CS-107 : Mini-projet 2
Jeux de type « RPG »

Q. Juppet, J. Sam, B. Jobstmann

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 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. Le jeu de base qu'il vous sera demandé de créer sera nommé ARPG en guise de clin d'oeil à une référence du genre[1]. 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 5 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 adéquet 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. Pour atteindre cet objectif, il vous sera notamment demandé d'enrichir le moteur de jeux de nouveaux composants, potentiellement utilisables dans des situations autres que celles des jeux que vous allez coder.

Le projet comporte quatre étapes :

- Étape 1 (« ARPG de base ») : au terme de cette étape vous aurez créé, en utilisant les outils du moteur de jeu fourni, une instance basique de RPG avec un acteur se déplaçant d'une aire à l'autre en passant par des portes et capable de couper des touffes d'herbe.
- Étape 2 (« Points de vie et ressources ») : lors de cette étape il vous sera demandé d'enrichir le moteur de jeu de fonctionnalités permettant l'affichage de graphismes statiques, comme le décompte des points de vie de l'acteur principal. Vous y ajouterez également l'outillage nécessaire à la gestion d'inventaires. Votre personnage principal pourra ainsi commencer à collecter des objets utiles.
- Étape 3 (« Monstres et batailles ») : cette étape permettra à votre acteur d'affronter des ennemis.
- Étape 4 (Extensions) : durant cette étape, diverses extensions plus libres vous seront proposées et vous pourrez enrichir à votre façon le jeu créé à l'étape précédente ou en créer d'autres.

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

1. Les ressources fournies dans le répertoire `res/` seront d'ailleurs dans des sous répertoires nommés `zelda`

FIG. 1 : Quelques scènes du jeu où le joueur se déplace d'aire en aire en collectant des objets et des points de vie et en affrontant divers créatures hostiles (ou pas).

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

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

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

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

(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 familier avec git, nous recommandons l'utilisation de GitLab : https ://gitlab.epfl.ch/, mais tout type de dépôt est acceptable pour peu qu'il soit privé.

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

2 ARPG de base (étape 1)

Le but de cette étape est de commencer à créer votre propre petit jeu de type ARPG.

Cette version de base contiendra un personnage principal capable de passer des portes et couper de l'herbe. Le passage des portes se fera selon le mécanisme plus général des *interactions entre acteurs*, tel que décrit dans le tutoriel 3. Ce jeu fera donc intervenir :

- un personnage principal :
- des portes qui permettront au personnage de les traverser. Il s'agira d'une interaction de contact (le personnage doit être dans une cellule contenant un acteur « porte » pour pouvoir traverser cette « porte »).
- des touffes d'herbe qui « accepteront » d'être coupées par le personnage. Il s'agira d'une interaction à distance (le personnage peut couper la touffe d'herbe se trouvant sur une case voisine).

Pour vérifier que vous maîtrisez le concepts d'`Interactor`, vous coderez également un nouveau type d'acteur, les « bombes », qui permettent aussi de couper de l'herbe mais d'une façon un peu moins écologiques[2].

Commencez par créer un nouveau paquetage `game.arpg`.

2.1 Préparation du jeu `ARPG`

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

- de la classe `ARPGPlayer` qui modélise un personnage principal, à placer dans `game.arpg.actor`; laissez cette classe vide pour le moment, nous y reviendrons un peu plus bas ;
- de la classe `ARPG`, équivalente à `Tuto2` à placer dans le paquetage `game.arpg`; cette classe héritera de `RPG` et n'aura donc plus besoin d'attribut spécifique pour le personnage, ce dernier étant hérité de la class `RPG`. N'oubliez pas d'adapter la méthode `getTitle()` qui retournera un nom de votre choix (par exemple *`"ZeldIC"`* ;-)) ;
- des classes `ARPGArea` équivalente à `Tuto2Area`, à placer dans un sous-paquetage `game.arpg.area`;
- des classes `Ferme`, `Village` et `Route` héritant de `ARPGArea`, à placer dans le paquetage `game.arpg.area` (elles sont équivalentes aux classes `Ferme`, et `Village` de `game.tutos.area.tuto2`) ; l'intitulé associé à `Route` est *`"zelda/route"`*;
- de la classe `ARPGBehavior` analogue à `Tuto2Behavior` à placer dans `game.arpg` et qui contiendra une classe publique `ARPGCell` équivalente de `Tuto2Cell`.

2. cela étant, la virtualité de nos mondes nous épargne, en apparence, les cas de consciences liés au réchauffement climatique

2.2 Ebauche du personnage principal

Pour le moment codez `ARPGPlayer` dans le même esprit que `GhostPlayer`. Il héritera cependant de la classe `Player` décrite dans le tutoriel 3 (ce qui en principe, devrait le dispenser de bien des lignes de code de son équivalent `GhostPlayer`). Son constructeur prendra en paramètre l'aire à laquelle il appartient et sa position de départ sous la forme d'une `DiscreteCoordinates`. L'image associée sera donnée directement dans le constructeur et aura pour le moment la valeur *`"ghost.1"`*. Nous vous donnerons des indications plus bas pour améliorer cette représentation graphique. La méthode `update` de `ARPGPlayer` ressemblera beaucoup à celle de `GhostPlayer` mais **ne devra plus tester si la cellule occupée suite à un déplacement correspond à une porte**. Il faudra, pour gérer proprement le passage de portes, mettre en place l'interaction avec les acteurs `Door`, selon le schéma suggéré par le tutoriel 3. Ceci vous sera demandé un peu plus loin. Comme `Interactor`, `ARPGPlayer` doit définir les méthodes :

