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

S. Abuzakuk, J. Sam, B. Jobstmann

Version 1.7

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 jeu de base qu'il vous sera demandé de créer est fortement inspiré des célèbre jeux de type Roguelike. La figure 1 montre un exemple de l'ébauche de base[1] 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 obligatoires :

- Étape 1 (« ICRogue de base ») : au terme de cette étape vous aurez créé, en utilisant les outils du moteur de jeu fourni, une instance basique de jeu avec un acteur se déplaçant dans une salle et capable de collecter des objets et lancer des projectiles ;
- Étape 2 (« Exploration d'un niveau ») : lors de cette étape le joueur deviendra capable d'explorer un *niveau de jeu* constitué de plusieurs salles ;
- Étape 3 (« Défis et ennemis ») : il s'agira de modéliser des *défis* qu'il faudra relever pour passer d'une salle à l'autre. Chemin faisant, il faudra collecter des objets utiles permettant de relever les défis ou de dévérouiller des portes. Le but est d'atteindre la salle ultime du « Boss ». Vaincre ce dernier est ce qui conditionnera le passage à un niveau suivant ou gagner.
- Étape 4 (« Génération aléatoire de niveaux ») : l'une des spécificités des jeux de type « RogueLike » réside dans la génération procédurale de niveaux de jeux. C'est à cette tâche que sera dédiée cette étape.
- Étape 5 (Extensions, facultif) : 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.

Coder quelques extensions (à choix) vous permet de gagner des points bonus et/ou de valoriser votre projet pour participer au concours.

[1] voir la vidéo de démonstration

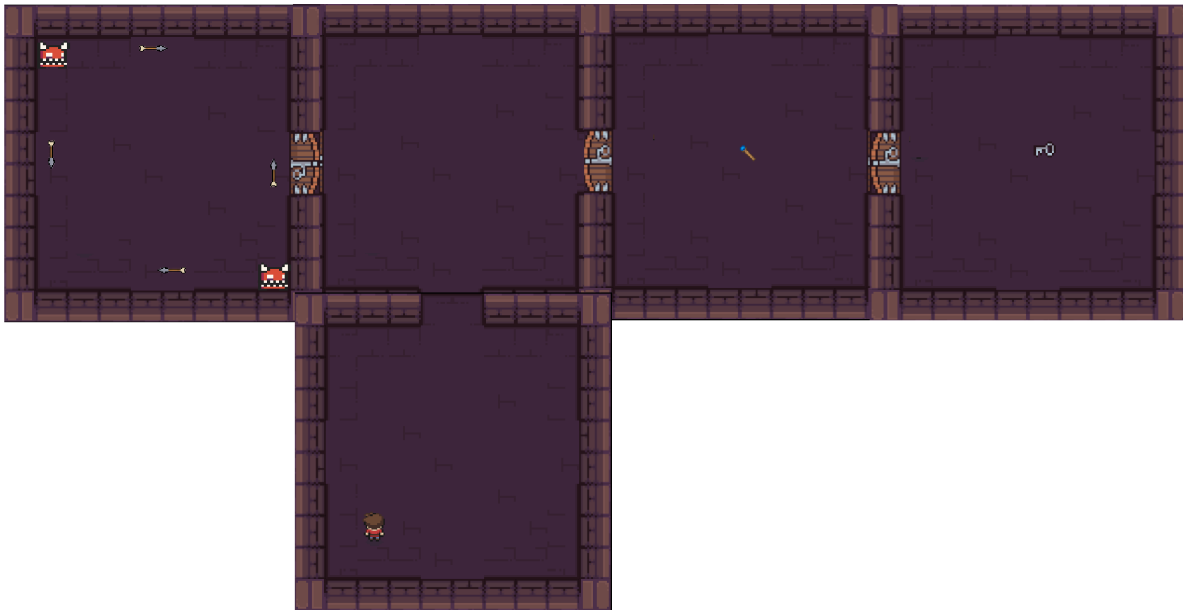


FIG. 1 : Exemple d'un niveau de jeu avec cinq salles. Le « joueur » se déplace de salle en salle (et seule la salle courante est visible). Son but sera de relever les défis associés à chaque salle pour atteindre et résoudre la salle ultime (ici celle en haut à gauche). Pour relever les défis et accéder aux diverses salles, il devra collectionner des objets et se défendre face à des ennemis.

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 votre IDE 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). 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.

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

4. 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.

2 ICRogue de base (étape 1)

Le but de cette étape est de commencer à créer votre propre petit jeu « ICRogue » dans une modeste lignée des jeux de type « RogueLike ».
Cette version de base contiendra un personnage principal capable de se promener dans une salle et d'y collecter des objets. La collecte d'objets 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 objets collectionables que ce dernier pourra ramasser en passant par dessus (« interactions de contact ») ou en les invoquant (« interaction à distance ») ;
- des projectiles qu'il pourra lancer.

Vous travaillerez dans le sous-paquetage fourni `game.icrogue`.

2.1 Préparation du jeu `ICRogue`

Préparez un jeu `ICRogue` en vous inspirant de `Tuto2` (voir la solution fournie dans le sous-paquetage `game.tutosSolution`). Ce dernier sera constitué pour commencer de :

- La classe `ICRoguePlayer` qui modélise un personnage principal, à placer dans `game.icrogue.actor` ; laissez cette classe vide pour le moment, nous y reviendrons un peu plus bas.
- La classe `ICRogue`, équivalente à `Tuto2` à placer dans le paquetage `game.icrogue` ; cette classe aura un `ICRoguePlayer` en guise de personnage. N'oubliez pas d'adapter la méthode `getTitle()` qui retournera un nom de votre choix (par exemple *`"ICRogue"`*).
- La classe `ICRogueRoom` équivalente à `Tuto2Area`, à placer dans le sous-paquetage `game.icrogue.area`.
- La classe `Level0Room` héritant de `ICRogueRoom`, à placer dans un sous paquetage `game.icrogue.area.level0.rooms` (équivalente de la classe `Ferme` ou `Village` de `game.tutosSolution.area.tuto2`).
- La classe `ICRogueBehavior` analogue à `Tuto2Behavior` à placer dans `game.icrogue` et qui contiendra une classe publique `ICRogueCell` équivalente de `Tuto2Cell`.

Différentes petites retouches vont cependant être nécessaires pour s'adapter à l'esprit du nouveau jeu. Il est conseillé d'être méticuleux dans la mise en oeuvre de ces adaptations (pour partir du bon pied ;-)).

2.1.1 Adaptation de `ICRogueRoom` et `Level0Room`

Les salles vont en fait faire partie d'une « carte » que nous introduirons lors de la modélisation des niveaux, comme illustré par la figure 2. Une `ICRogueRoom` aura ainsi naturellement comme caractéristique additionnelle ses *coordonnées dans cette carte* (de type `DiscreteCoordinates`).
Une `Level0Room` hérite donc de `ICRogueRoom` une position `[x][y]` sur une carte. Afin de pouvoir identifier une `Level0Room` de façon plus parlante dans ce contexte, vous modifierez sa définition de `getTitle()` de sorte à ce qu'elle retourne la chaîne *`"icrogue/level0"`*

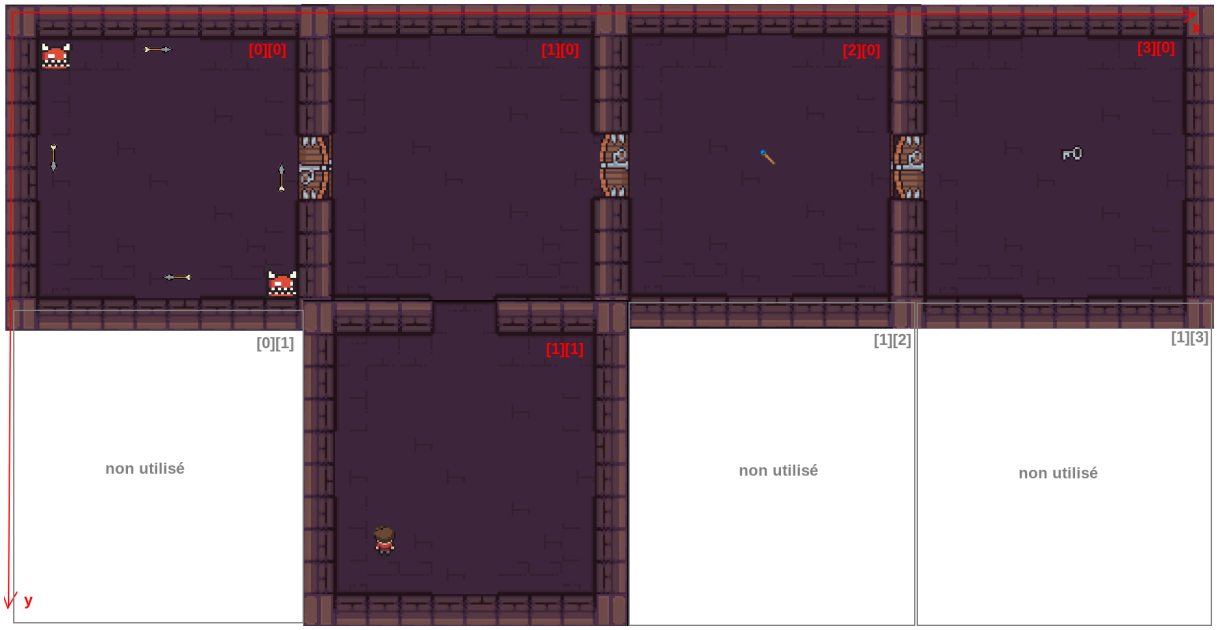


FIG. 2 : Exemple de « carte » de salles dans un niveau de jeu

auquel on concatène les coordonnées `x` et `y`. Par exemple, si la position de la `Level0Room` sur la carte est `[0][1]`, alors le titre sera *`"icrogue/level001"`*.
Le titre de la salle sera ainsi dépendant de sa position. En revanche, la grille associée à une `Level0Room` sera toujours la même, *quelque soit sa position*. Le nom de la grille (« behavior ») sera par exemple *`"icrogue/Level0Room"`* pour *toutes* les `Level0Room`.
Contrairement à ce qui était le cas dans `Tuto2`, il y a donc ici une distinction qui doit se faire entre le titre de l'aire et le nom de la grille associée[2]. Pour parvenir à gérer cette distinction, vous doterez `ICRogueRoom` d'un attribut permettant de stocker le nom de la grille (`behaviorName`) et d'un constructeur permettant d'initialiser les deux attributs spécifiques :

```
ICRogueRoom(String behaviorName, DiscreteCoordinates roomCoordinates)
```

`Level0Room` sera dotée quant à elle d'un constructeur :

```
Level0Room(DiscreteCoordinates roomCoordinates)
```

qui appelera le constructeur de sa super classe en lui donnant *`"icrogue/Level0Room"`* en guise de valeur pour l'attribut `behaviorName`.

Attention : la méthode `begin` de `ICRogueRoom` au moment de créer la grille `ICRogueBehavior` va utiliser comme paramètre `behaviorName` et non plus le titre de l'aire. De même l'acteur `Background` de `Level0Room` se construira au moyen du `behaviorName` (et non du titre) :

```
registerActor(new Background(this, behaviorName)); // avec un
    getter c'est mieux
```

Notez qu'il n'y pas d'acteur `Foreground` pour les `Level0Room`. Enfin, n'oubliez pas de redéfinir `getCameraScaleFactor` dans `ICrogueRoom` (avec la valeur de 11 par exemple).

