CS-107 : Mini-projet 2
Jeux coopératifs

J. Kalajdzic, J. Sam, A. Paulino

Version 1.4

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 familiarisés ces dernières semaines avec les fondamentaux d'un petit moteur de jeux adhoc (voir tutoriel) vous permettant de créer des jeux sur grille en deux dimensions 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 coopérative, nommée `ICcoOp`, de ce types de jeux impliquant **deux personnages principaux**. Ces derniers collaboreront entre eux pour atteindre des objectifs ou vaincre des ennemis. Le jeu de base qu'il vous sera demandé de créer est inpiré de jeux de type *Fire Boy and Water Girl*. 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.

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 `ICCoOp` où deux personnages principaux seront capables de se suivre d'une aire à l'autre et d'interagir avec des objets.
- Étape 2 (« Première coopération ») : lors de cette étape les deux personnages vont commencer à s'entraider pour affronter des murs hostiles. Des dialogues rudimentaires seront introduits de même que le décompte des points de vie des personnages.
- Étape 3 (« Adversaires bélliqueux ») : cette étape permettra à vos personnages d'affronter des ennemis et d'engager des batailles avec eux au moyen d'équipements.
- Étape 4 (« Défis finaux ») : cette étape introduira de nouvelles aires avec de nouveaux défis à relever de façon coopérative en exploitant la notion de signaux logiques.
- 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.

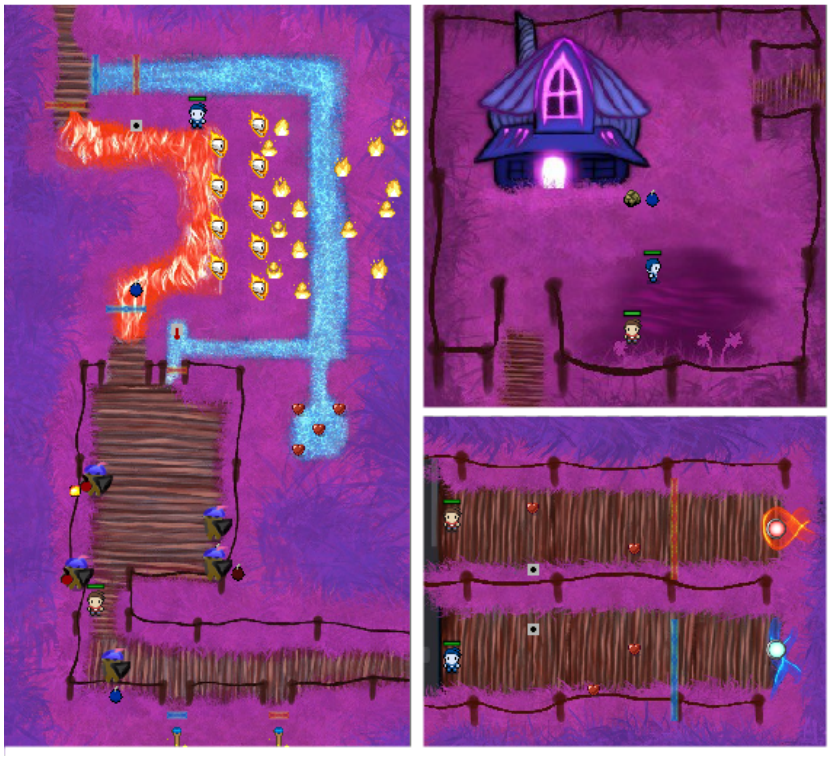

FIG. 1 : Quelques scènes du jeu où deux joueurs s'entraident d'aire en aire en collectant des objets et en affrontant diverses créatures hostiles (ou pas).

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.

(suite sur le page suivante . . .)

4. Les indications peuvent être parfois très détaillées. **Cela ne veut-tart pas dire pour autant qu'elles soient exhaustives**. Les méthodes et attributs nécessaires à la réalisation des traitements voulus ne sont évidemment pas tous décrits et ce sera à vous de les introduire selon ce qui vous semble pertinent et en respectant une bonne encapsulation.
5. Votre projet **ne doit pas être stocké sur un dépôt public** (de type github). Pour ceux d'entre vous 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 ICoop de base (étape 1)

Le but de cette étape est de commencer à créer votre propre jeu de type `ICoop`.

Cette version de base contiendra deux personnages principaux contrôlables individuellement et capables d'interagir avec des objets. Ce dernier point sera mis en oeuvre selon le mécanisme plus général des *interactions entre acteurs*, tel que décrit dans le tutoriel 3. Ce jeu fera donc intervenir :

- deux personnages principaux ;
- des portes que les personnages pourront traverser. Il s'agira d'une interaction de contact ;
- un explosif que les personnages pourront activer (par interaction à distance) ;
- un rocher que l'explosif une fois activé pourra faire exploser (par interaction à distance à nouveau).

Les deux personnages se suivront lors des transitions d'aires.

Vous travaillerez dans le paquetage fourni `icoop`.

2.1 Préparation du jeu `ICoop`

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

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

- de la classe `ICoopPlayer` qui modélise un personnage principal, à placer dans `icoop.actor` ; laissez cette classe vide pour le moment, nous y reviendrons un peu plus bas ;
- de la classe `ICoop`, équivalente à `Tuto2` à placer dans le paquetage `icoop` ; N'oubliez pas d'adapter la méthode `getTitle()` qui retournera un nom de votre choix (par exemple *`"ICoop"`*!) ;
- une classe `ICoopArea` équivalente à `Tuto2Area`, à placer dans un sous-paquetage `icoop.area` ;
- des classes `Spawn` et `OrbWay` héritant de `ICoopArea`, à placer dans le paquetage `icoop.area` (elles sont équivalentes aux classes `Ferme` ou `Village` dans le tutoriel. En guise de titre prenez *`"Spawn"`*`et` *`"OrbWay"`* afin que la correspondance avec les ressources graphiques se fasse bien.
- de la classe `ICoopBehavior` analogue à `Tuto2Behavior` à placer dans `icoop` et qui contiendra une classe publique `ICoopCell` équivalente de `Tuto2Cell`.

Vous ferez en sorte que les positions initiales des personnages principaux à venir soit dictées par une méthode retournant des valeurs spécifiques pour chaque aire : par exemple `(13,6)` pour le personnage rouge et `(14,6)` pour le bleu dans l'aire `Spawn` et `(1,12)` et `(1,5)` pour l'aire `OrbWay`.

Pour le type de jeux qui nous intéressent, il sera plus adapté de centrer la vue au mieux la vue dans la fenêtre. Pour cela, vous ajouterez simplement la redéfinition suivante dans

`ICoopArea` :

```
@Override
public boolean isViewCentered() { return true; }
```

2.2 Adaptation de `ICoopBehavior`

Dans un premier temps, il s'agit d'apporter quelques nouveautés aux classes `ICoopBehavior` et `ICoopCell`. Le type énuméré décrivant les types de cellules aura un champ supplémentaire indiquant si la cellule peut-être survolée ou pas (un champ `canWalk` similaire à `isWalkable` et un champ `canFly` indiquant si l'on peut survoler la cellule ou pas, et qui peut être utilisé plus tard dans le projet) :

```
NULL(0, false, false),
WALL(-16777216, false, false),
IMPASSABLE(-8750470, false, true),
INTERACT(-256, true, true),
DOOR(-195580, true, true),
WALKABLE(-1, true, true),
ROCK(-16777204, true, true),
OBSTACLE(-16723187, true, true)
```

Par ailleurs, la nature des cellules ne sera pas le seul élément conditionnant le fait qu'elle accepte de laisser entrer un acteur (via sa méthode `canEnter`) . Les acteurs qui y sont déjà présents auront aussi leur mot à dire. Dans le cas du jeu `ICoop`, une cellule pourra avoir dans son ensemble d'entités (hérité de `Cell`) au plus un acteur qui soit non traversable : deux acteurs non traversables ne peuvent pas y cohabiter ! C'est la méthode `takeCellSpace()` qui indique si un acteur est traversable (accepte qu'on lui « marche dessus ») ou non. Il l'est si sa méthode `takeCellSpace()` retourne `false`.

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

2.2.1 Tâche

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

2.3 Ebauche des personnages principaux

Pour le moment codez `ICoopPlayer` dans le même esprit que `GhostPlayer`.

Une différence importante est que les personnages, tout comme de nombreux acteurs à venir, vont « servir » un élément naturel (élément « eau » ou élément « feu » typiquement).

Vous ferez donc en sorte d'introduire une catégorie `ElementalEntity`. Tout acteur se comportant comme un `ElementalEntity` devra définir une méthode `element()` retournant quel élément naturel l'acteur sert. Ce sera le cas évidemment pour les personnages principaux. L'élément servi par le personnage sera spécifié à la construction.

FIG. 2 : Aire d'accueil

2.3.1 Dessin

Une autre différence importante avec `GhostPlayer` est que l'image qui servira à dessiner un `ICoopPlayer` dépendra de son *orientation* et elle sera *animée*.

Vous utiliserez donc, 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]. Comme nous aurons à créer deux personnages, le nom de l'image permettant de créer l'animation va évidemment différer de l'un à l'autre. Vous considérerez que ce nom est paramétrable à la construction du personnage.

2.3.2 Comportement

Un `ICoopPlayer` est un acteur non traversable et orienté par défaut vers le bas. Il se comporte (se met à jour) comme une `MoveableAreaEntity` mais en plus :

- il doit pouvoir être déplacé au moyen de touches à la manière du `GhostPlayer` codé dans le tutoriel ;
- son animation courante doit également être mise à jour selon l'algorithme suivant : si un déplacement est en cours, l'animation courante doit subir un `update`, sinon, elle doit subir un `reset`.

Une des particularités des jeux `ICoop` est qu'il faudra contrôler deux personnages et par conséquent, utiliser deux jeux de touches différents. Le jeu de touches spécifique à un personnage donné peut être paramétré à la construction comme un `KeyBinding.PlayerKeyBindings`. 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 `keys.up()`, `keys.down()`, `keys.right()` et `keys.left()` où `keys` est le jeu de touches associé au personnage.

Une fois cette ébauche `ICoopPlayer` complétée, vous pouvez commencer à adapter la mé-

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

FIG. 3 : Porte allant de l'aire "Spawn" à l'aire "OrbWay"

thode `begin` de `ICoop` et à tester votre jeu. Pour commencer, vous n'introduirez que l'un des deux personnages.

Au lancement, un joueur de type `ICoopPlayer` sera donc créé à la position qui lui est destinée dans l'aire courante. Le nom des images associées à son animation de base sera *`"icoop/player"`* et les touches à utiliser pour le contrôler seront celles spécifiées par `KeyBindings.RED_PLAYER_KEY_BINDINGS`.

2.3.3 Tâche

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

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

1. que le jeu démarre en affichant un `ICoopPlayer` qui regarde vers l'avant ;
2. que le personnage puisse être déplacé librement sur toute l'aire au moyen des touches spécifiées par `KeyBindings.RED_PLAYER_KEY_BINDINGS` (`W`,`A`, `S` et `D` si vous ne changez rien) mais qu'il ne doit pas pouvoir en sortir ;
3. que sa représentation graphique s'adapte bien à son orientation ;
4. et que ses mouvements soient animés.

2.4 Portes : interactions de contact

Il vous est demandé maintenant d'appliquer le schéma des interactions suggéré dans la 3ème partie du tutoriel (Lien cliquable)[2].

La première interaction à laquelle nous nous intéressons est une interaction de contact avec un acteur « porte » qui permettra à un `ICoopPlayer` de transiter d'une aire à une autre.

