CS-107 : Mini-projet 2
Jeux de labyrinthe

M. Alaoui, J. Sam

Version 1.5

Table des matières

1 Présentation

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

Vous vous êtes familiarisé.es 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 de type RPG. Le but de ce mini-projet est d'en tirer parti pour créer une ou plusieurs petites déclinaisons concrètes d'une variante « jeu de labyrinthe », nommée `ICMaze`. Le personnage principal aura à se déplacer dans un décor labyrinthique généré de façon procédurale, se complexifiant de niveau en niveau et peuplé d'entités hostiles ou utiles. La figure 1 montre quelques fragments de l'ébauche de base que vous pourrez enrichir ensuite à votre guise, au gré de votre fantaisie et imagination. La section 6 contient aussi une petite vidéo d'exemple de jeu auquel vous pourriez aboutir.

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.

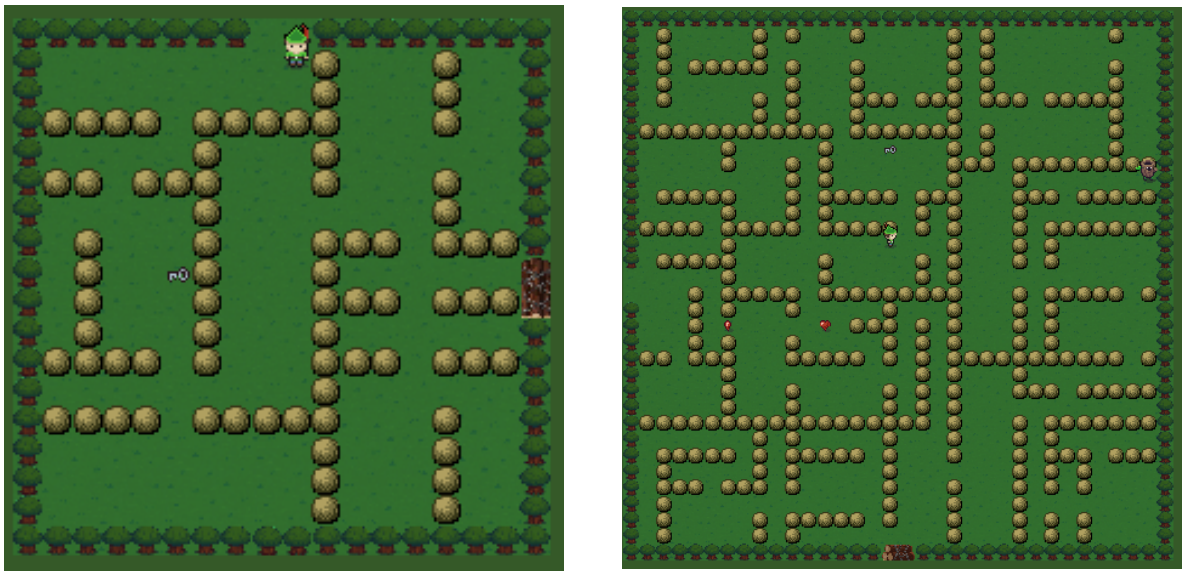

Fig. 1 : Exemple de scènes du jeu où le personnage résout des labyrinthes de plus en plus complexes et où il peut faire différents types de rencontres plus ou moins amicales.

Le projet comporte quatre étapes **obligatoires** et une facultative :

- Étape 1 (« Jeu de base ») : au terme de cette étape vous aurez créé, en utilisant les outils du moteur de jeu fourni, une instance basique de `ICMaze` où un personnage principal pourra collecter des objets et traverser des portails.

- Étape 2 (« Génération de labyrinthes ») : lors de cette étape le jeu sera enrichi par un mécanisme permettant de générer des labyrinthes qui conditionneront le déplacement du personnage.
- Étape 3 (« Niveaux et ennemis ») : cette étape permettra de générer procéduralement des niveaux de jeux peuplés d'ennemis que le personnage devra affronter pour pouvoir circuler dans les labyrinthes.
- Étape 4 (« Finalisation ») : cette étape finalisera le jeu dans sa partie obligatoire en introduisant un nouvel ennemi, des mécaniques de jeu basées sur la notion de signal ainsi que des dialogues rudimentaires.
- Étape 5 (Extensions, facultatives) : 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.

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

1. **Vous ne modifierez pas le code de `game-engine`**. Méfiez-vous à ce propos de proposition de résolution de problèmes de IntelliJ.
2. Le projet sera codé avec les outils Java standards (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 IntelliJ vous propose d'importer sur votre machine. Le projet utilise notamment la classe `Color`. Il faut utiliser la version `java.awt.Color` et non pas d'autres implémentations provenant de divers packages alternatifs.
3. Vos méthodes seront documentées selon les standards javadoc (inspirez-vous du code de la classe `TextGraphics` de `game-engine`).
4. 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.
5. Les indications peuvent être parfois très détaillées. **Cela ne veut pas dire pour autant qu'elles soient exhaustives**. Les méthodes et attributs nécessaires à la réalisation des traitements voulus ne sont évidemment pas tous décrits et ce sera à vous de les introduire selon ce qui vous semble pertinent et en respectant une bonne encapsulation.
6. Votre projet **ne doit pas être stocké sur un dépôt public** (de type github). Pour ceux d'entre vous familiers 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 ICMaze de base (étape 1)

Le but de cette étape est de commencer à créer votre propre jeu de type `ICMaze`.Cette version de base contiendra un personnage principal capable de se promener entre deux aires et d'y collecter des objets. Ces fonctionnalités se coderont selon le mécanisme 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 ») ;
- des *portails* qui lui permettront de transiter d'une aire à l'autre.

Vous travaillerez dans le paquetage fourni `icmaze`.

2.1 Préparation du jeu `ICMaze`

> La solution du tutoriel est fournie dans le dossier `tutos` et vous pourrez directement vous en inspirer pour commencer.

Préparez un jeu `ICMaze` en vous inspirant de `Tuto2`. Ce dernier sera constitué pour commencer :

- de la classe `ICMazePlayer` qui modélise un personnage principal, à placer dans `icmaze.actor` ; laissez cette classe vide pour le moment, nous y reviendrons un peu plus bas ;
- de la classe `ICMaze`, équivalente à `Tuto2` à placer dans le paquetage `icmaze` ; N'oubliez pas d'adapter la méthode `getTitle()` qui retournera un nom de votre choix (par exemple *`"ICMaze"`*!) ;
- une classe `ICMazeArea` équivalente à `Tuto2Area`, à placer dans un sous-paquetage `icmaze.area` ;
- des classes `Spawn` et `BossArea` héritant de `ICMazeArea`, à placer dans le paquetage `icmaze.area.maps` (elles sont équivalentes aux classes `Ferme` ou `Village` dans le tutoriel. En guise de titre prenez par exemple *`"icmaze/Spawn"`*et *`"icmaze/Boss"`*.
- de la classe `ICMazeBehavior` analogue à `Tuto2Behavior` à placer dans `icmaze.area` et qui contiendra une classe publique `ICMazeCell` équivalente de `Tuto2Cell`.

Note : le nom du `ICMazeBehavior` doit être *`"SmallArea"`*, aussi bien pour `Spawn` que pour `BossArea` afin que la correspondance avec les ressources graphiques se fasse bien.

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 ;-)).

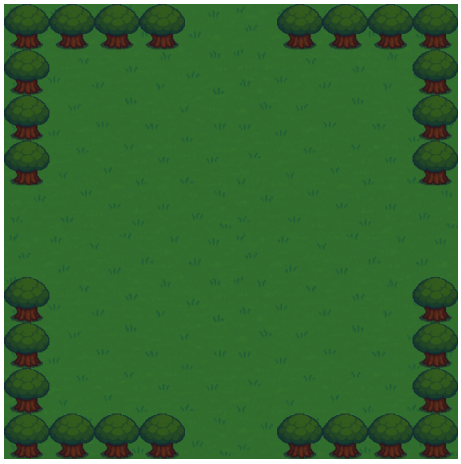

FIG. 2 : Aire d'accueil

2.2 Adaptation de `ICMazeArea`

Les aires concrètes utilisent toutes les deux le même nom de grille (*`"SmallArea"`*). Contrairement au tutoriel, il n'y a donc plus de lien univoque entre le titre de l'aire et le nom de la grille associée (« behavior »). La classe `ICMazeArea` doit donc aussi être caractérisée par le nom de sa grille, et un constructeur doit permettre d'initialiser ce nom. C'est ensuite ce même nom qui sera utilisé de manière appropriée dans la méthode `begin` de `ICMazeArea` pour associer une grille à l'aire (au lieu de recourir à `getTitle`).

Note : cela a un impact sur la création des acteurs de type `Background` qui doivent être construits avec le constructeur prenant en paramètre un titre (celui de la grille en l'occurence).

Comme facteur d'échelle de la caméra, vous pouvez prendre la valeur `11`.

2.3 Adaptation de `ICMazeBehavior`

Dans un premier temps, il s'agit d'apporter quelques nouveautés aux classes `ICMazeBehavior` et `ICMazeCell`.

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),
WALL(-14112955, false),
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 `ICMazeCell` le garantisse.

Procédez aux adaptations suggérées ci-dessus dans la classe `ICMazeBehavior`.

2.3.1 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Lancez votre jeu `ICMaze`. Vous vérifierez que l'aire vide de la figure 2 s'affiche.

2.4 Acteurs de `ICMaze`

Maintenant que les bases fondamentales sont posées, il vous est demandé de commencer à modéliser les acteurs des jeux de type `ICMaze`. Ils seront codés dans le sous-paquetage `icmaze.actor`.

Vous considérerez que tous les acteurs impliqués dans un jeu `ICMaze`, les `ICMazeActor`, 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 `ICMazeActor` 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, une seule catégorie plus spécifiques de `ICMazeActor` est à introduire : le *personnage principal* qu'il faudra diriger au moyen de touches.

2.4.1 Le personnage principal : `ICMazePlayer`

Un `ICMazePlayer` est un acteur non traversable. Il se comporte (se met à jour) comme un `ICMazeActor` mais doit en plus pouvoir être déplacé au moyen des flèches directionnelles à la manière du `GhostPlayer` codé dans le tutoriel.

Un `ICMazePlayer` peut se trouver dans différents état, représentables au moyen d'un type énuméré. Pour le moment, on peut considérer qu'il peut être soit dans l'état « inoccupé/au repos » (`IDLE`) ou en demande d'interaction à distance (`INTERACTING`). Vous ferez en sorte qu'il ne soit sensible aux touches gérant son déplacement que s'il est dans l'état `IDLE`.

Note : Nous reviendrons à tout cela un peu plus loin, mais il peut être une bonne idée d'utiliser un `switch` sur les états possible du personnage dans la méthode `update` de `ICMazePlayer`.

2.4.2 Jeux de touches

Dans l'ébauche de jeu élaborée dans le tutoriel, les touches utilisées par le jeu étaient codées « en dur ». Il est préférable de faire en sorte que cela soit configurable et une classe `KeyBindings` vous est fournie à cet effet dans le paquetage `icmaze`. Référez-vous à l'annexe 7.1 pour comprendre ce type de données. Adaptez votre méthode de déplacement de sorte à ce que les touches « flèches directionnelles » telles qu'utilisées par le tutoriel soient remplacées par `PLAYER_KEY_BINDINGS.up()`, `PLAYER_KEY_BINDINGS.down()`, `PLAYER_KEY_BINDINGS.right()` et `PLAYER_KEY_BINDINGS.left()`.

2.4.3 Représentation graphique d'un `ICMazePlayer`

L'image qui sert à dessiner un `ICMazePlayer` ne sera pas figée comme celle de `GhostPlayer` mais va dépendre de son *orientation* et sera *animée*.

Vous utiliserez pour dessiner le personnage, un objet de type `OrientedAnimation`. Il y aura plusieurs animations possibles au fur et à mesure que le projet évoluera, mais pour l'instant seule l'animation de base est à prévoir. Elle pourra se construire comme indiqué dans l'annexe 7.3.1[1].