[2]le titre de l'aire est ce qui l'identifie dans l'ensemble des aires d'un jeu, le `behaviorName` est ce qui identifie la grille associée ; cette grille est ici toujours la même pour toute aire d'un certain type

2.1.2 Adaptation de `ICRogue`

Un jeu de type `ICrogue` est constitué de niveaux et chaque niveau est appelé à contenir plusieurs salles.
Au lieu d'avoir comme attribut les titres de chacune des salles possibles (comme dans `Tuto2`), le jeu `ICrogue` aura donc plutôt comme attribut un *niveau* . Pour le moment, ce concept n'est pas introduit et l'on va tout simplement assimiler un niveau à une salle unique . En guise et place de niveau, il y aura donc dans `ICRogue` un attribut *salle courante* de type `Level0Room`.
La méthode `createAreas()`, quant à elle, est à remplacer par une méthode `initLevel()`, en charge de créer le "niveau" et dont le rôle est :

- d'affecter à la salle courante une `Level0Room` de coordonnées (0,0) ;
- d'ajouter cette salle à l'ensemble des aires du jeu ;
- de désigner la salle courante comme l'aire courante du jeu (`setCurrentArea`) ;
- de créer un `ICRoguePlayer` (prévoyez cet appel avec un commentaire, nous y reviendrons plus tard) ;
- et de le faire entrer dans l'aire courante (instruction aussi à prévoir avec un commentaire).

Notez qu'il n'est plus nécessaire de centrer la caméra sur le personnage car l'on veut avoir une vue d'ensemble sur toute la salle (si l'on ne fait rien la caméra est centrée sur le centre de l'aire courante).

2.1.3 Adaptation de `ICRogueBehavior`

Dans l'esprit, `ICRogueBehavior` et `ICRogueCell` sont équivalentes à `Tuto2Behavior` et `Tuto2Cell`.
Le type énuméré décrivant le types des cellules et leur « traversabilité » pourra être décrit comme suit :

```
NONE(0,false),        // Should never been used except in the
   toType method
GROUND(-16777216, true), // traversable
WALL(-14112955, false),  // non traversable
HOLE(-65536, true);
```

Néanmoins, la nature du « décor » ne sera désormais plus le seul élément qui va conditionner le déplacement du personnage. Dans le cas de ce nouveau type de jeu, la présence d'un autre acteur qui ne se laisserait pas « marcher dessus » entravera aussi le déplacement du personnage. Un objet se laisse marcher dessus (est traversable) si sa méthode `takeCellSpace()` retourne `false`.
Concrètement, deux entités pour lesquelles la méthode `takeCellSpace()` retourne vrai ne peuvent cohabiter les deux dans une même cellule. Vous devrez donc faire en sorte que la méthode `canEnter()` des `ICRogueCell` le garantisse.
Procédez aux adaptations suggérées ci-dessus dans la classe `ICRogueBehavior`.

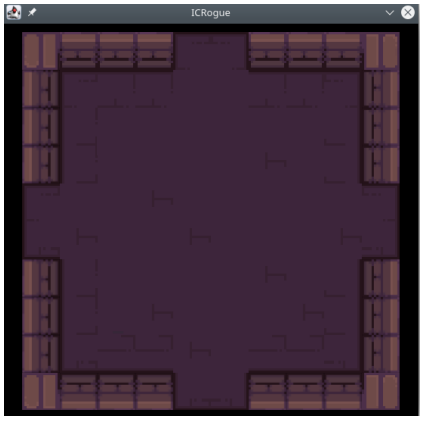


Fig. 3 : Première salle du jeu ICRogue

2.1.4 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Lancez votre jeu `ICRogue`. Vous vérifierez que la salle vide de la figure 3 s'affiche.

2.2 Acteurs des jeux « ICRogue »

Maintenant que les bases fondamentales sont posées, il vous est demandé de commencer à modéliser les acteurs des jeux de type `ICRogue`. Ils seront codés dans le sous-paquetage `game.icrogue.actor`.
Vous considérerez que tous les acteurs impliqués dans un jeu `ICRogue`, les `ICRogueActor`, sont des acteurs qui évoluent sur une grille (`MoveableAreaEntity`) et qu'à ce niveau d'abstraction, ils n'évoluent pas de façon spécifique. Leur aire d'appartenance, leur orientation et leur position de départ sont données à la construction. Les cellules qu'ils occupent se définissent comme pour le `GhostPlayer` (méthode `getCurrentCells()`), mais par défaut, ils sont traversables (`takeCellSpace()` retournant `false`).
En terme de fonctionnalité, tout `ICRogueActor` est capable d'entrer dans une aire donnée à une position donnée et de quitter l'aire qu'il occupe. En tant que `Interactable` il peut être l'objet d'interactions de contact uniquement (à ce niveau d'abstraction).
À ce stade du projet, deux catégories plus spécifiques de `ICRogueActor` sont à introduire : le *personnage principal* et des *projectiles*. Nous nous limiterons pour le moment au personnage principal.

2.2.1 Le personnage principal

La classe `ICRoguePlayer`, suggérée plus haut sera codée pour le moment dans le même esprit que `GhostPlayer`. Vous devez cependant réviser les liens d'héritage car il s'agit évidemment aussi d'un `ICRogueActor`. L'image qui servira à le dessiner dépendra de son orientation. Voici le code permettant d'extraire les `Sprite` nécessaires en fonction de l'orientation :

```
 //bas
new Sprite("zelda/player", .75f, 1.5f, this,
       new RegionOfInterest(0, 0, 16, 32), new Vector(.15f, -.15f));
// droite
```

```
new Sprite("zelda/player", .75f, 1.5f, this,
        new RegionOfInterest(0, 32, 16, 32), new Vector(.15f,
            -.15f));
// haut
      new Sprite("zelda/player", .75f, 1.5f, this,
              new RegionOfInterest(0, 64, 16, 32), new Vector(.15f,
                  -.15f));
// gauche
      new Sprite("zelda/player", .75f, 1.5f, this,
               new RegionOfInterest(0, 96, 16, 32), new Vector(.15f,
                   -.15f));
```

(voir la section 6.6 du tutoriel si vous souhaitez comprendre les détails de ce code).
Un `ICRoguePlayer` se comporte (se met à jour) comme un `ICRogueActor` mais en plus :

- il doit pouvoir être déplacé au moyen des flèches directionnelles à la manière du `GhostPlayer` codé dans le tutoriel ;
- le fait d'appuyer sur la touche `X` lui fait lancer des *boules de feu* : une boule de feu, ayant la même orientation que lui, doit être créée à la position qu'il occupe à chaque fois que la touche `X` est pressée. Pour le moment, prévoyez simplement ce traitement en commentaire.

Enfin, un `ICRoguePlayer` sera non traversable.
Pour pouvoir mettre en œuvre la boule de feu, il vous sera demandé d'introduire plus généralement la notion de *projectile*, ce qui nous occupera dans la section suivante.
À partir du moment où `ICRoguePlayer` est codé, vous pouvez finaliser la méthode `begin` de `ICRogue` (instructions prévues en commentaire) et commencer à tester votre jeu.
Au lancement, un joueur de type `ICRoguePlayer` sera créé à la position qui lui est destinée dans l'aire de démarrage (prenez $(2, 2)$ dans `Level0Room` par exemple). Il sera orienté vers le haut.
Pour faciliter les tests, vous doterez `ICRogue` du contrôle suivant : la touche `R` doit permettre de faire une réinitialisation (« reset ») du jeu c'est à dire le redémarrer dans les mêmes conditions que la toute première fois qu'on le lance (pensez à la modularisation des traitements ici).

2.2.2 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données.
Lancez votre jeu `ICRogue`. Vous vérifierez :

1. que le jeu démarre en affichant le `ICRoguePlayer` selon la figure 4 ;
2. qu'il peut être déplacé librement sur toute l'aire au moyen des flèches directionnelles mais qu'il ne doit pas pouvoir en sortir ;
3. que sa représentation graphique s'adapte bien à son orientation ;
4. qu'il ne peut pas marcher sur les murs du bord ;
5. et que la touche `R` permet de réinitialiser le jeu selon la spécification décrite plus haut.

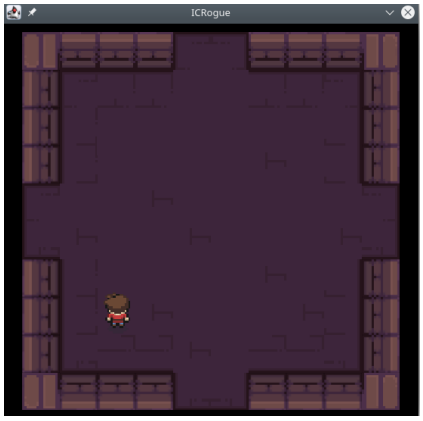


Fig. 4 : Placement du personnage

2.2.3 Projectiles

Les projectiles seront modélisés dans un sous-paquetage `game.icrogue.actor.projectiles`. Les projectiles (classe abstraite `Projectile`) se consument sous certaines conditions (par exemple la boule de feu finit par s'éteindre). Ce sont des `ICRogueActor` que l'on peut dessiner au moyen d'un `Sprite` spécifique. Ils évoluent (méthode `update`) en se déplaçant et sont caractérisés par :

- Le nombre de « frames » utilisées pour le calcul du déplacement ;
- les points de dommages qu'ils peuvent infliger (un entier) ;
- un état indiquant s'ils se sont consumés ou pas (`isConsumed`, un booléen).

Pour le déplacement, il suffira d'invoquer la méthode `move` en lui passant en paramètre le nombre de « frames ».
L'aire d'appartenance d'un projectile, son orientation, sa position de départ dans l'aire, le nombre de points de dommage qu'il inflige et le nombre de « frames » utilisées pour son déplacement sont donnés à la construction. Il doit également être possible de ne pas spécifier les deux dernières valeurs explicitement à la construction. Dans ce cas, elles prendront par défaut les valeurs `DEFAULT_DAMAGE` (valant 1 par exemple) et `DEFAULT_MOVE_DURATION` (valant 10 par exemple) respectivement.
Le `Sprite` associé ne peut avoir de valeur concrète à ce niveau d'abstraction. Il peut également prendre potentiellement plusieurs valeurs pour un même projectile (par exemple une boule de feu pourrait devenir de plus en plus rouge au fur et à mesure qu'elle se déplace). Par conséquent, un « setter » est d'avantage indiqué pour l'initialisation de sa valeur et le concept de projectile est donc abstrait.
Fonctionnellement un projectile se comporte comme un objet qui peut se consumer. L'interface fournie `Consumable` (paquetage `game.icrogue.actor.projectiles`) permet de modéliser cela. Un objet qui se consume offrira typiquement les méthodes `void consume()` codant ce qui se produit lorsque l'objet se consume et une méthode `boolean isConsumed()` permettant de tester si l'objet est consumé.
La méthode `void consume()` d'un projectile consistera simplement à ce stade à attribuer la valeur `true` à l'attribut `isConsumed`.
Les cellules occupées par un projectile se définissent comme pour le `ICRoguePlayer` (méthode `getCurrentCells()`), et par défaut, ils sont traversables (`takeCellSpace` retournant

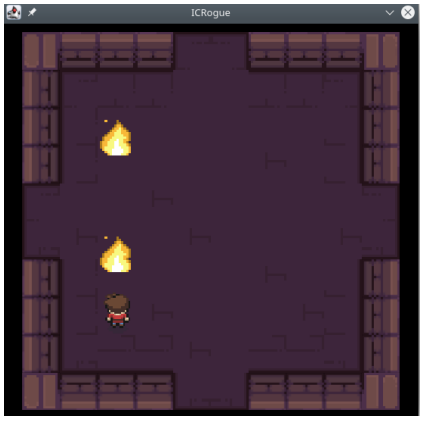


FIG. 5 : Lancement de boules de feu avec la touche `X`.