[2]Un complément vidéo est aussi disponible pour expliquer la mise en œuvre des interactions :mp2-interactions.mp4

2.4.1 Portes

Comme expliqué dans le tutoriel, si une entité subit des interactions elle doit être modélisée comme un `Interactable` ; si elle en fait subir, elle doit être modélisé comme un `Interactor` (et elle peut être les deux).

Comme toute première catégorie de `Interactable`, nous disposons déjà des `ICoopCell` (par exemple s'il est en contact avec certains types de cellules, le personnage pourrait adopter un autre comportement comme se mettre à glisser). Nous n'exploiterons pas cette facette pour le moment mais choisissons plutôt d'introduire un nouvel `Interactable`, « porte », qui aura une utilité plus immédiate.

Une porte, `Door`, à coder dans le paquetage `icoop.actor` est un acteur de type `AreaEntity` qui permet de transiter vers une aire de destination.

Cet acteur est caractérisé par :

- le nom de l'aire vers laquelle il permet de transiter (une chaîne de caractères) ;
- et l'ensemble des coordonnées d'arrivée (`DiscreteCoordinates`) dans l'aire de destination. Cet ensemble sera de taille variable même si a priori dans le projet il aura une taille de deux : il s'agit dans l'ordre des coordonnées d'arrivée du personnage rouge puis de celles du personnage bleu.

Le tutoriel a introduit la notion de signal qui peut commencer à être exploitée ici pour rendre le jeu plus attractif en le faisant dépendre de résolution d'énigmes. On peut imaginer par exemple que pour obtenir une arme avec laquelle se défendre, le personnage ait à trouver une clé qui permettrait d'ouvrir une porte vers une salle contenant l'arme voulue. L'outillage fourni, permettrait typiquement de répertorier un acteur clé comme comme un signal attaché à cette porte : une fois le signal activé (la clé ramassée), la porte s'ouvre.

Un acteur « porte » sera donc aussi caractérisé par un *signal* (de type `Logic`) qui permettra de modéliser sous quelles conditions elle est ouverte ou pas.

Le constructeur d'une `Door` prend donc en paramètres : le nom de l'aire de destination, le signal conditionnant son ouverture, les coordonnées d'arrivée possible dans l'aire de destination, l'aire à laquelle elle appartient, la position de sa cellule principale et éventuellement la liste des coordonnées des cellules qu'elle occupe en plus de sa cellule principale (exprimée au moyen d'une ellipse dans un second constructeur). La méthode `getCurrentCells()` doit donc être redéfinie dans `Door` pour retourner l'ensemble de ces coordonnées occupées.

Une `Door` est un acteur traversable qui accepte les interactions de contact et dont la méthode de dessin ne fait rien par défaut. En effet, les portes coincideront avec des éléments du décor et ne se dessineront pas en tant que telles. De ce fait leur orientation n'est pas très importante.

Il vous est demandé d'introduire ce nouvel acteur et d'en enregistrer une instance aux caractéristiques suivantes dans l'aire `Spawn` :

destination	coord. arrivée	signal	cellule principale	autres coord.
"OrbWay"	(1,12)(1,5)	Logic.TRUE	(19,15)	(19,16)

Ceci revient à positionner une porte à l'endroit indiqué dans la figure 3.

Enregistrez aussi les portes permettant de faire la transition dans l'autre sens de `OrbWay` à `Spawn` :

destination	coord. arrivée	Signal	cellule principale	autres coord.
"Spawn"	(18,16),(18,15)	Logic.TRUE	(0,14)	(0,13),(0,12),(0,11), (0,10)
"Spawn"	(18,16),(18,15)	Logic.TRUE	(0,8)	(0,7), (0,6), (0,5), (0,4)

La valeur `Logic.TRUE` indique que ces portes sont ouvertes sans condition. Les emplacements des portes peuvent être retrouvés en comptant les coordonnées des cases rouges dans les images "behavior" (par exemple, voir `resources/images/behaviors/Spawn.png`)

2.4.2 `ICoopPlayer` comme `Interactor`

Maitenant que nous disposons d'une entité concrète pouvant subir des interactions, intéressons nous à `ICoopPlayer` comme entité les faisant subir. Commencez par mettre en place le fait que tous les `ICoopPlayer` deviennent des `Interactor` (ils peuvent faire subir des interactions à des `Interactable`).

Comme `Interactor`, `ICoopPlayer` 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 formant son champ de vision consistent en l'unique cellule à laquelle il fait face :

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

En tant que `Interactor`, il voudra systématiquement toutes les interactions de contact (`wantsCellInteraction` retourne toujours `true`). Le fait qu'il soit demandeur d'interactions à distance (`wantsViewInteraction`) sera conditionné par l'utilisateur du jeu : sa touche `keyBindings.useItem())` sera utilisée pour indiquer que le personnage veuille une interaction à distance. Par exemple, si un personnage est en face d'un explosif, pour indiquer que l'on souhaite qu'il l'active, on appuie sur la touche correspondant à `keyBindings.useItem())` (par défaut, `E` pour le joueur rouge et `O` pour le bleu).

Nous pouvons maintenant procéder à la gestion concrète des interactions possibles à ce stade : dans le sous-paquetage `icoop.handler`, complétez l'interface `ICoopInteractionVisitor` héritant de `AreaInteractionVisitor`. Cette interface doit fournir une définition par défaut des méthodes d'interaction `interactWith` de tout `Interactor` du jeu de `ICoop` avec :

- une cellule du jeu (`ICoopCell`) ;
- un personnage principal du jeu (`ICoopPlayer`) ;
- une porte (`Door`)

Ces définitions (par défaut) auront un corps vide pour exprimer le fait que par défaut, l'interaction consiste à ne rien faire (`ICoopPlayer` en tant que `Interactor` du jeu `ICoop`, devra fournir si nécessaire 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 `ICoopInteractionVisitor`.

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

Cette méthode peut contenir plus de code selon les cas, mais elle est pour le moment suffisante pour les trois types de `Interactable` à disposition, à savoir : `ICoopPlayer`, `ICoopCell` et `Door`.

Pour que `ICoopPlayer` puisse gérer plus spécifiquement les interactions qui l'intéressent, définissez dans la classe `ICoopPlayer`, une classe imbriquée privée `ICoopPlayerInteractionHandler` implémentant `ICoopInteractionVisitor`. Ajoutez-y les définitions nécessaires pour gérer plus précisément l'interaction avec les portes de sorte à ce que lors d'une interaction de contact avec la porte, si le signal de cette dernière est activé (méthode `isOn()`), le personnage :

1. quitte son aire courante (méthode `leaveArea()`)
2. arrive dans l'aire de destination dans la première coordonnée d'arrivée possible pour celle-ci (méthode `setCurrentArea`).

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 !

2.4.3 Tâche

Il vous est demandé de coder les concepts et traitements décrits précédemment conformément aux spécifications et contraintes données. Lancez votre jeu `ICoop`. Vérifiez alors que le personnage principal peut transiter de la salle *`"Spawn"`* à la salle *`"OrbWay"`* (et vice-versa) par l'emplacement correspondant à la porte de la figure 3. La porte du bas dans la salle `Orbway` ne sera atteignable que lorsque le second personnage sera introduit.

2.5 Deuxième personnage

Il s'agit maintenant d'introduire ce qui fera la spécificité des jeux `ICoop` ; la présence d'un second personnage. Son introduction va tout d'abord poser la question de comment centrer proprement la caméra qui, jusqu'ici suit l'unique personnage comme dans le tutoriel.

L'idée adoptée est de centrer la caméra sur un point à équidistance des deux personnages. Pour simplifier les calculs requis un acteur `CenterOfMass` est fourni.

Il vous est maintenant demandé de faire évoluer le jeu `ICoop` de sorte à :

- ce qu'il soit caractérisé par un second personnage ;

- ce que sa méthode `begin` crée ce personnage et centre la caméra sur un objet `CenterOfMass` construit au moyen des deux personnages.

Le second personnage aura comme position initiale celle qui lui est dictée dans l'aire courante, comme image associée *`"icoop/player2"`*, et les clés permettant de le contrôler seront celles données par `KeyBindings.BLUE_PLAYER_KEY_BINDINGS`.

Si ce personnage transite via une porte, il doit arriver sur la **seconde** coordonnée d'arrivée de l'aire d'arrivée. Vous ferez également en sorte que les deux personnages se suivent ; c'est-à-dire que si l'un passe une porte l'autre le suive dans l'aire de destination en y arrivant au niveau de la coordonnée d'arrivée qui y est prévue pour lui par cette porte. Les deux personnages vont donc toujours occuper la même aire !

Enfin, pour s'assurer que le facteur d'échelle initialement choisi ne soit pas en défaveur de l'observation conjointe des deux personnages, la méthode `update` du jeu le recalcule dynamiquement (et le reaffecte au moyen de `setCameraScaleFactor`) au moyen d'une formule heuristique telle que :

```
max(DEFAULT_SCALE_FACTOR, DEFAULT_SCALE_FACTOR * 0.75
                        + distance(position_playerA, position_playerB) / 2)
```

Vous êtes libre d'affiner cette formule à votre guise. La distance peut être calculée comme la norme du vecteur `position_playerA - position_playerB` (méthodes `sub` et `getLength` des `Vector`).

2.5.1 Tâche

Une fois les modifications suggérées ci-dessus implémentées, lancez votre jeu `ICoop`. Vous vérifierez :

- qu'un second personnage apparaît au lancement du jeu ;
- qu'il peut être contrôlé au moyen de ses touches spécifiques ;
- qu'il se comporte comme le premier personnage mais se dessine avec une animation colorée en bleu ;
- que chaque personnage peut transiter de *`"Spawn"`* à *`"OrbWay"`* et vice-versa, que dans ce cas, l'autre le suit et que les personnages arrivent dans l'aire à la position de destination prévue pour eux par la porte franchie. Un exemple de passage de porte est fourni dans la vidéo suivante : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/secondPlayer.mp4

2.6 Explosifs : interactions à distance

Pour bien assoir le mécanisme de gestion des interactions, il s'agit dans la dernière partie de cette étape d'introduire deux nouveaux acteurs, les explosifs et les obstacles, avec l'idée qu'un explosif peut faire exploser certains obstacles *à distance*.

2.6.1 Obstacles et explosifs

Un *obstacle* est un acteur dérivant de `AreaEntity` et dessinable par défaut au moyen du sprite `"rock.2"`. Par défaut, il est non traversable et accepte inconditionnellement tout type d'interaction. Les *rochers* sont un type particulier d'obstacles, dessinables au moyen du sprite `"rock.1"`, et qui peuvent être détruits : ils ne sont pas traversables et ils ne se dessinent que s'ils ne sont pas détruits.

Un *explosif* est un acteur traversable héritant de `AreaEntity` et jouant le role d'`Interactor`. Il peut être activé et est associé à un retardateur (un entier) initialisé à sa création. Il est créé désactivé. Lorsqu'il est activé, à chacun de ses `update`, son retardateur est décrémenté (par exemple d'une unité). Lorsque le retardateur est à zéro, l'explosion a lieu (l'explosif est en état d'explosion), ce qui cause l'affichage d'une animation particulière, puis il disparaît du jeu. Un explosif, en tant que `Interactor`, est en demande d'interaction à distance ou de contact lorsqu'il est en état d'explosion. Pour le moment il interagit à distance uniquement avec les rochers. L'effet de l'interaction avec ces derniers sera évidemment de les détruire. Le champ d'action, `getFieldOfViewCells()`, de l'explosif est l'ensemble des 4 cellules immédiatement voisines horizontalement et verticalement à sa cellule principale. `getCurrentCells()` sera codé comme pour `ICoopPlayer`. En tant que Interactable un `explosif` peut subir des interactions à distance s'il n'a pas explosé, et des interactions de contact seulement s'il n'est pas activé et n'a pas explosé.

