le considérerez comme spécifique au jeu de type `superpacman`. Cet acteur occupera sa cellule principale. Il sera traversable en fonction d'un signal le caractérisant : la barrière n'est « traversable » que si le signal qui lui est attaché est activé. Pour le moment, ce type d'acteur n'accepte aucun type d'interaction (ni à distance, ni de contact) et le gestionnaire d'interaction du jeu n'en fera rien. Il ne sera par ailleurs dessiné (visible) que si son signal est activé. Le signal attaché à un `Gate` doit pouvoir être spécifié à sa construction.

Le visuel des `Gate` se fera au moyen de la ressource *`"superpacman/gate"`*. La région `(0,0)` sera extraite si la barrière est orientée à la verticale (`DOWN` ou `UP`) et `(0,64)` sinon.

Pour démontrer que vous avez bien compris le concept de `Signal`, il vous est enfin de demandé de faire en sorte qu'une aire de jeu `SuperPacmanArea` soit elle même interprétable comme un signal. En clair, une `SuperPacmanArea` doit devenir un signal qui est activé lorsque tous les diamants sont collectés.

3.4.1 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Pour tester vos développements, vous ajouterez des barrières dans votre jeu.

L'aire `Level0` sera dotée de deux barrières dont voici les caractéristiques :

orientation	position	signal
RIGHT	(5,8)	key
LEFT	(6,8)	key

où `key` est la clé créée dans `Level0`.

FIG. 6 : Placements attendus pour les barrières de l'aire `Level0`

Vous ajouterez aussi à `Level1` un premier `Gate` en position `(14,3)` et un second en position `(15,3)` , tous deux orientés à droite. En exploitant le fait qu'une aire peut être un signal, vous ferez en sorte que cette barrière s'ouvre lorsque tous les diamants de `Level1` sont collectés.

Vous ajouterez enfin quatre clés (`key1`, `key2`, `key3` et `key4`) au niveau 2, avec pour positions respectives `(3, 16)`, `(26, 16)`, `(2, 8)` et `(27, 8)` et vous ajouterez à ce niveau les barrières aux caractéristiques suivantes :

orientation	position	signal
RIGHT	(8,14)	key1
DOWN	(5,12)	key1
RIGHT	(8,10)	key1
RIGHT	(8,8)	key1
RIGHT	(21,14)	key2
DOWN	(24,12)	key2
RIGHT	(21,10)	key2
RIGHT	(21,8)	key2
RIGHT	(10,2)	key3 et key4
RIGHT	(19,2)	key3 et key4
RIGHT	(12,8)	key3 et key4
RIGHT	(17,8)	key3 et key4
RIGHT	(14,3)	tous les diamants collectés
RIGHT	(15,3)	tous les diamants collectés

Vous vérifierez alors :

1. que les barrières apparaissent bien aux endroits prévus (voir les figures 6 et 7) et qu'elles sont bloquantes pour le personnage ;

2. que le personnage peut ramasser les clés et que cela a pour effet d'ouvrir les barrières qui en dépendent selon ce qui a été prévu dans les descriptifs ci-dessus ;