`false`).
Toutes sortes de projectiles concrets peuvent être envisagées. Il vous est demandé pour cette partie de coder un projectile « boule de feu » (`Fire`) dont les caractéristiques sont les suivantes :

- nombre de points de dommage : 1
- nombre de « frames » pour le déplacement : 5

Le `Sprite` associé peut être extrait des ressources fournies au moyen de la tournure :

```
new Sprite("zelda/fire", 1f, 1f, this,
                new RegionOfInterest(0, 0, 16, 16), new
                   Vector(0, 0)));
```

Une boule de feu disparaît lorsqu'elle se consume (n'existe plus comme acteur).
Une fois la boule de feu codée, vous pouvez compléter le comportement de `ICRoguePlayer` de sorte à ce que la touche 'X' crée une boule de feu à la position du personnage (`getCurrentMainCellCoordinates`) et orientée comme lui.

2.2.4 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données.
Lancez votre jeu `ICRogue`. Vous vérifierez :

1. que la touche `X` permet au personnage de lancer des boules de feu en face de lui ;

À ce stade les boules de feu ne disparaîssent pas encore (ce sera le cas lorsque vous aurez codé les interactions un peu plus bas).

2.3 Objets à collecter

Il s'agit maintenant de modéliser des objets que le personnage principal pourra collecter, ce qui va conditionner le déroulement du jeu. Les objets à collecter seront implémentés dans un sous-paquetage `game.icrogue.actor.items`.
Il vous est demandé d'introduire une classe abstraite `Item` modélisant les objets « ramassables » et héritant de la classe `CollectableAreaEntity` fournie dans la maquette (paquetage `areagame.actor`).

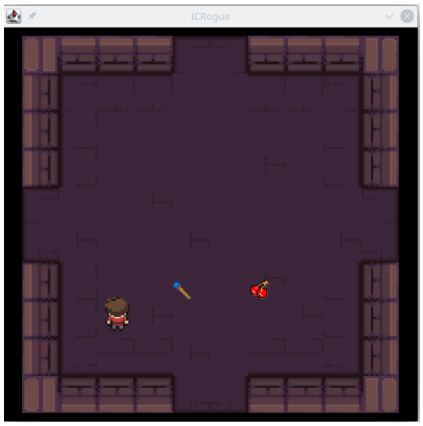


FIG. 6 : Position initiale du « joueur » et des acteurs `Cherry` et `Staff`.

Un `Item` est un objet traversable par défaut. Il est dessiné au moyen d'un `Sprite` qui le caractérise. Il ne se dessine cependant que s'il n'est pas collecté (méthode `isCollected()` de `CollectableAreaEntity`).
En tant que `Interactable`, par défaut, il peut être l'objet d'interactions de contact uniquement[3].
Deux types concrets de `Item` sont à coder à ce stade : les cerises (`Cherry`) et les bâtons (`Staff`). Un bâton est un `Item` qui accepte les interactions à distance (le personnage pourra l'invoquer s'il est suffisamment proche de lui[4]). Son aire d'appartenance, son orientation et sa position de départ sont données comme paramètres à la construction. Le `Sprite` associé se construit au moyen de la tournure :

```
new Sprite("zelda/staff_water.icon", .5f, .5f, this));
```

Une « cerise » est codée dans le même esprit, mais n'accepte pas les interactions à distance. Le `Sprite` associé se construit au moyen de la tournure :

```
new Sprite("icrogue/cherry", 0.6f, 0.6f, this)).
```

Complétez enfin le code de `Level0Room` de sorte à ce qu'elle enregistre à sa création :

- un bâton en position `(4,3)` orienté vers le bas ;
- une cerise en position `(6,3)` orientée vers le bas.

2.3.1 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Lancez votre jeu `ICRogue`. Vous vérifierez que le bâton et la cerise apparaissent (figure 6).

[3] souvenez vous des méthodes `isCellInteractable` et `isViewInteractable`
[4] ce n'est pas à vous de tester la proximité à un objet car cela est fait par la moteur de jeu

2.4 Interactions

Il vous est demandé maintenant d'appliquer le schéma suggéré par le tutoriel[5] pour mettre en place les premières interactions entre les `ICRogueActor` codés jusqu'ici.

2.4.1 Collecte d'objets

Dans un premier temps il s'agit de permettre aux `ICRoguePlayer` de collecter des objets.
Pour cela, commencez par mettre en place le fait que tous les `ICRoguePlayer` deviennent des `Interactor` (ils peuvent faire subir des interactions à des `Interactable`, par exemple pour les ramasser).
Comme `Interactor`, `ICRoguePlayer` doit définir les méthodes :

- `getCurrentCells` : ses cellules courantes (se réduiront à l'ensemble contenant uniquement sa cellule principale (comme vous avez déjà eu l'occasion de l'exprimer) ;
- `getFieldOfViewCells()` : les cellules de son champs de vision consistent en l'unique cellule à laquelle il fait face

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

En tant que `Interactor`, il voudra systématiquement toutes les interactions de contact (`wantsCellInteraction` retourne toujours vrai). Le fait qu'il soit demandeur d'interactions à distance (`wantsViewInteraction`) sera conditionné par l'utilisateur du jeu : la touche `W` sera utilisée pour indiquer que le personnage veut une interaction à distance. Par exemple, si notre personnage est en face d'un bâton, pour indiquer que l'on souhaite qu'il le ramasse, on appuie sur la touche `W`. Cette touche ne s'occupera évidemment pas que du cas spécifique du bâton. Elle sera uniquement employée pour faire basculer le personnage en mode « demande d'interaction à distance » (et `wantsViewInteraction` retournera `true` ou `false` selon la valeur de ce mode)
Intéressons-nous maintenant à la gestion concrètes des interactions.
Dans le sous paquetage `game.icrogue.handler`, complétez l'interface `ICRogueInteractionHandler` héritant de `AreaInteractionVisitor`. Cette interface doit fournir une définition par défaut des méthodes d'interaction de tout `Interactor` du jeu de `ICRogue` avec :

- une cellule du jeu (`ICRogueCell`) ;
- un personnage principal du jeu (`ICRoguePlayer`) ;
- une cerise ;
- un bâton ;
- et une boule de feu.

Ces définitions (par défaut) auront un corps vide pour exprimer le fait que par défaut, l'interaction consiste à ne rien faire. `ICRoguePlayer` en tant que `Interactor` du jeu `ICRogue`, doit fournir le cas échéant une définition plus spécifique de ces méthodes.
Tout `Interactable` concret doit maintenant indiquer qu'il accepte de voir ses interactions gérées par un gestionnaire d'interaction de type `ICRogueInteractionHandler`. Leur méthode `acceptInteraction` (vides jusqu'ici) doit être reformulée dans ce sens (dans chacune des classes concernées) :

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

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

Cette méthode peut contenir plus de code selon les cas (par exemple, on peut raisonnablement considérer que les projectiles consumés n'acceptent plus les interactions).
Pour que `ICRoguePlayer` puisse gérer plus spécifiquement les interactions qui l'intéressent, définissez dans la classe `ICRoguePlayer`, une classe imbriquée privée `ICRoguePlayerInteractionHandler` implémentant `ICRogueInteractionHandler`. Ajoutez-y les définitions nécessaires pour gérer plus spécifiquement l'interaction avec une cerise et avec un bâton.
Ces deux objets devront se voir collectés par interaction :

- la cerise par une interaction de contact ;
- le bâton par une interaction à distance (souvenez-vous que le personnage peut exprimer le souhait d'une interaction à distance au moyen de 'W').

Note : le codage de ces méthodes ne devrait pas faire plus que deux à trois lignes.
Conformément au tutoriel 3, pour que cela fonctionne, il faut que :

- `ICRoguePlayer` ait pour attribut son gestionnaire d'interaction spécifique (de type `ICRoguePlayerInteractionHandler`) ;
- que sa méthode `void interactWith(Interactable other, boolean isCellInteraction)` délègue la gestion de cette interaction à son gestionnaire (`handler` ci-dessous) :

  ```
  other.acceptInteraction(handler, isCellInteraction);
  ```

  souvenez vous que cette méthode `void interactWith` est invoquée automatiquement par le noyau de jeu pour tout `void Interactor` et ce, pour tout `void Interactable` avec lequel il est en contact ou qui est dans son champs de vision (revoir au besoin la section 6.3.1 du tutoriel).

2.4.2 Les projectiles comme `Interactor`

Vous comprenez maintenant les mécanismes généraux permettant à un acteur d'interagir avec un autre. D'autres acteurs que le personnage principal peuvent bien sûr être « demandeurs d'interaction ». Il s'agit maintenant de modéliser le fait que les projectiles sont des objets qui peuvent faire subir des interactions aux autres acteurs, c'est à dire qu'ils se comportent comme des `Interactor`.
Pour cela, `Projectile` redéfinit les méthodes `getCurrentCells` et `getFieldOfViewCells()` comme le fait `ICRoguePlayer`.
En tant que `Interactor`, un projectile voudra systématiquement toutes les interactions de contact et à distance (`wantsCellInteraction` et `wantsViewInteraction` retournent toujours vrai).
La boule de feu en tant que projectile concret aura un gestionnaire d'interaction qui mettra en œuvre l'interaction spécifique suivante : si la boule interagit (à distance) avec une cellule de type `WALL` ou `HOLE` elle se consume.
Modifiez enfin le comportement de `ICRoguePlayer` de sorte à ce qu'il ne puisse lancer de boules de feu au moyen de la touche `X` que s'il a collecté un bâton au préalable.

A ce stade vous modéliserez cette situation de façon très simple : par exemple un attribut booléen du personnage indiquant s'il a ramassé ou pas un bâton fera l'affaire. Cela peut bien sûr être sophistiqué à souhait plus tard.

2.4.3 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données.
Lancez votre jeu `ICRogue`. Vous vérifierez :

1. que le `ICRoguePlayer` se comporte comme pour les étapes précédentes mais qu'il peut ramasser une cerise en marchant dessus et le bâton s'il est dans son champ de vision et que la touche `W` a été appuyée ;
2. qu'il ne peut pas marcher sur le bâton ;
3. qu'il ne peut plus lancer de boules de feu avec la touche `X` sans avoir au préalable ramassé un bâton ;
4. que les boules de feu s'éteignent (disparaissent) en atteignant un mur (ou un trou).
5. et que les objets ramassés disparaissent (visuellement).