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

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

En tant que `Interactable`, le personnage `ARPGPlayer` sera l'objet d'interaction par contact ou a distance. Il sera non traversable (on ne peut pas lui marcher dessus !). En tant que `Interactor`, il voudra systématiquement toutes les interactions de contact. Le fait qu'il soit demandeur d'interactions à distance sera conditionné par l'utilisateur du jeu : la touche *`'E'`* sera utilisée pour indiquer qu'il veut une interaction à distance. Par exemple, si notre personnage est en face d'une touffe d'herbe, pour indiquer que l'on souhaite qu'il la coupe, on appuie sur la touche *`'E'`*. Cette touche ne s'occupera évidemment pas du cas spécifique de la touffe d'herbe. Elle sera uniquement employée pour faire basculer le personnage en mode « demande d'interaction à distance ». .

Intéressons-nous maintenant à la gestion concrètes des interactions.

2.3 Interaction avec les portes

Il vous est demandé d'appliquer le schéma suggéré par le tutoriel 3 pour mettre en place les interactions de `ARPGPlayer` avec l'acteur `Door`.

Pour cela, créez dans un sous paquetage `game.arpg.handler`, l'interface `ARPGInteractionVisitor` héritant de `RPGInteractionVisitor`, et qui fournit une définition par défaut des méthodes d'interaction de tout `Interactor` du jeu de `ARPG` avec :

- une cellule du jeu (`ARPGCell`)
- une personnage principal du jeu (`ARPGPlayer`)

Ces définitions (par défaut) auront un corps vide pour exprimer le fait que par défaut, l'interaction consiste à ne rien faire. Notez qu'il n'est pas nécessaire de définir l'interaction par défaut avec l'acteur « porte », `Door` du paquetage `game.rpg.actor`, qui est déjà prévue dans `RPGInteractionVisitor`.

`ARPGPlayer` en tant que `Interactor` du jeu de `ARPG`, doit fournir le cas échéant une définition plus spécifiques de ces méthodes.

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

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

2.3.1 Enregistrement des acteurs dans les aires

Le jeu `ARPG` créera au démarrage un personnnage `ARPGPlayer` placé initialement en `(6, 10)` (vous noterez que la classe `RPG` fournit une méthode `initPlayer`). L'aire de démarrage sera *`"zelda/Ferme"`*.

La méthode `createArea` de `Ferme` dans `game.arpg.area` enregistrera trois portes (conformément à l'image du « behavior » associé) et dont voici les caractéristiques :

porte	destination	coord. arrivée	orientation	cellule principale	autres coord.
1	Route	(1,15)	droite	(19,15)	(19,16)
2	Village	(4,18)	bas	(4,0)	(5,0)
3	Village	(14,18)	bas	(13,0)	(14,0), (15,0)

La méthodes `createArea` de `Village` dans `game.arpg.area` enregistrera les trois portes dont voici les caractéristiques :

porte	destination	coord. arrivée	orientation	cellule principale	autres coord.
1	Ferme	(4,1)	haut	(4,19)	(5,19)
2	Ferme	(14,1)	haut	(13,19)	(14,19),(15,19)
3	Route	(9,1)	haut	(29,19)	(30,19)

Enfin la méthode `createArea` de `Route` dans `game.arpg.area` enregistrera les deux portes :

porte	destination	coord. arrivée	orientation	cellule principale	autres coord.
1	Ferme	(18,15)	haut	(0,15)	(0,16)
2	Village	(29,18)	bas	(9,0)	(10,0)

Notez que selon l'image (« behavior ») associé à `Route`, il y a deux emplacements possibles non (encore) exploités correspondant à des visuels portes.

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

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

2.4 Animation du personnage

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

Voici quelques indications pour y parvenir :

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

2.5 Touffes d'herbe : interactions à distance

Il s'agit maintenant de coder un premier acteur avec lequel `ARPGPlayer` pourra interagir *à distance*. Il s'agira de l'acteur `Grass` dérivant de `AreaEntity` et se dessinant sous la forme d'une touffe d'herbe. Prenez la region `(0, 0, 16, 16)` de *`"zelda/grass"`* comme `Sprite` pour la représentation graphique. Une touffe d'herbe aura la particularité de pouvoir être *coupée*. Dès qu'elle l'est, elle disparaît de l'aire de jeu. Il s'agit d'un acteur qui n'est traversable qu'à condition d'avoir été coupé et accepte les interactions à distance et celle de contact. Ses cellules courantes (retour de `getCurrentCells()`) se réduiront à l'ensemble contenant uniquement sa cellule principale (pareil que pour `ARPGPlayer`). L'entête de son constructeur sera également analogue à celle de `ARPGPlayer`.

Complétez les méthodes d'interaction de `ARPGPlayer` de sorte à ce qu'actionner la touche *`'E'`*, lorsque `ARPGPlayer` fait face à une touffe d'herbe, permette de la couper.

Pour améliorer le visuel, vous pouvez animer la découpe de l'herbe au moyen de la ressource fournie *`"zelda/grass.sliced"`* (facultatif) .

2.5.1 Création des touffes d'herbes

Faites en sorte que l'aire `Route` de votre jeu `ARPG`, à sa création, crée des touffes d'herbes dans les cases `(i,j)` (avec `i` variant de `5` à `7` compris et `j` de `6` à `11` compris).