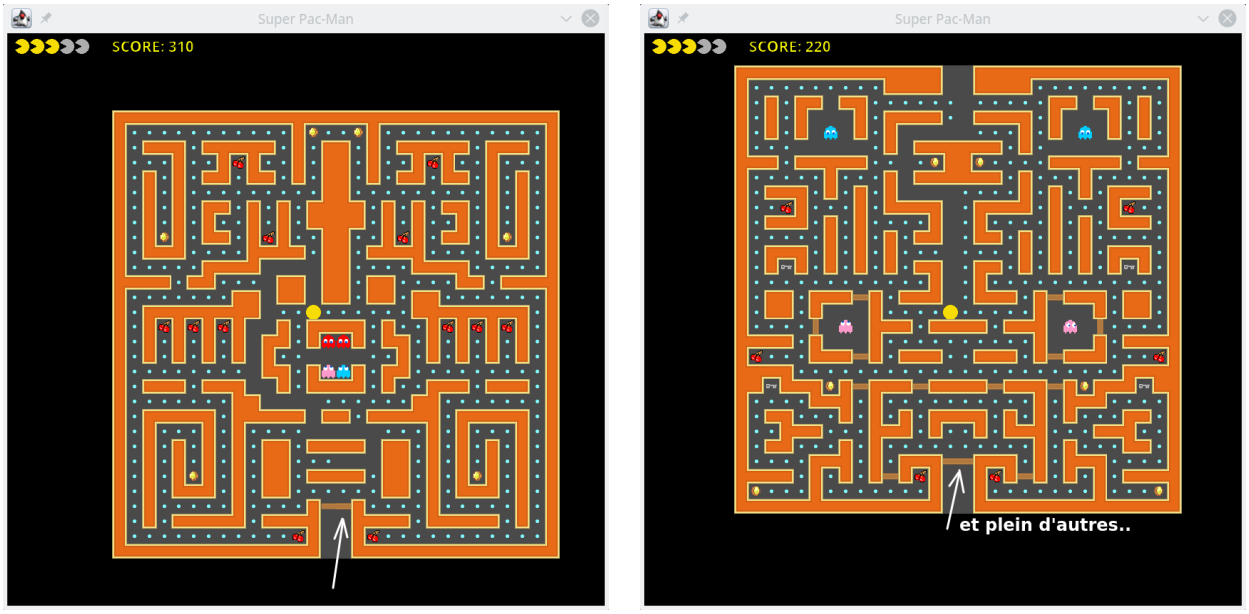

FIG. 7 : Placements attendus pour les barrières des aires `Level1` et `Level2`.

3. que les barrières ouvertes disparaissent visuellement ;
4. et que le passage du niveau 1 au niveau 2 est bien conditionné par la collecte de tous les diamants du niveau 1.

3.5 Validation de l'étape 2

Pour valider cette étape, toutes les vérifications des sections 2.5, 3.2.1, 3.3.2 et 3.4.1 doivent avoir été effectuées.

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

Pour programmer cette partie, il est important de bien lire l'énoncé dans son intégralité, en tout cas jusqu'à la description du premier type concret de fantôme (ne codez pas linéairement au fur et à mesure que vous lisez). Il est important de bien réfléchir à où placer les attributs et méthodes nécessaires, sans avoir recours à des accès trop intrusifs (pas de getters sur les fantômes du jeu par exemple).

4 Fantômes (étape 3)

Dans ce troisième volet du projet, il vous est demandé d'enrichir la conception en permettant l'ajout de « fantômes » que le personnage principal aura à affronter. Vous trouverez ci-dessous les spécifications à respecter ainsi que quelques indications.

Les fantômes sont des acteurs dont le but est d'attraper notre personnage principal en employant une stratégie propre à leur « personnalité ». Ils peuvent également se faire « manger » par ce dernier, dans certaines conditions.

Les fantômes sont des acteurs :

- capables de se déplacer sur une grille ;
- caractérisés par leur position « refuge » – un coin préféré du labyrinthe –; ils y apparaissent et s'y redirigent dans certaines situations ;
- « demandeurs » d'interactions (il ne font pas que les subir) ;
- qui sont inconditionnellement demandeurs d'interactions à distance, mais qui ne demandent pas d'interaction de contact (pour mimer le fait qu'un fantôme ne ressent pas de contact ;-)) ;
- avec lesquels il n'est possible d'avoir que des interactions de contact (par défaut) ;
- sur lesquels, par défaut, on peut marcher ;
- qui rapportent un score fixe (par exemple un `GHOST_SCORE` de 500) à celui qui parvient à les manger ;
- et qui ont par défaut comme champs de perception toutes les cellules de leur voisinage dans un « rayon » fixé par une constante commune à tous (5 par exemple). Il ne s'agira pas d'un rayon circulaire mais du carré entourant la position fantôme.

La position « refuge » sera donnée à la construction.