Lors de la mise à jour du personnage, dans l'état `IDLE`, son animation courante doit également être adaptée selon l'algorithme suivant : si un déplacement est en cours, l'animation courante doit subir un `update`, sinon, elle doit subir un `reset`.

Pour pouvoir tester ces développements, faites en sorte qu'au lancement du jeu un joueur de type `ICMazePlayer` soit créé à la position qui lui est destinée dans l'aire de démarrage ; prenez par exemple `(5,7)` dans `Spawn`. Il sera orienté vers le bas et dans l'état `IDLE`.

2.4.4 Tâche

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

Lancez votre jeu `ICMaze`. Vous vérifierez qu'il se comporte comme sur la petite vidéo suivante : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icmaze/startStep1.mp4 ; c'est-à-dire concrètement :

1. que le jeu démarre en affichant un `ICMazePlayer` qui regarde vers l'avant ;
2. que le personnage puisse être déplacé librement sur toute l'aire au moyen des touches prévues pour le déplacement (par défaut les flèche directionnelles) et qu'il ne puisse pas 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 ses mouvements soient animés.

[1] Référez-vous à la documentation de `OrientedAnimation` si vous souhaitez comprendre le rôle des paramètres dans la création de l'animation.

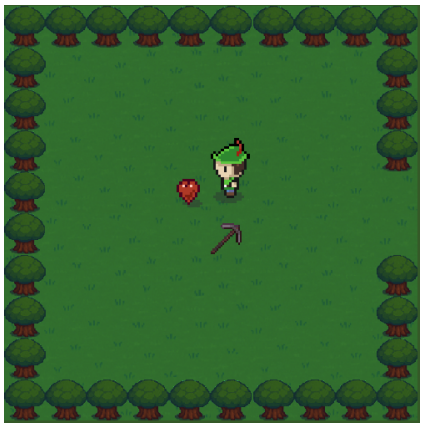

FIG. 3 : Position initiale des acteurs `Pickaxe` et `Heart` dans l'aire de démarrage.

2.5 Objets à collecter : interactions de contact

Il s'agit maintenant de modéliser des objets que le personnage principal pourra collecter et qui lui seront utiles le long des ses pérégrinations. Les objets à collecter seront implémentés dans un sous-paquetage `icmaze.actor.collectable`. En utilisant la classe `CollectableAreaEntity` fournie dans la maquette (paquetage `areagame.actor`), il vous est demandé de coder une hiérarchies d'objets à collecter spécifiques au jeux `ICMaze`. Par défaut, ces objets seront traversables, orientés vers le bas et n'accepteront que les interactions de contact[2]. Ils doivent disparaître de la simulation une fois collectés. Une sous-catégorie de ces objets représentera les objets à collecter qui pourront servir d'équipement. On considérera que tous les objets de cette dernière catégorie ont une orientation spécifique, donnée à la construction, et qu'ils peuvent être dessinés au moyen d'un `Sprite` qui les caractérise.

Deux types concrets d'objets à collecter sont à coder à ce stade : les pioches (`Pickaxe`) et les coeurs (`Heart`). Les pioches entrent dans la catégorie « équipement » mais pas les coeurs. Pour ces deux types d'objets, l'aire d'appartenance et la position de départ sont données comme paramètres à la construction. Pour ce qui est des la représentation graphique, le `Sprite` associé à la pioche se construit au moyen de la tournure :

```
new Sprite("icmaze/pickaxe", .75f, .75f, this));
```

Les coeurs sont quant à eux animés (voir 7.3.2).

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

- une pioche en position `(5,4)` orienté vers le bas ;
- un coeur en position `(4,5)`.

2.5.1 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Lancez votre jeu `ICMaze`. Vous vérifierez que la pioche et le apparaissent (figure 3).

[2] souvenez vous des méthodes `isCellInteractable` et `isViewInteractable`

2.5.2 Interactions de contact

Il vous est demandé maintenant d'appliquer le schéma suggéré par le tutoriel[3] pour mettre en place les premières interactions entre les `ICMazeActor` codés jusqu'ici. Dans un premier temps il s'agit de permettre aux `ICMazePlayer` de collecter des objets.

Pour cela, commencez par mettre en place le fait que tous les `ICMazePlayer` deviennent des `Interactor` : ils peuvent faire subir des interactions à des `Interactable`, par exemple pour les ramasser.

Comme `Interactor`, `ICMazePlayer` 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. Les interactions à distance seront, par contre, conditionnées par son état :

- `wantsCellInteraction` : retourne `true` inconditionnellement ;
- `wantsViewInteraction` : retourne `true` quand le personnage est dans l'état `INTERACTING`, et `false` autrement. Les modalités de basculement vers l'état `INTERACTING` seront abordées un peu plus loin.

Intéressons-nous maintenant à la gestion concrètes des interactions.

Dans le sous paquetage `icmaze.handler`, complétez l'interface `ICMazeInteractionVisitor` héritant de `AreaInteractionVisitor`. Cette interface doit fournir une définition par défaut des méthodes d'interaction de tout `Interactor` du jeu avec les *Interactables* connus à ce stade ; à savoir :

- une cellule du jeu (`ICMazeCell`) ;
- un personnage principal du jeu (`ICMazePlayer`) ;
- une pioche ;
- et un coeur.

Ces définitions (par défaut) auront un corps vide pour exprimer le fait que par défaut, l'interaction consiste à ne rien faire. `ICMazePlayer` en tant que `Interactor` du jeu `ICMaze`, 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 `ICMazeInteractionVisitor`. Leur méthode `acceptInteraction` (vides jusqu'ici) doit être reformulée dans ce sens et dans chacune des classes concernées :

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

Pour que `ICMazePlayer` puisse gérer plus spécifiquement les interactions qui l'intéressent,

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

définissez dans la classe `ICMazePlayer`, une classe imbriquée privée `ICMazePlayerInteractionHandler` implémentant `ICMazeInteractionVisitor`. Ajoutez-y les définitions nécessaires pour gérer plus spécifiquement l'interaction avec une pioche et avec un coeur.

Ces deux objets devront se voir collectés par interaction de contact. Le cas de l'interaction à distance sera testé un peu plus tard.

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 :

- `ICMazePlayer` ait pour attribut son gestionnaire d'interaction spécifique (de type `ICMazePlayerInteractionHandler`) ;
- 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).

2.5.3 Tâche

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

Lancez votre jeu `ICMaze`. Vous vérifierez :

1. que le `ICMazePlayer` se comporte comme pour les étapes précédentes mais qu'il peut ramasser la pioche et le coeur en leur marchant dessus ;
2. et que les objets ramassés disparaissent de l'aire.

2.6 Portails et clés : interactions à distance

Les « portails » sont des acteurs permettent de connecter des aires entre elles et dont l'ouverture peut être conditionnée par la possession d'une clé. Lorsque le personnage se place en face d'un portail, il pourra par exemple indiquer qu'il souhaite interagir avec (touche 'E' par défaut) et s'il possède la clé nécessaire, le portail pourra s'ouvrir.

Les « portails » peuvent être 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 en fonction de conditions plus ou moins complexes.

Un portail verrouillé correpond visuellement à un amas de rondins de bois solidement amarrés par une chaîne. Cette dernière peut être détachée au moyen d'une clé, ce qui ouvrira le passage. Il faut donc commencer par compléter un peu la hiérarchie des équipements.

2.6.1 Les clés

Les clés (classe `Key`) sont semblables en tout point à des `PickAxe`. La seule différence est qu'une `Key` est caractérisée par un identificateur entier (initialisé par une valeur passée en paramètre du constructeur). Le `ICMazePlayer` 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 portails. À des fins de tests, créez dans `Spawn` deux clés : l'une d'identifiant `Integer.MAX_VALUE` en position `(6,5)` et l'autre d'identifiant `Integer.MAX_VALUE-1` en position `(1,2)`.

2.6.2 Les portails

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

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

- un état (un type énuméré avec les valeurs `OPEN`, `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 `Integer.MIN_VALUE` par exemple).

L'aire d'appartenance d'un portail, 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 portail dépend de son état : on dessinera un `Sprite` spécifique s'il est invisible ou verrouillé 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("icmaze/invisibleDoor_"+orientation.ordinal(),
 (orientation.ordinal()+1)%2+1, orientation.ordinal()%2+1, this);

// pour verrouillé
 new Sprite("icmaze/chained_wood_"+orientation.ordinal(),
                (orientation.ordinal()+1)%2+1, orientation.ordinal()%2+1,
                    this);
```

En tant que `Interactable`, un `Portal` 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 portail occupe sa cellule principale et celle immédiatement à sa droite.

FIG. 4 : Les `ICMazeArea` ont toutes 4 portails placés aux même endroits : Ouest (W), Sud (S), Est (E), Nord (N) : dans cet exemple, les portails « Ouest » et « Sud » sont verrouillés, le portail nord est invisible et le portail « Est » est ouvert.

2.6.3 Aires avec portails

La classe `ICMazeArea`, ébauchée à l'étape précédente, représente le concept abstrait d'« aire dans un jeu ICMaze ». 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 portails* permettant des transitions vers d'autres aires.

Pour simplifier vous considérerez que :

- chaque `ICMazeArea` comporte systématiquement quatre portails, placés dans un ordre et à des endroits précis (voir figure 4),
- que chaque aire est un carré de taille `size x size` ; ce qui permettra de placer automatiquement les 4 portails aux positions : `(size / 2 , size + 1)` pour le portail « nord », `(size / 2, 0)` pour le portail « sud », `(0, size / 2)` pour le portail « ouest » et `(size + 1, size / 2)` pour le portail « est » ;
- et que les coordoonnées d'arrivée dans une aire soient : `(size / 2 + 1, size)` si l'on arrive par le portail « nord », `(size / 2 + 1 , 1)` par le portail « sud », `(1, size / 2 + 1)` par le portail « ouest » et `(size, size / 2 + 1)` par le portail « est ».

Avec cette modélisation, les coordonnées de destination du portail « Est », par exemple, seront les coordonnées d'arrivée depuis l'« Ouest ».

Les portails sont par défaut invisibles.

Pour commencer, vous introduirez le code d'un type énuméré `AreaPortals` dans `ICMazeArea` et dont la définition est :

```
public enum AreaPortals {
     N(Orientation.UP),
     W(Orientation.LEFT),
     S(Orientation.DOWN),
     E(Orientation.RIGHT);

     private final Orientation orientation;

     AreaPortals(Orientation orientation) {
         this.orientation = orientation;
     }

     public Orientation getOrientation() {
         return orientation;
     }
}
```

Le constructeur de `ICMazeArea` sera revisité de sorte à avoir la taille du côté de l'aire comme paramètre supplémentaire. Ce constructeur devra évidemment aussi initialiser son ensemble de portails.

Par ailleurs, l'aire `BossArea` sera désormais caractérisée par le portail qui permet d'y entrer (un `AreaPortals`, initialisé au moyen d'un paramètre à la construction).

Indication : le type énuméré de `AreaPortal` donne l'orientation vers laquelle le portail mène. Lors de la création du portail par le constructeur de `ICMazeArea`, il faudra associer au portail l'opposé de cette orientation (méthode `opposite()`). Ainsi, si le portail `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 cause de la vue du dessus du jeu.

Création des `ICMazeArea` La méthode `createArea` de `ICMazeArea` a pour rôle d'enregistrer les acteurs spécifiques à l'aire. Elle devra donc faire en sorte que les portails soient enregistrés comme acteurs afin que leur comportement puisse être simulé.

Les appels aux constructeurs des aires `Spawn` et `BossArea` seront revus de sorte à ce que la taille des aires vaille 8 et que :

- l'aire `Spawn` ait son portail « Est » visible et verrouillé au moyen de la clé d'identifiant `Integer.MAX_VALUE` ;
- l'aire `BossArea` ait son portail d'entrée ouvert.

Il faudra par ailleurs indiquer lors de la création des aires que :

- le portail d'entrée dans `BossArea` est le portail « Ouest » ;
- le portail « Est » de `Spawn` a pour aire de destination `BossArea` et que les coordonnées d'arrivée dans cette aire soient les coordonnées « Ouest » ;
- le portail « Ouest » de `BossArea` a pour aire de destination `Spawn` et que les coordonnées d'arrivée dans cette aire soient les coordonnées « Est ».

Vous introduirez pour cela toute méthode vous semblant utile en veillant à préserver au mieux l'encapsulation en évitant les « getters » intrusifs (comme par exemple un « getter » sur les portails d'une aire).

Interactions avec les portails `ICMazePlayer` doit enfin devenir capable de passer au travers des portails pour transiter d'une aire à l'autre et ce passage est conditionné potentiellement par la possession de clés.

Il faut donc compléter le comportement du `ICMazePlayer` pour intégrer le fait qu'il peut passer dans un état « en demande d'interaction » (`INTERACTING`) :

- si dans l'état `IDLE`, lorsque le personnage n'est pas en train de se déplacer, la touche de demande d'interaction est pressée :