2.6 Bombes

Vous comprenez maintenant les mécanismes généraux permettant à un acteur d'interagir avec un autre. D'autres acteurs que le personnage principal peuvent bien sûr être « demandeur d'interaction ». Pour l'illustrer, créez un nouvel acteur, nommé `Bomb`, héritant de `AreaEntity` et jouant le role d'`Interactor`. Une bombe sera associée à un retardateur (un entier) initialisé à sa création. A chacun de ses `update`, le retardateur est décrémenté (par exemple d'une unité). Lorsque le retardateur est à zéro, la bombe explose (se met en état d'explosion) et disparaît du jeu. La bombe est en demande d'interaction à distance ou de contact lorsqu'elle est en état d'explosion. Pour le moment elle interagit à distance uniquement avec les touffes d'herbe. L'effet de l'interaction avec une touffe d'herbe sera de la couper. Le champs d'action, `getFieldOfViewCells()`, de la bombe est l'ensemble des cellules immédiatement voisines à sa cellule principale. `getCurrentCells()` sera codé comme pour `ARPGPlayer`.

La bombe peut être représentée graphiquement au moyen de la ressource *`"zelda/bomb"`*. Elle doit disparaître après son explosion qui doit s'accompagner d'un visuel spécifique (par exemple au moyen de la ressource *`"zelda/explosion"`*).

Question 1

Comment feriez-vous en sorte (conceptuellement) pour que les bombes, au lieu de couper les touffes d'herbes les brûlent ?

Pour valider cette étape, placez une bombe près des touffes d'herbes de `Route`. Au bout d'un moment, la bombe doit exploser détruisant l'herbe. Bien sûr l'on pourrait faire en sorte que la bombe soit déposée par le personnage, pour cela la notion d'inventaire, à disposition du joueur, serait bienvenue, nous allons nous y atteler un peu plus tard dans le projet.

Question 2

La logistique mise en place, telles qu'exposées 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 déplaçent 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.

2.7 Validation de l'étape 1

Pour valider cette étape, vous vérifierez :

1. que l'aire `Route` contient un « champs » de 3 x 6 touffes d'herbes ;
2. que `ARPGPlayer` ne peut pas marcher sur l'herbe ;
3. qu'il peut marcher sur les portes ;
4. qu'il peut interagir à distance avec l'herbe au moyen de la touche `'E'` lorsqu'il est dans une case immédiatement voisine seulement. Il coupe alors l'herbe qui doit disparaître ; il peut alors marcher sur l'ancien emplacement occupé par l'herbe ;
5. que s'il a réussi à couper une touffe d'herbe, il faut à nouveau faire usage de la touche `'E'` pour lui faire couper une autre touffe d'herbe ;
6. qu'il peut passer de `Ferme` à `Village` par les portes en (4,0) et en (13,10) dans `Ferme`;
7. qu'il peut passer de `Ferme` à `Route` par la porte en (19,15) dans `Ferme`;
8. qu'il peut passer de `Route` à `Ferme` par la porte en (0,15) dans `Route`;
9. qu'il peut passer de `Route` à `Village` par la porte en (9,0) dans `Route`;
10. qu'il peut passer de `Village` à `Ferme` par les portes es en (4,19) et 13,19) dans `Village`;
11. et qu'il peut passer de `Village` à `Route` par la porte en (29,19) dans `Village`;
12. que la bombe explose au bout d'un laps de temps ; qu'elle coupe les touffes d'herbe de son voisinage immédiat en explosant. L'explosion cause la disparition de la bombe et celles des touffes d'herbes touchées.

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

3 Points de vie et ressources (étape 2)

Dans cette seconde partie du projet, il s'agit de doter le personnage principal d'un système de ressources (inventaire basique pour stocker de l'argent et des équipement par exemple). Ceci lui permettra de survivre dans le monde hostile que nous nous apprêtons à lui créer. Les graphismes seront élaborés de sorte à permettre de visualiser les points de vie du personnage ainsi que les ressources dont il dispose. Notez que dans un premier temps, la notion d'inventaire sera très rudimentaire (pas de graphisme pour séléctionner les éléments de l'inventaire à utiliser par exemple). Vous trouverez ci-dessous les spécifications à respecter ainsi que quelques indications.

3.1 Points de vie

Le personnage principal (`ARPGPlayer`) sera doté de points de vie (un `float`). Le nombre de points de vie est plafonné à une valeur maximale (par exemple 5), identique pour tous les personnages de ce type. On supposera que le nombre de points de vie est fixé à ce maximum lors de la création du personnage.

3.2 Ressources

Le personnage disposera de ressources (argent, équipements etc.) qu'il peut stocker dans un *inventaire* contenant divers *articles*.

La notion générique d'inventaire (`Inventory`) et celle d'article d'inventaire (`InventoryItem`) doivent être utilisables dans tout jeu de type RPG et ne pas être spécifiques au jeu ARPG que vous êtes en train d'implémenter.

3.2.1 Articles d'inventaires

La spécification *fonctionnelle* d'un article d'inventaire (`InventoryItem`) est qu'il doit être possible d'accéder à :

- son nom (un `String`) ;
- son poids (un `float`) ;
- et son prix (un `int` pour simplifier)

Le poids des articles permettra potentiellement de plafonner ce que l'inventaire est capable de stocker.