4.1 Fantômes effrayés et personnage « invulnérable »

Les bonus (classe `Bonus`) lorsque collectés, rendent le personnage principal invulnérable pendant un certain temps. Cet état d'invulnérabilité effraie les fantômes de l'aire, peu

importe leur position. Ils s'animent alors différemment, ce que vous pouvez mettre en oeuvre en utilisant les ressources graphiques *"superpacman/ghost.afraid"*. Pour le moment anticipez simplement le fait que les fantômes ont une méthode permettant de tester s'ils sont effrayés et mettez en place les mécanismes décidant de l'invulnérabilité du personnage.

Pour mettre en oeuvre la vulnérabilité pendant un temps limité, vous pouvez utiliser un attribut du personnage qui simule un « timer ». Ce dernier pourra simplement être un `float` initialisé à une valeur fixe et décrémenté de `deltaTime` à chaque pas de simulation (sans pouvoir descendre en dessous de zéro).

La valeur d'initialisation sera considérée comme spécifique au bonus (une constante valant 10 par exemple). A vous de gérer proprement les (re)initialisations de ce « timer ». Le personnage pourra alors être considéré comme invulnérable tant que le timer a une valeur strictement positive par exemple.

4.2 Interaction entre le personnage et les fantômes

Le modèle suggéré pour gérer les interactions entre le personnage principal et un fantôme est le suivant :

- Si le fantôme voit le personnage (ce dernier entre dans son champs de perception), il gère cette interaction à distance. Le plus simple dans ce cas est qu'il « mémorise » le personnage. Cela lui permettra de mettre en oeuvre des stratégies pour le suivre par exemple.
- Si le personnage et le fantôme se croisent sur une case, c'est le premier qui va gérer l'interaction de contact (le fantôme lui ne ressent pas de contact). Si le personnage est vulnérable il doit se faire manger. Sinon, c'est le fantôme qui se fait manger et le personnage principal augmente son score de `GHOST_SCORE` (ou équivalent). Lorsqu'un fantôme se fait manger, il réapparaît à sa position refuge et « oublie » le personnage.

Le fait que le personnage se fasse manger se traduit par le fait qu'il perd un point de vie et réapparaît à sa case de naissance (`PLAYER_SPAWN_POSITION` dépendant de l'aire). Attention alors à ce qu'il soit bien désenregistré de la case où il était pour s'enregistrer dans sa case de départ. Les fantômes doivent aussi réapparaître dans leur case refuge. Voici les instructions à utiliser pour se quitter la position courante :

```
getOwnerArea().leaveAreaCells(this, getEnteredCells());
```

et investir dans la position courante :

```
getOwnerArea().enterAreaCells(this, getCurrentCells());
```

Plusieurs sortes de fantômes sont envisageables ; voire d'autres types de créatures hostiles ou amicales). Nous vous demandons de coder les trois déclinaisons dont les spécifications sont décrites ci-dessous ; à savoir les fantôme de type « Blinky » (le lunatique), « Inky » (le prudent) et « Pinky » (le fuyant) (voir la figure 8 tout un programme :-)). Tout ces fantômes ont néanmoins un comportement général commun, décrit ci-dessous.

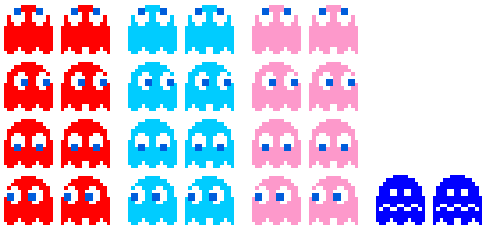

FIG. 8 : Dans l'ordre (et dans tous leurs états) : Blinky, Inky, Pinky, et « mort de peur »

4.3 Comportement général des fantômes

La méthode `update` (comportement par défaut) des fantômes peut être résumée comme suit :

- Si le fantôme est en cours de déplacement, il n'y a rien à faire si ce n'est de gérer les animations (on laisse simplement le déplacement se terminer en animant l'acteur de façon appropriée). Les animations dépendent de son état (effrayé ou non).
- Sinon, le fantôme choisit une orientation (une méthode `getNextOrientation` est préconisée) ; s'oriente selon cette direction (méthode `orientate`) et se déplace selon cette orientation (méthode `move` avec 18 par exemple comme nombre par défaut de « frames » utiles au déplacement).