L'explosif peut être représenté graphiquement au moyen d'une animation. Il doit disparaître après son explosion qui doit s'accompagner d'une autre animation spécifique. Référez-vous à l'annexe 7.3.2 pour des indications plus précises à ce sujet.

Vous ferez aussi enfin en sorte que lorsqu'un personnage entre en interaction à distance avec un explosif il soit activé. Souvenez-vous que les personnages expriment le fait de vouloir une interaction à distance au moyen de leur touche `useItem()` (par défaut `'E'` et `'O'` respectivement).

Créer pour finir un explosif et un rocher en positions respectives `(11,10)` et `(10, 10)` de `Spawn`.

2.6.2 Tâche

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

Lancez votre jeu `ICoop`. Vous vérifierez qu'il se comporte comme sur la petite vidéo suivante : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step1Explosion.mp4 ; c'est-à-dire concrètement :

- que les personnages ne puissent pas traverser le rocher mais puissent traverser les explosifs ;
- que chaque personnage puisse activer l'explosif par une interaction à distance mais ne puisse le faire qu'à distance ;
- qu'une fois l'explosif activé, le retardateur soit déclenché et que l'explosion ait lieu au bout d'un moment ;
- que l'explosion s'accompagne d'une animation et qu'il n'y ait plus de trace de l'explosif une fois cette animation terminée ;

- que l'explosion détruise le rocher et que l'emplacemement de ce dernier, laissé vide, soit traversable par les personnages.

2.7 Reset

Pour faciliter les tests, vous doterez `ICoop` du contrôle suivant : la touche `KeyBindings.RESET_GAME` ('R' par défaut) doit permettre de faire une réinitialisation ("reset") du jeu et `KeyBindings.RESET_AREA` ('T' par défaut) la réinitialisation de l'aire.

Dans le premier cas, il s'agit de redémarrer dans les mêmes conditions que la toute première fois qu'on le lance. Dans le second, c'est **uniquement** l'aire courante qui est redémarrée. Dans les deux situations, les méthodes `begin` devraient rendre de loyaux services. Pour le redémarrage de l'aire courante, il faudra trouver le moyen d'y recréer/replacer les personnages à leur position de « spawn ». 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 les deux touches de « reset » fonctionnent comme attendu et que les personnages réapparaissent bien aux positions de « spawn » dictées spécifiquement par les aires. Testez les deux touches dans les deux aires existantes.

2.8 Validation de l'étape 1

Pour valider cette étape, toutes les vérifications des sections 2.2.1, 2.3.3, 2.4.3, 2.5.1, 2.6.2 et 2.7.1 doivent avoir été effectuées.

Le jeu `ICoop` 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 Première coopération (étape 2)

Cette seconde partie du projet a pour but d'introduire des acteurs hostiles qui malmèneront nos personnages principaux en leur infligeant des dommages. Des moyens de se prémunir de ces dangers seront introduits pour leur permettre de survivre au cours de leurs désormais périlleux périples.

Plus précisément, il s'agit d'étoffer le jeu en introduisant les nouveaux éléments suivants :

- les personnages deviendront vulnérables à certains types de dommages qui leur feront perdre des points de vie ; vous anticiperez des dommages d'eau, de feu ou physiques et ces dommages ne s'appliqueront pas qu'aux personnages ;
- des murs de feu infligeront des dégâts de feu aux personnages ;
- des murs d'eau infligeront des dommages d'eau ;
- des orbes (de type feu ou eau) leur permettront d'acquérir l'invulnérabilité à ces types de dommages ;
- et des plaques de pression permettront de désactiver les murs : le personnage bleu pourra se poser sur une plaque de pression qui désactivera le mur de feu (et reciproquement pour le mur d'eau et le personnage rouge) ; ceci permettra aux deux personnages de commencer à coopérer pour affronter ces murs hostiles.

La mise en oeuvre de ces nouveaux éléments va à nouveau dépendre de la notion de signal déjà utilisée pour les portes à l'étape précédente.

Lors de cette étape, les graphismes seront élaborés pour permettre de visualiser l'état de santé des personnages au moyen d'une barre de vie. Des dialogues simples fourniront également quelques indices utiles en cours de jeu.

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

3.1 Barre de vie

Fig. 4 : Barres de vie des personnages

Il s'agit maintenant de les doter `ICoopPlayer` d'une barre de vie visualisant leur « état de santé » (voir la figure 4). Une classe `Health` est fournie dans le répertoire `actor` pour vous simplifier la mise en oeuvre de cet aspect. Vous considérerez qu'un `ICoopPlayer` a un nombre de points de vie maximal (un entier identique pour tous les personnages et valant par exemple 5). À chaque `ICoopPlayer` sera associée une barre de vie initialisée par une tournure telle que :

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

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 point 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). La barre de vie doit évidemment être dessinée avec le personnage.

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 qu'un `ICoopPlayer` :

- puisse devenir invulnérable (temporairement ou pas) à un certain type de dommage (un seul à la fois et au départ il est vulnérable à tout) ;
- puisse perdre un nombre de points de vie donné lié à un type de dommage donné (par exemple l'explosif inflige un dommage physique causant la perte d'un nombre donné de points de vie) ;
- et qu'une fois qu'il a subi une perte de point de vie (quelle qu'en soit la cause), il bénéficie d'une petite période d'immunité dont la durée est identique pour tous les personnages.

La méthode permettant la perte des points liée à un type donné de dommage peut être implémentée selon l'algorithme suivant : s'il est invulnérable au dommage en question ou si le personnage est dans sa période d'immunité alors rien ne se passe. Sinon, sa barre de vie doit refléter la perte du nombre de points de vie voulu.

Compléter la classe `Explosive` de sorte à ce que l'interaction de l'explosif avec un `ICoopPlayer` lui inflige un dommage physique causant la perte d'un nombre fixe de points de vie. Ce nombre sera identique pour tous les explosifs (par exemple 2). La durée de la période d'immunité sera aussi considérée comme identique pour tous les personnages pour simplifier.

Indication : vous pouvez implémenter la période d'immunité en vous inspirant des modalités de mise en oeuvre du retardateur de l'explosif. Comme durée de la période d'immunité, vous pouvez prendre une valeur comme 24.

Retouche au dessin des personnages : Afin de permettre de visualiser l'état d'immunité, modifiez le dessin des `ICoopPlayer` 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, aussi bien pour le reset du jeu que celui de l'aire courante. Par simplification, vous considérerez aussi qu'ils repartent avec un temps d'immunité nul.

Fin de vie : adaptez votre jeu `ICoop` de sorte à ce que la perte de tous les points de vie de l'un ou l'autre des personnages provoque un « reset » de l'aire courante.

FIG. 5 : Dialogues d'aide.

3.1.1 Tâche

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

Lancez votre jeu `ICoop`. Vous vérifierez qu'il se comporte comme sur la petite vidéo suivante : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step2-1.mp4 ; c'est-à-dire concrètement :

1. qu'une barre de vie s'affiche à proximité de chaque personnage, qu'elle soit complètement verte au départ et qu'elle suive le personnage associé dans ses déplacements ;
2. que l'explosif une fois explosé fasse subir des pertes de points de vie au personnage qui est à proximité (par exemple si le personnage qui a activé l'explosif ne s'en est pas suffisamment éloigné) ;
3. et que le dessin du personnage 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 traduise par un « reset » de l'aire courante. La notion d'invulnérabilité à un type de dommage sera quant à elle testée un peu plus loin.

3.2 Dialogues

De nombreuses touches sont déjà impliquées dans les interactions et il est utile de pouvoir les documenter au lancement du jeu. Par ailleurs, ce type de jeu peut rapidement devenir injouable sans quelques indications dispensées 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 5.

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("orb_water_msg");
```

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 `ICoop` :

- 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 `ICoop` 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`).

3.2.1 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. En revanche, ce dialogue ne doit pas se ré-afficher si l'on utilise la touche `AREA_RESET`.

3.2.2 Tâche

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

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.

3.3 Objets à collecter

Lors de leurs déplacements, les personnages principaux pourront collecter des objets leur apportant des avantages (soins, invulnérabilité, etc), ou constituant des équipements utiles.

Il vous est demandé maintenant d'introduire une classe abstraite `ICoopCollectable` modélisant le comportement général par défaut des objets « ramassables » dans les jeux de type `ICoop`. Cette classe héritera de la classe `CollectableAreaEntity` fournie dans la maquette (paquetage `areagame.actor`). Par défaut, un `ICoopCollectable` est traversable et en tant `Interactable`, ne se prête à ce niveau d'abstraction qu'à des interactions de contact (cela peut être évidemment rediscuté dans les sous-classes). Un `ICoopCollectable` doit enfin disparaître de l'aire au moment de sa collecte.

3.3.1 Collecte d'explosifs

Il vous est demandé de modifier votre code, de sorte à ce que les personnages puissent en faire la collecte par interaction de contact.

Indication : Utilisez le paramètre booléen `isCellInteraction` pour différencier les interactions de contact et à distance dans les méthodes `interactWith` spécifiques aux acteurs.

3.3.2 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications données, lancez votre jeu `ICoop`. Vous vérifierez que les explosifs puissent désormais être collectés par interaction de contact avec les personnages mais seulement s'ils ne sont pas activés ou en train d'exploser (libre à vous d'en ajouter aux endroits de votre choix). Une fois collectés, ils disparaîssent alors simplement de l'aire. Vérifiez également que les interactions à distance avec les explosifs fonctionnent comme elles le faisaient auparavant.

3.3.3 Les `ElementalItem`

Vous introduirez ensuite une catégorie particulière de `ICoopCollectable`. Il s'agit des objets « ramassable » servant un élément naturel (`ElementalItem`). Ils se comporteront à la fois comme des `ElementalEntity` et comme des signaux logiques (`Logic`) et vous considérerez qu'ils ne sont pas instantiables à ce niveau d'abstraction.

L'aire d'appartenance d'un `ElementalItem`, sa position de départ dans cette dernière, son orientation et l'élément naturel qu'il sert sont donnés à la construction.

En tant que `Logic` un `ElementalItem` est « allumé » (`isOn()` retournant `true`) lorsqu'il est collecté et « éteint » (`isOff()` retournant `true`) sinon.

Pour pimenter un peu les interactions, il vous est demandé de garantir qu'un `ElementalItem` ne se laissera collecter que par une entité de la même catégorie élémentaire (un `Item` de feu ne se laissera collecter que par un acteur de feu par exemple).

3.3.4 Les orbes

Un premier type concret de `ElementalItem` est à implémenter à ce stade : une orbe modélisée par la classe `Orb`. Une orbe est un `ElementalItem` qui protège celui qui la ramasse contre certains types de dommages.

Une *orbe d'eau* donne protection contre les dommages d'eau et sa collecte doit s'accompagner de l'affichage du dialogue `orb_water_msg`.

Une *orbe de feu* donne protection contre les dommages de feu et sa collecte doit s'accompagner de l'affichage du dialgue `orb_fire_msg`. Son aire d'appartenance, l'élément qu'il sert (feu ou eau) et sa position de départ sont donnés comme paramètres à la construction. Elle est orientée par défaut vers le haut. Une orbe se dessine au moyen d'une animation. Référez-vous à l'annexe 7.3.3 pour des indications plus précises à ce sujet.

Lorsqu'un personnage entre en interaction de contact avec une orbe, un dialogue doit s'afficher : *`"orb_water_msg"`* pour les orbes d'eau et *`"orb_fire_msg"`* pour celles de feu. Une fois le dialogue terminé, l'orbe doit disparaître (avoir été collectée).

Indication : Un type énuméré `OrbType`, spécifiant l'élément servi, le dialogue à afficher et les `spriteYDelta` (tels qu'utilisés par le code de l'animation), peut être une bonne idée ici !

Question 2

Comment proposez vous de permettre à une orbe de communiquer au jeu le message à activer, sans recourir à des getters intrusifs ?

Pour tester ce nouveau type d'objets, vous ferez en sorte que l'aire `Orbway` soit dotée à sa création d'une orbe de feu en position `(17,12)` et d'une orbe d'eau en position `(17,6)`.

3.3.5 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications et contraintes données, lancez votre jeu `ICoop`. Vous vérifierez qu'il se comporte comme sur la petite vidéo suivante : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step2-3.mp4 ; c'est-à-dire concrètement :

1. que la salle `Orbway` apparaisse dotée d'une orbe de feu et d'une orbe d'eau aux positions voulues et toutes deux animées ;
2. que le personnage rouge puisse collecter l'orbe de feu et que le personnage bleu agisse de façon analogue avec l'orbe d'eau ;
3. que le dialogue spécifique à chaque type d'orbe s'affiche lors de l'interaction de contact avec les personnages ;
4. et que les orbes disparaissent à la fin de l'affichage de ces dialogues.

Le fait qu'une `ElementalItem` ne puisse être collecté que par une personnage servant le même élément sera testé un peu plus loin dans le projet.

3.4 Murs servant un élément

Un mur servant un élément (`ElementalWall`) est un mur qui se comporte comme un `ElementalEntity`. Il peut être actif ou pas (ce qui peut être modélisé au moyen d'un attribut de type `Logic`). Il peut être détruit et doit disparaître de l'aire s'il l'est.

Son aire d'appartenance, son orientation, sa position de départ, le signal logique qui conditionne le fait qu'il soit actif ou pas ainsi que le « sprite » permettant de le dessiner sont donnés à la construction. Cet acteur est traversable par défaut et ne se dessine que s'il est actif.

En tant que `Interactable`, un `ElementalWall` peut être l'objet d'interactions de contact et à distance. En tant que `Interactor` il fait systématiquement subir les interactions de contact mais pas celles à distance.

Lorsqu'il interagit avec un personnage `ICoopPlayer` et qu'il est actif, il lui fait subir des dommages. On considérera que la méthode faisant subir des dommages ne peut pas être définie à ce niveau d'abstraction.

L'ensemble des sprites permettant de le dessiner doit être extrait de la façon suivante :