Articles d'inventaires et acteurs La question se pose naturellement de savoir si un article d'inventaire est un acteur : va t'on stocker dans un inventaire un `Actor` ou une autre forme de représentation ? De façon générale, un acteur est une entité qui a un comportement

propre et qui évolue au cours du temps : on l'enregistre dans l'aire et cette dernière se charge d'invoquer sa méthode `update`. Un article stocké dans un inventaire n'a pas forcément ce rôle. Dans certains cas, il s'agit simplement d'un équipement indissociable de celui à qui il appartient. De plus, tant que l'article est stocké dans un inventaire, il n'apparaît en principe pas dans l'aire. Vous coderez donc un `InventoryItem` comme une entité indépendante de `Actor`. Vous noterez que ce choix de conception n'empêche pas un article d'inventaire de se transformer en acteur si nécessaire. Par exemple, si le personnage sort une bombe de son inventaire pour la poser par terre, la bombe devient un acteur qui va évoluer comme tous les acteurs.

Pour être plus précis, `InventoryItem` sera 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. Tous les articles d'un modèle donné seront donc supposés avoir un même nom, un même poids et un même prix. Pour simplifier, nous parlerons cependant d'*articles* d'inventaire (plutôt que de *modèles d'articles* d'inventaire).

ARPGItem Votre jeu `ARPG`, disposera de sa propre implémentation concrète d'articles d'inventaires. Il vous est demandé d'implémenter le concept `ARPGItem`, une spécialisation de `InventoryItem` modélisant les articles d'inventaire spécifiques au jeu `ARPG`.

Dans la version de base de `ARPG` qu'il vous est demandé de rendre, vous considérerez que les articles d'inventaires possibles dans le jeu sont : des flèches (`Arrow`), des épées (`Sword`), des bâtons de magiciens (`Staff`), des arcs (`Bow`), des bombes (`Bomb`) et des clés de château (`CastleKey`).

Pour le moment, il n'est pas nécessaire d'anticiper les acteurs correspondant aux articles d'inventaires.

Indications :

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

Choisissez les prix que vous voulez pour chacun des articles suggérés. Vous pouvez dans un premier temps considérer que tout `ARPGItem` a un poids nul (ce qui implique que l'on peut en avoir autant que l'on veut dans l'inventaire).

Vous anticiperez aussi que tout `ARPGItem` a une représentation graphique et est donc caractérisé par un `Sprite` (vous pouvez utiliser les images `arrow.icon`, `sword.icon`, `bow.icon`, `bomb` , `staff_water.icon` et `key` du dossier `res/images/sprites/zelda`).

3.2.2 Inventaires

Un inventaire (`Inventory`) sera caractérisé par :

- le poids maximal qu'il peut porter (donné à la construction) ;

- l'ensemble des articles de type `InventoryItem` qu'il contient et leur quantité (par exemple 3 `Arrow` et 1 `Bow`). Cet ensemble doit être vide à la construction de l'inventaire.

Il doit être possible d'y ajouter ou d'en supprimer une quantité donnée d'un `InventoryItem` donné et de savoir si un `InventoryItem` donné s'y trouve stocké.

Un article ne pourra être ajouté s'il fait dépasser le poids maximal toléré de l'inventaire (dans ce cas la méthode d'ajout ne fait rien). De façon analogue, la méthode qui tente de supprimer une quantité donnée d'un `InventoryItem` ne fait rien si cet article n'existe pas dans l'inventaire ou s'il existe en quantité insuffisante.

En terme d'encapsulation, seule la méthode testant si un article donné est dans l'inventaire sera publique. Les méthodes d'ajout et de retrait ainsi que le constructeur devront rester protégés.

Vous coderez enfin `ARPGInventory`, une spécialisation de `Inventory` spécifique au jeu `ARPG`. Sa particularité sera de pouvoir aussi consigner un montant d'argent ainsi que la fortune possédés par le propriétaire de l'inventaire (des `int`). La fortune sera le montant d'argent auquel s'additionne la somme du prix des articles de l'inventaire (s'il y a 3 épées ce sera 3 fois le prix d'une épée plus le montant en argent). Le montant en argent peut être augmenté par des ajouts direct d'argent (méthode `addMoney(int money)` ou équivalent). La classe `ARPGInventory` offrira une méthode publique permettant de connaître le montant d'argent ainsi que la fortune stockés dans l'inventaire (`getMoney()` et `getFortune()`ou équivalents).

Le montant d'argent initial sera donné à la construction. Vous donnerez une grande valeur quelconque au poids maximal d'un `ARPGInventory`.

Indications :

- il est judicieux de faire en sorte que les méthodes d'ajout et de retrait d'un article dans un inventaire retournent un booléen indiquant si le retrait/ajout a pu se faire ou pas ;
- une table associative est un bon moyen d'apparier une description d'article à une quantité (voir l'annexe 2 du tutoriel) ;
- les concepts d'acteurs et d'inventaires sont très liés (un `ARPGInventory` va appartenir à un acteur de `ARPG` dans la plupart des cas et ce dernier va avoir besoin d'y mettre/puiser des articles). Tenez en compte lors du choix des paquetages dans lesquels coder les concepts demandés.

3.2.3 Inventaire de `ARPGPlayer`

Vous commencerez par modéliser le fait que tout inventaire a un propriétaire (`Inventory.Holder`). Un propriétaire d'inventaire est caractérisé par une méthode `boolean possess(InventoryItem item)` ou équivalent indiquant s'il possède l'article concerné.