Question 1

La logistique mise en place, telle 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 œuvre un jeu de Pacman 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, toutes les vérifications des sections 2.1.4, 2.2.2, 2.2.4, 2.3.1 et 2.4.3 doivent avoir été effectuées.

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

Cette partie reste délibérément guidée car elle constitue le coeur du fonctionnement du jeu. Tous les détails ne sont cependant évidemment pas donnés.

3 Exploration d'un niveau (étape 2)

Dans les jeux de type « Roguelike », le personnage principal doit explorer de multiples salles. Il s'agira dans la suite du projet de modéliser le fait que :

- le jeu est constitué de *niveaux* et que chaque niveau est constitué d'un ensemble de *salles* ;
- le passage d'une salle à l'autre se fait au moyen de *connecteurs* (points de passages) ;
- qu'à chaque salle est associé un défi, comme supprimer un ennemi. Lorsque ce défi est résolu, tous les connecteurs s'ouvrent, sauf ceux qui nécessitent la possession d'une clé (ceux qui sont verrouillés). Le personnage doit donc avoir au préalable ramassé une telle clé (possiblement dans une autre salle) ;
- il existe une salle particulière (salle du « boss ») dont il faut avoir trouvé la clé pour y pénétrer. Le but est de vaincre cet ennemi, en s'étant suffisamment équipé au préalable dans les autres salles pour pouvoir le faire. Lorsque le « boss » est vaincu, le niveau est réussi et il est possible de transiter au niveau suivant.

Lors de cette étape, il s'agit de commencer à modéliser la notion de *connecteurs*, de *salle avec connecteur* et de *niveau* de jeu en faisant abstraction des défis et des ennemis.

3.1 Points de passage entre salles

Nous appellerons « connecteurs » les points de passage d'une salle à l'autre. Ils peuvent être fermés, ouverts, verrouillés ou invisibles. Il est naturel de les modéliser comme des acteurs (`AreaEntity`), car il ne s'agit pas uniquement d'éléments du décor : ils peuvent en tant que points de passage avoir des *comportements*, comme passer de l'état ouvert à l'état fermé fonction de conditions complexes.
Un connecteur est verrouillé lorsque son ouverture est conditionnée par la possession d'une clé. Il faut donc commencer par compléter un peu la hiérarchie de `Item`.

3.1.1 Les clés

Les clés (classe `Key`) sont semblables en tout point à des `Cherry`. La seule différence est qu'une `Key` est caractérisée par un identificateur entier (initialisé par une valeur passée en paramètre du constructeur). Le `ICRoguePlayer` les ramasse en marchant dessus. Il doit mémoriser les identifiants des clés ramassées car c'est ce qui lui permettra dans certains cas d'ouvrir des points de passage.

3.1.2 Les connecteurs

Il vous est donc demander d'introduire le concept de `Connector` (dans le sous paquetage `icrogue.actor`).

Un `Connector` est caractérisé spécifiquement (et au minimum) par :

- un état (un type énuméré avec les valeurs `OPEN`, `CLOSED`, `LOCKED` et `INVISIBLE` est une bonne option ici) ;
- le nom de l'aire de destination (un `String` ; souvenez vous que chaque aire à un nom retourné par sa méthode `getTitle()`) ;
- les coordonnées d'arrivée dans l'aire de destination (il s'agit des coordonnées de la cellule où l'on arrive donc des `DiscreteCoordinates`) ;
- et l'identificateur de la clé qui permet de l'ouvrir (un entier). Par défaut il n'y a pas besoin de clé (valeur de l'identifiant : une constante `NO_KEY_ID` valant un entier par exemple).

L'aire d'appartenance d'un connecteur, sa position et son orientation seront donnés comme paramètres à la construction et il est invisible par défaut au moment de sa création.
Le dessin du connecteur dépend de son état : on dessinera un `Sprite` spécifique s'il est invisible, verrouillé ou fermé et rien sinon. Le code pour extraire les sprites en fonction de l'état et les initialiser dans le constructeur vous est donné pour simplifier :

```
// pour invisible:
 new Sprite("icrogue/invisibleDoor_"+orientation.ordinal(),
 (orientation.ordinal()+1)%2+1, orientation.ordinal()%2+1, this);
 // pour fermé:
 new Sprite("icrogue/door_"+orientation.ordinal(),
 (orientation.ordinal()+1)%2+1, orientation.ordinal()%2+1, this);
// pour verrouillé
 new Sprite("icrogue/lockedDoor_"+orientation.ordinal(),
                (orientation.ordinal()+1)%2+1, orientation.ordinal()%2+1,
                   this);
```

En tant que `Interactable`, un `Connector` est défini comme un objet traversable s'il est dans l'état ouvert. Il accepte toujours les interactions à distance. Le corps de sa méthode `getCurrentCells` vous est donnée par simplification :

```
DiscreteCoordinates coord = getCurrentMainCellCoordinates();
return List.of(coord, coord.jump(new
    Vector((getOrientation().ordinal()+1)%2,
    getOrientation().ordinal()%2)));
```

qui implémente le fait que le connecteur occupe sa cellule principale et celles immédiatement à sa gauche et à sa droite.

3.2 Salles avec connecteurs

La classe `ICRogueRoom`, ébauchée à l'étape précédente, représente le concept abstrait de « salle dans un jeu ICRogue ». Il vous est demandé maintenant d'étoffer un peu ce concept et de modéliser le fait qu'il est également caractérisé par *un ensemble de connecteurs* (un tableau dynamique).
La liste des coordonnées des connecteurs, la liste de leur orientations respectives ainsi que les coordonnées de la salle dans la carte sont aussi données comme paramètres au constructeur d'une `ICRogueRoom` :

```
ICRogueRoom(List<DiscreteCoordinates> connectorsCoordinates,
            List<Orientation> orientations,
            String behaviorName, DiscreteCoordinates roomCoordinates)
```

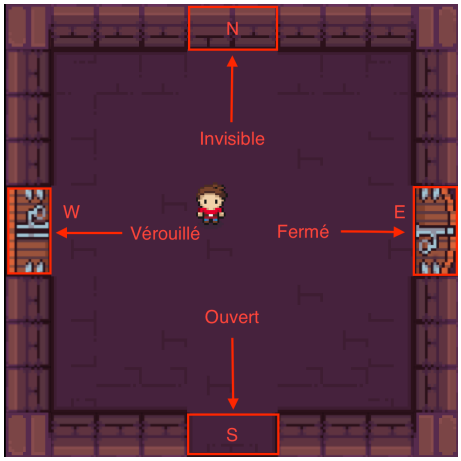


FIG. 7 : Les `Level0Rooms` ont toutes 4 connecteurs placés aux même endroits : Ouest (W), Sud (S), Est (E), Nord (N)

Ce constructeur devra évidemment utiliser ces données pour initialiser son ensemble de connecteurs.
Afin de faciliter les implémentations ultérieures, le constructeur de `ICRogueRoom` stockera les connecteurs dans son ensemble de connecteurs dans l'ordre indiqués par ses paramètres. Par exemple, construire une `ICRogueRoom` en lui passant en paramètre `{(4,5),(2,3)}` pour les positions des connecteurs et `{LEFT, DOWN}` pour leur orientation stockera le connecteur en position `(4,5)` est orienté à gauche (`LEFT`) comme premier connecteur de son ensemble.
La méthode `createArea` de `ICRogueRoom` devra bien sûr faire en sorte que les connecteurs soient enregistrés comme acteurs afin que leur comportement puisse être simulé.

3.2.1 Connecteurs spécifiques

A priori il peut y avoir des salles avec des ensembles quelconques de connecteurs situés à des endroits quelconques. Pour simplifier la mise en place et les tests, nous allons partir de l'idée que les `Level0Room` sont des salles qui ont toutes 4 connecteurs, placés dans un ordre et à des endroits précis (voir figure 7).
Introduisez et complétez de façon adéquate le code d'un type énuméré `Level0Connectors` dans `Level0Room` et dont le début de la définition est :

```
public enum Level0Connectors implements ConnectorInRoom {
        // ordre des attributs: position, destination, orientation
        W(new DiscreteCoordinates(0, 4),
          new DiscreteCoordinates(8, 5), Orientation.LEFT),
        S(new DiscreteCoordinates(4, 0),
          new DiscreteCoordinates(5, 8), Orientation.DOWN),
        E(new DiscreteCoordinates(9, 4),
          new DiscreteCoordinates(1, 5), Orientation.RIGHT),
        N(new DiscreteCoordinates(4, 9),
          new DiscreteCoordinates(5, 1), Orientation.UP);
       ...
```

Ce type modélise les caractéristiques des `Connector` d'une `Level0Room` à savoir : le fait

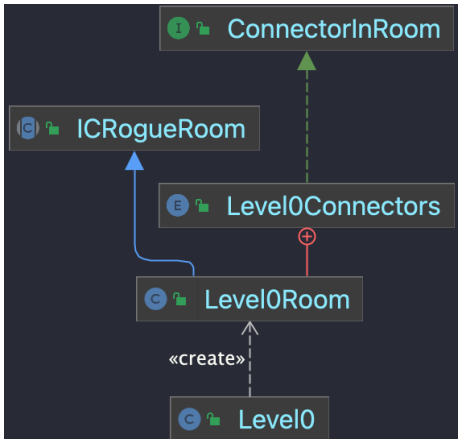


FIG. 8 : `Level0Room` hérite de `ICRogueRoom`. Elle contient une type énuméré `Level0Connectors` qui décrit les caractéristiques de ses connecteurs. Il est possible d'appliquer à chaque valeur du type énuméré les méthodes `getIndex` et `getDestination`. L'existence de ces méthodes est dictée par l'interface `ConnectorInRoom`. `getIndex` permet d'identifier un connecteur par sa position dans l'ensemble des connecteurs de la salle. `Level0` est un niveau (codé en section 3.5). C'est lui qui créera les `Level0Rooms`.

qu'il y a 4 connecteurs, que le connecteur à la position zéro dans l'ensemble des connecteurs est celui identifié par W etc. L'idée est d'utiliser ce type énuméré pour créer les connecteurs dans un ordre donné et avec des caractéristiques données.
L'interface fournie `ConnectorInRoom` permet de manipuler les valeurs de l'énumération au travers d'un type plus abstrait qui leur impose de fournir :

- une méthode `DiscreteCoordinates getDestination()` qui retourne les coordonnées de destination du connecteur correspondant à la valeur du type énuméré ;
- et une méthode `int getIndex()` donnant l'indice de ce connecteur dans l'ensemble des connecteurs de la salle. Cet indice correspond simplement à la position dans le type énuméré (souvenez vous de `ordinal()`).

Définissez ensuite dans l'énumération `Level0Connectors` les méthodes :

```
static List<Orientation> getAllConnectorsOrientation()
static List<DiscreteCoordinates> getAllConnectorsPosition()
```

retournant respectivement la liste des orientations et des positions des connecteurs **dans leur ordre de définition dans le type énuméré**.
Ces deux méthodes devront être utilisées dans le constructeur de `Level0Room` pour qu'il invoque de façon appropriée le constructeur de sa super-classe. Ainsi, pour ce type précis de salle, si la méthode `getIndex()` imposée par l'interface `ConnectorInRoom` retourne une position dans le type énuméré, cela correspondra à la position de l'objet correspondant dans la liste des connecteurs de la salle.