```
Sprite[] wallSprites =  RPGSprite.extractSprites(spriteName,
   4, 1, 1, this, Vector.ZERO, 256, 256)
```

Pour le dessiner concrètement, il faudra utiliser celui de ces sprites qui correspond à son orientation :

```
wallSprites[getOrientation().ordinal()]
```

Deux types spécifiques de `ElementalWall` seront introduits :

- les murs de feu qui se dessinent au moyen de sprites *"fire_wall"* et qui font subir des dommages à `ICoopPlayer` en lui infligeant un dommage de feu lui faisant perdre un point de vie ;

- les murs d'eau qui se dessinent au moyen de sprites *"water_wall"* et qui font subir des dommages à `ICoopPlayer` en lui infligeant un dommage d'eau lui faisant perdre aussi un point de vie.

3.4.1 Rôle de la catégorie élémentaire

Pour complexifier les interactions, il vous est demandé de garantir que des entités qui ne sont pas de même catégorie élémentaire, ne peuvent pas cohabiter sur la même cellule de la grille. Ainsi, un mur d'eau ne laissera passer qu'un personnage d'eau et un mur de feu qu'un personnage de feu.

Question 3

Quelles modification proposez-vous d'apporter à `ICoopBehavior.ICoopCell` pour satisfaire cette contrainte ? (les tests sur la catégorie élémentaires ne sont pas considérés comme des tests de type car on ne teste pas la nature réélle des instance mais uniquement une de leurs facettes de comportement).

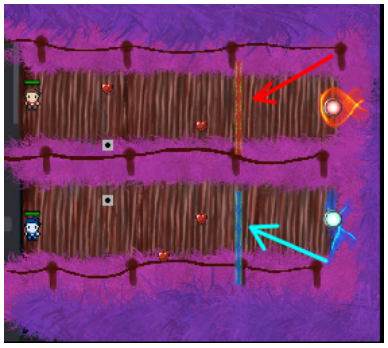

FIG. 6 : Les obstacles désignés par les flèches rouge et bleue sont constitués de 5 murs juxtaposés verticalement.

Pour tester ce nouveau type d'acteurs, vous ferez en sorte que `OrbWay` soit dotée à sa création de cinq murs de feu orientés à gauche en positions `(12, 10+i)` et cinq murs d'eau en positions `(12, 4+i)`, où `i` est le numéro du mur (avec `i` allant de 0 à 4) (voir la figure 6). À ce stade, ils seront actifs en permanence : le signal associé sera `Logic.TRUE`.

Vous ajouterez aussi à des fins de test un mur d'eau en position `(7,12)` et un mur de feu en position `(7,6)`. Ces deux murs seront associés à un signal toujours actif.

3.4.2 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications et contraintes données, lancez votre jeu `ICoop`. Vous vérifierez qu'il se comporte comme sur la petite vidéo suivante : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step2-4.mp4 ; c'est-à-dire concrètement :

1. que les déplacements sur les aires restent conformes à ce qu'ils étaient dans les étapes précédentes ;

2. que la salle `Orbway` apparaisse dotée des murs de feu et d'eau tels qu'ils apparaissent dans la vidéo ;
3. que les murs ne laissent passer que des personnages de leur catégorie élémentaire ;
4. et qu'ils font subir des dommages aux personnages lorsqu'ils les traversent.

3.5 Entraide, soins et périodes d'invulnérabilité

Le but est maintenant de doter les personnages de moyens de se prémunir des dommages causés par les murs. Pour cela, trois moyens leur seront offerts :

- la collecte des orbes va leur permettre de devenir invulnérables aux dommages ;
- les personnages pourront collecter des coeurs qui permettront de restaurer des points de vie ;
- chaque personnage pourra aider l'autre en désactivant son mur (nous commençons ici à mettre en place un mécanisme de coopération entre eux !).

Il vous est donc demandé pour commencer de compléter l'interaction entre `ICoopPlayer` et l'orbe de sorte à ce que la collecte de cette dernière le rende invulnérable au dommage dont elle assure la protection : l'orbe de feu protège des dommages de feu et celle d'eau agit de façon analogue.

Introduisez ensuite un nouvel objet « ramassable », l'acteur *coeur* (`Heart`) dont l'interaction de contact avec un `ICoopPlayer` pemettra à ce dernier regagner un point de vie (et au coeur d'être collecté). Un *coeur* ne sert aucun élément naturel (il n'y pas de coeur d'eau et de feu !).

Du point de vue graphique un coeur s'animera selon les indications de l'annexe 7.3.4.

Créez enfin un acteur *plaque de pression* (`PressurePlate`) qui se comporte comme un signal logique (`Logic`). Lorsqu'un personnage lui marche dessus il devient actif (`isOn()` retournant `true`).Vous disposez de la ressource graphique *`"GroundPlateOff"`* pour créer le `RPGSprite` permettant de le dessiner. Cet acteur est orienté par défaut vers le bas.

Pour tester vos nouveaux développements, vous ferez en sorte que `OrbWay` soit dotée à sa création de quatre coeurs aux positions de votre choix (par exemple en (`(8,4)`, `(10,6)`, `(5,13)` et `(10,11)`). Vous créerez aussi une première plaque de pression en position `(5,7)` et une seconde en position `(5,10)`. Enfin, vous modifierez la construction des murs de sorte à ce que le mur de feu se désactive lorsque la première plaque de pression est active et que le mur d'eau se désactive lorsque la seconde plaque est active. Ainsi, chaque personnage pourra aider l'autre à accéder à l'orbe sans subir les dommages des murs.

3.5.1 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications et contraintes données, lancez votre jeu `ICoop`. Vous vérifierez qu'il se comporte comme sur la petite vidéo suivante : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step2-5.mp4 ; c'est-à-dire concrètement :

1. que le personnage bleu puisse désactiver le mur de feu en marchant sur la plaque de pression qui lui est accessible ;
2. qu'un comportement analogue s'observe pour le personnage rouge ;
3. que la collecte des orbes puisse se faire sans dommage une fois les murs désactivés ;

4. et que les murs réapparaissent quand les personnages quittent la plaque de pression qui les conditionne.

3.6 Aire du labyrinthe

Pour conclure cette étape riche en nouveautés, il vous est demandé de coder une nouvelle aire nommée `Maze` (et ayant pour titre `"Maze"`). Vous y enregistrerez des acteurs conformément à la configuration de l'annexe 7.2. Cette salle peut être accédée depuis la salle `Spawn` au moyen d'une porte configurée comme suit :

destination	coord. arrivée	signal	cellule principale	autres coord.
"Maze"	(2,39)(3,39)	Logic.TRUE	(4,0)	(5,0)

L'accès inverse de `Maze` à `Spawn` n'est pas possible (aucune porte prévue : les personnages sont pour le moment piégés dans l'aire du labyrinthe ... ce qui devrait entretenir un certain suspens !). Les positions de « spawn » dans Maze sont `(2,39)` pour le personnage rouge et `(3,39)` pour le bleu.

3.6.1 Tâche

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

1. que les personnages puissent accéder à l'aire du labyrinthe depuis la porte de départ par la porte voulue et vice-versa (comme sur la vidéo https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step2-6.mp4) ;
2. que les différents composants apparaissent et se comportent selon la spécification l'annexe 7.2.

3.7 Validation de l'étape 2

Pour valider cette étape, toutes les vérifications des sections 3.1.1, 3.2.2, 3.3.2, 3.3.5, 3.4.2, 3.5.1 et 3.6.1 doivent avoir été effectuées.

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

4 Adversaires et batailles (étape 3)

Dans ce troisième volet du projet, il vous est demandé d'enrichir la conception en permettant l'ajout d'adversaires que les personnages auront à affronter ainsi que différents équipements plus « martiaux » pour mettre en scène des batailles épiques. Un concept rudimentaire d'inventaire sera introduit qui permettra aux personnages de stocker et de sélectionner les équipements à utiliser.

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

4.1 Équipements

Le personnage disposera d'équipements qu'il peut stocker dans un *inventaire* contenant divers *articles*.

La notion générique d'inventaire (`Inventory`) et celle d'article d'inventaire (`InventoryItem`) sont fournies dans la maquette (dans `areagame.handler` du module `game-engine`).

4.1.1 Articles d'inventaire

La spécification *fonctionnelle* d'un article d'inventaire (`InventoryItem` fourni dans `game-engine`) est qu'il est possible d'accéder à :

- son nom (un `String`) ;
- et à la poche de l'inventaire dans laquelle il se trouve (un `int` qui désigne le numéro de la poche pour simplifier).

Vous noterez qu'une poche peut contenir plusieurs articles d'inventaire, nous n'utiliserons d'ailleurs qu'une poche dans le projet pour simplifier.

Un `InventoryItem` doit être vu comme la description d'un *modèle* d'article que l'on peut stocker dans un inventaire (par exemple le modèle d'article « épée »), plutôt qu'un objet physique. Pour simplifier, nous parlerons cependant d'*articles* d'inventaire (plutôt que de *modèles d'articles* d'inventaire).

ICoopItem Votre jeu `ICoop`, disposera de sa propre implémentation concrète d'articles d'inventaires. Il vous est demandé d'implémenter le concept `ICoopItem`, une spécialisation de `InventoryItem` modélisant les articles d'inventaire spécifiques au jeu `ICoop`. Vous pouvez placer ce concept dans le répertoire `handler` du votre jeu.

Dans la version de base de `ICoop` qu'il vous est demandé de rendre, vous considérerez que les articles d'inventaires possibles dans le jeu sont : des épées (`Sword`), des clés de feu et d'eau (`FireKey`, `WaterKey`), des bâtons de feu et d'eau (`FireStaff` et `WaterStaff`) et des explosifs (`Explosive`).

Pour le moment, il n'est pas nécessaire d'anticiper des acteurs « épée », « clé » ou « bâton » ; il s'agit uniquement de représenter des modèles d'articles d'inventaires.

Indications :

- codez `ICoopItem` comme un type énuméré ;
- un type énuméré peut très bien implémenter une interface.

Vous pouvez choisir librement le nom associé à chacun des articles prévus. Vous anticiperez aussi que tout `ICoopItem` a une représentation graphique et est donc caractérisé par un `Sprite` (vous pouvez utiliser les images `sword.icon`, `key_blue`, `key_red`, `staff_water.icon`, `staff_fire.icon` et `explosive` du dossier `resources/images/sprites/`). Pour simplifier, dans la version de base du projet on considérera que l'inventaire n'a qu'une seule poche et que donc tous les articles appartiennent par défaut à cette poche d'identifiant zéro.

4.1.2 Inventaires

Un inventaire abstrait (classe `Inventory` fournie dans `game-engine`) est caractérisé par un ensemble de poches (`Pocket`) dans lesquelles il est possible d'ajouter ou de supprimer une quantité donnée de `InventoryItem`. Examinez dans les grandes lignes ces classes pour anticiper les fonctionnalités qu'elles vous offrent[3]. Les poches de l'inventaire sont numérotées (indices entiers). Notez que l'interface imbriquée `Inventory.Holder` permet de désigner fonctionnellement les détenteurs d'inventaire : on pourra les interroger pour savoir s'ils ont tel ou tel article dans leur inventaire au moyen de la méthode `possess`.

Vous coderez `ICoopInventory`, une spécialisation de `Inventory` spécifique au jeu `ICoop` qui se construit par défaut avec une poche unique portant le nom de votre choix. Vous pouvez aussi placer ce concept dans le répertoire `areagame.handler` du votre jeu.

Vous compléterez ensuite la modélisation du personnage principal `ICoopPlayer` de sorte à le doter d'un inventaire de type `ICoopInventory`. A la création du personnage, quelques articles, dont un ou plusieurs explosifs et épées, seront placés « en dur » dans son inventaire pour simplifier[4]. `ICoopPlayer` devra naturellement être répertorié comme propriétaire d'inventaire. Pour préserver l'encapsulation, il ne fournira pas son inventaire au monde extérieur.

Vous considérerez que `ICoopPlayer` ne peut utiliser plus d'un article d'inventaire à la fois. L'article en cours d'utilisation sera référencé comme étant l'*équipement courant*. Vous doterez `ICoopPlayer` de nouvelles fonctionnalités permettant :

- de passer d'un article de l'inventaire à l'autre (c'est à dire de changer son équipement courant) au moyen de la touche `keyBindings.switchItem()` ; passer d'un article à l'autre se fera de façon circulaire (après le dernier de la liste on revient au premier) ; vous noterez que la classe `Inventory` offre la méthode `contains(InventoryItem item)` permettant de savoir si l'un des poches de l'inventaire contient l'article `item` ;
- et d'invoquer l'utilisation de l'équipement courant au moyen de la touche `keyBindings.useItem()`.

Indications : pour que le jeu soit plus facilement jouable, il est préférable que les articles d'inventaires apparaissent toujours dans le même ordre lorque lorsque l'on transite d'un article à l'autre. Un attribut qui donne cet ordre peut être utile. On n'affichera cependant dans cet ordre que les articles que le personnage possède effectivement dans son inventaire.

[3] l'interface imbriquée GUI n'est pas utile dans la version de base du projet

[4] vous pourrez améliorer ce point dans la partie extensions du projet

FIG. 7 : Zones d'affichage graphique du statut des personnages (une zone pour chaque personnage)

L'utilisation de l'équipement courant se fera bien entendu au cas par cas[5] : chaque équipement est un cas particulier à gérer de façon spécifique.

Pour le moment vous ne programmerez que l'utilisation de l'explosif (tous les autres cas seront considérés comme des cas par défaut où le personnage ne fait rien).

Utiliser une équipement de type « explosif » revient à créer un acteur « explosif » en face du personnage. Cette création ne sera possible que si la case en face du personnage permet le placement d'un acteur. Une bombe utilisée doit être retirée de l'inventaire (la quantité de cet article dans l'inventaire sera diminuée d'une unité).

L'explosif sera doté d'interactions supplémentaires :

- avec d'autres explosifs en les faisant exploser ;
- avec les murs en les détruisant.

4.1.3 Collecte et inventaire

Lors de l'étape précédente, trois types d'acteurs « ramassables » ont été introduits, les explosifs, les orbes et les coeurs. Une nuance doit désormais être établie entre eux : certains objets collectés peuvent être stockés dans l'inventaire lors de la collecte (feront partie des équipements) et d'autres non. Typiquement, on considérera que les orbes et les coeurs, une fois collectés, n'apparaîtront pas dans l'inventaire alors que les explosifs oui. Il vous est demandé d'adapter votre code pour que ce soit le cas.

4.1.4 Graphismes

Afin de pouvoir visualiser des informations spécifiques au personnage (comme l'équipement courant de son inventaire), il vous est demandé maintenant d'utiliser et compléter la classe

[5] les types énumérés se prêtent fort bien à l'usage du switch

fournie `ICoopPlayerStatusGUI`. Cette classe modélise un « objet graphique décrivant le statut d'un personnage principal ». La coquille fournie permet de dessiner le petit cercle brun servant de support aux articles d'inventaire dans la figure 7 (objet graphique `gearDisplay`) ; elle peut être enrichie à votre guise pour ajouter d'autres informations. Complétez la méthode de dessin existante de cette classe pour faire afficher dans le cercle brun le sprite de l'article d'inventaire courant de chaque personnage. Il s'agira concrètement de construire et dessiner une image :