Vous compléterez ensuite la modélisation du personnage principal `ARPGPlayer` de sorte à le doter d'un inventaire de type `ARPGInventory`. A la création du personnage, quelques articles, dont une ou plusieurs bombes, seront placés « en dur » dans son inventaire pour

simplifier[3]. `ARPGPlayer` devra naturellement être répertorié comme propriétaire d'inventaire. Pour préserver l'encapsulation, il ne fournira pas son inventaire au monde extérieur.

Vous considérerez que `ARPGPlayer` 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 `ARPGPlayer` de nouvelles fonctionnalité 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 `Tab`; passer d'un article à l'autre se fera de façon circulaire (après le dernier de la liste on revient au premier) ;
- et d'invoquer l'utilisation de l'équipement courant au moyen de la barre d'espace.

L'utilisation de l'équipement courant se fera bien entendu au cas par cas[4] : chaque équipement est un cas particulier à gérer de façon spécifique.

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

Utiliser une équipement de type « bombe » revient à créer un acteur bombe 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).

La bombe sera dotée d'une interaction supplémentaire, avec le personnage principal cette fois : elle lui fera perdre un nombre constant de points de vie (2 par exemple).

Question 3

Comment feriez-vous pour rendre l'inventaire moins spécifique ?(on ne vous demande pas de l'implémenter)

3.2.4 Graphismes statiques

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

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

3. vous pourrez améliorer ce point dans la partie extensions du projet
4. les types énumérés se prêtent fort bien à l'usage du switch

FIG. 2 : Affichage graphique du status du personnage principal

Afin de pouvoir visualiser les différents éléments mis en place, il vous est demandé de mettre en place leur représentation graphique selon l'exemple de la figure 2. Ces informations seront encapsulées dans un seul concept, celui d'« objet graphique décrivant le statut d'un personnage principal » (`ARPGPlayerStatusGUI`). Un tel objet sera naturellement un `Graphics`.

Indications :

- Les différents objets graphiques constituant un `ARPGPlayerStatusGUI` peuvent être dessinés comme des `ImageGraphics` du répertoire `game.actor`. Les sprites fournis suivants peuvent être utilisés : *`"zelda/gearDisplay"`*, *`"zelda/heartDisplay"`*, *`"zelda/coinsDisplay"`*, *`"zelda/digits"`* et les sprites associés aux articles d'inventaires prévus pour le jeu `ARPG`.

- Voici un exemple de comment placer le petit cercle brun destiné à accueillir le graphique de l'équipement courant :

```
float width = canvas.getScaledWidth();
float height = canvas.getScaledHeight();
Vector anchor = canvas.getTransform().getOrigin().sub(new
   Vector(width/2, height/2));
ImageGraphics gearDisplay = new
   ImageGraphics(ResourcePath.getSprite("zelda/gearDisplay"),
   1.5f, 1.5f, new RegionOfInterest(0, 0, 32, 32),
   anchor.add(new Vector(0, height - 1.75f)), 1, DEPTH);
gearDisplay.draw(canvas);
```

  vous pouvez vous en inspirer pour le codage de la classe `ARPGPlayerStatusGUI`

Il y aura autant de coeurs dessinés que le nombre maximal possible de points de vie. Un coeur sera dessiné en rouge, partiellement rouge ou grisé selon le nombre de points de vie

effectif. Si par exemple le nombre de points de vie du personnage supérieur à 3 et inférieur à 4, il y aura 3 coeurs rouges, un partiellement rouge et un gris.

Voici enfin les contraintes de codage à respecter :

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

Notez que l'on ne vous demande que l'affichage du montant d'argent. Si vous souhaitez afficher la fortune, vous pouvez ajouter à votre jeu un nouveau contrôle permettant de passer de l'affichage du montant d'argent à l'affichage de la fortune et vice-versa.

3.2.5 Tâche

Il vous est demandé de :

- coder les concepts décrits précédemment conformément aux spécifications et contraintes données.
- tester vos développements de façon « ad'hoc » en attribuant différents objets d'inventaire et différents montants d'argent au personnage principal. Comme indiqué précédemment, vous le ferez pour le moment « en dur » dans le constructeur du personnage.

Vous vérifierez alors que :

1. le montant d'argent est correctement affichée visuellement en fonction des avoirs du personnage en argent ;
2. que les points de vie s'affichent correctement et que le personnage en perd s'il dépose une bombe devant lui et qu'il reste sur place (faites lui déposer plusieurs bombes l'une après l'autre) ;
3. qu'il est bien possible de passer d'un équipement à l'autre au moyen de la touche `Tab` et que l'équipement courant s'affiche correctement ;
4. que seuls les équipements explicitement ajoutés à l'inventaire apparaîssent lorsque on passe d'un équipement à l'autre au moyen de la touche `Tab`;
5. que seul l'équipement courant est utilisable (à vérifier avec la bombe) ;
6. que la bombe n'est plus sélectionnable comme équipement courant s'il n'y en avait qu'une et que le personnage l'a utilisée ;
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.

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

3.3 Collecte d'objets

Afin que le personnage principal puisse récupérer des points de vie perdus ou amasser des richesses, il vous est maintenant demandé d'introduire dans votre mini-projet, le concept d'objet « ramassable ». Ce concept est *a priori* intéressant pour tout jeu sur grille et il est raisonnable d'introduire le concept général de `CollectableAreaEntity`. Un objet « ramassable » a pour caractéristique fondamentale de pouvoir être ramassé et de disparaître de l'aire simulée au moment où il l'est. Dans votre jeu `ARPG`, un objet ramassable aura une déclinaison spécifique : il sera automatiquement ramassé lorsqu'on lui marche dessus.