Le mécanisme des choix de l'orientation ne peut en principe pas être géré à ce niveau d'abstraction, chaque catégorie concrète de fantôme ayant ses propres stratégies.

- Vous veillerez à ce que chaque classe contiennent ce qui lui est spécifique et que les super-classes n'aient pas connaissance de leurs sous-classes ;
- Le code des méthodes `interactWith` entre fantôme et personnage principal se résume à quelques lignes. Si le code devient trop verbeux chez vous, n'hésitez pas à nous solliciter car cela peut refléter une mauvaise compréhension des modalités de mise en place des interactions.

4.4 `Blinky` le lunatique

Un `Blinky` est un fantôme lunatique qui pourchasse sa proie au hasard sans jamais s'arrêter. Sa spécificité est donc simplement de changer aléatoirement de direction de déplacement (orientation) à chaque pas de simulation. L'orientation est choisie au hasard parmi les 4 possibles. Le rôle de `getNextOrientation` sera donc ici retourner l'orientation ainsi tirée au hasard.

Pour tirer au hasard un entier compris entre `0` et `MAX`, utilisez l'instruction :

```
int randomInt = RandomGenerator.getInstance().nextInt(MAX);
```

Vous pourrez prendre `superpacman/ghost.blinky` comme représentation graphique de ce fantôme.

Indication : Vous noterez que la classe `Orientation` fournit une méthode `fromInt`.

4.5 Ajout des fantômes au jeu

L'ajout des fantômes au jeu se fera selon les mêmes modalités que les objets à collecter. Vous modifierez donc `SuperPacmanBehavior.registerActors` de sorte à ce que si le type de la cellule est `FREE_WITH_BLINKY`, un acteur de type `Blinky` soit enregistré dans l'aire.

Une fois des fantômes concrets ajoutés au jeu, vous ajouterez les mécanismes permettant de faire transiter les fantômes à l'état « effrayé ». Il faudra ajouter pour cela à la méthode `update` du jeu, le fait que les fantômes doivent devenir effrayé ou pas selon l'état de vulnérabilité du personnage.

4.5.1 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données.

Vous vérifierez alors :

1. que deux fantômes `Blinky` apparaîssent bien au milieu du `Level1` comme dans la figure 7 ;
2. qu'ils s'animent proprement (voir la vidéo de démonstration) quand ils ne sont pas effrayés ;
3. qu'ils s'animent proprement (voir la vidéo de démonstration) quand le player collecte un bonus (transition à l'état « effrayé ») ;
4. que les `Blinky` choisissent au hasard une nouvelle orientation à chaque pas de simulation ;
5. qu'ils ne se déplacent qu'à l'intérieur de l'aire de jeu ;
6. que `SuperPacmanPlayer` se fait manger par `Blinky` s'il est vulnérable et qu'ils se croisent sur une case ;
7. que lorsqu'il se fait manger `SuperPacmanPlayer` réapparaît à son endroit de « naissance », et que les fantômes sont replacés à leur position « refuge » ;
8. que les `Blinky` se font manger par `SuperPacmanPlayer` s'il est invulnérable et que les `Blinky` mangés réapparaissent alors dans leur zone refuge ;
9. et que le score de `SuperPacmanPlayer` croît de `GHOST_SCORE` (ou équivalent) à chaque fois qu'il mange un `Blinky`.

Notez que pour tester plus rapidement, vous pouvez faire en sorte que, momentanément pendant que vous développez cette partie, la première aire du jeu soit `Level1`. Vous pouvez aussi très bien enregistrer des fantômes « à la main » (comme les clés par exemples) dans le niveau `Level0`.

Passons maintenant aux deux autres catégories de fantômes.

`Inky` et `Pinky` sont tous deux des fantômes qui suivent un chemin en se déplaçant. Lisez les descriptifs de chacun avant de vous lancer dans la mise en oeuvre dans cette partie. Certains comportements communs peuvent en effet sans doute être « factorisés » dans une super-classe commune.