```
new ImageGraphics(ResourcePath.getSprite(sprite_name), 0.5f,
    0.5f, new RegionOfInterest(0, 0, 16, 16), anchor.add(new
    Vector(0.5f, height - 1.25f)), 1, DEPTH);
```

où `sprite_name` est le nom du sprite associé au `ICoopItem` sélectionné par le personnage concerné.

Note : Si le paramètre `flipped` d'un `ICoopPlayerStatusGUI` est `true`, sa méthode `draw` l'affichera à droite, sinon à gauche.

4.1.5 Tâche

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

1. qu'il soit bien possible de passer d'un équipement à l'autre au moyen des touches spécifiques à chaque personnage (par défaut `Q` et `U`) et que l'équipement courant s'affiche correctement ;
2. que seuls les équipements explicitement ajoutés à l'inventaire apparaissent lorsque on passe d'un équipement à l'autre au moyen de la touche `Tab` ;
3. que seul l'équipement courant soit utilisable (à vérifier avec l'explosif) ; au moyen des touches spécifiques à chaque personnage (par défaut `E` et `O`) ;
4. que l'explosif, en explosant, fasse exploser un autre explosif à proximité et détruise les murs ;
5. qu'il ne soit plus sélectionnable comme équipement courant s'il n'y en avait qu'un et que le personnage l'ait utilisé ;
6. que si l'inventaire est vide d'explosif pour un personnage et qu'il en collecte un, cet article réapparaisse dans l'inventaire ;
7. et que la description graphique du statut du personnage continue à se faire proprement lorsque l'on passe d'une aire à l'autre.

Dans la partie obligatoire du projet, il n'est pas attendu à ce que le « reset » de l'aire courante restaure l'état de l'inventaire pour qu'il soit tel que lorsque l'aire a été abordée la première fois. Par exemple, si on collecte un objet dans l'inventaire et que l'on « reset » ensuite l'aire, il n'est pas attendu de défaire la collecte précédente. Vous pourrez vous preoccuper de cette question dans les extensions si vous le souhaitez.

4.2 Ennemis

Les ennemis (`Foe`) 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 ; ils disparaîssent alors de l'aire de jeu après avoir été l'objet d'un affichage particulier. Il s'agira d'une animation qui les montre en train de mourir et qui sera la même pour tout type d'ennemis (*`"icoop/vanish"`*) ;
- qui peuvent être sensibles à un certain nombre de types de dommages (ils ont une « liste de vulnérabilités ») ;
- 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 ;
- qui comme les personnages peuvent perdre des points de vie lié à un type de dommages donné ;
- et qu'une fois qu'ils ont subi une perte de point de vie (quelle qu'en soit la cause), bénéficient d'une petite période d'immunité dont la durée est identique pour tous les ennemis.

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 vulnérabilités possibles seront les mêmes que pour les personnages (vulnérabilités aux dommages physiques, de feu ou d'eau).

Chaque type d'ennemi aura une liste de vulnérabilités spécifiques, définie à la construction (par exemple les crânes lanceurs de feu seront sensibles à l'eau et aux dommages physiques mais pas au feu). Un ennemi naît avec le nombre maximal de point de vie.

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 7.3.8).

Toutes sortes de catégories d'ennemis sont envisageables. Nous vous demandons de coder les deux déclinaisons dont les spécifications sont décrites ci-dessous ; à savoir des « crânes lanceurs de feu », et des « artificiers » (poseurs d'explosifs) ... tout un programme :-).

Indications pour cette partie :

- l'usage de types énumérés combinés à l'instruction `switch` est une option naturelle pour coder le fait que l'évolution (`update`) d'un acteur dépende de son état (comme ce sera le cas pour la plupart des ennemis) ;
- vous veillerez à ce que chaque classe contienne ce qui lui est spécifique et que les super-classes n'aient pas connaissance de leurs sous-classes ;
- le code des méthodes `interactWith` se résume pour les cas suivants à quelques lignes. Si le code devient trop verbeux chez vous, n'hésitez pas à nous solliciter car cela peut refléter une mauvaise compréhension des modalités de mise en place des interactions.

Comme certains ennemis vont lancer des projectiles (par exemple les crânes lanceurs de feu de la figure 8), il vous est d'abord demandé de modéliser ce concept.

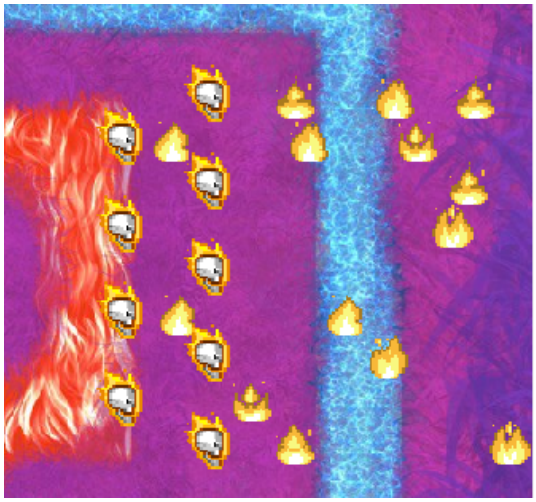

FIG. 8 : crânes lanceurs de feu

4.2.1 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. Ces deux caractéristiques sont fixées à la construction. 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.

A ce niveau d'abstraction, les interactions avec les différents acteurs concrets ne peuvent cependant encore être définies.

Comme première déclinaison concrètes de projectiles, vous coderez des flammes.

Indication : vous pouvez introduire une catégorie `Unstoppable` modélisant les entités capables de survoler des cellules (notamment celles sur lesquelles on ne peut habituellement pas marcher). Réfléchissez ensuite à la modification à apporter `ICoopBehaviour` et `ICoopCell` de sorte à ce que un `Unstoppable` puisse investir n'importe quel type de cellule.

Flammes Une flamme (`Fire`) est un projectile qui disparaît lorsqu'il est stoppé dans sa course. Elle se dessine au moyen d'une animation (voir 7.3.6).

Par interaction de contact :

- elle essaie de causer des points de dommages de feu aux ennemis quels qu'ils soient et aux personnages `ICoopPlayer` (1 point par exemple mais libre à vous de prendre d'autres valeurs) ; il s'agit ici d'essayer car ces dommages ne s'infligeront pas si l'entité ciblée est invulnérable aux dommages de feu ;
- elle fait exploser les explosifs.

4.2.2 Crânes lanceurs de feu

Un crâne lanceur de feu (`HellSkull`) est un ennemi vulnérable aux dommages physiques et à l'eau. Son nombre de points de vie maximal est une constante identique pour tous les ennemis de ce type (par exemple `1` si vous voulez vous simplifier les combats futurs ;-), mais libre à vous !). On peut lui marcher dessus. Il impose des interactions de contact (s'il n'est pas mort) mais pas d'interaction à distance. La représentation graphique d'un `HellSkull` peut se faire au moyen d'une animation (voir 7.3.9).

Le `HellSkull` peut lancer des « flammes » (voir figure 8).

Comportement général à des intervalles de temps `t`, le `HellSkull` génère une flamme qu'il place en face de lui (si l'emplacement est accessible). L'intervalle `t` est re-généré aléatoirement après chaque lancer. Il est tiré au hasard entre deux bornes dont vous pouvez choisir les valeurs (par exemple `0.5f` et `2.f`).

Pour tirer au hasard un float compris entre deux bornes `MIN` et `MAX`, utilisez l'instruction :