Vous introduirez deux types d'objets de ce type :

1. les pièces de monnaie (`Coin`) : vous considérerez que toutes les pièces ont la même valeur (50 par exemple).
2. les coeurs (`Heart`) : vous considérerez que chaque coeur ramassé rapporte le même nombre de points de vie (1 par exemple)

Les interactions attendues sont que le personnage principal augmente son montant d'argent de la valeur de chaque pièce ramassée et qu'il augmente son nombre de points de vie à chaque coeur qu'il ramasse ; ce nombre étant plafonné au maximum possible. Les `Coin` et `Heart` peuvent être animés (tourner autour d'eux même). Vous pouvez mettre en place cette animation en bonus si vous le souhaitez.

Enfin vous modifierez votre classe `Grass` de sorte à ce que si une touffe d'herbe est coupée, une pièce ou un coeur apparaîsse à sa place.

Pour tirer au sort l'apparition d'une pièce ou d'un coeur vous utiliserez l'algorithme suivant :

1. tirer un `double` au hasard ;
2. si ce nombre est inférieur à une constante `PROBABILITY_TO_DROP_ITEM`;
   (a) tirer un autre `double` au hasard ;
   (b) si ce nombre est inférieur à une constante `PROBABILITY_TO_DROP_HEART` créer un coeur, sinon créer un pièce.

Si les deux constantes valent 0.5 par exemple, cet algorithme revient à dire que lorsque la touffe d'herbe est coupée, il y a 50% de chances de voir apparaître un objet et que dans ce cas il y a encore 50% de chances que cet objet soit un coeur.

Indication : Pour tirer un `double` au hasard, utilisez l'instruction :

```
RandomGenerator.getInstance().nextDouble();
```

Il faut pour cela inclure `RandomGenerator` du paquetage fourni `math`.

3.3.1 Tâche

Il vous est demandé de :

- coder les éléments suggérés ci-dessus conformément aux spécifications et contraintes décrites ;
- les tester en créant les conditions permettant de ramasser des coeurs ou des pièces de monnaie.

Vous vérifierez que :

1. le fait de couper de l'herbe permet de faire gagner des points de vie et de l'argent de façon aléatoire au personnage en marchant dessus ;
2. que le fait de ramasser des pièces ou de coeur se répercute proprement sur l'affichage du statut ;
3. que le fait de marcher sur un objet ramassable le fait disparaître de l'aire de jeu ;
4. et que les points vérifiés dans la sections 3.2.5 sont toujours fonctionnels.

3.4 Objets dépendants de signaux

Le tutoriel a introduit la notion de signal qui peut être exploitée pour rendre le jeu plus attractif en le faisant dépendre de la résolution d'énigmes. On peut imaginer par exemple que pour obtenir une arme avec laquelle se défendre, le personnage principal ait à allumer des torches ce qui permettrait d'ouvrir une porte vers une salle contenant l'arme voulue. L'outillage fournit, permet typiquement de répertorier les torches comme des signaux attachés à la porte. Pour commencer à travailler avec la notion de signal, il vous est demandé de mettre en oeuvre un nouvel acteur nommée `CastleDoor`, dérivant de l'acteur fourni `Door`, et dont l'ouverture peut être faite à la demande. Ce type de porte est créée fermée et acceptera des interactions à distance. `ARPGPlayer` interagit avec `CastleDoor` de la façon suivante :

- si la porte est ouverte il l'a passe et la referme derrière lui ;
- sinon, s'il possède une clé, il peut interagir à distance avec la porte pour demander son ouverture.

La clé sera modélisée au moyen d'un nouveau type d'acteurs, les `CastleKey`. Il s'agit d'un simple objet ramassable, associé au graphisme *`"zelda/key"`* et que le personnage principal stocke dans son inventaire lorsqu'il le ramasse.

Vous considérerez que la clé une fois utilisée reste dans l'inventaire.

Le visuel des `CastleDoor` correpondra à leur état (ouvert ou fermé) : vous disposez pour cela des ressources *`"zelda/castleDoor.close"`* et *`"zelda/castleDoor.open"`*.

Vous exploiterez le fait que `Door` dépende d'un signal pour contrôler la fermeture et l'ouverture de la `CastleDoor`.

3.4.1 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Pour tester vos développements, vous ajouterez au jeu deux nouvelles aires `RouteChateau` et `Chateau`.

L'aire `RouteChateau` sera dotée de deux portes dont voici les caractéristiques :

type de portes	destination	coord. arrivée	orientation	cellule principale	autres coord.
Door	Route	(9,18)	bas	(9,0)	(10,0)
CastleDoor	Chateau	(7,1)	haut	(9,13)	(10,13)

`Chateau` sera dotée d'une porte dont voici les caractéristiques :

type de portes	destination	coord. arrivée	orientation	cellule principale	autres coord.
Door	RouteChateau	(9,13)	bas	(7,0)	(8,0)

Vous ajouterez également la porte permettant le passage de `Route` à `RouteChateau` :

type de portes	destination	coord. arrivée	orientation	cellule principale	autres coord.
Door	RouteChateau	(9,1)	haut	(9,19)	(10,19)

Faites également en sorte qu'une `CastleKey` soit créée dans `Ferme`, par exemple en `(6,6)`.