4.6 Graphe associé à la grille

La section 6.10 du tutoriel vous a montré qu'un graphe peut être associé à la grille du jeu. C'est précisément au moyen d'un tel graphe, que les chemins suivis par `Inky` et `Pinky` seront construits. Il vous est donc demandé dans un premier temps, d'associer un graphe de type `AreaGraph` à `SuperPacmanAreaBehavior`. Ce graphe sera construit dans le constructeur de `SuperPacmanAreaBehavior` selon l'algorithme suivant :

- pour toutes les cellules qui ne sont pas des murs, ajouter un noeud au graphe au moyen de la méthode `addNode`.

Les booléens requis par cette méthode se calculent en fonction de la présence de murs. Par exemple, `hasLeftEdge` peut se calculer de façon analogue à ceci :

```
x > 0 && type de la cellule (x-1,y) != SuperPacmanCellType.WALL,
```

Vous veillerez à bien modulariser ces traitements.

4.7 `Inky` le prudent

`Inky` est un fantôme qui se déplace autour du point d'apparition (point « refuge »). Il attaque le personnage s'il est à sa portée et retourne rapidement à sont point « refuge » lorsqu'il est effrayé. Comme ce sera le cas pour le fantôme suivant (`Pinky`), `Inky` se déplace en suivant des plus courts chemins vers une position cible :

- dans le cas où il est effrayé, cette position est une case atteignable, choisie au hasard parmi toutes celles à une distance inférieure à une constante (par exemple `MAX_DISTANCE_WHEN_SCARED`, valant 5) de sa position refuge ;
- s'il n'est pas effrayé, et s'il a repéré le personnage, la position cible sera celle du personnage ;

- sinon, il s'agira d'une position choisie au hasard parmi toutes celles à une distance inférieure à une constante (par exemple `MAX_DISTANCE_WHEN_NOT_SCARED` , valant 10) de sa position refuge : il accepte de se trouver un peu plus loin de son refuge.

Une case atteignable choisie au hasard sera toute case de la grille vers laquelle il existe un (plus court) chemin.

Pour faire en sorte que le fantôme se déplace en suivant un chemin, il suffit qu'à chaque pas de simulation, il choisisse l'orientation lui permettant de suivre ce chemin, ce sera le rôle de `getNextOrientation`.

4.7.1 Calcul d'une case aléatoire atteignable

Pour calculer une case aléatoire atteignable dans un certain rayon autour de la position refuge, il suffit de répéter la recherche d'une case atteignable au hasard sur toute la grille, tant que la distance entre cette case et la position refuge est supérieure au rayon (la classe `DiscreteCoordinates` offre a méthode `distanceBetween`).

4.7.2 Méthode `getNextOrientation`

Dans le cas de `Inky`, cette méthode a pour rôle d'évaluer à chaque pas de simulation la position cible à atteindre (via un plus court chemin).

Le choix de la cible dépend de l'état du fantôme comme décrit plus haut. Une fois la cible calculée, l'orientation candidate est celle permettant de s'orienter vers le plus court chemin entre la position courante `getCurrentMainCellCoordinates()` et la position de la cible `targetPos`. Ce plus court chemin, `path`, de type `Queue<Orientation>` s'obtient ainsi :

```
path = graph.shortestPath(getCurrentMainCellCoordinates(),
   targetPos)
```

où `graph` est le graphe de la grille associée à l'aire dans laquelle évolue le fantôme. Attention à accéder à ce graphe sans casser l'encapsulation. Rappelons alors que l'orientation à prendre pour suivre ce chemin est donnée par :

```
path.poll();
```

Vous noterez qu'il est possible qu'un tel plus court chemin n'existe pas (imaginez par exemple que le personnage ait à un moment donné le pouvoir de traverser les murs et de se cacher dans une zone inatteignable). Dans ce cas `Inky` prendra pour cible n'importe quelle case atteignable, choisie au hasard.

Vous noterez aussi que la case cible n'a pas besoin d'être réévaluée à chaque pas de simulation. Il est donc intéressant d'en faire un attribut du fantôme.