```
float randomFloat =
   RandomGenerator.getInstance().nextFloat(MIN, MAX);
```

Interactions Lors d'interactions de contact, le `HellSkull` inflige des dommages de feu causant la perte d'une nombre donné de points de vie aux personnages principaux. Le nombre de points de dommage infligé est une constante identique pour tous les `HellSkull`, et par simplification aussi identique quelle que soit l'entité en interaction.

Pour tester ces développements, faites en sorte que l'aire `Maze` soit dotée de 10 `HellSkull` orientés à droite et placées aux coordoonnées `(12,33)`, `(12,31)`, `(12,29)`, `(12,27)`, `(12,25)`, `(10,33)`, `(10,32)`, `(10,30)`, `(10,28)` et `(10,26)`.

Quelques astuces peuvent être utiles au debugging (notamment dans l'aire `Maze`) :

- il est possible de dé-zoomer en jouant sur le `DEFAULT_SCALE_FACTOR` dans `ICoopArea` (essayez la valeur de 39.f au lieu de 13.f par exemple) ;
- de placer artificiellement le centre de masse sur une zone à observer (par exemple pour debugger les `BombFoe` il peut être utile de changer momentanément les positions de « spawn » des personnages en les faisant apparaître en `(3,10)` et `(4,10)`) ;
- en commentant la gestion de la fin de vie des personnages pour observer les comportements des ennemis plus tranquillement ;-).

FIG. 9 : Les artificiers (ennemis poseurs d'explosifs)

4.2.3 Tâche

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

1. que les 10 crânes de feux apparaîssent aux positions voulues ;
2. qu'ils lancent tous des flammes avec des laps de temps aléatoires ;
3. que les flammes se déplacent en face des crânes qui les lancent ; qu'elles survolent toutes les cellules y compris celles sur lesquelles on ne peut pas marcher, qu'elles s'animent et qu'elles disparaissent lorsqu'elles atteignent les limites de l'aire ;
4. que les flammes lancées par le crâne en haut à gauche ne causent pas de dommage au crâne aligné à sa droite (celui-ci ne disparaît jamais) ;
5. qu'en explosant, les explosifs ne causent pas de dommages aux crânes lanceurs de feu ;
6. que les crânes lanceurs de feu font subir des dommages aux personnages par contact direct ;
7. que les flammes font aussi subir des dommages aux personnages ;
8. et que les projectiles lancés détruisent ce qui se trouve sur leur passage tant qu'ils n'ont pas arrêté leur course et qu'ils disparaissent d'eux-mêmes une fois la course terminée.

4.2.4 Artificier

L'artificier (`BombFoe`, voir figure 9) est un ennemi vulnérable aux dommages physiques et aux dommages par le feu. Le nombre de points de vie maximal est une valeur commune à toutes ses instances (2 par exemple). Il est créé orienté vers le bas.

Il peut être dans différents états qui vont conditionner son comportement. Par défaut, il est dans l'état « inoccupé » (« idle »). Dans cet état, l'artificier ne fait rien d'autre que se déplacer. Autrement il peut être en train d'attaquer ou de se protéger.

La représentation graphique de l'artificier se fera au moyen d'animations spécifiques : *`"icoop/bombMonster"`* dans le cas général et *`"icoop/bombMonster.protecting"`* lorsqu'il est en train de se protéger.

Interactions L'artificier est « demandeur » d'interaction à distance mais pas d'interaction de contact. Il ne peut interagir que s'il est inoccupé ou en train d'attaquer. Lorsqu'il est en train d'attaquer, son champ d'interaction (de perception) est l'unique cellule en face de lui. Sinon, son champ de perception (élargi) est un ensemble constant de cellules en face de lui (le même nombre pour tous les artificiers, par exemple 8). Il se dessine au moyen d'une animation (voir 7.3.10).

L'artificier interagit avec les personnages principaux. L'interaction consiste à basculer en mode « attaque » lorsqu'un de ces personnages est dans son champ d'interaction. Pour le moment, vous ne vous préoccuperez pas de comment les personnages principaux combattent leurs ennemis.

Comportement général L'artificier a des moments d'inaction durant lesquels il ne fait absolument rien (même pas se déplacer), peu importe son état. Ce temps d'inaction est nul à la création de l'artificier et sera codé comme un nombre de pas de simulation. Il ne peut dépasser une certaine valeur (par exemple `24`, commune à tous les ennemis de ce type).

Le comportement de l'artificier est le suivant lorsque son temps d'inaction est écoulé et qu'il ne bénéficie pas d'une période d'immunité :

- s'il est dans l'état inoccupé (« idle ») et qu'aucun déplacement n'est en cours, il se déplace aléatoirement (voir plus bas) puis peut entamer un moment d'inaction dont la durée est tirée aléatoirement entre zéro et le temps d'inaction maximal ;
- s'il est en mode « attaque », il se déplace en mode ciblé vers le personnage qu'il a mémorisé comme cible jusqu'à être à une distance suffisamment proche, par exemple une distance 3[6]. Plus de détails sont donnés ci-dessous. Il largue alors un explosif dans la case en face de lui, si elle est libre et bascule dans l'état « protection » ;
- dans l'état « protection », il se déplace plus lentement (libre à vous de le mettre en oeuvre comme vous le souhaitez) et oublie sa cible. Il reste un certain temps dans cet état (une durée aléatoire tirée entre deux constantes caractéristiques de la classe, par exemple 72 et 120), puis rebascule dans l'état « idle ».

Lorsqu'il est en période d'immunité, il bascule en mode inoccupé avec un temps d'inaction nul.

Déplacement aléatoire Le déplacement aléatoire de l'artificier se fait un peu comme celui d'un personnage principal. La différence réside dans le fait que ce déplacement n'est pas contrôlé via le clavier et que le changement d'orientation se fait aléatoirement. La probabilité de changement d'orientation sera tirée au hasard et s'il y a lieu, la nouvelle orientation sera aussi choisie au hasard parmi les 4 possibles. Par exemple l'artificier aura 40% de chances de changer d'orientation et s'il le fait, il choisira au hasard et de manière équiprobable la direction à prendre. Pour tirer au hasard un entier compris entre 0 et MAX, utilisez l'instruction :

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

Pour adopter le comportement voulu dans 40% des cas par exemple vous pouvez tirer un double au hasard :

[6] La distance peut être calculée au moyen de la méthode `distanceBetween` des `DiscreteCoordinates`