Indication : le type énuméré de `Level0Connector` donne l'orientation vers laquelle le connecteur mène. Lors de la création du connecteur par le constructeur de `ICRogueRoom`, il faudra associer au connecteur l'opposé de cette orientation (méthode `opposite()`. Ainsi, si le connecteur `W` a pour orientation associée (`Orientation.Left`) c'est parcequ'il mène le personnage vers la gauche mais son `Sprite` doit être affiché comme allant vers la droite (à

FIG. 9 : De gauche à droite : connecteurs invisibles, ouverts, fermés/verrouillés

cause de la vue du dessus du jeu).
La figure 8 illustre la conception attendue pour cette partie.

Retouches à `ICRogueRoom` : Nous allons tester cette partie de façon un peu *ad hoc* en procédant au préalable à quelques retouches à la classe `ICRogueRoom`.
En effet, comme nous n'avons pas encore modélisé la notion de défi et de niveau, la logique qui régit l'ouverture et la fermeture des connecteurs ne peut pas encore être mise en oeuvre. Nous allons donc "tricher" un peu. Ajoutez pour cela les contrôles suivants à la méthode `update` de `ICRogueRoom` :

- si l'on appuie sur la touche `O`, tous les connecteurs passent à l'état ouvert ;
- si l'on appuie sur la touche `L`, le connecteur à la position zéro, se verrouille avec une clé d'identifiant 1 ;
- et si l'on appuie sur la touche `T`, l'ensemble des connecteurs "switchent" d'états (passent de ouvert à fermé et vice versa, les connecteurs verrouillés doivent le rester).

3.2.2 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Vous vérifierez alors (voir la figure 9) :

1. que la salle commence par s'afficher en étant complètement fermée (connecteurs dans l'état invisible) ;
2. que la touche `O` permet de les ouvrir et la touche `T` de tous les fermer ;
3. que la touche `L` permet de verrouiller le connecteur ouest (W), il apparaît alors avec un visuel différent des connecteurs fermé) ;
4. et qu'appuyer à nouveau sur la touche `T` permet de tout ouvrir/fermer sauf le connecteur verrouillé.

3.3 Niveaux

Un niveau, matérialisé par une classe `Level` (à coder dans le sous-paquetage `icrogue.area`), est caractérisé par :

- une « carte » de salles (tableau à deux dimensions `width x height` de `ICRogueRoom`) ;

- les coordonnées d'arrivée dans les salles (on présume par simplification que c'est la même pour toutes les salles d'un niveau, `DiscreteCoordinates`) ;
- la position sur la carte de la salle du « boss »[6] ;
- et le nom de la salle de départ du niveau (qui correspondra à son titre).

Parmi les fonctionnalités utiles associées à un niveau, il vous est demandé de coder celles listées ci-dessous. Afin de préserver l'encapsulation vous veillerez à définir ces méthodes comme protégées. Vous ajouterez toutes les méthodes nécessaires en veillant toujours à préserver l'encapsulation : en particulier, vous n'introduirez pas de getters intrusifs (comme par exemple un getter retournant les salles d'un niveau ou les connecteurs d'une salle).

- `void setRoom(DiscreteCoordinates coords, ICRogueRoom room)` permettant d'affecter la salle `room` à la position `[coords.x][coords.y]` de la carte ;
- `void setRoomConnectorDestination(DiscreteCoordinates coords, String destination, ConnectorInRoom connector)` : permettant d'affecter une destination au connecteur à la position `connector.getIndex()` dans la salle à la position `[coords.x][coords.y]` de la carte (en clair, permettant par exemple de faire en sorte que le 3ème connecteur de la salle `[coords.x][coords.y]` mène vers la salle *`"icrogue/level000"`*) ;
- `void setRoomConnector(DiscreteCoordinates coords, String destination, ConnectorInRoom connector)` faisant la même chose que la précédente mais marquant en plus le connecteur comme fermé ;
- `void lockRoomConnector(DiscreteCoordinates coords, ConnectorInRoom connector, int keyId)` permettant d'affecter une clé au connecteur à la position `connector.getIndex()` dans la salle `[coords.x][coords.y]` de la carte ; cette méthode marquera aussi ce connecteur comme verrouillé ;
- et un « setter » qui permet de donner une valeur au nom de la salle de départ à partir de coordonnées discrètes. Par exemple si le paramètre du « setter » vaut `[0][0]`, il affectera au nom de la salle de départ le titre de la salle à la position `[0][0]` de la carte).

Le constructeur d'un `Level` prendra en paramètre les coordonnées d'arrivée dans les salles ainsi que les dimensions de la carte. Il fera en sorte que la position de la salle du « boss » sur la carte soit par défaut `[0][0]`. Enfin, il invoquera une méthode `generateFixedMap`, que l'on ne sait pas définir à ce niveau d'abstraction mais qui aura pour vocation de remplir la carte de salles.

Question 2

Quel est selon vous l'avantage d'avoir eu recours à l'interface `ConnectorInRoom` ? Quelles seraient les alternatives de conception que vous pourriez envisager pour identifier un connecteur dans une salle.

La classe `Level` est censée représenter un niveau de jeu abstrait. Dans ce qui suit, il s'agira de créer un type concret de niveau, `Level0`. Pour rendre les choses un peu plus intéressantes, nous allons d'abord diversifier un peu les salles qui peuvent constituer un tel type de niveau.

[6] Rappelons que le but ultime de jeux de type « RogueLike » est d'atteindre une telle salle en s'étant suffisamment équipé au préalable !

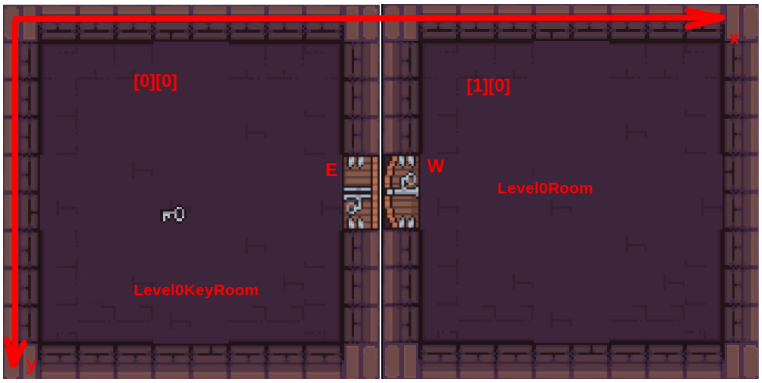


FIG. 10 : Les deux salles sont des `Level0Room`. La salle `[0][0]` est associée à une clé. Cette clé sert à déverrouiller son connecteur « Est » (E).

3.4 Hiérarchie de salles

Vous l'aurez compris, la classe `Level0Room` modélise en fait une salle basique du futur niveau `Level0`. Il vous est demandé maintenant de complexifier et spécialiser un peu ce concept en introduisant (dans le sous-paquetage `icrogue.area.level0.rooms`) les classes :

- `Level0ItemRoom` : qui sont des `Level0Room` abstraites[7] contenant une liste de `Item` ; il devra être possible d'ajouter un `Item` donné à l'ensemble ;
- `Level0KeyRoom` et `Level0StaffRoom` : qui sont des salles avec un `Item` unique : une clé (`Key`) pour la première et un bâton (`Staff`) pour la seconde. La position de cet `Item` unique sera la même pour toutes les instances créées (par exemple `[5][5]`). Les constructeurs prendront en paramètre les coordonnées de la salle sur la carte (et l'identificateur de la clé pour `Level0KeyRoom`). Ils seront en charge de créer leur unique `Item`.

3.5 Premier niveau concret : Level0

Il vous est maintenant demandé de programmer un premier niveau de jeu concret nommé `Level0` (à placer dans un sous-paquetage `icrogue.area.level0`). Il s'agira bien sûr aussi d'un `Level`. Pour être instantiable, `Level0` doit définir concrètement la méthode `generateFixedMap`.
Afin de faciliter vos tests, il vous est demandé de créer deux petites cartes au moyen deux méthodes utilitaires `generateMap1` et `generateMap2`. La méthode `generateFixedMap` de `Level0` pourra invoquer l'une ou l'autre à choix (vous pouvez y invoquer les deux méthodes et commenter l'une ou l'autre).
La méthode `generateMap1` devra créer la carte de la figure 10 et `generateMap2` celle de la figure 11. Vous trouverez sous ce lien un exemple de comment coder ces méthodes (à adapter à votre code pour ce qui est des clés typiquement).
On suppose que la salle de départ a toujours les mêmes coordonnées pour les niveaux de type `Level0` (une constante commune valant `[1][0]` par exemple). La position d'arrivée du personnage dans les salles du niveau aura aussi un valeur par défaut (par exemple `[2][2]`). La classe `Level0` aura un constructeur par défaut utilisant ces données et créant une carte

[7] l'intérêt d'en faire un classe abstraite deviendra plus clair lorsque nous lierons les salles aux défis

FIG. 11 : La salle `[0][0]` est verrouillée par la clé de la salle `[3][0]`.

4x2. Ces modalités de construction seront généralisées lors des étapes ultérieures.

3.6 Adaptation du code existant

Retouche à `ICRogue` : Lors de l'étape précédente, un jeu était caractérisé par une simple salle. Il convient naturellement de remplacer maintenant cette salle par un *niveau* (`Level`).La méthode `initLevel()` devra donc créer un niveau (de type `Level0`), ajouter toutes les salles du niveau comme aire du jeu (attention à ne pas introduire de « getters » intrusifs), désigner la salle de départ du niveau comme aire courante du jeu et faire entrer le `ICRoguePlayer` cette aire.
Par ailleurs, un jeu de type `ICRogue` est conçu de sorte à ce que si l'on se rend dans une salle déjà visitée, on doit la retrouver dans l'état où on l'a laissée (voir les paramètres de `setCurrentArea`).
Vous veillerez à bien adapter le code à ce niveau en remplaçant l'ancien contenu de façon adéquate.

Retouche à `ICRoguePlayer` : `ICRoguePlayer` doit enfin devenir capable de passer au travers des connecteurs pour transiter d'une salle à l'autre et ce passage est conditionné potentiellement par la possession de clés.
Il faut donc compléter le schéma d'interaction entre un `ICRoguePlayer` et `Connector` de sorte à ce que :

- si le joueur est en interaction à distance, il essaie de déverrouiller le connecteur (le connecteur passe de verrouillé à ouvert si le joueur est en possession de la clé associée) ;
- s'il est en interaction de contact et qu'il n'est pas en train de se déplacer (`!isDisplacementOccurs()`), il puisse transiter vers la destination du connecteurs.

Indication : Pour le passage vers la salle de destination, introduisez un booléen indiquant si le personnage est en train de passer un connecteur, qui deviendra vrai lorsque c'est le cas (au moment de l'interaction) et dont la valeur peut être interrogée par le jeu. C'est en effet au jeu `ICrogue` lui même de gérer les transitions d'une salle à l'autre (comme c'est dans le cas dans `Tuto2` avec l'invocation de `switchArea`). Il vous est donc suggéré de coder une méthode `switchRoom` (remplaçant `switchArea` mais réalisant un travail analogue) et qui sera invoquée lorsque le personnage est dans l'état « en train de passer une porte ».