Par ailleurs, la cible ne doit en réalité être réévaluée que quand il y a eu transition d'état ; c'est-à-dire :

1. lorsque le fantôme transite de l'état effrayé à l'état non-effrayé (ou vice-versa) ;

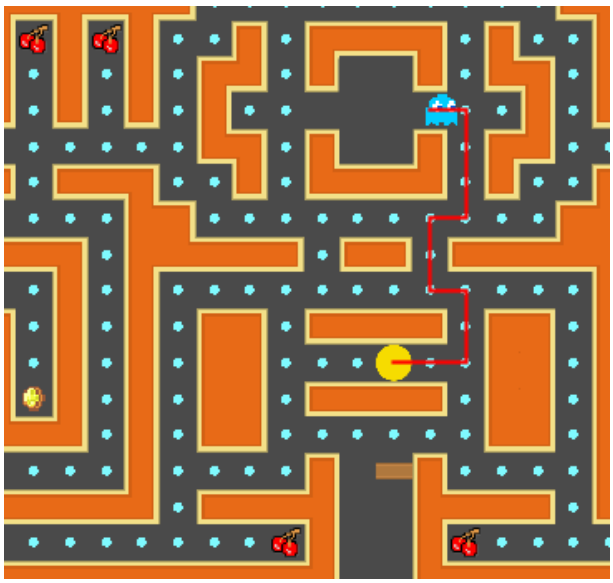

FIG. 9 : Dessin explicite du chemin suivi par un fantôme.

2. ou lorsqu'il passe de l'état où il mémorise le personnage à l'état où il l'oublie (et vice-versa) ;

3. ou encore lorsqu'il a atteint sa position cible.

Il faudra bien entendu compléter `SuperPacmanBehavior.registerActors` de sorte à ce que si le type de la cellule est `FREE_WITH_INKY`, un acteur de type `Inky` soit enregistré dans l'aire.

Notez que pour faciliter la vérification des comportements, vous pouvez faire dessiner explicitement les chemins suivis. Pour cela, il suffit de déclarer un attribut de type « chemin graphique » (par exemple `graphicPath` de type `Path`). A chaque fois que l'on calcule un nouveau chemin (`path`), on peut alors mettre à jour sa contrepartie graphique par cette tournure :

```
graphicPath = new Path(this.getPosition(), new
   LinkedList<Orientation>(path));
```

Il suffit alors de faire dessiner le chemin graphique (s'il n'est pas nul) par la méthode de dessin du fantôme, ce qui permet de voir explicitement apparaître le chemin suivi, comme montré dans la figure 9.

Pour affiner le comportement de `Inky`, faites enfin en sorte que sa vitesse de déplacement (nombre de « frames » utilisées par `move`) soit plus rapide lorsqu'il est effrayé.

4.7.3 Tâches

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données.

Vous vérifierez alors :

1. que `Inky` se déplace en suivant des chemins dans les limites da la grille ;
2. qu'il peut mémoriser le personnage lorsqu'il rentre dans son champs de perception (il commence à le suivre) ;
3. que la position ciblée pour la construction de ces chemins dépend de l'état du fantôme : une case au hasard proche de sa position refuge quand il est effrayé, la position du personnage principal lorsqu'il entre dans son champs de vision et une case au hasard plus éloignée potentiellement de sa position refuge sinon ;
4. que `Inky` se laisse effrayer selon les mêmes modalités que `Blinky` ;
5. que le personnage se fait manger par `Inky` selon des modalités analogues à ce qui est le cas avec `Blinky` ;
6. que `Inky` se fait manger par le personnage principal selon des modalités analogues à ce qui est le cas avec `Blinky` , y compris pour la gestion du score, mais qu'en plus il oublie sa cible.