Vous vérifierez alors :

1. que `ARPGPlayer` peut transiter vers les nouvelles aires aux endroits prévus ;
2. qu'il peut ramasser la clé qui devient alors un élément visible de son inventaire ;
3. qu'il peut alors ouvrir la porte du château en se plaçant devant et en invoquant son souhait d'interagir à distance (touche `E`) ;
4. que le fait d'utiliser la clé ne la fait pas sortir de l'inventaire ;
5. que le visuel de la porte change alors et qu'il peut se rendre dans le château en passant la porte ;
6. qu'il peut revenir en arrière mais trouve la porte fermée lorsqu'il est à nouveau dans l'aire `RouteChateau` ;
7. et qu'il ne peut ouvrir la porte du château s'il n'a pas de clé dans son inventaire.

3.5 Validation de l'étape 2

Pour valider cette étape, toutes les vérifications des sections 2.7, 3.2.5, 3.3.1 et 3.4.1 doivent avoir été effectuées.

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

4 Monstres et batailles (étape 3)

Dans ce troisième volet du projet, il vous est demandé d'enrichir la conception en permettant l'ajout de « monstres » que le personnage principal aura à affronter et de différents équipements plus « martiaux » pour mettre en scène des batailles épiques. Vous trouverez ci-dessous les spécifications à respecter ainsi que quelques indications.

4.1 Monstres

Les monstres 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 ; il 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 de monstre (`"zelda/vanish"`) ;
- qui peuvent déposer des objets sur l'aire simulée en mourant ;
- qui peuvent être sensibles à un certain nombre de types de dommages (ils ont une « liste de vulnérabilités ») ;
- qui sont « demandeurs » d'interactions (il ne font pas que les subir) ;
- avec lesquels il est possible par défaut d'avoir des interactions aussi bien à distance que de contact ;
- sur lesquels, par défaut, on ne peut marcher que s'ils sont morts ;
- et qui perdent des points de vie s'ils subissent une attaque sur l'une de leur vulnérabilité (le type de l'attaque et le nombre de points de vie perdus lors de l'attaque dépend de l'attaquant).

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 de monstre.

Les vulnérabilités possibles seront des vulnérabilités : « par dommage physique » (par exemple des coups d'épée), « par le feu » ou « par la magie ».

Chaque type de monstre aura une liste de vulnérabilités spécifiques, définie à la construction (par exemple les monstres « tronc » seront sensibles au feu et aux dommages physiques mais pas à la magie). Un monstre naît avec le nombre maximal de point de vie.

Toutes sortes de catégories de monstres sont envisageables. Nous vous demandons de coder les trois déclinaisons dont les spécifications sont décrites ci-dessous ; à savoir des « crânes volants », des « monstres tronc » et des « seigneurs des ténèbres » (tout un programme :-)).

Indications pour cette partie :

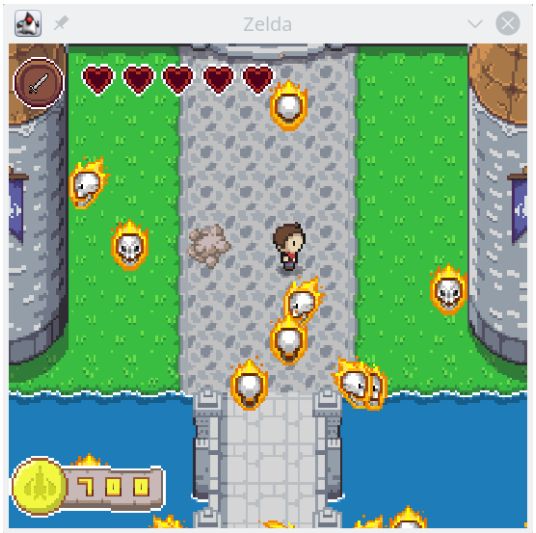


FIG. 3 : Crânes volants (le petit nuage derrière le personnage est un crâne volant en train de disparaître)

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

4.1.1 Crâne volant

Un crâne volant (`FlameSkull`) est un monstre vulnérable aux dommages physiques et à la magie. Sa durée de vie est fixée à la construction entre deux valeurs seuils constantes `MIN_LIFE_TIME` et `MAX_LIFE_TIME`, identiques pour tous les monstres de ce type. La représentation graphique ou l'animation d'un `FlameSkull` peut se faire au moyen de la ressource (*`"zelda/flameSkull"`*).

La durée maximale de vie d'un crâne volant est une constante identique pour toutes les instances (par exemple `350`).

Le déplacement du crâne volant se fait un peu comme celui du personnage principal. La différence réside dans le fait que ce déplacement n'est pas contrôlé via le clavier (le monstre se déplace tout seul) et qu'il y a des changements de direction aléatoires. La probabilité de changement de direction sera tirée au hasard et s'il y a lieu, la nouvelle orientation sera aussi choisie au hasard parmi les 4 possibles. Par exemple le crâne volant aura 60% de chances de changer d'orientation et s'il le fait, il choisira au hasard la direction à prendre. Pour tirer au hasard un entier compris entre 0 et MAX, utilisez l'instruction :

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

Une autre différence avec le personnage principal est que le crâne volant peut *survoler* des zones normalement non traversables par le personnage (comme l'eau par exemple).