  ```
  keyboard.get(PLAYER_KEY_BINDINGS.interact()).isPressed()
  ```

  le personnage doit basculer dans l'état `INTERACTING`.
- dans l'état `INTERACTING` si la touche de demande d'interaction n'est plus appuyée

  ```
  !keyboard.get(PLAYER_KEY_BINDINGS.interact()).isDown()
  ```

  le personnage rebascule dans l'état `IDLE`.

Le schéma d'interaction entre `ICMazePlayer` avec `Portal` doit aussi être adapté de sorte à ce que :

- si le personnage est dans l'état `INTERACTING`, il essaie de déverrouiller le portail (le portail passe de verrouillé à ouvert si le personnage est en possession de la clé associée) ;
- s'il est en interaction de contact, il transite vers la destination du portail.

Indication : seul le jeu connait toutes les aires et est capable de faire transiter le personnage d'une aire à l'autre au moyen des méthodes `leaveArea` et `setCurrentArea`, par conséquent, ce traitement ne peut pas être directement codé dans la méthode d'interaction du personnage avec la porte. Cette méthode doit se contenter d'informer le personnage qu'il est en train de traverser une porte (et laquelle), et le personnage doit pouvoir fournir les informations utile au jeu !

2.6.4 Tâche

Lancez votre jeu `ICMaze`. Vous vérifierez qu'il se comporte comme sur la petite vidéo suivante : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icmaze/step1Final.mp4 ; c'est-à-dire concrètement :

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. que les portails censés êtres invisibles le soient bien ;
3. que la portail « est » de la classe `Spawn` soit verrouillé ;
4. que le personnage ne peut pas transiter vers l'aire `BossArea` via ce portail tant qu'il est verrouillé ;
5. qu'il peut collecter les clés ;
6. que la clé d'identifiant `Integer.MAX_VALUE -1`, même ramassée ne lui permet pas d'ouvrir le portail verrouillé au moyen de la clé d'identifiant `Integer.MAX_VALUE` ;

7. qu'une fois la clé d'identifiant `Integer.MAX_VALUE` ramassée, il peut ouvrir le portail verrouillé associé à cette clé au moyen de la touche `E` (par défaut) et transiter vers l'aire suivante en marchant sur le portail ouvert ;
8. que les portails une fois ouverts le restent et que des aller-retours entre les aires sont possibles via les portails ouverts.

2.7 Reset

Pour faciliter les tests, vous doterez `ICMaze` du contrôle suivant : la touche `KeyBindings.RESET_GAME` ('R' par défaut) doit permettre de faire une réinitialisation ("reset") du jeu. Il s'agit de redémarrer le jeu dans les mêmes conditions que la toute première fois qu'on le lance. Notez que les méthodes `begin` d'un jeu ou d'une aire (tels que fournis par la maquette) vident les collections impliquées (aires, acteurs etc.) ; il n'est donc pas nécessaire de s'en préoccuper.

2.7.1 Tâche

Vérifiez que la touche de « reset » fonctionne comme attendu et que les acteurs (personnage, clé, coeur et pioches) réapparaissent bien aux positions originales. Testez cette touche dans les deux aires existantes.

2.8 Validation de l'étape 1

Pour valider cette étape, toutes les vérifications des sections 2.3.1, 2.4.4, 2.5.1, 2.5.3, 2.6.4 et 2.7.1 doivent avoir été effectuées.

Le jeu `ICMaze` dont le comportement vérifie les étapes de validation ci-dessus est à rendre à la fin du projet.

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 oeuvre 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.

3 Labyrinthes (étape 2)

Cette seconde partie du projet a pour but de créer de façon procédurale des labyrinthes dans des aires. Les murs des labyrinthes seront matérialisés par des acteurs *« rocher »* que le personnage ne pourra pas traverser.

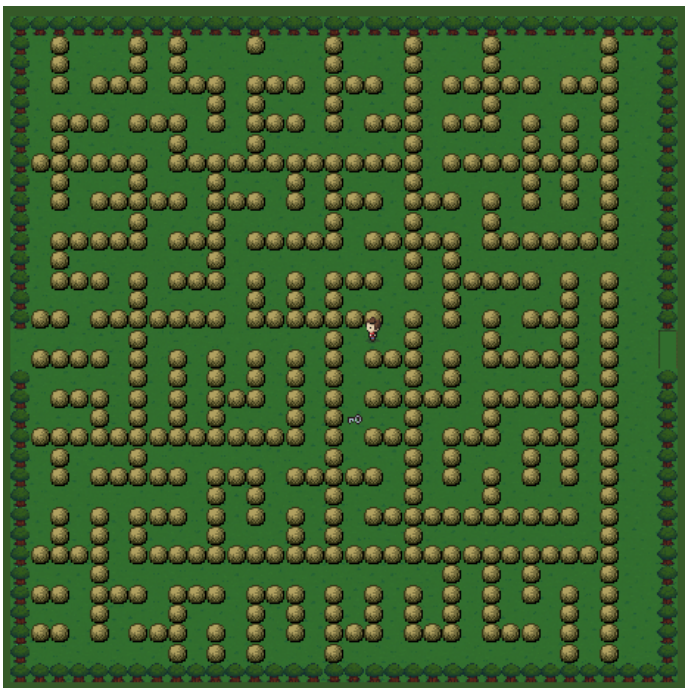

Fig. 5 : Exemple de labyrinthe généré par la méthode de division récursive.

La méthode de génération de labyrinthes à implémenter est appelée "Division récursive". L'algorithme consiste à diviser récursivement l'espace en sous-régions jusqu'à ce que l'on atteigne une granularité souhaitée. Il sera utilisé pour générer un tableau d'entiers, de la taille de l'aire, où la valeur zéro représentera une case vide et la valeur 1 un obstacle. Une fois ce tableau généré, il suffira de placer un acteur *« rocher »* sur toutes les cases correspondant à la valeur 1.

La figure 5 donne un exemple typique d'aire où des rochers non traversables auront été placés conformément à un schéma de labyrinthe généré algorithmiquement.

Note : un tableau de booléens pourrait suffire a priori mais l'utilisation d'entiers facilite des extensions futures (d'autres valeurs d'entiers pourraient en effet correspondre à d'autres sortes d'obstacles, par exemple le chiffre 2 pourrait représenter de la lave).

Un type particulier d'aire entre en scène : les aires comportant des labyrinthes. La taille de l'aire y aura une influence particulière puisqu'elle conditionnera la création de labyrinthes plus ou moins complexes (plus une aire est grande, plus il est possible d'y créer un labyrinthe étendu et complexe). Il vous sera demandé concrètement de placer entre l'aire `Spawn` et l'aire `BossArea` déjà crées, une séquence de trois aires labyrinthiques de tailles croissantes : une `SmallArea`, une `MediumArea` et `LargeArea`.

3.1 Aires labyrinthiques

Une *aire labyrinthique* est une `ICMazeArea` caractérisée par un *portail d'entrée* (de type `AreaPortals`, permettant d'entrer dans le labyrinthe associé), un *portail de sortie*, l'identifiant de *la clé* qui permet de déverrouiller le portail de sortie et un *niveau de difficulté*, de type entier.

Ces données seront initialisées à la construction au moyen de valeurs passées en paramètres.

Trois catégories concrètes d'aires labyrinthiques sont à introduire :

- `SmallArea` de taille 8 et avec *`"SmallArea"`* comme nom de grille associée ;
- `MediumArea` de taille 16 et avec *`"MediumArea"`* comme nom de grille ;
- et `LargeArea` de taille 32, avec *`"LargeArea"`* comme nom de grille.

Le titre associé à l'aire sera codé de sorte à indiquer le nom de la grille et l'identifiant de la clé permettant d'en sortir ; par exemple, selon le format

`"icmaze/<nom_de_la_grille>[identifiant_clé]"`

Pour le moment vous intégrerez les nouvelles aires au jeu en faisant en sorte que le portail de sortie soit celui à l'est et celui d'entrée à l'ouest. `Spawn` doit permettre de rentrer dans `SmallArea` au moyen d'un portail verrrouillé, comme il le faisait pour rejoindre `BossArea` auparavant. `SmallArea` doit permettre ensuite de rentrer dans `MediumArea` qui, à son tour, doit permettre de rentrer dans `LargeArea`. Cette dernière permettra enfin de rentrer dans `BossArea`. Les portails de sortie des aires labyrinthiques seront créés ouverts pour le moment.

Enfin, les niveaux de difficulté des aires labyrinthiques seront assignés à la construction et pour cette étape, vous choisirez le niveau de difficulté `Difficulty.HARDEST` pour toutes les aires.

La classe fournie `Difficulty` (paquetage `icmaze`) permet en effet de modéliser, par l'intermédiaire de constantes, des valeurs possibles pour le niveau de difficulté du labyrinthe associé à une aire.

Important : afin de faciliter les modifications ultérieures du jeu, il est recommandé de faire en sorte que la création des aires dans `ICMaze.createAreas()` se fasse par le biais d'une méthode utilitaire, que vous pourrez par exemple appeler `generateHardCodedLevel()`, par opposition à la génération procédurale des aires qui se fera plus tard dans le projet.

3.1.1 Rochers

Un *rocher* (classe `Rock`) est un acteur dérivant de `AreaEntity` et dessinable par défaut au moyen du sprite *`"rock.2"`* (avec hauteur et largeur de 1). Par défaut, il est non traversable et accepte inconditionnellement tout type d'interaction. Sa méthode `getCurrentCells` est identique à celle du personnage principal. Pour le moment, le personnage principal a pour seule interaction avec les rochers celle par défaut (qui consiste à ne rien faire). Vous créerez un rocher dans `Spawn` à une position de votre choix.

3.1.2 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications et contraintes données, lancez votre jeu `ICMaze`. Vous vérifierez que :

- que le rocher est non traversable ;
- et que le personnage peut transiter de l'aire `Spawn` à l'aire `BossArea` en empruntant les portails « Est » via les aires "labyrinthique (qui ne les sont pas encore tout à fait) intermédiaires.

3.2 Génération de labyrinthes

Une classe `MazeGenerator` est fournie dans le paquetage `icmaze`. Elle sera en charge de générer les labyrinthes à proprement parler et il s'agit de la compléter. Pour cela, vous la doterez d'une méthode statique principale :