3.6.1 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Commencez par tester avec la carte la plus simple puis avec celle à 5 salles. Vous vérifierez alors :

1. que les contraintes de déplacement établies à l'étape précédente restent valides (le personnage ne peut pas marcher sur les « murs » ou sortir de la carte) ;
2. qu'il peut se promener sur chacun des salles de la carte au travers des connecteurs ouverts ;
3. qu'il peut collecter le bâton (s'il y a lieu) et la clé ;
4. qu'une fois la clé ramassée, il peut ouvrir le portail verrouillé associé à cette clé au moyen de la touche `W` ;
5. et qu'une fois le bâton ramassé, le personnage peut toujours lancer des boules de feu avec la touche `X`.

Note : l'état des connecteurs est spécifique à chaque salle. Ainsi, il est normal que dans la seconde carte par exemple, si vous transitez de `[0][1]` à `[0][2]` le connecteur ouest de `[0][2]` soit fermé même si le connecteur est de `[0][1]` a été ouvert.

3.7 Validation de l'étape 2

Pour valider cette étape, toutes les vérifications des sections 3.2.2 et 3.6.1 doivent avoir été effectuées.

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

Cette partie est délibérément moins guidée. Il est important de bien réfléchir à où placer les attributs et méthodes nécessaires, sans avoir recours à des accès trop intrusifs.

4 Défis et ennemis (étape 3)

Cette étape a pour but de compléter la logique générale du jeu en intégrant les défis dont la résolution permettra l'ouverture des connecteurs ainsi que quelques ennemis pour pimenter un peu le tout.

4.1 Salles dépendant de signaux

Le tutoriel a introduit la notion de *signal* qui peut être exploitée pour mettre en place la logique du jeu. L'objectif est de faire en sorte que les connecteurs fermés d'une salle s'ouvrent lorsque les défis qui y sont associés sont relevés. Par exemple, pour une `Level0StaffRoom`, le défi à relever sera simplement de ramasser le bâton qui y est présent. On dira qu'une salle est « résolue » lorsqu'elle a été visitée par le joueur et qu'il aura réussi à relever les défis qui lui sont spécifiques.
L'idée est donc de faire en sorte que chaque salle *se comporte comme un* signal logique (`Logic`) qui est activé lorsqu'elle est résolue et désactivé dans le cas contraire.
Apportez les modifications nécessaires à votre code de sorte à ce que :

- une `ICRogueRoom` se comporte comme un signal logique et qu'elle doit impérativement avoir été visitée par le joueur pour pouvoir être résolue ;
- une `Level0ItemRoom` soit résolue lorsqu'en plus d'avoir été visitée, tous ses items ont été collectés.

Indication : introduisez un attribut booléen indiquant si une salle a été visitée et faites en sorte que cette valeur devienne vraie lorsque le joueur entre dans la salle (souvenez vous que le personnage dispose d'une méthode pour entrer dans une aire).

4.2 Salles avec ennemis

Il vous est maintenant demandé de créer des salles peuplées d'ennemis. Le défi associé sera évidemment de les battre.

4.2.1 Ennemi

Un ennemi, à coder dans le sous-paquetage `icrogue.actor.enemies`, est un `ICRogueActor` qui peut être mort ou vivant et dont on peut interroger l'état (mort ou vivant). Il dispose d'une méthode le faisant mourir et se construit sur la base des informations suivantes : son aire d'appartenance, son orientation, et sa position initiale dans l'aire (`DiscreteCoordinates`).
Vous ne considérerez à ce stade qu'un seul type concret d'ennemis (qui jouera donc le rôle de « boss ») : `Turret` qui aura la particularité de lancer des flèches à intervalles de temps

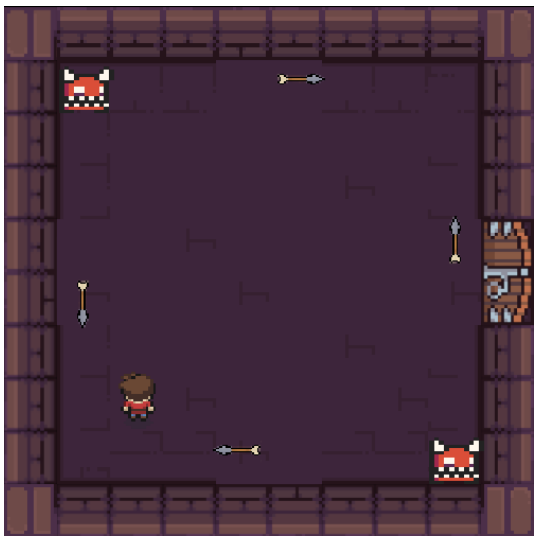


FIG. 12 : Ennemi « tourettes » lançant des flèches : dans cet exemple, la tourette en haut à gauche lance des flèches vers la droite et vers le bas et celle en bas à droite lance des flèches vers le haut et vers la gauche.

réguliers. Libre à vous dans les extensions d'enrichir à volonté la panoplie des adversaires et leur degré de bellicosité :-)
Le `Turret` peut se dessiner au moyen du Sprite :

```
new Sprite("icrogue/static_npc", 1.5f, 1.5f,
this, null, new Vector(-0.25f, 0));
```

Il peut envoyer des flèches dans plusieurs directions. Il est donc caractérisé par l'ensemble des directions (`Orientation`) vers lesquelles il lance ses flèches. Cet ensemble est précisé à la construction (on peut par exemple créer un `Turret` qui envoie des flèches vers le haut et vers le bas ou uniquement vers la gauche).
Les flèches (`Arrow`) sont des projectiles (un peu comme la boule de feu et qui se construisent de façon analogue) caractérisés par des points de dommage spécifiques (1 par exemple).
Le sprite associé se calcule par la tournure :

```
new Sprite("zelda/arrow", 1f, 1f, this,
    new RegionOfInterest(32*orientation.ordinal(), 0, 32, 32),
       new Vector(0, 0)));
```

où orientation est l'orientation de la flèche.
Une flèche interagit par contact avec le joueur en lui infligeant ses points de dommages, et disparaît une fois qu'elle l'a touché.
Pour que le `Turret` puisse lancer ses flèches à intervalles de temps réguliers, vous le doterez d'un temps d'attente spécifique (par exemple `COOLDOWN` valant `2.f` par exemple) et d'un compteur comptant le temps écoulé entre deux lancements de flèche. Le compteur sera initialisé à zéro et incrémenté de `dt` à chaque pas de la simulation. Lorsqu'il atteint la valeur de `COOLDOWN`, l'ennemi passe à l'attaque (et le compteur se réinitialise bien sûr proprement).
Une attaque consiste à lancer une flèche dans chacune des directions possibles (voir la figure 12). De son côté, le joueur peut se défendre en lançant des boules de feu ou en marchant sur l'ennemi. Si un `Turret` a une interaction de contact avec une une boule de

feu, il meurt (et la boule de feu disparaît). Il meurt aussi si le personnage lui marche dessus.

4.2.2 Salles avec des ennemis

Une `Level0EnemyRoom` est une `Level0Room` caractérisée par une *liste d'ennemis actifs*.
Elle dispose d'une méthode permettant de donner une valeur à cette liste (seules les sous-classes de `Level0EnemyRoom` et les classes du même paquetage ne devraient être habilitées à pouvoir recourir à cette méthode).
Une `Level0EnemyRoom` est résolue lorsque la liste des ses ennemis actifs est vide. Elle a la charge :

- d'enregistrer l'ensemble de ses ennemis actifs lors du démarrage de l'aire (méthode `createArea`) ;
- de désenregistrer ceux morts entre deux pas de simulation.

Une `Level0TurretRoom` est une `Level0EnemyRoom` contenant deux `Turret`, tous deux orientés vers le haut, l'un positionné en `(1, 8)` et lançant des flèches en bas et à droite et l'autre positionné en `(8,1)` et lançant des flèches en haut et à gauche.

4.3 Logique du jeu

Pour compléter l'ensemble, on souhaite conditionner la logique générale du jeu à la réussite d'un niveau. On souhaite modéliser le fait que :

- tout niveau de jeu a une salle du « boss » associée[8] ; les coordonnées de cette salle dans la carte sont une caractéristique du niveau et elles valent par défaut `[0][0]` ;
- un niveau se comporte comme un signal logique : il est résolu lorsque sa salle du « boss » l'est.

Indication : pour rester compatible avec les phases antérieure du développement du jeu, vous couvrirez le cas où il n'existe pas de salle du « boss » : le niveau est en fait résolu si la salle du « boss » ne vaut pas `null` et qu'elle est résolue. Les contrôles `T`, `O` et `L` n'ont plus de raison d'être, vous pouvez les commenter (à garder éventuellement comme touche « triche » à des fins de tests).
Lorsqu'un niveau de jeu est résolu, le jeu peut passer au niveau suivant et s'il n'y en a plus se termine. Le jeu se termine donc soit dans ce cas soit lorsque le joueur meurt. Pour simplifier, un simple message `"GameOver"` ou `"Win"` dans la console suffit. Libre à vous dans les extensions de le coder différemment et d'introduire d'autres niveaux.

4.3.1 Carte avec salle du « boss »

Pour pouvoir tester l'ensemble, faites en sorte que `Level0` crée la carte de la figure 13 au moyen d'une méthode utilitaire `generateFinalMap()`. Il s'agit de la même carte que celle générée par `generateMap2()` mais où la salle `[0][0]` est désormais une `Level0TurretRoom`.

4.3.2 Tâche

Il vous est demandé de :

[8] le « boss » peut en réalité être n'importe quel défi plus complexe que les autres, il s'agit de l'« ultime défi du niveau »

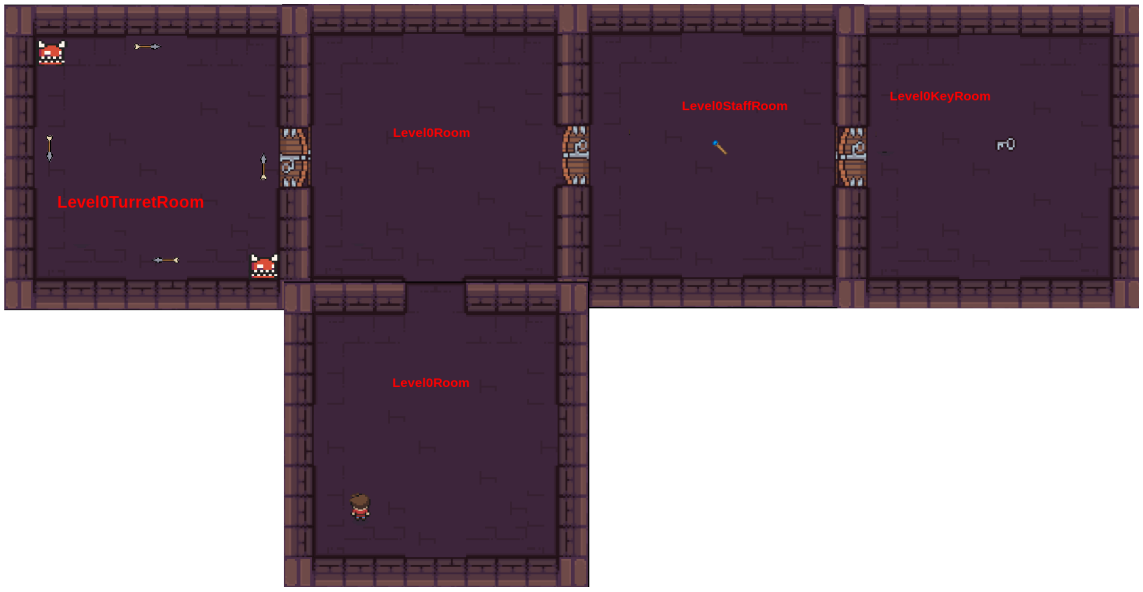


FIG. 13 : La salle `[0][0]` est toujours verrouillée par la clé de la salle `[3][0]`.