```
double randDouble = RandomGenerator.getInstance().nextDouble();
```

et adopter le comportement voulu si le nombre tiré est inférieur à 0.4.

Indication : l'appel à la méthode `move` peut être paramétré par une valeur `ANIMATION_DURATION / speedFactor`. `speedFactor` peut varier en fonction de l'état de l'artificier (par exemple valoir 2 pour une vitesse d'animation normale, 6 pour rapide et 1 pour lent). Le pas de déplacement aléatoire se fera à une vitesse d'animation normale.

Déplacement ciblé L'algorithme simple suivant sera utilisé pour choisir la direction :

i. calculer `v`, le vecteur séparant l'artificier de sa cible (méthode `sub` des `Vector`), `deltaX` la composante en `x` de ce vecteur et `deltaY` sa composante en `y` ;
ii. si la valeur absolue de `deltaX` est plus grande que celle de `deltaY` (l'artificier est plus loin horizontalement que verticalement de sa cible), orienter l'artificier selon le vecteur `(deltaX, 0)` sinon selon le vecteur `(0, deltaY)`. La méthode `Orientation.fromVector` peut être utilisée pour convertir un `Vector` en `Orientation` ;
iii. si le changement d'orientation n'a pas pu se faire, un pas de déplacement à vitesse rapide aura lieu.

Pour tester ces développements, faites en sorte que l'aire `Maze` soit dotée de 4 `BombFoe` placés aux coordoonnées `(5,15)`, `(6,17)`, `(10,17)` et `(5,14)`.

4.2.5 Tâche

Une fois les concepts décrits précédemment codés conformément aux spécifications et contraintes données, lancez votre jeu `ICoop`. Vous vérifierez qu'il se comporte comme sur la petite vidéo suivante : https ://proginsc.epfl.ch/wwwhiver/mini-projet2/videos/icoop/step3-3.mp4 ; c'est-à-dire concrètement :

1. le `BombFoe` se déplace seul en changeant aléatoirement de direction dans les mêmes limites que les personnages ;
2. qu'il puisse larguer des bombes aléatoirement lorsqu'un personnage est dans son champ de perception élargi ;
3. qu'il se mette en mode protection avec un graphisme spécifique lorsqu'il a réussi à larguer une bombe ;
4. et qu'il observe aléatoirement des moments d'inaction.

4.3 Batailles

Il est temps de permettre aux personnages de se défendre un peu mieux contre leurs ennemis. Vous allez commencer par l'équiper en le dotant d'une nouvelle arme : un bâton capable de lancer des boules d'eau ou de feu. Les bâtons rouges lanceront des boules de feu et les bâtons bleus des boules d'eau.

FIG. 10 : Etats « attaque » pour le personnage avec visuels spécifiques : à gauche pour l'attaque avec l'épée et à droite pour celle avec le bâton

4.3.1 Bâtons et boules de feu et d'eau

Un bâton (`Staff`) est un `ElementalItem` se dessinant au moyen d'une animation (voir 7.3.5).

Les boules d'eau et de feu se codent de façon analogue aux flammes. Elles se dessineront aussi au moyen d'une animation (voir 7.3.7). Les boules de feu ou d'eau causeront 1 point de dommage de type dégât d'eau ou de feu aux ennemis, ils feront exploser les bombes et détruiront les rochers. Elles seront stoppées dans leur course en les touchant. Enfin, les bâtons une fois collectés par les personnages devront intégrer l'inventaire.

Vous créerez un bâton rouge en position `(13,2)` de `Maze` et un bâton bleu en position `(8,2)`.

4.3.2 Tâches

Vérifiez que les bâtons apparaîssent bien dans `Maze`, qu'ils peuvent être collectés mais uniquement par les personnage de leur couleur et qu'ils apparaissent alors dans l'inventaire. Le lancé de boules d'eau et de feu sera testé un peu plus bas.

4.3.3 Utilisation des armes par les personnages

La méthode qui gère l'utilisation des équipements de `ICoopPlayer` ne se charge pour le moment que de la bombe. Il vous est demandé de l'étendre pour prendre en compte la gestion de l'épée et du bâton.

Il faut pour cela modéliser le fait que le personnage lui aussi peut être dans différents états qui vont conditionner son comportement (par exemple, s'il est dans l'état « attaquant avec l'épée », alors il pourra attaquer un ennemi).

Les états à anticiper sont, outre l'état « inoccupé/idle » similaire à ce qui a été décrit plus haut pour certains ennemis, ceux le modélisant en train d'attaquer : « attaquant avec son épée » et « attaquant avec son bâton ».

La transition d'un état à l'autre est conditionnée par l'appel à la méthode qui gère l'utilisation des équipements (invocable par les contrôles prévus) et bien sûr par l'équipement

sélectionné. Par exemple, le personnage transite à l'état « attaquant avec son épée », lorsque le joueur a sélectionné l'épée comme équipement courant et qu'il a manifesté le souhait d'utiliser cet équipement en appuyant sur la touche prévue (par défaut, `E` et `O`). Le passage à un des modes d'attaque ne doit pouvoir se faire que si le personnage est en mode inoccupé (il ne peut pas passer directement du mode « attaquant avec son épée » au mode « attaquant avec son bâton » par exemple). Ce passage doit se traduire par un visuel spécifique (voir la figure 10 ainsi que l'annexe 7.3.1).

Le retour à l'état inoccupé doit être fait après que les animations liées au passage à un nouvel état soient terminées et les attaques menées à bien :

- Une attaque à l'épée se termine lorsque l'utilisateur cesse d'utiliser la touche correspondant à l'utilisation de l'équipement. Le personnage retourne à l'état inoccupé dès que l'interaction demandée s'est terminée ; l'interaction comprenant le temps d'animation de l'attaque à l'épée.
- une attaque au bâton se termine de façon analogue. Vous ferez en sorte qu'avant de transiter à l'état inoccupé, une boule d'eau ou de feu soit alors lancée par le personnage, selon sa couleur.

Il vous est demandé de modifier la méthode de mise à jour de `ICoopPlayer` de sorte à intégrer la transition vers de nouveaux états (et conformément aux descriptions ci-dessus). Le `ICoopPlayer` ne se déplacera comme il le faisait auparavant que dans l'état inoccupé.

Vous modifierez aussi le code pour :

- que le personnage puisse avoir une interaction à distance pendant tout le temps où il utilise son épée (tant qu'il est dans l'état « attaquant avec l'épée » il peut interagir à distance) ;
- qu'en attaquant avec l'épée, le personnage puisse causer des dommages physiques aux ennemis ;
- que le personnage puisse lancer des boules d'eau ou de feu en attaquant avec son bâton ;
- que les boules d'eau/de feu causent des dommages d'eau/feu aux ennemis et détruisent les rochers.

4.3.4 Tâches

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

1. que les personnage principaux continuent à pouvoir se déplacer sous contrôle du clavier lorsqu'ils sont inoccupés ;
2. qu'ils puissent transiter à la demande dans différents modes d'attaque après sélection des équipements, bâton ou épée ;
3. que l'utilisation des équipements choisis se fasse via les touches anticipées et que cela se traduise par le visuel approprié ;
4. qu'il puisse lancer des projectiles d'eau en utilisant le bâton et que ces derniers sont stoppés dans leur course s'ils rencontrent des ennemis ou des rochers ;
5. qu'il puisse détruire les ennemis en leur infligeant des dommages physiques (en les frappant de son épée) ;
6. qu'il puisse infliger des dommages physiques aux crânes de feu et aux artificiers (épée) ;

7. qu'il puisse infliger des dommages d'eau (bâton) aux ennemis ;
8. et que les ennemis disparaissent dans un petit nuage en mourant.

4.4 Validation de l'étape 3

Pour valider cette étape, toutes les vérifications des sections 4.1.5, 4.2.3, 4.2.5, 4.3.2 et 4.3.4 doivent avoir été effectuées.

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

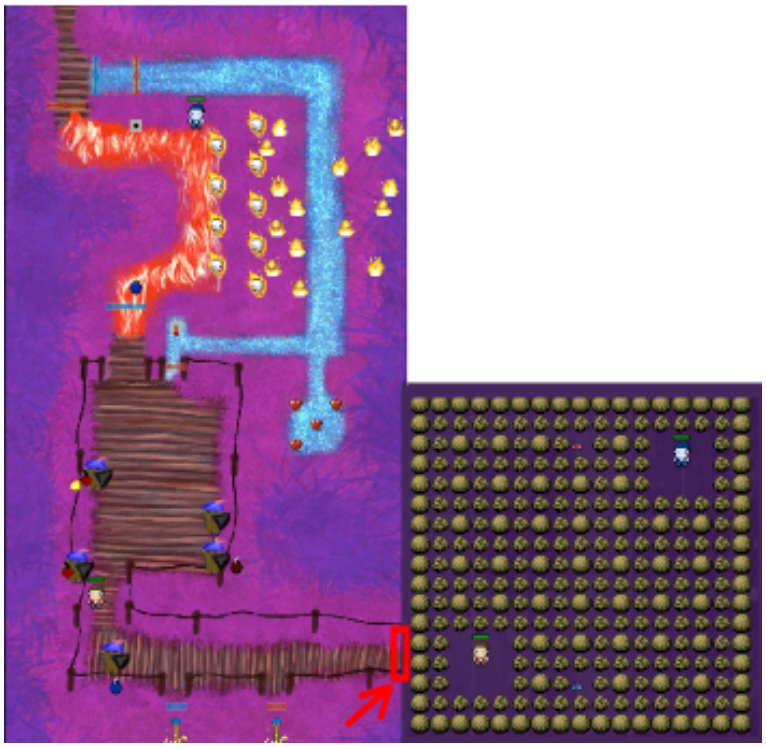

FIG. 11 : Passage de `Maze` (à gauche) à `Arena` (à droite) : l'aire `Arena` ne permet pas de retour en arrière vers `Maze`, elle est parsemée d'obstacles non franchissables et de rochers destructibles.

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 qu'en sortant de l'aire `Maze` par la porte de la figure 11, les personnages se trouvent prisonniers d'une aire sans porte, l'aire `Arena`. Ils devront y détruire des rochers pour collecter les clés de leur couleur. Lorsque chacun aura ramassé la clé de sa couleur (catégorie élémentaire), un téléporteur apparaîtra qui leur permettra de rejoindre la salle initiale `Spawn`.

S'ils ont préalablement collecté les bâtons de leur couleur dans l'aire précédente, ils pourront « ouvrir » la porte du manoir et gagner la récompense finale.

La mise en oeuvre de cette étape devra s'appuyer sur la notion de signal logique (`Logic`) : par exemple la collecte des deux clés peut être modélisée comme un objet « et » logique (`AND` dans le paquetage `logic` de `game-engine`) sur les deux clés qui sont elles-mêmes des `Logic`.

5.1 Clés et téléporteurs

Une *clé* est un `ElementalItem` se dessinant au moyen du sprite :

```
new Sprite("icoop/key_red", 0.6f, 0.6f, this);
```

s'il sert l'élément de feu ou

```
new Sprite("icoop/key_blue", 0.6f, 0.6f, this);
```

s'il sert celui d'eau.

Un téléporteur n'est autre qu'une porte dessinable qui n'occupe comme position que sa cellule principale (telle que retournée par `getCurrentMainCellCoordinates()`). Il ne se dessine que s'il est actif. Comme « sprite » permettant de le dessiner, vous pouvez utiliser :