```
int[][] createMaze(int width, int height, int difficulty)
```

qui a pour vocation de créer une grille de dimension `width × height` en faisant appel a une méthode récursive de division. Le paramètre `difficulty` représente la taille minimale des sous-régions à diviser. Plus cette valeur est grande, plus le labyrinthe aura de grands couloirs (et donc plus le labyrinthe sera facile !).

La méthode de division récursive est le cœur de l'algorithme. Elle s'applique à une sous-région du labyrinthe, définie par :

- `(x, y)` : les coordonnées de la cellule en haut à gauche de la sous-région ;
- `width` : la largeur de la sous-région ;
- `height` : sa hauteur.

Le comportement de la méthode est le suivant :

1. **Cas de base** : si la sous-région est trop petite (inférieure ou égale à `difficulty` en largeur ou hauteur), on arrête la division.
2. On choisit aléatoirement d'ajouter un mur horizontal ou vertical, en privilégiant la direction la plus longue ; ceci permet de maintenir un labyrinthe homogène.
3. On choisit aléatoirement un emplacement pour le mur, en veillant à le placer à une position impaire ; pour conserver un motif de grille compatible avec des chemins d'une seule cellule de largeur.
4. On insère une ouverture (passage) dans ce mur à une position paire, choisie aléatoirement.
5. On applique récursivement la méthode de division récursive aux deux nouvelles sous-régions ainsi créées.

Remarques importantes :

- L'utilisation de positions paires/impaires permet de garantir que les murs et les chemins ne se superposent pas et que le labyrinthe soit toujours cohérent.
- Une méthode `printMaze` est fournie afin de déboguer votre implémentation.
- Les méthodes `nextBoolean()` et `nextInt()` du générateur aléatoire fourni (voir 7.2) permettent de faire les choix au hasard requis.

3.2.1 Intégration du labyrinthe aux aires

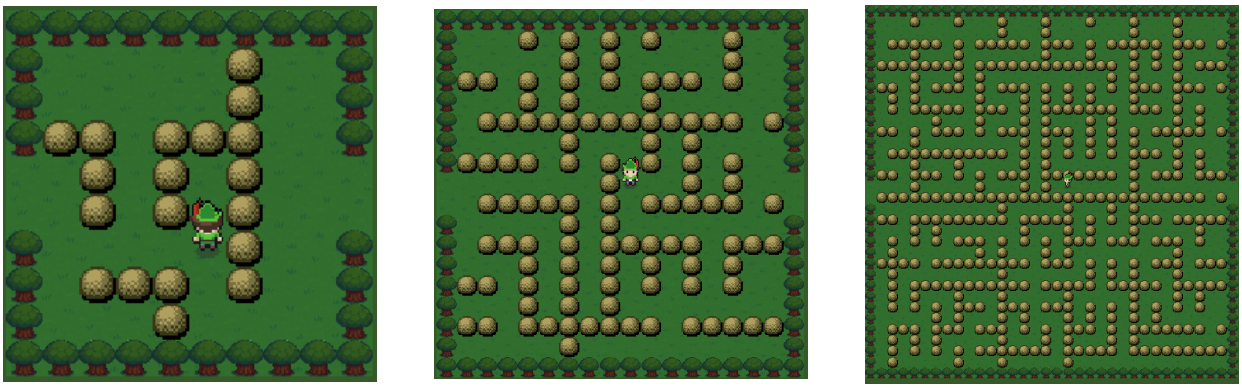

FIG. 6 : Exemple de séquence d'aires labyrinthiques, de gauche à droite : une `SmallArea`, une `MediumArea` et une `LargeArea`

Il vous est maintenant demandé de compléter le code des aires labyrinthiques de sorte à ce que leur méthode `createArea` fasse appel à la méthode `MazeGenerator.createMaze`. Un acteur `Rock` sera créé dans chacune des cellules correspondant à la valeur `1` dans le labyrinthe construit par cette méthode, pour peu que cela ne bloque pas directement l'entrée ou la sortie du labyrinthe.

Astuces :

- Aidez vous de la méthode fournie `printMaze` pour examiner les labyrinthes créés par `createMaze`.
- Utilisez un facteur d'échelle élevé pour avoir une vision d'ensemble pour tester et « debugger ». Par exemple, la vidéo de la section 3.3.1 et faite avec une méthode `getScaleFactor()` retournant 40 au lieu de 11.

.

Facteur d'échelle dynamique Pour améliorer le confort de jeu, il est sans doute avantageux de définir la méthode `getScaleFactor()` de sorte à ce qu'elle permette d'adapter dynamiquement le facteur d'échelle à la taille de l'aire.

Voici une façon heuristique possible de le faire :

```
public float getCameraScaleFactor() {
    return (float) Math.min(size * DYNAMIC_SCALE_MULTIPLIER,
       MAXIMUM_SCALE);
}
```

Avec par exemple `DYNAMIC_SCALE_MULTIPLIER` valant `1.375` et `MAXIMUM_SCALE` valant `30`.

3.2.2 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications et contraintes données, lancez votre jeu `ICMaze`. Vous vérifierez que l'aire `Spawn` débouche sur une séquence de trois aires labyrinthiques de difficulté croissante comme sur la figure 6. Vous vérifierez plus précisément :

- que des labyrinthes résolvables de difficultés croissantes sont générés dans les aires labyrinthiques ;
- que les portails d'entrée et de sortie de chaque aire labyrinthique ne sont pas bloquées par des rochers ;
- que le personnage peut marcher sur les chemins entre les rochers, mais pas sur les rochers eux-mêmes ;
- qu'entre l'aire `Spawn` et l'aire `BossArea` se trouvent désormais trois aires labyrinthiques de difficulté et taille croissantes ;
- et que l'aire la plus complexe permet de déboucher sur l'aire `BossArea`.

3.3 Placement des clés dans les labyrinthes

La section 6.5 du tutoriel vous montre qu'un graphe (`AreaGraph`) peut être associé à une grille. Un tel graphe permet d'obtenir des informations importantes comme quelle cellule de la grille est atteignable à partir d'une autre et par quel plus court chemin possible typiquement.

Il s'agit maintenant de compliquer un peu la vie de notre personnage en verrouillant les portails de sortie des aires labyrinthiques. On placera toutefois aléatoirement dans chacune de ces aires, une clé permettant d'ouvrir son portail de sortie. La clé devra évidemment avoir pour identifiant l'entier donné à la construction de l'aire et devra être placée sur les chemins et non sur les rochers.

C'est ici que la notion de graphe associé à la grille devient utile. Supposons que l'on associe une graphe de type `AreaGraph` à toute aire `ICMazeArea` et que l'on y ajoute par le biais de la méthode `addNode` un noeud pour chaque cellule faisant partie d'un chemin du labyrinthe, alors il est possible de trouver la liste des coordonnées discrètes de ces cellules au moyen de la méthode `keySet()`. Une fois ces coordonnées obtenues dans une liste `lst`, il suffit de les reordonner aléatoirement avec un appel

```
Collections.shuffle(lst, RandomGenerator.rng);
```

et d'utiliser la première coordonnée de cette liste pour obtenir une position aléatoire où placer une clé. Pour rappel, `RandomGenerator` est un classe fournie dans le paquetage `icmaze` (voir 7.2).

Il vous est donc demandé ici de retoucher votre code de sorte à ce :

- que les portails de sortie soient verrouillés au moyen d'une clé (dont l'identifiant est celui donné à la construction) ;
- qu'un `AreaGraph` soit associé à `ICMazeArea` ;
- que la méthode `createArea` des aires labyrinthiques permette d'ajouter un noeud à ce graphe pour toute cellule qui n'est pas occupée par un rocher ;

- et qu'elle crée une clé à une position choisie au hasard parmi les positions appartenenant à un chemin. À noter que les clés ne seront pas les seuls acteurs à pouvoir être générés dans les labyrinthe et il peut être utile de penser d'emblée à modulariser le code en charge de ces créations.

3.3.1 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications et contraintes données, lancez votre jeu `ICMaze`. Vous vérifierez alors que :

- que les portails de sorties des aires labyrinthiques sont verrouillés ;
- qu'une clé apparaît à une position aléatoire et non occupée par une rocher dans chacune de ces aires ;
- et que la clé permet de déverrouiller le portail.

Le parcours du personnage de salle en salle devrait ressembler à l'exemple de cette vidéo : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icmaze/step2.1.mp4

3.4 Interaction avec les murs

Pour apporter quelques facilités aux personnages, il vous est enfin demandé de lui permettre de briser des rochers avec la pioche s'il l'a ramassée. La destruction d'un rocher s'accompagnera de sa disparition dans un petit nuage (voir 7.3.3) et le rocher disparait complètement de la simulation au terme de cette animation.

Le rocher sera assorti de points de vie et chaque coup asséné avec la pioche l'affaiblira d'un point. Lors de sa disparition un coeur sera généré à sa place avec une probabilité de votre choix (par exemple une avec une chance sur deux).

La mise en oeuvre de ces traitements implique de modéliser l'utilisation de la pioche par le personnage et donc un nouvel état possible du personnage : l'état « attaquant avec sa pioche » (`ATTACKING_WITH_PICKAXE`).

Le comportement du personnage doit donc à nouveau être adapté et vous ferez en sorte que :

- si le personnage est dans l'état `IDLE` et qu'il n'est pas en cours de déplacement, si la touche d'utilisation de la pioche est pressée :