4.8 `Pinky` le fuyant

`Pinky` est un type de fantôme capable de traquer le personnage de façon plus globale dans le labyrinthe car il ne cherche pas à rester proche de sa position refuge. Il sera par ailleurs capable de fuir le personnage, ce qui le rendra plus difficile à manger. `Pinky` a un comportement analogue à `Inky` aux différences près suivantes :

- Dans le cas où il n'est pas effrayé, et qu'il n'a pas mémorisé le personnage, il va simplement choisir au hasard n'importe quelle position atteignable.
- Lorsqu'il est effrayé, il va chercher à s'éloigner du personnage. S'il a mémorisé ce dernier, il va prendre pour cible un point atteignable au hasard lui permettant d'être au moins à une distance donnée de lui (prenez comme distance
une constante `MIN_AFRAID_DISTANCE` valant 5 par exemple). Sinon, il ciblera n'importe quelle position atteignable au hasard.

Le nombre d'essais pour trouver une position l'éloignant suffisamment du personnage sera limité (utilisez par exemple une constante `MAX_RANDOM_ATTEMPT` valant 200). `Pinky` sera ajouté à l'aire de jeu façon analogue à ce qui est fait pour `Blinky` et `Pinky`.

4.9 Calcul de chemins et obstacles

Les noeuds d'un graphe peuvent être désactivés au moyen de la fonctionnalité `setSignal` de `AreaGraph`. Ils ne seront alors plus considérés comme des étapes possibles d'un plus court chemin. C'est par ce biais qu'il est possible de tenir compte de la présence d'obstacles éventuels sur un chemin. La présence d'une barrière va donc pouvoir ainsi, par exemple, être prise en compte par l'algorithme calculant le plus court chemin. Modifiez le constructeur de `Gate` de sorte ce que si la barrière est fermée, le noeud correspondant à sa position dans le graphe associé à la grille soit désactivé.

4.9.1 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Pour tester cette partie, utilisez le `Level2` (voir la figure 7) où vous aurez commenté la création de l'un des `Gate` emprisonnant l'un des `Pinky`. Vérifiez alors :

1. que `Pinky` se déplace en suivant des chemins dans les limites de la grille ;
2. qu'il peut mémoriser le personnage lorsqu'il rentre dans son champs de perception (il commence à le suivre) ;
3. que lorsqu'il n'est pas effrayé et n'a pas mémorisé le personnage, il peut cibler n'importe quelle position atteignable ;
4. que lorsqu'il est effrayé et qu'il connaît la position du personnage il va essayer de le fuir ;
5. que `Pinky` se laisse effrayer selon les mêmes modalités que `Inky` ;
6. que le personnage se fait manger par `Pinky` selon des modalités analogues à ce qui est le cas avec `Inky` ;
7. que `Pinky` se fait manger par le personnage principal selon des modalités analogues à ce qui est le cas avec `Inky`.
8. que les obstacles sont bien pris en compte dans le calcul des chemins (le `Pinky` complètement enfermé ne devrait pas pouvoir fuir un personnage qui s'en rapproche).

4.10 Validation de l'étape 3

Pour valider cette étape, toutes les vérifications des sections 2.5, 3.2.1, 3.3.2, 3.4.1, 4.5.1, 4.7.3 et 4.9.1 doivent avoir été effectuées.

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

5 Extensions (étape 4)

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

La mise en oeuvre est libre et très peu guidée. Seules quelques suggestions sont données ci-dessous. Une estimation de barème pour les extensions suggérées est donnée, mais n'hésitez pas à nous solliciter pour une évaluation plus précise si vous avez une idée particulière. Un petit bonus sera attribué si vous faites preuve d'inventivité dans la conception du jeu.

Vous pouvez coder vos extensions dans le jeu `SuperPacman` au moyen de niveaux supplémentaires ou dans un nouveau jeu utilisant la logistique que vous avez mise en place dans les étapes précédentes.

Vous prendrez soin de **commenter soigneusement** dans votre `README`, les aires accessibles ainsi que les modalités de jeu. Nous devons notamment savoir quels contrôles utiliser et avec quels effets sans aller lire votre code. Voici un exemple de jeu auquel vous pourriez aboutir ainsi qu'un `README` (partiel) correpondant qui explique comment jouer :

- vidéo d'exemple de jeu : ZeldICDemo.mp4
- fichier `README.md` correspondant : README.md

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

5.1 Nouveaux acteurs

Toutes sortes d'acteurs peuvent être envisagés. En particulier, la composante "signal" peut être tirée à profit pour créer des scénarios de jeu liés à la résolution d'énigmes. Une liste (non exhaustive) de suggestions est données ci-dessous.