```
RPGSprite("shadow", 1, 1, this, new RegionOfInterest(0, 0, 32,
   32))
```

5.2 Aire `Arena`

L'aire `Arena` est accessible depuis `Maze` via une porte à la spécification suivante :

destination	coord. arrivée	Signal	cellule principale	autres coord.
"Arena"	(4,5),(14,15)	Logic.TRUE	(19,6)	(19,7)

À sa création, y sont enregistrées une clé rouge en position `(9,16)`, une clé bleue en position `(9,4)` et un portail (fermé, donc invisible) en position (10,10). Le portail assurera la transition vers l'aire `Spawn` (les personnages y arriveront à des positions de votre choix).

De nombreux rochers et obstacles parsèment l'aire `Arena` et il serait fastidieux de tous les enregistrer manuellement à la création de l'aire. Il vous est donc demandé de modifier le code de sorte à ce que la grille puisse dicter à l'aire la création d'un acteur `Rock` partout où il y a une cellule de type `ROCK` et la création d'un acteur `Obstacle` partout où il y a une cellule de type `OBSTACLE`.

Vous veillerez à coder ceci sans getter intrusif !

5.3 Porte du manoir

Nous vous proposons de finaliser la partie obligatoire du projet en associant à l'emplacement de la porte du manoir dans l'aire `Spawn`, un nouveau type de porte qui cause l'ouverture d'un dialogue lorsque les personnages essayeront d'interagir par contact avec. Il s'agira du dialogue *`"key_required"`* si la porte est fermée et *`"victory"`* si elle est ouverte. La figure 12 vous montre le premier type de dialogue (correspondant à une porte encore fermée).

La porte occupera la position `(6,11)` pour coincider avec les graphismes. Comme vous vous arrêterez à ce stade pour le développement du jeu en lui même, il n'est pas prévu que la

FIG. 12 : Porte du manoir : le but du jeu, dans sa partie obligatoire, est de réussir à l'ouvrir !

porte fasse transiter les personnages vers une nouvelle aire (ce que vous pourrez corriger dans les extensions si vous le souhaitez). Vous pouvez donc choisir comme salle de destination la salle `Spawn` elle-même.

Les personnages auront donc réussi leur mission lorsqu'ils auront pu ouvrir la porte. Il vous est pour cela demandé d'étendre votre code pour modéliser le fait que :

- les aires aient un challenge spécifique à réussir. Pour l'aire `Maze` le challenge à réussir est de collecter les deux bâtons. Pour l'aire `Arena` il est de collecter les deux clés et pour l'aire `Spawn` il consiste à réussir à ouvrir la porte du manoir. Pour l'aire `OrbWay`, libre à vous (et on peut considérer qu'une aire a un challenge toujours réussi).
- le comportement des acteurs d'une aire puisse être conditionné par des signaux externes. Par exemple, la porte du manoir a son ouverture conditionnée par la réussite du challenge des aires `Arena` et `Maze`.

Indication : les aires peuvent se comporter comme des signaux (`Logic`)!

Vous veillerez à coder ceci :

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

5.4 Tâche

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

1. que la porte du manoir soit initialement fermée dans l'aire `Spawn` ; le dialogue indiquant que la porte est fermée doit s'afficher lorsque les personnages essaient de la passer ;
2. que la nouvelle porte prévue dans `Maze` permette de transiter vers `Arena` ;

3. qu'une fois qu'ils l'ont franchie, les deux personnages y soient au départ piégés ;
4. que l'aire `Arena` se présente comme sur la figure 11 ;
5. que les personnages puissent y détruire les rochers (au moyen du bâton) mais pas les obstacles ;
6. que chacun d'eux puisse récupérer la clé de sa couleur une fois qu'il y a accédé ;
7. que la collecte des deux clés fasse apparaître un portail et que dès que l'un des personnages le franchit (par contact), les deux personnages soient libérés de l'arène et téléportés dans l'aire `Spawn` ;
8. et que la porte du manoir y soit ouverte si les personnages ont aussi collecté les bâtons dans `Maze`.

Une petite difficulté est à anticiper ici concernant le dialogue associé à la porte du manoir : l'apparition du dialogue se fait lors d'une interaction de contact et il n'est pas possible de déplacer le personnage tant que le dialogue n'est pas entièrement déroulé. Or, au moment où le dialogue se termine, le personnage sera encore en interaction de contact avec la porte et le dialogue recommencera en boucle.

Question 4

Comment proposez vous de résoudre ce petit problème sans introduire dans le jeu des tests spécifiques sur la nature du dialogue courant ?

5.5 Validation de l'étape 4

Pour valider cette étape, toutes les vérifications de la section 5.4 doivent avoir été effectuées.

Le jeu `ICoop` 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. Notez que du point de vue graphiques, deux salles additionnelles sont fournies pour vous permettre de coder des extensions (`SanctumEntrance` et `Sanctum`). 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, `ICoop`, mais il est alors **impératif de préserver les fonctionnalités obligatoires et la testabilité des composants demandés**.

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

Vous prendrez soin de **commenter soigneusement** dans votre fichier `HELP`, 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 `HELP` (partiel) correpondant qui explique comment jouer :

- vidéo d'exemple de jeu : ICoop.mov

Un ébauche de fichier `HELP.md` correspondant à un jeu des années antérieures vous est donné en guise d'exemple : HELP.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.

- 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)
- ajouter un objet `Box` ou `Safe`, dont l'ouverture serait dirigée par un signal aurait pour contenu un ou plusieurs objets ; (~3pts)
- etc.

6.2 Pause, fin de jeu et « reset »(~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).

La « reset » d'une aire, tel que codé actuellement, ne permet pas de revenir dans l'aire dans l'exact état où on l'a abordée la première fois, en particulier pour ce qui touche à le gestion de l'inventaire. Il est possible, comme extension, de garder une trace de cet état pour le

restaurer de façon appropriée.

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 `HELP` 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 ».[7]

Si vous souhaitez concourir, vous devrez nous envoyer d'ici au **19.12 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).

[7] 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 soient définies dans les variables `RED_PLAYER_KEY_BINDINGS` et `BLUE_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 `ICoopPlayer` 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.useItem()` (et de façon analogue, `keys.right()`, `keys.down()`, etc) pour référencer une touche donnée : celle associée à l'utilisation d'équipement 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.useItem()).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 Configuration de base de l'aire du labyrinthe

Selon la figure 13, la configuration de base suggérée pour l'aire Maze est la suivante :

- Coordonnées de départ des personnages dans l'aire : `(2, 39)`, `(3, 39)`.
- Deux murs d'eau orientés à gauche, en positions `(4, 35)` et `(4, 36)`, toujours actifs.
- Deux murs de feu orientés à gauche en positions `(6, 35)`, et `(6, 36)` désactivés lorsque la plaque de pression 4 est active (lorsqu'un personnage marche dessus).
- Deux murs de feu orientés en bas, en positions `(2, 34)` et `(3, 34)`, toujours actifs.
- Plaque de pression en position `(6,33)`.
- Explosif en position `(6, 25)`.
- Deux murs d'eau orientés vers le bas en positions `(5,24)` et `(6,24)` toujours actifs.

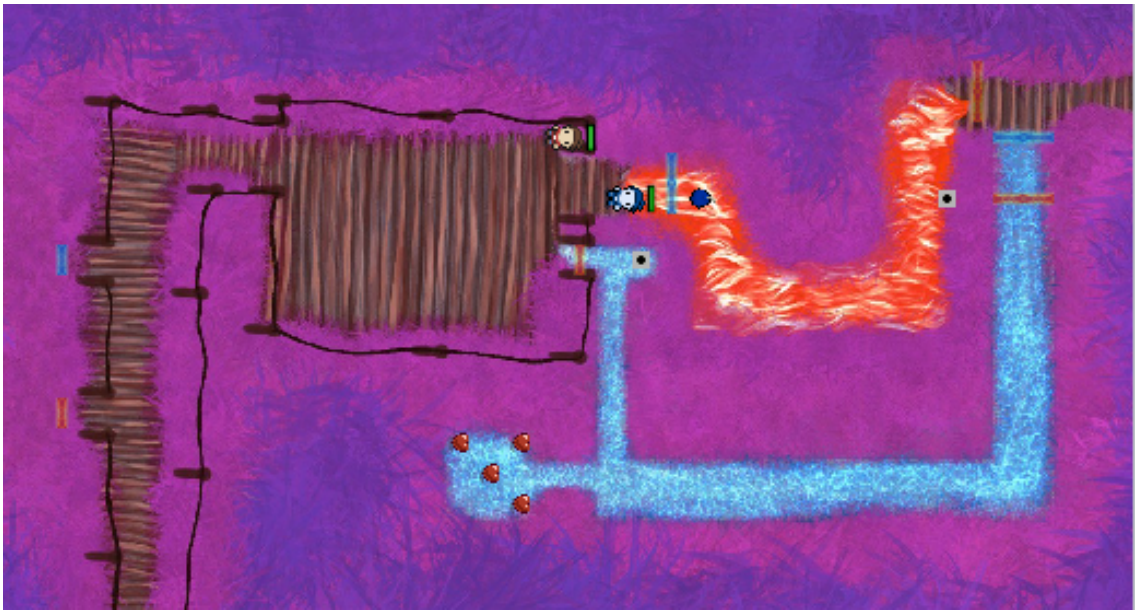

FIG. 13 : Configuration de base de l'aire `Maze` (vue sur le flanc)

- Plaque de pression en position `(9,25)`. Vous pouvez en bonus remplacer cet acteur par un levier (voir extensions).
- Mur de feu orienté vers le bas, en position `(8,21)` qui disparait si le composant précédent est actif.
- Une série de coeurs en positions `(15, 18)`, `(16, 19)`,`(14, 19)` et `(14, 17)`.
- Mur d'eau orienté vers le bas, en position `(8,4)` toujours actif.
- Mur de feu orienté vers le bas, en position `(13,4)` toujours actif.

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 `ICoopPlayer`

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, `prefix` valant *`"icoop/player"`* pour le joueur rouge et *`"icoop/player2"`* pour le bleu.

Utilisation de l'épée :

```
final Vector anchor = new Vector(-.5f, 0);
```

```
final Orientation[] orders = {DOWN, UP, RIGHT, LEFT};
... new OrientedAnimation(prefix+".sword",
                          SWORD_ANIMATION_DURATION, this,
                          anchor, orders, 4, 2, 2, 32, 32)
```

avec `SWORD_ANIMATION_DURATION` valant 2.

Utilisation du bâton :

```
final Vector anchor = new Vector(-.5f, -.20f);
final Orientation[] orders = {DOWN, UP, RIGHT, LEFT};
... new OrientedAnimation(name , STAFF_ANIMATION_DURATION, this,
                          anchor, orders, 4, 2, 2, 32, 32)
```

avec `STAFF_ANIMATION_DURATION` valant 2 et `name` valant `icoop/player2.staff_water` pour le bâton bleu et `icoop/player.staff_fire` pour le rouge.

7.3.2 Explosif

Explosif non explosé :

```
new Animation("icoop/explosive", 2, 1, 1, this, 16, 16,
              ANIMATION_DURATION/2, true)
```

Explosion :

```
new Animation("icoop/explosion", 7, 1, 1, this, 32, 32,
              ANIMATION_DURATION/7, false)
```

avec `ANIMATION_DURATION` valant 24

7.3.3 Orbes

```
final Sprite[] sprites = new Sprite[ANIMATION_FRAMES];
for (int i = 0; i < ANIMATION_FRAMES; i++) {
  sprites[i] = new RPGSprite("icoop/orb", 1, 1, this,
  new RegionOfInterest(i * 32, spriteYDelta, 32, 32));
}
.. new Animation(ANIMATION_DURATION / ANIMATION_FRAMES, sprites)
```

où `ANIMATION_DURATION` vaut 24, `ANIMATION_FRAMES` vaut 6 et `spriteYDelta` vaut zéro pour les orbes bleues et 64 pour les rouges.

7.3.4 Coeurs

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

où `ANIMATION_DURATION` vaut 24.

7.3.5 Bâtons

```
new Animation(staff_name , 8, 2, 2, this, 32, 32,
              new Vector(-0.5f, 0),
              ANIMATION_DURATION/8, true);
```

où ANIMATION_DURATION vaut 32 et staff_name vaut *"icoop/staff_water"* pour le bâton bleu et *"icoop/staff_fire"* pour le rouge.

7.3.6 Flammes

```
new Animation("icoop/fire", 7, 1, 1, this, 16, 16, 4, true)
```

7.3.7 Boules 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 *"icoop/magicWaterProjectile"* pour les boules d'eau et *"icoop/magicFireProjectile"* pour celles de feu.

7.3.8 Mort des enemmis

Les ennemis disparaissent dans un petit nuage en mourant :

```
new Animation("icoop/vanish", 7, 2, 2, this, 32, 32, new
   Vector(-0.5f, 0f), ANIMATION_DURATION/7, false);
```

où ANIMATION_DURATION vaut 24.

7.3.9 Crânes lanceurs de feu

```
 Orientation[] orders = new Orientation[]{Orientation.UP,
    Orientation.LEFT, Orientation.DOWN, Orientation.RIGHT};
... new OrientedAnimation("icoop/flameskull",
                          ANIMATION_DURATION/3, this,
                          new Vector(-0.5f, -0.5f), orders,
                          3, 2, 2, 32, 32, true)
```

où ANIMATION_DURATION vaut 12.

7.3.10 Artificiers

```
final Vector anchor = new Vector(-0.5f, 0);
final Orientation[] orders = {DOWN, RIGHT, UP, LEFT};
// état "non protégé":
...new OrientedAnimation("icoop/bombFoe", ANIMATION_DURATION/3,
                         this, anchor, orders, 4, 2, 2, 32, 32,
                         true)
// état "protégé":
...new OrientedAnimation("icoop/bombFoe.protecting",
   ANIMATION_DURATION/3,
```

```
                                this, anchor, orders, 4, 2, 2, 32, 32,
                                false)
```

où `ANIMATION_DURATION` vaut 24.