  ```
  PLAYER_KEY_BINDINGS.pickaxe()).isPressed()
  ```

  et que la pioche est collectée, le personnage bascule dans l'état `ATTACKING_WITH_PICKAXE` où il doit alors changer d'animation (voir l'annexe 7.3.1)
- s'il est dans l'état `ATTACKING_WITH_PICKAXE`, son animation courante doit être mise à jour et lorsque l'animation est terminée (méthode `isCompleted`), le personnage rebascule dans l'état `IDLE`.

3.4.1 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications données, lancez votre jeu `ICMaze`. Vous vérifierez que le personnage est capable de détruire les

rochers (une fois qu'il a ramassé la pioche dans `Spawn`) comme sur la petite vidéo suivante : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icmaze/step2.2.mp4

3.5 Validation de l'étape 2

Pour valider cette étape, toutes les vérifications des sections 3.1.2, 3.2.2, 3.3.1, et 3.4.1 doivent avoir été effectuées.

Le jeu `ICMaze` dont le comportement vérifie les étapes de validation ci-dessus est à rendre à la fin du projet.

4 Niveaux et ennemis (étape 3)

Dans ce troisième volet du projet, il vous est demandé d'enrichir la conception en permettant l'ajout d'adversaires qui compliqueront la progression du personnage dans les labyrinthes. Il s'agira également de permettre de générer les aires labyrinthiques *de façon procédurale*.

Vous trouverez ci-dessous les spécifications à respecter ainsi que quelques indications.

4.1 Ennemis

Les ennemis (`Ennemy`) sont des acteurs :

- capables de se déplacer sur une grille ;
- dotés de points de vie et d'un nombre de points de vie maximal ;
- qui meurent si leur nombre de points de vie est inférieur ou égal à zéro ;
- qui sont « demandeurs » d'interactions (ils ne font pas que les subir) ;
- avec lesquels il est possible par défaut d'avoir des interactions aussi bien à distance que de contact ;
- sur lesquels, par défaut, on ne peut marcher que s'ils sont morts ;
- et qui comme le personnage principal peuvent perdre un nombre donné de points de vie (par exemple au cours d'une interaction).

Vous considérerez que le nombre de points de vie maximal ne peut être défini à ce niveau d'abstraction, mais qu'il doit être garanti de pouvoir y accéder pour tout type d'ennemi.

Les ennemis disparaissent de l'aire de jeu lorqu'ils meurent (nombre de points de vie inférieur ou égal à zéro). Une petit animation se joue juste avant leur disparition (voir 7.3.5).

PathFinderEnemy Toutes sortes de catégories d'ennemis sont envisageables mais puisqu'il s'agit d'un jeu de labyrinthes, il est raisonnable d'envisager une catégorie d'ennemis capables de se diriger en suivant des chemins labyrinthique (les `PathFinderEnemy`).

Ce type d'ennemis se distingue par le fait qu'ils ont la capacité de décider dans quel sens se déplacer selon une stratégie qui leur est propre. Ceci se fera par le biais d'une méthode (par exemple `Orientation getNextOrientation()`) qui ne peut se définir à ce niveau d'abstraction mais qui aura des déclinaisons concrètes pour des sous-classes particulières. Par exemple, `getNextOrientation` pourra retourner l'orientation permettant de se rapprocher d'une cible selon le plus court chemin qui existe entre l'ennemi et la cible.

Nous vous demandons de coder pour le moment une seule implémentation concrète de `PathFinderEnemy`, les « monstre troncs ».

Vous considérerez que tous les `PathFinderEnemy` ont comme champs de perception toutes les cellules de leur voisinage dans un « rayon » fixé par une constante spécifique à chaque sous-classe concrète. Il ne s'agira pas d'un rayon circulaire mais du carré entourant la position l'ennemi. Ils sont « demandeur » d'interaction à distance mais pas d'interaction de contact.

FIG. 7 : Monstres tronc : l'un en déplacement ciblé et l'autre endormi

4.1.1 `LogMonster`

Le monstre tronc (`LogMonster`, voir figure 7) est un `PathFinderEnemy` qui a un rayon de perception spécifique (5 par exemple).

Il peut être dans différents états qui vont conditionner son comportement : *« en train de dormir »*, *« se dépaçant au hasard »* ou *« se déplaçant à la poursuite d'une cible »* et cet état est donné à la construction. Sa transition d'un état à un autre est impactée par le niveau de difficulté de l'aire dans laquelle il se trouve. Il est de surcroit capable de mémoriser une cible (en l'occurrence un `ICMazePlayer`).

Représentation graphique Le monstre tronc se dessine au moyen d'une animation qui change en fonction de son état (voir l'annexe 7.3.6).

Interactions Le personnage principal peut désormais subir des attaques de la part du monstre tronc. Il devient hélas mortel ce qui se met en oeuvre par le fait qu'il a un nombre de points de vie courant et un nombre de points de vie maximal (une constante commune à toutes les instances de `ICMazePlayer` et valant par exemple 5). Le personnage nait avec ce nombre de point de vie maximal. Lorsque son nombre de points de vie est en dessous de zéro le personnage meurt et la partie doit faire l'objet d'un « reset ».

Le monstre tronc ne peut interagir que s'il n'est pas en train de dormir. Lorsqu'il interagit avec un `ICMazePlayer`, s'il se trouve juste en face de lui, il lui fait perdre un nombre fixe de points (par exemple 1), sinon (plus de distance) il le mémorise. De son côté, le personnage peut faire subir des dommages au monstre tronc en utilisant sa pioche : à nouveau un nombre fixe de points par attaque (par exemple 1 point).

Comportement général Le comportement du monstre tronc implique des temps d'attente qui peuvent être mis en oeuvre au moyen de la classe fournie `CoolDown` (paquetage `icmaze`). Un objet de type `Cooldown` est un compteur qui peut être initialisé avec un temps (à attendre) et sa méthode `ready` retournera `true` lorsque ce temps est écoulé. Le monstre tronc observe des temps d'attente pour se réorienter à des moments voulus ou avant la transition d'un état à un autre. Deux objets de type `Cooldown` peuvent donc être utilisés pour servir chacun de ces motifs.

Le comportement du « monstre tronc » peut être décrit comme suit :

- s'il est en train de dormir :
  1. si son compteur de reorientation est écoulé, il se tourne à gauche (la classe `Orientation` dispose d'une méthode `hisLeft()`)
  2. si son compteur de transition d'état est écoulé, il transite avec une probabilité `pTransition` à l'état « déplacement aléatoire » qui se calcule selon la formule

     ```
     (double) Difficulty.HARDEST / difficulty;
     ```

     où `difficulty` est le degré de difficulté de l'aire à laquelle appartient le monstre ; ainsi plus le niveau de difficulté est élevé plus la probabilité de transition l'est aussi.
- s'il est en mode « déplacement aléatoire » :
  1. si son compteur de re-orientation est écoulé, il change aléatoirement d'orientation et se déplace d'un pas (voir la description du mode de déplacement un peu plus bas) ;
  2. si son compteur de transition est écoulé et s'il a une cible en vue, il transite à l'état « déplacement ciblé » avec la probabilité `pTransition`.
- s'il est en mode « déplacement ciblé » : s'il n'a plus de cible, il rebascule en mode « déplacement aléatoire ». Sinon :
  1. il estime comment s'orienter en fonction de la stratégie de parcours du labyrinthe qui lui est spécifique (méthode `getNextOrientation` décrite plus bas). Si son compteur de réorientation est écoulé et qu'une nouvelle orientation a pu être calculé, il se déplace d'un pas ;
  2. si son compteur de transition est écoulé, il ressombre dans le sommeil avec une probabilité valant `1 - pTransition` ; ainsi plus le niveau est élevé, plus la probabilité de se rendormir est faible.

Vous pouvez par exemple utiliser `0.75f` comme temps d'attente pour la ré-orientation et `3.f` pour les transitions d'état.

Mode de déplacement et méthode `getNextOrientation` Le déplacement d'un cran se fera naturellement avec la méthode `move` héritée ses acteurs mobiles sur grille. Vous pouvez prendre par exemple 10 comme nombre par défaut de « frames » utiles au déplacement.

La méthode `getNextOrientation` a, quant à elle, pour rôle d'évaluer, à chaque pas de simulation, la position cible à atteindre `via un plus court chemin`. Lorsque la cible est repérée, 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 (`AreaGraph`) associé à l'aire dans laquelle évolue le monstre. L'orientation à prendre pour suivre ce chemin est alors donnée par :

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

> Le graphe n'est en principe pas atteignable directement depuis la classe `LogMonster` et vous veillerez à éviter le codage de « getters » intrusifs comme un `getGraph` dans `ICMazeArea`.

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).

Création de `LogMonster` Il vous est maintenant demandé d'enrichir la méthode `createArea` des aires labyrinthiques de sorte à ce qu'en plus du placement de clés, elle crée des « monstres troncs » à des positions aléatoires du labyrinthe. Le nombre maximal d'ennemis par labyrinthe sera plafonné (par exemple à 3) et le nombre effectifs d'ennemis à générer sera calculé en fonction du niveau de difficulté associé à l'aire : plus l'aire est difficile et plus le nombre d'ennemis sera élevé.

Nous vous proposons la formule heuristique suivante pour calculer la probabilité d'ajouter un ennemi :

```
diffRatio = Math.min(1.0, (double) Difficulty.HARDEST /
   difficulty);
```

Ainsi, tant que le nombre d'ennemis maximal n'est pas atteint, on décide d'ajouter un ennemi supplémentaire si `rng.nextDouble()< 0.25 + 0.60 * diffRatio`. Une fois le nombre d'ennemis décidé, ils seront créés à des positions aléatoires et d'une manière analogue à ce qui a été fait pour les clés.

L'état de départ des `LogMonster` se décidera aléatoirement, avec encore une fois un algorithme *ad'hoc* utilisant trois valeurs :

- la probabilité dêtre en mode déplacement ciblé : `pTarget= 0.10 + 0.70 * diffRatio` (dépendante du niveau de difficulté) ;
- la probabilité d'être en mode « déplacement aléatoire » : `pRandom=0.20` ;
- et la probabilité d'être endormi, `pSleeping`, qui vaudra 1 moins les deux précédentes.

En clair, on tirera une valeur aléatoire `rng.nextDouble()`. Si la valeur tirée est inférieure à `pSleeping` le monstre sera créé endormi, si elle est inférieure à `pSleeping + pRandom`, il sera créé en mode « déplacement aléatoire » et sinon en mode « déplacement ciblé ».

4.1.2 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications et contraintes données, lancez votre jeu `ICMaze`. Vous vérifierez alors :

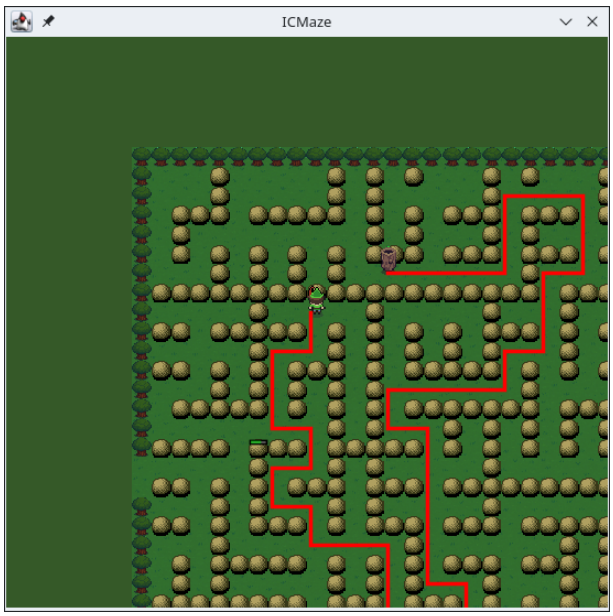

Fig. 8 : Dessin explicite du chemin suivi par un monstre tronc.

- que des monstres troncs sont désormais générés automatiquement dans les labyrinthes ;
- qu'il peuvent naître dans différents états et que plus l'aire est difficile moins ils sont endormis au départ et plus facilement il peuvent se déplacer en mode ciblé ;
- qu'il peuvent se déplacer aléatoirement ;
- que dans l'état endormi ils sont inoffensifs et s'affichent avec une animation spécifique ;
- qu'ils peuvent transiter d'un état à l'autre et que la transition de « endormi » vers « déplacement ciblé » est de plus en plus probable au fur et à mesure que l'aire est difficile et que plus l'aire est difficile, moins ils se rendorment ;
- qu'en mode déplacement ciblés ils sont capables de traquer le personnage à travers le labyrinthe ;
- qu'une fois à proximité ils infligent des dommage au personnage et sont capable de l'« occire » ce qui se traduit par un « reset » de la partie.

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 monstre tronc, ce qui permet de voir explicitement apparaître le chemin suivi, comme montré dans la figure 8.

FIG. 9 : Personnage avec barre de vie

4.2 Barres de vie

Maintenant que le personnage doit faire face à des ennemis pouvant lui infliger des dommages, il peut être utile de pouvoir visualiser leur barre de vie d'une façon explicite (voir la figure 9).

Une classe `Health` est fournie dans le répertoire `actor` pour vous simplifier la mise en oeuvre de cet aspect. À tout `ICMazePlayer` et `LogMonster` sera associée une barre de vie initialisée par une tournure telle que :