- coder les éléments suggérés ci-dessus conformément aux spécifications et contraintes décrites ;
- les tester au moyen de la carte générée par `generateFinalMap`.

Vous vérifierez que :

1. le niveau `Level0` est conforme à la carte de la figure 13 (si l'on a choisi cette dernière comme carte au lancement du jeu `ICRogue`) ;
2. le joueur peut se déplacer dans le niveau `Level0` et transiter de salle par les connecteurs ouverts ;
3. que les connecteurs non liés à des clés s'ouvrent conformément au défis de chaque type de salles ;
4. que la clé de la salle du boss permet de l'ouvrir ;
5. que le boss (`Turret`) lance des flèches à intervalles de temps réguliers et que ces dernières peuvent tuer le joueur ;
6. que le joueur peut accéder à la salle du « Boss » s'il a collecté la clé associée au préalable ;
7. que le joueur peut tuer l'ennemi en lui marchant dessus ou en lui lançant des boules de feu (vous pouvez « tricher » sur la valeur du `COOLDOWN` pour arriver plus vite à vos fins ;-)). Il aura au préalable ramassé le bâton ;
8. que le niveau est résolu lorsque le joueur trucide les `Turret` ou qu'il meurt (ce qui se solde par un le message *`"Win"`* ou *`"Game Over"`* selon le cas).

4.4 Validation de l'étape 3

Pour valider cette étape, toutes les vérifications de la section 4.3.2 doivent avoir été effectuées.

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

Pour coder cette partie vous êtes relativement libres dans les détails de mise en oeuvre. Votre implémentation doit adhérer à la spécification et respecter les principe d'une bonne conception orientée-objets. Ne codez pas cette partie en lisant linéairement l'énoncé. Il est important d'avoir une bonne compréhension de ce qu'il y a à implémenter et de où il est judicieux de placer les méthodes suggérées.

5 Génération aléatoire de niveau (étape 4)

Il s'agit pour cette dernière étape d'introduire un des éléments qui font la spécificité des jeux de type « RogueLike » : la *génération aléatoire de niveaux*. Pour l'essentiel, il s'agit de doter les niveaux de jeu d'une méthode `generateRandomMap` qui sera justement en charge de créer les salles de façon aléatoire.
L'algorithme suggéré comporte deux phases :

1. la première décide du *placement des salles* ; elle choisit quelles coordonnées de la carte correspondront à des salles ;
2. la seconde *génère les salles* aux endroits décidés par la première phase.

Ces deux phases se déroulent en principe selon la même logique quel que soit le niveau. Pour les mettre en oeuvre, une idée simple consiste à considérer que chaque niveau :

- est caractérisé par un ensemble de *types de salles* lui est spécifique `{Type1, Type2... TypeN}` dans un ordre prédéterminé (par exemple `Level0` aurait pour types de salles caractéristiques : `TurretRoom, StaffRoom, Boss_key, Spawn, Normal` dans cet ordre) ;
- peut être généré sur la base d'une distribution souhaitée de ses salles caractéristiques. Cette distribution peut être simplement un ensemble d'entiers décrivant le nombre de chaque salles type de salles : par exemple la distribution souhaitée 1, 4, 1, ... pour `Level0` signifierait que l'on souhaite 1 salle de type `TurretRoom`, 4 salles de type `StaffRoom`, etc.

Modifiez le constructeur de `Level` de sorte à ce qu'il ait l'entête suivant :

```
Level(boolean randomMap, DiscreteCoordinates startPosition,
   int[] roomsDistribution, int width, int height}
```

Son rôle sera de générer une carte fixe de taille `width*height` si `randomMap` vaut `false` (en utilisant `generateFixedMap`). Si le paramètre vaut `true`, ce constructeur devra générer une carte aléatoire au moyen de `generateRandomMap` en fonction de `roomsDistribution` : la carte aléatoire sera de taille `nbRooms * nbRooms` où `nbRooms` est la somme du nombre de salles spécifiées par `roomsDistribution` (par exemple 7 pour une distribution `{1,4,2}`).

5.1 Placement des salles

Il vous est demandé de coder une méthode protégée :

```
MapState[][] generateRandomRoomPlacement()
```

qui permet de décider du placement des salles sur la carte pour un niveau quelconque.

Le type `MapState` permet à l'algorithme de placement de fonctionner en marquant le statut d'une position donnée en cours d'exécution de l'algorithme. Il peut se définir comme suit :

```
protected enum MapState {
    NULL,               // Empty space
    PLACED,             // The room has been placed but not yet
       explored by the room placement algorithm
    EXPLORED,           // The room has been placed and
       explored by the algorithm
    BOSS_ROOM,          // The room is a boss room
    CREATED;            // The room has been instantiated in
       the room map

    @Override
    public String toString() {
        return Integer.toString(ordinal());
    }
}
```

le tableau retourné par `generateRandomRoomPlacement`, que l'on appellera *carte des placements*, est évidemment de même dimensions que la carte à créer.
L'algorithme suggéré pour le placement des salles est ensuite le suivant :

1. initialiser toutes les entrées de la carte des placements à (`MapState.NULL`) ;
2. commencer par marquer la salle au centre de cette carte comme placée (`PLACED`) ;
3. puis, tant qu'il reste un nombre `roomsToPlace` (>0) de salles à placer, pour chaque salle placée mais non explorée :
   (a) calculer le nombre `freeSlots` d'emplacements possibles autour de cette salle (les emplacements en haut en bas à gauche et à droite qui sont dans les limites de la carte et valant `MapState.NULL`) ;
   (b) tirer au hasard un nombre maximal de salles à placer (valant au plus le minimum entre `roomsToPlace` et `freeSlots`) ;
   (c) placer ces salles au hasard dans les emplacements possibles disponibles (il y a différentes façons de faire cela et vous êtes libres de le mettre en oeuvre comme vous le souhaitez) ;
   (d) marquer la salle comme explorée.
4. placer enfin la salle du « boss » : choisir une place libre voisine d'une salle placée et la marquer avec la valeur `MapState.BOSS_ROOM`.

La classe `RandomHelper` fournie dans le sous-paquetage `game.icrogue` met à votre disposition un outillage minimal pour la gestion de l'aléatoire. Pour tous les tirages aléatoires liés à la génération de la carte, vous pouvez utiliser :

- la tournure :

  ```
  RandomHelper.roomGenerator.nextInt(borneMin, borneMax)
  ```

  pour tirer aléatoirement un entier entre `borneMin` (comprise) et `borneMax`(non comprise) ;

- la méthode :

  ```
  List<Integer> RandomHelper.chooseKInList(int k,
     List<Integer> list)
  ```

  qui retourne `k` entiers choisis au hasard dans la liste `list`. Ceci permet par exemple, si l'on a un ensemble de `N` positions potentielles de la carte d'en choisir `k` parmi `N` (la méthode donnera les indices à exploiter dans l'ensemble en question). Exemple d'utilisation :

  ```
  RandomHelper.chooseKInList(3, List.of(1,4,4,5,6));
  ```

  renvoie 3 éléments au hasard de la liste, par exemple : (1,4,4) ou (4,5,1) ou (1,5,6) etc.

Vous noterez que la génération n'est en réalité que pseudo-aléatoire : à partir d'une même « graine » (`ROOM_SEED` du `RandomHelper.java`) ce sera toujours la même séquence de nombres qui sera tirée (cela peut faciliter le déboguage).
Enfin, pour faciliter la vérification des placements aléatoires, vous pouvez incorporer à votre code la méthode utilitaire :

```
private void printMap(MapState[][] map) {
    System.out.println("Generated map:");

    System.out.print("  / ");
    for (int j = 0; j < map[0].length; j++) {
        System.out.print(j + " ");
    }
    System.out.println();
    System.out.print("--/-");
    for (int j = 0; j < map[0].length; j++) {
        System.out.print("--");
    }
    System.out.println();

    for (int i = 0; i < map.length; i++) {
        System.out.print(i + " / ");
        for (int j = 0; j < map[i].length; j++) {
            System.out.print(map[j][i] + " ");
        }
        System.out.println();
    }
    System.out.println();
```

```
}
```

qui permet d'afficher une carte sur le terminal.

5.2 Génération des salles

La création des salles peut se faire sur la base d'un schéma de placement `roomsPlacement` de type `MapState[][]` et d'une distribution souhaitée `roomDistribution` de type `int[]` (à vous de voir s'il vous semble pertinent d'en faire des attributs ou des paramètres de méthodes). L'algorithme simple suggéré est :

- Pour toute entrée d'indice `i` dans `roomDistribution` :
  - il y a `k = roomDistribution[i]` salles de type `i` à placer : choisir pour cela au hasard `k` emplacements exploitables (tous ceux valant `PLACED` ou `EXPLORED` dans `roomsPlacement`) ;
  - créer à ces emplacements une salle de type `i` (attention à bien créer des nouvelles salles à chaque fois !) ;
  - marquer les positions correspondantes comme `CREATED` dans `roomsPlacement`.
- mettre en place les connecteurs entre salles créées ;
- créer la salle du « boss » à l'endroit qui lui est réservé et mettre en place les connecteurs qui la relie à ses voisines.

Note : il est considéré que cet algorithme est le même pour tout type de niveaux. Il est entendu cependant que la création effective d'une salle de type `i` ne peut se faire que dans les niveaux concrets. Il en va de même pour la salle du « boss » et pour la mise en place des connecteurs. Vous veillerez à ce que votre implémentation respecte cela.
Pour la mise en place des connecteurs, il est raisonnable d'introduire une méthode protégée :

```
setUpConnector(MapState[][] roomsPlacement, ICRogueRoom room)
```

invoquée dans l'algorithme général et chargée de la création des connecteurs.
Sa définition concrète ne pourra évidemment se faire que dans les sous-classes concrètes de niveaux. Par exemple, dans `Level0` cette méthode peut itérer sur chacun des connecteurs possibles pour ce type de niveau. Pour chacun des connecteurs permettant d'entrer dans une salle (coordonnées de destinations correspondant à une salle différente de `MapState.NULL`), elle doit mettre à jour de façon appropriée ses coordonnées de destination et son état. Tous les connecteurs seront créés fermés. Les coordonnées de destination peuvent se calculer selon la tournure suivante ;

```
DiscreteCoordinates destination =
   roomCoordinates.jump(connector.getOrientation().toVector());
```

où `roomCoordinates` désigne les coordonnées de la salle à laquelle ces connecteurs appartiennent.
Cette méthode doit aussi être en charge de faire en sorte que tous les connecteurs de la salle du « boss » soient verrouillés au moyen de la clé spécifique au niveau.