- nouveaux personnages avec des modalités de déplacement et de comportement spécifiques ; pouvant être hostiles ou amicaux à l'égard du joueur ; (~4pts/personnage) ;
- passages permettant de se téléporter vers certains niveaux/endroits du jeu ; (~2pts)
- divers types de murs (par exemple des murs de labyrinthe momentanéments traversables ou qui peuvent être détruits dans certaines conditions) ; (~2pts à 4 pts)
- divers acteurs comme des projectiles ou des acteurs pouvant servir de signaux (torches, plaques de pression, boutons-pressions, bombes faisant exploser des murs etc.) ; (~4pts/acteur)

- signaux avancés pour puzzle (oscillateurs , signaux avec retardateur) : un oscillateur est un signal dont l'intensité varie au cours du temps ; ((~4pts/signal)
- changement des modalités de déplacement du personnage principal en fonction de son état (par exemple augmentation temporaire de sa vitesse) ; (~1 à 2pts)
- complexifier les comportements (par exemple doter les acteurs de la capacité de lancer des projectiles ~2pts) ;
- ajouter de nouveaux type de cellules avec des comportements appropriés (eau, glace, feu, etc.) ; (~2pts/cellules)
- ajouter de nouveaux contrôles avancés (interactions, actions, déplacements, etc.) ; (~2pts/controle basique, et plus en fonction de la nature de l'extension) ;
- ajouter des événements aléatoires (personnages, signaux, etc.) ; (~ 4pts)
- etc.

5.2 Générateur automatique de labyrinthe (~ 10pts)

Des labyrinthes peuvent être générés aléatoirement(https://en.wikipedia.org/wiki/Maze_generation_algorithm. Vous pouvez implémenter de tels algorithmes pour la génération de nouveaux niveaux de jeux. Le plus simple est alors d'enrichir les modalités de construction des grilles (on pourrait les construire par algorithme plutôt qu'en utilisant des couleurs dans une image, dans certains cas).

5.3 Pause et fin de jeu (~2 à 4pts)

Mettre en place un système de pause et de reprise du jeu ou gérer les fin de partie (typiquement quand le personnage n'a plus de point de vie) compléterait naturellement votre jeu.

La notion d'aire peut-être exploitée pour introduire la mise en pause des jeux. Sur requête du joueur, le jeu peut basculer en mode pause puis rebasculer en mode jeu. Vous pouvez également introduire la gestion de la fin de partie (si le personnage n'a plus de points de vie).

5.4 Mode multi-joueur(~5 à 10 pts)

Deux super-pacman pourraient prendre place sur la grille (et interagir de façon cooperative ou hostile). **Ne pas inclure cependant de fonctionnalités réseau** qui dépassent de trop loin la portée de ce cours.

En réalité, la base que vous avez codée peut être enrichie à l'envie. Vous pouvez aussi laisser parler votre imagination, et essayer vos propres idées. S'il vous vient une idée originale qui

vous semble différer dans l'esprit de ce qui est suggéré et que vous souhaitez l'implémenter pour le rendu ou le concours (voir ci-dessous), il faut la faire valider avant de continuer (en envoyant un mail à CS107@epfl.ch).

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

5.5 Validation de l'étape 4

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

6 Concours

Ceux d'entre vous qui ont terminé le projet avec un effort particulier sur le résultat final (gameplay intéressant, effets visuels, extensions intéressantes/originales etc.) peuvent concourir au prix du « meilleur jeu du CS107 ».[4]

Si vous souhaitez concourir, vous devrez nous envoyer d'ici au **18.12 à 9 :00** un petit « dossier de candidature » par mail à l'adresse **cs107@epfl.ch**. Il s'agira d'une description de votre jeu et des extensions que vous y avez incorporées (sur 2 à 3 pages en format .pdf avec quelques copies d'écran mettant en valeur vos ajouts).

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