```
new Health(this, Transform.I.translated(0, 1.75f), MAX_LIFE,
    friendly);
```

où `friendly` vaut `true` pour le personnage et `false` pour l'ennemi !. Le premier paramètre permet d'attacher la barre à l'acteur courant (comme les points de vie dans le tutoriel), le second indique de combien on décale la barre de l'acteur dans son repère relatif et le 3ème est un entier indiquant le nombre maximal de points de vie. Le dernier paramètre permet de colorier la barre de santé en vert ou rouge selon que l'acteur concerné soit amical ou hostile (par défaut amical). Une barre de vie peut également être associée aux rochers si vous le souhaitez.

Étudiez les fonctionnalités de la classe `Health` pour voir comment faire en sorte que les points de vie qu'elle reflète augmentent ou diminuent. Modélisez ensuite le fait :

- que `ICMazePlayer` puisse gagner des points de vie lors de la collecte de coeurs (par exemple 1 point par coeur) ;
- et qu'une fois qu'il a subi une perte de point de vie (quelle qu'en soit la cause), tout acteur bénéficie d'une petite période d'immunité dont la durée est identique pour tous les instances de sa classe. Durant la période d'immunité l'acteur est insensible aux

dommages.

La barre de vie associée à un acteur doit évidemment être dessinée par la méthode de dessin de ce dernier. Il est possible de ne le faire que lorsque l'acteur n'est pas dans une période d'immunité et qu'il y a encore des points de vie à afficher.

Indication : vous pouvez implémenter les périodes d'immunité au moyen d'un compteur de type `Cooldown`. Comme durée de la période d'immunité, vous pouvez prendre une valeur comme 24 par exemple.

Retouche au dessin des personnages : Afin de permettre de visualiser l'état d'immunité, modifiez le dessin du personnage, des rochers et des ennemis de sorte à ce que lorsqu'ils sont en état d'immunité, le dessin de l'animation ne se fasse qu'une fois sur deux (par exemple uniquement quand le compteur du temps d'immunité est pair).

Retouche au « reset » : les touches de « reset » doivent désormais assurer que les personnages repartent avec une barre de vie complète. Vous considérerez aussi qu'ils repartent sans immunité.

4.2.1 Tâche

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

Lancez votre jeu `ICMaze`. Vous vérifierez alors :

1. qu'une barre de vie s'affiche à proximité du personnage (ça peut être tout le temps ou en période d'immunité selon votre choix), qu'elle soit complètement verte au départ et qu'elle suive le personnage associé dans ses déplacements ;
2. que le monstre tronc fasse perdre de points de vie au personnage qui est à proximité et que cela se traduise au niveau de sa barre de vie ;
3. que la collecte de coeur permette de restaurer des points de vie et que cela se reflète aussi dans la barre de vie ;
4. que le dessin du personnage et du monstre tronc clignote lorsqu'il vient de subir des dommages (indiquant qu'il a bénéficié d'un peu d'immunité).

Vous vérifierez aussi que les « reset » permettent de restaurer les points de vie et que la « mort » d'un des personnages se traduit par un « reset » de l'aire courante..

4.3 Génération procédurale des niveaux

Jusqu'ici les aires labyrinthiques étaient créées « en dur » dans le jeu. Il s'agit maintenant de remplacer ces aires par une succession d'aires générées de façon procédurale. L'algorithme qui vous est suggéré permet de générer un niveau *linéaire*, c'est-à-dire une suite d'aires connectées sans embranchements.

L'idée est la suivante :

1. Le niveau commence par l'aire de départ (`Spawn`).

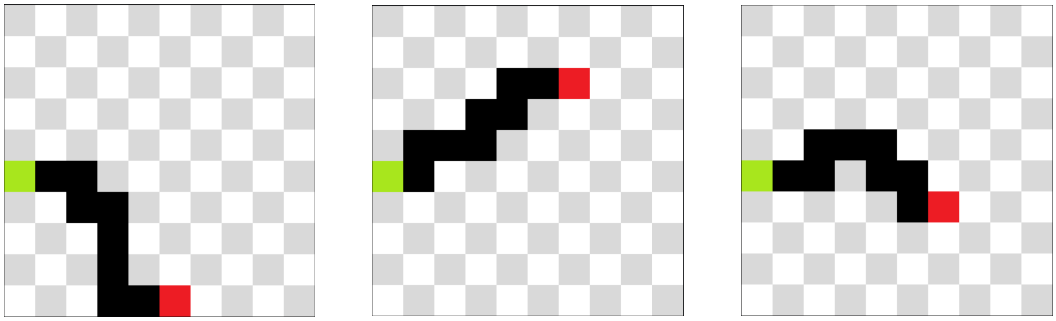

FIG. 10 : Trois exemples possibles d'un niveau de longueur 8

2. Il est ensuite composé d'une succession, de longueur donnée, d'aires générées dynamiquement.
3. Chaque nouvelle aire est placée dans une direction allant **« vers l'avant »** (nord, est ou sud) (voir la figure 10).
4. La difficulté et la taille des aires augmentent progressivement à mesure que le joueur avance.
5. Le niveau se termine par l'aire du « Boss » (`BossArea`).

Ce générateur garantit que les aires ne se superposent pas, et gère les connexions bidirectionnelles entre les aires. C'est une classe `LevelGenerator` qui sera en charge de réaliser les traitements requis. Cette classe comportera la méthode principale :

```
public static ICMazeArea[] generateLine(ICMaze game, int length)
```

qui a pour rôle de génèrer un niveau linéaire de longueur `length` et retourne un tableau d'aires (`ICMazeArea[]`) dans l'ordre où elles doivent être chargées dans le jeu. Le paramètre `game` est une instance du jeu, transmise à chaque aire pour sa création et `length` est le nombre strictement positif d'aires, **hors aire de départ et aire du « Boss »**.

Comportement détaillé de `generateLine` Les aires sont placées dans un référentiel fictif dont le point de départ est la salle de départ. Dans ce référentiel, chaque salle occupe une position discrète [i][j]. L'algorithme à mettre en oeuvre dans la méthode `generateLine` :

1. Crée une aire de départ (`Spawn`) à la position `[x][y]=[0][0]`.
2. À chaque itération et tant que la longueur du niveau n'est pas atteinte :
   - Choisit une direction libre parmi « Nord », « Est » ou « Sud » (jamais « vers l'arrière »). Par exemple, si le choix se fait à l'est, ce sera la salle en position `[x+1][y]` qui sera créée.
   - S'assure que la position cible n'est pas déjà occupée (stockée dans un `Set`, voir à ce sujet l'Annexe 2 du tutoriel).
   - Calcule la difficulté de l'aire à créer selon la progression du joueur (en fonction du nombre de salles déjà générée, voir plus bas).
   - Crée une aire de type `SmallArea`, `MediumArea` ou `LargeArea` selon le niveau de difficulté choisi.
   - Connecte l'aire précédente à la nouvelle aire, et inversement.
3. À la fin :
   - Ajoute une `BossArea` à la suite de la dernière aire.
   - Connecte les deux.

Le tableau retourné contient toutes les aires dans l'ordre, du `Spawn` à la `BossArea`.

Évolution de la difficulté et de la taille des aires La difficulté du niveau augmente progressivement à mesure que le nombre `i` d'aires générées augmente. Soit une mesure de la progression calculée comme suit :

```
double progress = (double) (i + 1) / length;
```

Cette valeur représente un pourcentage d'avancement (allant de `0` à `1`) dans la génération du niveau.

On la convertit ensuite en une constante de difficulté :

```
int difficulty = switch ((int) (progress * 4)) {
    case 0 -> Difficulty.EASY;
    case 1 -> Difficulty.MEDIUM;
    case 2 -> Difficulty.HARD;
    default -> Difficulty.HARDEST;
```

En parallèle, cette valeur de progression est également utilisée pour déterminer dynamiquement le type des aires :