5.3 Retouche au code existant

Pour compléter l'ensemble, il est nécessaire de finaliser la classe `Level0` de manière adéquate. Il faut d'abord y décrire la distribution de salles souhaitées pour ce type de niveaux. Vous pouvez pour cela utiliser le type énuméré ébauché ci-dessous (à compléter par vos soins) :

```
public enum RoomType {
        TURRET_ROOM(3), // type and number of roon
        STAFF_ROOM(1),
        BOSS_KEY(1),
        SPAWN(1),
        NORMAL(1);
//..
}
```

Vous pouvez formuler ce type énuméré comme il vous semble le plus pertinent de la faire. Dans celui qui est suggéré, l'idée serait d'avoir trois `TurretRoom`, une `Level0StaffRoom`, une `Level0KeyRoom` qui contiendrait la clé de la salle du « boss ». Toutes les autres seraient des `Level0Room`. Celle correspondant à `SPAWN` serait de surcroît la salle de démarrage du niveau.

Indication : dotez le type énuméré d'une méthode qui retourne une liste de distribution telle que nécessaire au constructeur de la super-classe.

La classe `Level0` sera désormais dotée de 2 surcharges du constructeur :

- `Level0(boolean randomMap)` qui invoque de manière appropriée le constructeur de sa super-classe en lui passant sa liste de distribution spécifique. Les dimensions pour la génération de carte fixes valent toujours par défaut `[4][2]` ;
- `Level0()` qui fait par défaut de la génération aléatoire.

Vous pouvez à loisir changer l'appel à ce constructeur dans `ICRogue` pour (en lui passant rien ou `false` de sorte à ce que soit généré un niveau fixe ou un niveau aléatoire).

Enfin, la salle du « Boss » et l'aire de démarrage ne sont donc plus fixes. Faites en sorte que les valeurs « en dur » initialisées jusqu'ici lors de la construction des niveaux ne soient valable que pour les cartes fixes (non aléatoires).

5.3.1 Tâche

Il vous est demandé de coder les éléments suggérés ci-dessus conformément aux spécifications et contraintes décrites et de les tester (en visualisant le résultat au moyen de la fonction d'impression suggérées, que vous pouvez invoquer dans `generateRandomMap`).

Vous vérifierez :

1. qu'il est désormais possible de générer à la demande (en fonction du paramètre passé au constructeur de `Level0` dans `ICRogue`) des niveaux de façon aléatoire ;
2. que les niveaux générés obéissent bien aux contraintes de la distribution souhaitée (nombre de salles de chaque type adéquat, pas de salles déconnectées des autres, présence d'une salle du « Boss ») ;
3. que les connecteurs sont correctement positionnés et fonctionnels ;
4. que la logique générale du jeu reste respectée pour les deux types de niveaux (aléatoires et fixes).

5.4 Validation de l'étape 4

Pour valider cette étape, toutes les vérifications de la section 5.3.1 doivent avoir été effectuées.

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

6 Extensions (étape 5)

Pour obtenir des points bonus (pouvant compenser d'éventuels malus sur la partie obligatoire) ou pour participer au concours, vous pouvez coder quelques extensions librement choisies. Au plus 20 points seront comptabilisés : coder beaucoup d'extensions pour compenser les faiblesses des parties antérieures n'est donc pas une option possible.
La mise en oeuvre est libre et très peu guidée. Seules quelques suggestions sont données ci-dessous. N'hésitez pas à nous solliciter pour une évaluation du barème de vos extensions si vous décidez de partir dans une direction précise. Un petit bonus pourra aussi être attribué si vous faites preuve d'inventivité dans la conception du jeu.
Vous pouvez coder vos extensions dans le jeu `ICRogue` au moyen de niveaux supplémentaires ou dans un nouveau jeu utilisant la logistique que vous avez mise en place dans les étapes précédentes. Le jeu de base (avec le niveau `Level0`) doit pouvoir rester jouable tel que demandé dans la partie obligatoire.
Vous prendrez soin de **commenter soigneusement** les modalités de jeu de votre extension dans votre `README`. Nous devons notamment savoir quels contrôles utiliser et avec quels effets sans aller lire votre code. Voici un exemple de `README` (partiel) correspondant qui explique comment jouer à un jeu :

- exemple de fichier `README.md` tiré d'un ancien projet : 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 points nécessaires ;-). Une estimation du nombre de points associés aux extensions est donnée. Cela peut valoir plus si un effort plus important que prévu est dédié.

6.1 Pistes d'extensions

En réalité, la base que vous avez codée peut être enrichie à l'envie. En particulier, la composante « signal » peut être tirée à profit pour créer des défis de plus en plus complexes. Vous trouverez « en vrac » quelques suggestions d'extensions dans les parties qui suivent.

6.1.1 Nouveaux types de salles

(nombre de points dépendant de l'effort réalisé)
L'extension la plus naturelle est sans doute d'ajouter de nouveaux types de salles caractérisées par de nouveaux contenus et défis. Par exemple on peut imaginer d'ajouter une salle caractérisée par un magasin où le personnage pourrait acheter des objets utiles. Pour les nouveaux contenus et défis vous pouvez vous inspirer de la section suivante. Il est aussi possible d'ajouter de nouveaux types de niveaux (avec potentiellement d'autres connecteurs).
Il est important dans vos ajouts de respecter l'esprit des jeux de type « RogueLike » : les salles doivent toujours pouvoir être visitées dans un ordre quelconque avant d'arriver à la salle finale d'un niveau. En clair, il ne doit pas y avoir de conditions explicites dans les niveaux imposant de devoir visiter une salle avant une autre.

6.1.2 Nouveaux acteurs ou extensions du joueur

Toutes sortes d'acteurs peuvent être envisagés et leur rendu visuel peut être amélioré.

- animations : au lieu de représenter les acteur au moyen d'un `Sprite` unique, il est possible de plutôt leur associer une animation qui serait une séquence de `Sprite` affichés tour à tour pour donner une illusion de mouvement ; la maquette offre une classe `Animation` que vous pouvez exploiter pour cela ; (~2 à 3pts)
- nouveaux objets à collecter, voire modélisation d'un système de ressources : or, argent, bois, nourriture, doses de soins, etc ; (~2 à 5pts)
- divers acteurs pouvant servir de signaux : orbes, torches, plaques de pression, leviers ; (~2 à 5 pts)
- signaux avancés pour les défis (oscillateurs , signaux avec retardateur) : un oscillateur est un signal dont l'intensité varie au cours du temps ; (~4pts/signal)
- toutes sortes d'ennemis avec des modalités de déplacement et de comportement spécifiques ; (~2 à 5 pts/personnage, jusqu'à 10 points si une IA est codée)
- introduction de personnages amicaux qui peuvent aider le personnage (notamment au moyen de dialogues, voir plus bas) ; (~2 à 5 pts/personnage)
- créer un personnage suiveur à l'image du Pikachu de Red dans Pokémon jaune ; (~3pts)
- créer un ou plusieurs événements de scénario se déclenchant avec des signaux. Par exemple, une type de salle peut être lié à un signal qui une fois « allumé » entraînerait l'apparition d'un personnage. Ce dernier donnant à son tour un objet ou une consigne pour résoudre le défi de la salle. On peut aussi imaginer des salles avec un nombre variable d'ennemis et qu'un événement (comme être resté trop longtemps dans la salle) double le nombre d'ennemis etc. (nombre de points dépendant de la complexité de l'ajout)
- complexifier les comportements (par exemple doter les personnages de périodes d'immunité) ; (~2pts)
- ajouter de nouveaux type de cellules avec des comportements appropriés (eau, glace, feu, etc.) ; (~2pts/cellules)
- ajouter une ombre ou un reflet au joueur et à certains acteurs ; (~3pts)
- ajouter de nouveaux contrôles avancés (interactions, actions, déplacements, etc.) ; (~2pts/-control)
- ajouter des événements aléatoires (décors, sigaux, etc.) ; (~4pts)
- etc.

6.1.3 Augmenter le part de l'aléatoire

(nombre de points dépendant de l'effort réalisé)
Les jeux de type « RogueLike » sont souvent caractérisés par une part importante d'aspects aléatoires : salle avec un nombre aléatoire d'ennemis ou d'objets, aspects des acteurs changeant aléatoirement, etc. Toute idée dans ce sens est bienvenue.

6.1.4 Dialogues, pause et fin de jeu

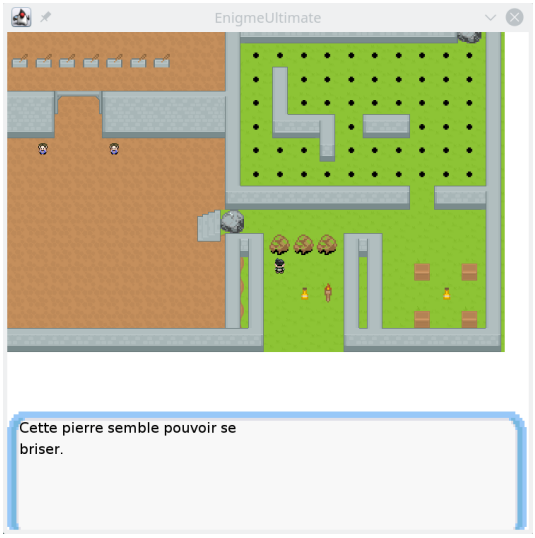
Si les défis sont complexes le jeu peut aussi rapidement devenir injouable sans quelques indications dispensées à bon escient. Introduire la possibilité de recourir à quelques dialogues peut donc être un plus fort agréable. Par ailleurs, mettre en place un système de pause et de reprise du jeu ou gérer les fins avec un écran de "game over" (typiquement quand le personnage n'a plus de point de vie) compléterait naturellement votre jeu.
Quelques indications vous sont fournies ci-dessous pour aller dans ce sens.

Dialogues

(nombre de points dépendant de l'effort réalisé pour mettre en place les dialogues).
Il est possible d'attacher des textes aux acteurs (un peu comme nous l'avons fait avec `SimpleGhost`). Vous pouvez initier des dialogues dans certaines situations. Par exemple, lorsque le personnage demande une interaction avec un acteur, une indication peut alors s'afficher pour mettre le joueur sur la piste de ce qu'il faut faire pour que l'interaction devienne possible.
Des textes prédéfinis peuvent être stockés dans des fichiers en format `.xml` comme celui fourni dans `res/strings/enigme_fr.xml`. A titre d'exemple :

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

retournera la chaîne de caractères `"utilisez une CastleKey!"`. Le fichier `xml` associé au jeu est défini dans `Play`.
Alternativement, et de façon un peu plus proche de ce qui se fait dans les jeux de type *GameBoy*, vous pouvez exploiter la classe fournie `Dialog` (de `rpg.actor`) et obtenir des visualisations de dialogues ressemblant à ceci :



Pause du jeu et fin de partie (~2 à 4pts)

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 a atteint un objectif ou a été battu par exemple).

Vous pouvez, de manière générale, 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 !

6.2 Validation de l'étape 5

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

7 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 ».[9]
Si vous souhaitez concourir, vous devrez nous envoyer d'ici au **22.12 à 18 :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).

> Les projets gagnants feront l'objet d'une présentation lors de la semaine de la rentrée (février).

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