```
double r = rng.nextDouble();
if (r < progress * progress)
    return new LargeArea(...);
if (r < progress)
    return new MediumArea(...);
return new SmallArea(...);
```

Interprétation :

- En début de niveau (`progress` proche de 0), les aires sont généralement petites.
- Vers le milieu (`progress` $\approx$ `0.5`), les `MediumArea` deviennent plus fréquentes.
- Vers la fin (`progress` $\approx$ `1`), les grandes aires (`LargeArea`) deviennent très probables (car `progress` devient grand).

Ainsi, globalement, plus le joueur avance :

- Plus les aires deviennent difficiles.
- Plus elles deviennent grandes, ce qui permet d'augmenter l'intensité du jeu de manière fluide.

Cette montée en puissance est entièrement procédurale et donne une impression de progression naturelle, sans nécessiter de script manuel.

Note : En raison des tirages aléatoires, il demeure possible cependant que le personnage entre dans une aire plus facile après une plus difficile.

Retouche à (`ICMaze`) Afin de pouvoir intégrer la génération procédurale des aires, il vous est enfin demandé de modifier la méthode (`createAreas` de `ICMaze` de sorte à ce que

la création des salles labyrinthiques se fasse au moyen de la méthode `generateLine` (au lieu de l'ajout manuel des trois salles `SmallArea`, `MediumArea` et `LargeArea`). Le nombre d'aires labyrinthiques à générer peut être considéré comme une constante caractéristique du jeu (et vous pourrez jouer sur sa valeur pour créer un nombre plus ou moins important d'aires à des fins de test).

> Lorsque vous procéderez à cette modification, il est recommandé de simplement commenter l'appel à `generateHardCodedLevel()` plutôt que le supprimer (i.e. conserver l'ajout en dur des salles de façon commentée). Ceci permettra de "revenir en arrière" si vous n'arrivez pas à coder correctement la génération procédurale des niveaux.

4.3.1 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications et contraintes données, lancez votre jeu `ICMaze`. Vous vérifierez alors que :

- des aires labyrinthiques de difficultés croissantes sont automatiquement générées ;
- que la transition d'une aire à l'autre peut se faire sans problème dans les deux sens ;
- que l'aire de départ rester la salle `Spawn` et que la salle finale est la `BossArea` ;
- que le nombre d'aires labyrinthiques créées est conforme à la valeur stipulée dans le jeu ; en particulier qu'avec la longueur zéro le personnage passe directement de `Spawn` à `BossArea`.

4.4 Validation de l'étape 3

Pour valider cette étape, toutes les vérifications des sections 4.1.2, 4.2.1, et 4.3.1 doivent avoir été effectuées.

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

5 Défi final (étape 4)

Cette dernière étape est beaucoup plus libre. Seule une spécification des fonctionnalités à implémenter et quelques indications vous seront fournies.

Vous devrez faire en sorte que lorsque le personnage entre dans l'aire `BossArea` le portail d'entrée se verrouille empêchant tout retour en arrière. Il aura alors à affronter un ennemi belliqueux, naturellement nommé `Boss`, qui utilisera des projectiles pour lui causer du tort. L'issue du combat pourra avoir un impact sur la réouverture du portail d'entrée et sur le contenu des aires déjà visitées.

Enfin, lors de cette étape, la notion de dialogue sera également introduite, ce qui permettra de donner des indications utiles au joueur en début de partie.

La mise en oeuvre de cette étape devra s'appuyer sur la notion de signal logique (`Logic`) : par exemple l'éviction du `Boss` et la collecte de la clé qu'il lâche au moment de sa mort vont conditionner le comportement de la `BossArea` en tant que `Logic` et ce signal pourra à son tour conditionner le comportement d'autres aires.

5.1 Ennemi ultime : le `Boss`

Piège fatal Commencez par modifier le code existant de sorte à ce que le portail d'entrée de l'aire `BossArea` soit créé fermé au moyen d'une clé d'identifiant -1. Vérifiez alors que le personnage, une fois dans la `BossArea`, se trouve piégé à l'intérieur.

Il s'agit ensuite d'implémenter l'enemi `Boss` ainsi que les projectiles qu'il pourra lancer.

5.1.1 Projectiles

Un projectile est acteur mobile sur grille, « demandeur d'interaction », capable de survoler des cellules de la grille et sur lequel on peut marcher.

Un projectile est caractérisé par une vitesse de déplacement et la distance maximale qu'il peut atteindre depuis son point de départ. On pourra considérer ces caractéristiques comme fixes et identiques pour toutes les instances (par exemple 1 pour la vitesse et 7 pour la distance maximale). Il se déplace dans la même direction (celle de son orientation) et disparaît lorsqu'il a fini sa course (atteint sa distance maximale). L'attribut vitesse sert à moduler la vitesse de déplacement (par des tournures du type `move(MOVE_DURATION/speed)`). La distance maximale pourra être codée comme un entier. À chaque cycle de simulation on considérera pour simplifier que la distance restant à parcourir diminue d'une unité.

Par défaut, il n'accepte ni interaction de contact, ni interaction à distance. Il peut faire subir de son côté une interaction de contact tant qu'il n'a pas fini sa course (tout ce qu'il touche sur sa trajectoire peut être impacté).

Un projectile doit pouvoir aussi être arrêté dans sa course avant d'avoir atteint sa distance maximale.

Une seule sous-classe concrète de projectiles est à anticiper à ce stade, à choix les `WaterProjectile` ou `FireProjectile`. Ils se dessinent au moyen des animations décrites en 7.3.4 et font subir un nombre fixe de points de dommage au personnage en cas de contact (par exemple 1 point).

5.1.2 L'acteur `Boss`

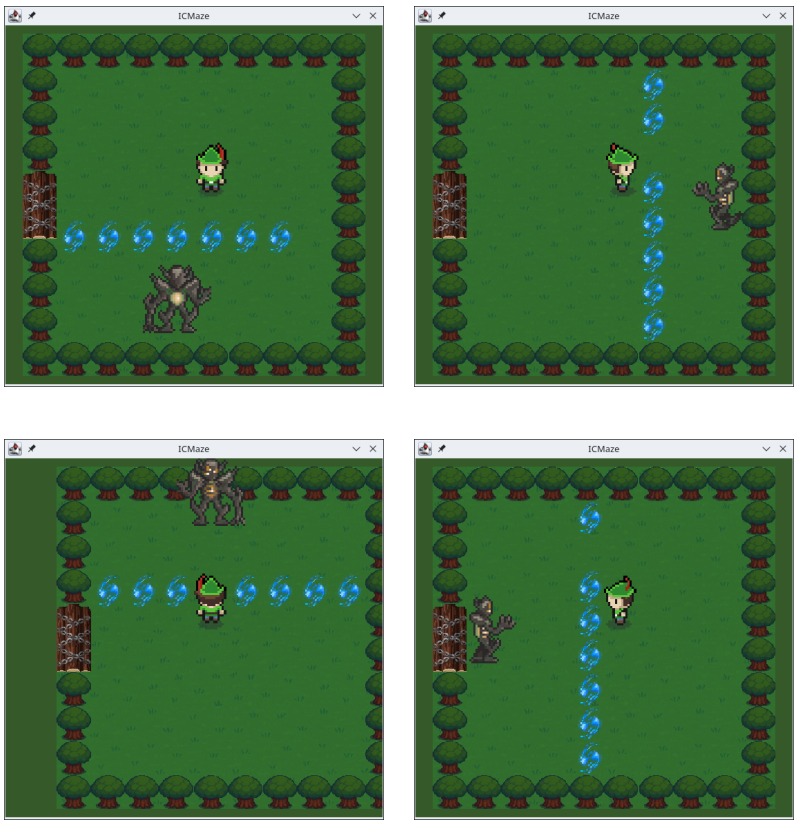

FIG. 11 : L'ennemi `Boss` peut se téléporter aléatoirement à l'une des positions illustrées par cette figure, il émet des barrages de projectiles sur toute la longueur ou largeur de l'aire, hormis une case choisie au hasard.

L'ennemi ultime (`Boss`, voir figure 11) n'a pas de spécificité pour pouvoir agir dans une labyrinthe. Son nombre de points de vie maximal est une valeur commune à toutes ses instances (5 par exemple). Il se dessine au moyen de l'animation *`"icmaze/boss"`*

Le personnage peut l'attaquer avec sa pioche et dans ce cas il inflige une nombre donné de pénalité (appelons le `damage`), selon l'algorithme simple suivant :

- S'il est attaqué pour la première fois, il se téléporte à un point de l'aire choisi au hasard (voir ci-dessous comment il procède). Il commence alors à émettre des barrages de projectiles autour de lui à intervalles de temps régulier.
- Sinon il se téléporte aussi, mémorise le fait qu'il a déjà été attaqué mais perd en plus `damage` points de vie.
- Lorsqu'il meurt, il lâche une clé permettant d'ouvrir le portail d'entrée de `BossArea`. Il faudra donc veiller à ce que l'identifiant de la clé soit choisi de manière adéquate.

Le comportement du `Boss` est simple et se résume à ce qui suit ;

- S'il n'a jamais été touché, il ne fait rien de spécifique.
- Sinon, si son animation courante n'est pas complétée, elle se met à jour, et sinon il gère l'émission de barrages de projectiles à intervalles de temps réguliers et fait subir un `reset` à son animation.

Téléportation à une position aléatoire Pour simplifier, les positions auxquelles le `Boss` se téléporte sont fixes : au milieu et au bord de chaque paroi extérieure (comme le montre la figure 11) et il s'y téléportera en regardant vers l'avant. À chaque fois l'une de ces positions sera choisie au hasard en évitant de garder la position courante.

Barrages de projectiles Les projectiles sont tirés depuis la ligne ou la colonne immédiatement devant le `Boss` et sur chaque case de cette ligne/colonne hormis une choisie au hasard (voir la figure 11). Cette case vide permettra au personnage de trouver un passage pour continuer à attaquer le `Boss`.

Pour tester ces développements, faites en sorte que l'aire `BossArea` soit dotée d'un `Boss` placé au milieu de l'aire. Réduisez temporairement le nombre d'aires labyrinthiques à zéro dans le jeu pour pouvoir tester plus facilement.

5.1.3 Tâche

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

1. que le portail d'entrée de `BossArea` est verrouillé ;
2. que l'ennemi `Boss` apparaît immobile dans cette aire ;
3. qu'il s'active dès qu'il subit la première attaque du personnage et qu'il se téléporte alors aléatoirement à l'une des positions montrées dans la figure 11 ;
4. qu'une fois activé, il lance, à intervalles de temps réguliers, des barrages de projectiles en face de lui ;
5. que ces projectiles couvrent toute la ligne ou colonne en face du `Boss`, à l'exception d'une case choisie au hasard ;
6. que les projectiles se déplacent en face du `Boss` et font subir des dommages au personnage ;
7. qu'ils disparaissent une fois les limites de l'aire atteinte ;
8. que le `Boss` lâche une clé après sa disparition lorsqu'il est vaincu par le personnage ;
9. et que la clé lâchée par le `Boss` permet d'ouvrir le portail d'entrée de `BossArea`.

5.2 Signaux logiques

Le tutoriel a introduit la notion de *signal* qui peut être exploitée pour finaliser la logique générale du jeu. L'objectif est de faire en sorte que le comportement d'une aire, et potentiellement des acteurs qui y sont présents, dépendent de signaux logiques.

Plus précisément, on souhaite mettre en place le fait que toute aire soit associée à un défi et qu'elle soit considérée comme réussie, lorsque le défi est surmonté. Le comportement de

l'aire et des acteurs qui y figurent peut alors dépendre de la réussite du défi. Le défi d'une aire peut de surcroit dépendre d'une défi d'autres aires.

L'idée est donc de faire en sorte que chaque aire *se comporte comme un* signal logique (`Logic`) qui est activé sous certaines conditions et que ce signal logique puisse être communiqué à d'autres aires ou acteur pour en conditionner le comportement.

Apportez les modifications nécessaires à votre code de sorte à ce qu'il respecte les conditions suivantes :

- Une `ICMazeArea` se comporte comme un signal logique allumé quand son défi est résolu et éteint sinon. On dira que l'aire est résolue lorsque son défi est surmonté.
- Le défi de la `BossArea` est de vaincre le `Boss` et de ramasser la clé qu'il lâche en mourant.
- Les aires labyrinthiques sont résolues lorsque `BossArea` l'est aussi. Dans ce cas les labyrinthes sont désactivés (ils ne s'affichent plus et les rochers ne font plus barrage) et les monstres troncs deviennent perpetuellement endormis.
- L'aire `Spawn` est résolue lorsque l'aire du `Boss` l'est aussi. Dans ce cas, un trésor de votre choix apparaît qui sera la récompense du personnage (et la réussite de la partie).

Vous veillerez à coder ceci :

- sans recourir à des getters intrusifs ;
- sans que les aires n'aient à se connaître mutuellement en tant que telles (il n'y a pas de raison qu'une aire ait connaissance de l'existence d'une autre aire du jeu en tant qu'aire) ;
- et sans que le acteurs ne mémorisent des informations trop spécifiques (collecte d'une clé spécifique par exemple) ; on devrait pouvoir changer la nature du défi d'une aire sans que cela n'ait d'impact sur le code des acteurs ou des autres aires.

Remarque : à des fins de tests il est recommandé de conserver l'exécutabilité de la méthode `generateHardCodedLevel()`. Vous veillerez à y apporter les modifications rendues nécessaires par vos nouveaux ajouts.

5.2.1 Tâche

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

1. que cela fait apparaître le portail d'entrée de `BossArea` et que le personnage peut l'ouvrir une fois la clé collectée ;
2. que les aires labyrinthiques sont désormais sans labyrinthe apparent et que les monstres tronc qui s'y trouvent sont endormis ;
3. qu'un trésor apparaît dans l'aire de `Spawn` ;
4. et que la touche `reset` permet de redémarrer le jeu dans son état initial (défis non résolus).

FIG. 12 : Dialogues simples pour documenter les touches à utiliser ou donner des recommandations au joueur.

5.3 Dialogues (optionnel)

De nombreuses touches sont déjà impliquées dans les interactions et il est utile de pouvoir les documenter au lancement du jeu. Certains passages du jeu peuvent aussi bénéficier de quelques conseils dispensé à bon escient. L'idée est donc maintenant d'introduire une composante « dialogues » pour répondre à ces besoins, à l'image de l'exemple de la figure 12.

La maquette fournit déjà une classe `Dialog` (dans le paquetage `engine.actor`) utilisable à cette fin. Pour créer un dialogue, il suffit de l'associer à un texte présent dans les ressources (sous-dossier `dialogs`), par exemple :

```
new Dialog("welcome");
```

La méthode `update` des `Dialog` permet de dérouler le texte. La méthode `isCompleted` retourne `true` lorsque tout le texte a été déroulé.

Les dialogues seront utilisés ici en guise d'indication fournie par le jeu à l'utilisateur. Ceci peut être mis en oeuvre en dotant `ICMaze` :

- d'un attribut de type `Dialog`, qui modélise l'indication à dispenser à un moment donné (le « dialogue actif ») ; on considérera qu'un dialogue est actif si cet attribut ne vaut pas `null` ;
- et d'une méthode `setActiveDialog` permettant d'affecter une valeur à cet attribut.

Il vous est demandé ensuite de retoucher la classe `ICMaze` de sorte que :

- l'aire courante n'évolue plus lorsqu'un dialogue est actif dans le jeu (l'aire sera alors uniquement dessinée) ; le dialogue actif devra évidemment aussi être dessiné ;
- le dialogue actif (s'il y en a un) se mette à jour lorsque l'on presse sur la touche `NEXT_DIALOG` :

```
kbd.get(KeyBindings.NEXT_DIALOG).isPressed()
```

où `kbd` est l'objet `Keyboard` associé au jeu ;
- et que le jeu puisse reprendre son cours normal lorsqu'un dialogue actif se termine (méthode `isCompleted()` des `Dialog`).

5.3.1 Activation d'un dialogue

La question qui se pose ensuite est de déterminer comment affecter une valeur à l'attribut « dialogue actif » du jeu sachant qu'en général, ce sera un composant du jeu et non pas le jeu lui même qui saura quand il faut le faire. Pour donner un exemple concret, si l'on souhaite qu'une indication soit affichée lorsqu'un personnage interagit avec un acteur, c'est dans la méthode de gestion de cette interaction que doit être appelée la méthode instanciant le dialogue actif du jeu. Cette méthode (ou la classe à laquelle elle appartient) doit alors avoir connaissance du jeu. De même si une indication doit être dispensée quand une aire donnée commence, l'aire doit avoir connaissance du jeu pour permettre à ce dernier d'instancier le dialogue actif. Il vous est suggéré d'utiliser ici la notion d'interface pour éviter de dévoiler l'intégralité du jeu aux composants qui ont besoin de le connaître pour y activer un dialogue.

Codez pour cela une interface `DialogHandler` qui aurait pour seul contenu une méthode `void publish(Dialog)`. Faites ensuite en sorte que le jeu implémente cette interface et redéfinisse la méthode `publish` afin qu'elle affecte le dialogue à publier au dialogue courant du jeu. Pour terminer, retouchez la classe `Spawn` de sorte à ce que :

- l'aire connaisse le jeu auquel elle appartient mais *uniquement* en tant que `DialogHandler` (vue abstraite moins intrusive) ;
- lors du premier appel à sa méthode `update`, le dialogue actif du jeu soit le dialogue *`"welcome"`*. Ce dialogue ne doit pas se re-afficher si on revient dans l'aire *`"Spawn`* après avoir visité une autre aire.

Dialogue de bienvenue et « reset » : La touche `GAME_RESET` permet de redémarer le jeu comme au départ, il est donc normal que la dialogue d'accueil s'affiche à nouveau suite à l'utilisation de cette touche.

5.4 Tâche

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

1. qu'au lancement du jeu le dialogue d'introduction s'affiche et puisse être déroulé au moyen de la touche `NEXT_DIALOG` ;
2. que pendant que le dialogue s'affiche les personnages ne répondent plus aux touches ;
3. qu'une fois le dialogue terminé, le jeu retrouve le comportement qu'il avait jusqu'ici ;
4. et que les touches de « reset » fonctionnent comme souhaité par rapport au dialogue de bienvenue.

5.5 Validation de l'étape 4

Pour valider cette étape, toutes les vérifications des sections 5.1.3 et 5.2.1 doivent avoir été effectuées. Les vérifications de la section 5.4 sont optionnelles.

Le jeu `ICMaze` 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 du projet, ou pour participer au concours, vous pouvez coder quelques extensions librement choisies. Au plus 15 points seront comptabilisés (coder beaucoup d'extensions pour compenser les faiblesses des parties antérieures n'est dont donc pas une option possible).

La mise en oeuvre est libre et très peu guidée. Seules quelques suggestions et indications vous sont données ci-dessous. Une estimation de barème pour les extensions suggérées 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 existant, `ICMaze`, mais il est alors **impératif de préserver les fonctionnalités obligatoires et la testabilité des composants demandés**.

Vous pouvez aussi, alternativement, créer un nouveau jeu `ICMazeExtension` utilisant la logistique que vous avez mise en place dans les étapes précédentes.

Vous prendrez soin de **commenter soigneusement** dans votre fichier `README`, les modalités de jeu de vos extensions. 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 (extrait court) : ICMaze.mp4

Une ébauche de fichier `README.md` correspondant à un jeu des années antérieures vous est donné en guise d'exemple : 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 ;-).

Notez qu'il existe dans le matériel fourni, quelques ressources pour des aires additionnelles.

6.1 Nouveaux acteurs ou extensions des personnages

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

- nouveaux `PathFinderEnemy` avec des stratégies différentes pour circuler dans les labyrinthes ; (~4 à 6 points)
- modélisation d'un système de ressources (or, argent, bois, nourriture, doses de soins etc) ; (~4 à 6 points)
- ajouter un objet `Box` ou `Safe`, dont l'ouverture serait dirigée par un signal et qui aurait pour contenu un ou plusieurs objets ; (~3pts)

- divers acteurs pouvant servir de signaux (orbes, torches, plaques de pression, leviers) ; (~4pts)
- acteurs de décors animés ; (~2pts)
- signaux avancés pour puzzle (oscillateurs, signaux avec retardateur) : un oscillateur est un signal dont l'intensité varie au cours du temps ; (~4pts/signal)
- toute sorte de personnages avec des modalités de déplacement et de comportement spécifiques ; pouvant être hostiles ou amicaux à l'égard du joueur ; (~3 pts à 6 pts selon la complexité du personnage)
- en particulier, des personnages vendeurs disposant d'un inventaire et chez qui les personnages principaux pourront acheter des équipements après avoir collecté des pièces de monnaie ; la notion d'inventaire devra être enrichie pour permettre à des articles de transiter d'un inventaire à l'autre ; ceci impliquera de coder un menu graphique permettant d'afficher les inventaires dans leur totalité (avec le nombre d'articles de chaque type et de sélectionner un article, par exemple celui que l'on veut acheter) (~5 à 10 points) : Notez que le menu associé à un inventaire peut être codé même sans personnage vendeur.
- créer des nouveaux modes de déplacement pour les personnages (en courant, nageant, etc.) avec une adaptation adéquate des sprites/animations et éléments de décor ; (~3 à 5 pts)
- créer des personnages suiveurs 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 un personnage qui arrive dans l'aire pour donner un objet ou donner une consigne. (~3 à 5 pts)
- ajouter de nouveaux types de cellules avec des comportements appropriés (eau, glace, feu, etc) ; (~2pts/cellule)
- implementer un cycle jour/nuit qui pourrait servir de signal ou qui conditionnerait le comportement des personnages (par exemple il ne peut plus avancer s'il fait trop noir et il devrait se munir d'une lampe de poche) ; (~5pts)
- ajouter une ombre ou un reflet au joueur et à certains acteurs ; (~2 à 3pts)
- ajouter de nouveaux contrôles avancés (interactions, actions, déplacements, etc) ; (~2pts/-contrôle)
- ajouter des événements aléatoires (décors, signaux, etc.) ; (~4pts)
- ajouter des dialogues à choix multiples ; (~3 à 6 pts)
- etc.

6.2 Pause et fin de jeu (~2 à 5pts)

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 les personnages ont atteint un objectif ou ont été battus par exemple).

En réalité, la base que vous avez codée peut être enrichie à l'envi. Vous pouvez aussi laisser parler votre imagination, et essayer vos propres idées.

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

L'annexe 3 du tutoriel vous donne des indications pour enrichir les ressources graphiques.

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

6.3 Validation de l'étape 5

Comme résultat final du projet, créez un scénario de jeu bien documenté dans le `README` et 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.

6.4 Concours

Les personnes qui ont terminé le projet avec un effort particulier sur le résultat final (gameplay intéressant, richesse de aires de jeu, 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 13 :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).

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

7 Annexes

7.1 KeyBindings

Une classe `KeyBindings` est fournie pour faciliter la configuration des touches utilisables dans le projet. Elle est codée au moyen de la notion de `record` qui ne sera vue qu'au second semestre. Un `record` n'est autre qu'un moyen abrégé d'écrire une forme restreinte de classe. Pour ce projet il suffit de savoir que :

- les touches à utiliser pour les contrôles liés au personnage sont définies dans la variable `PLAYER_KEY_BINDINGS` (et peuvent être modifiées librement selon vos préférences) ;
- le type `PlayerKeyBindings` permet de faire en sorte que **dans l'ordre** les touches correspondent aux contrôles : « déplacement vers le haut », « la gauche », « le bas », « la droite », « changement d'équipement » (sélection de l'équipement courant dans l'inventaire) et « utilisation de l'équipement courant » ;
- pour doter un `ICMazePlayer` d'un ensemble de touches spécifique, il suffit de le doter d'un attribut de type `KeyBindings.PlayerKeyBindings` ;
- lorsque `keys` est un objet de type `KeyBindings.PlayerKeyBindings`, il suffit d'écrire `keys.pickaxe()` (et de façon analogue, `keys.right()`, `keys.down()`, etc) pour référencer une touche donnée : celle associée à l'utilisation de la pioche par exemple. La méthode `moveIfPressed` des personnages peut invoquer :

  ```
  moveIfPressed(Orientation.LEFT,keyboard.get(keys.left()));
  ```

  ou lors de l'`update` d'un personnage, on peut tester si la touche d'utilisation d'équipement est pressée par :

  ```
  keyboard.get(keys.pickaxe()).isPressed()
  ```

Note : La configuration par défaut est pour un clavier QWERTZ. Pour un clavier AZERTY on aurait plutôt :

```
RED_PLAYER_KEY_BINDINGS = new PlayerKeyBindings(Z, Q, S, D, A,
   E);
```

dans `KeyBindings.java`.

7.2 Génération de nombres aléatoires

Java permet de générer des nombres dits « pseudo-aléatoires » au moyen d'une classe nommée `Random`. Dans le paquetage `icmaze`, vous trouverez une classe `RandomGenerateur` qui offre une instance de générateur de type `Random`. Ceci permet de n'avoir qu'un seul objet de type `Random` dans le projet et de permettre de créer des situations de test déterministes à des fins de debugging, comme expliqué n peu plus bas.

Au travers de l'objet `RandomGenerator.rng`, il est possible d'invoquer toutes les méthodes de génération aléatoire offertes par l'API de la classe Random, par exemple :

```
RandomGenerator.rng.nextInt(10);
```

permet de tirer un nombre entier au hasard (avec distribution de probabilité uniforme) entre 0 et 10.

Les algorithmes implémentés par les générateurs de nombres pseudo-aléatoires utilisent de que l'on apelle une « graine » (« seed ») qui par défaut est aléatoire. Pour une même valeur de la graine c'est toujours la même séquence de nombre qui est tirée. On peut donc jouer sur cette caractéristique pour permettre de créer des situations de tests reproductibles au moment du debugging (autrement les programmes avec des comportements aléatoires seraient ardus à corriger, car il n'est pas garanti que d'une exécution à l'autre on soit dans les mêmes conditions).

Si vous voulez forcer le générateur à partir d'une graine donnée, il vous suffit de remplacer la ligne :

```
public static Random rng = new Random();
```

par

```
public static Random rng = new Random(val);
```

où `val` est la valeur de la graine choisie (peu importe laquelle), dans la classe `RandomGenerator`.

7.3 Création des animations

Le code des animations n'est pas très intéressant à produire. Il vous est donc donné un exemple de code pour chacune des classes.

7.3.1 `ICMazePlayer`

Animation de base :

```
final Vector anchor = new Vector(0, 0);
final Orientation[] orders = { DOWN, RIGHT, UP, LEFT };
 ... new OrientedAnimation(prefix, ANIMATION_DURATION, this,
                           anchor, orders, 4, 1, 2, 16, 32,
                           true)
```

avec `ANIMATION_DURATION` valant 4, par exemple, et `prefix` valant *`"icmaze/player"`*.

Utilisation de la pioche

```
final Vector anchor = new Vector(-.5f, 0);
final Orientation[] orders = {DOWN, UP, RIGHT, LEFT};
... new OrientedAnimation(prefix+".pickaxe",
                          PICKAXE_ANIMATION_DURATION, this,
                          anchor, orders, 4, 2, 2, 32, 32)
```

avec `PICKAXE_ANIMATION_DURATION` valant 5.

7.3.2 Coeurs

```
new Animation("icmaze/heart", 4, 1, 1, this, 16, 16,
                ANIMATION_DURATION/4, true)
```

où ANIMATION_DURATION vaut 24.

7.3.3 Disparition dans un nuage

```
new Animation("icmaze/vanish", 7, 2, 2, this, 32, 32, new
   Vector(-0.5f, 0.0f), ANIMATION_DURATION/7, false);
```

où ANIMATION_DURATION vaut 24.

7.3.4 Projectiles d'eau ou de feu

```
new Animation(name, 4, 1, 1, this, 32, 32,
               ANIMATION_DURATION/4, true)
```

où ANIMATION_DURATION vaut 12 et name vaut *"icmaze/waterProjectile"* pour les projectiles d'eau et *"icmaze/magicFireProjectile"* pour ceux de feu.

7.3.5 Mort des enemmis

Les ennemis disparaissent dans un petit nuage en mourant (comme en 7.3.3).

7.3.6 Monstres troncs

En eveil :

```
 // En mode déplacement ciblé
 Orientation[] orders = new Orientation[]{Orientation.DOWN,
    Orientation.UP, Orientation.RIGHT, Orientation.LEFT};
... new OrientedAnimation("icmaze/logMonster",
                          ANIMATION_DURATION/3, this,
                          new Vector(-0.5f, 0.25f), orders,
                          4, 2, 2, 32, 32, true)

// En mode déplacement aléatoire
orders = new Orientation[]{Orientation.DOWN, Orientation.UP,
   Orientation.RIGHT, Orientation.LEFT};
... new OrientedAnimation("icmaze/logMonster_random",
                          ANIMATION_DURATION/3, this,
                          new Vector(-0.5f, 0.25f), orders,
                          4, 2, 2, 32, 32, true)
// En mode endormi
orders = new Orientation[]{Orientation.DOWN, Orientation.LEFT,
   Orientation.UP, Orientation.RIGHT};
... new OrientedAnimation("icmaze/logMonster.sleeping",
```

```
                                ANIMATION_DURATION/3, this,
                                new Vector(-0.5f, 0.25f), orders,
                                1, 2, 2, 32, 32, true)
```

où ANIMATION_DURATION vaut 30.

7.3.7 Boss

```
final Vector anchor = new Vector(-0.5f, 0);
final Orientation[] orders = {DOWN, RIGHT, UP, LEFT};
...new OrientedAnimation("icmaze/boss", ANIMATION_DURATION/4,
                         this, anchor, orders, 3, 2, 2, 32, 32,
                         true)
```

où ANIMATION_DURATION vaut 60.