Indication : vous pouvez introduire une catégorie `FlyableEntity` modélisant les entités capables de survoler des cellules (notamment celles sur lesquelles on ne peut habituellement pas marcher). Un `FlyableEntity` est caractérisé fonctionnellement par une méthode boolean `canFly()`. Cette méthode retournera `true` par défaut (toutes les entités volantes peuvent "survoler" par défaut). Vous modifierez ensuite `ARPGBehaviour` et `ARPGCell` de sorte à ce que `IMPASSABLE` soit répertoriée comme type de cellule que l'on peut survoler et que les entités `FlyableEntity` puissent y entrer. Bien sûr cela doit se faire sans test explicite sur la valeur `IMPASSABLE` (on peut imaginer d'ajouter par la suite d'autres types de cellules que l'on peut survoler). Notez enfin qu'un `FlyableEntity` est supposé *toucher* les cellules qu'il survole : le contact physique avec la cellule demeure.

Interactions L'acteur `FlameSkull` est « demandeur » d'interaction de contact tant qu'il est vivant. Il ne fait pas subir d'interaction à distance et l'on peut marcher dessus. Il peut interagir avec :

- tous les monstres vulnérables aux dommages de feu (en leur faisant perdre un nombre fixe de points de vie (par exemple `1.f`) ;
- avec le personnage principal en lui faisant perdre des points de vie (la même valeur que pour les monstres) ;
- avec les touffes d'herbe en les coupant ;
- et avec les bombes en les faisant exploser.

Les interactions possibles de la bombe doivent être enrichie de sorte à ce qu'en explosant, elle inflige des dommages physiques à tout monstre sensible à ce genre de dégâts.

4.1.2 Tâche

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Pour tester vos développements, vous doterez l'aire `RouteChateau` de deux contrôles spécifiques (qui ne doivent opérer que dans cette aire) : le fait d'appuyer sur la touche `S` devrait permettre la création d'un `FlameSkull` dans l'aire (par exemple en position (8,10)). Appuyer sur la touche `B` permettra de faire apparaître une bombe à une position donnée.

Vous ajouterez aussi quelques touffes d'herbe dans l'aire.

Vous vérifierez alors que :

1. les `FlameSkull` se déplacent seuls en opérant des changements de directions aléatoires ;

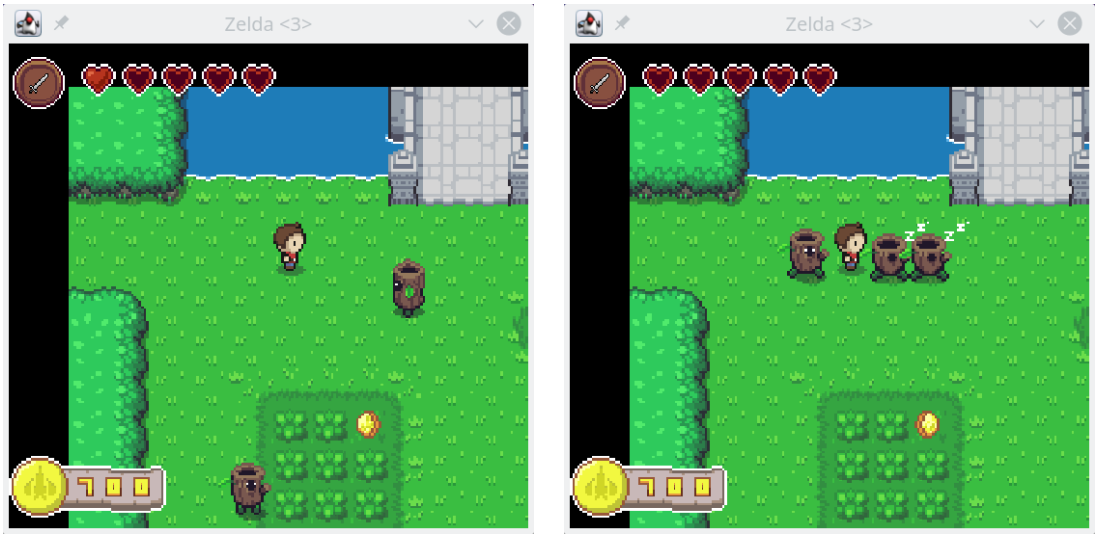


FIG. 4 : Monstres « tronc » (à droite, en train de dormir après une attaque)

2. qu'ils disparaîssent d'eux même après un certain temps en s'évanouissant dans un petit nuage ;
3. qu'ils sont capables de survoler le plan d'eau ;
4. qu'ils coupent l'herbe en passant dessus ;
5. qu'ils font exploser la bombe en passant dessus et qu'il subisse alors le dommage lié à l'explosion en disparaîssant à leur tour ;
6. et qu'ils font perdre des points de vie au personnage en passant dessus.

4.1.3 Monstre « tronc »

Le monstre tronc (`LogMonster`, voir figure 4) est un monstre vulnérable aux dommages physiques et aux dommages par le feu.

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

A chacun des états du monstre correspondra une représentation graphique/animation spécifique : *`"zelda/logMonster"`*, *`"zelda/logMonster.sleeping"`* et *`"zelda/logMonster.wakingUp"`*.

Interactions Le monstre tronc 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 champs d'interaction (de perception) est l'unique cellule en face de lui. Sinon, son champs d'interaction est un ensemble constant de cellules en face de lui (le même nombre pour tous les monstres tronc, par exemple 8).

Le monstre tronc n'interagit qu'avec les acteurs de type `ARPGPlayer`. L'interaction consiste à basculer en mode « attaque » lorsqu'un `ARPGPlayer` est dans son champs d'interaction et de faire perdre des points de vie à ce dernier lorsqu'il est devenu capable de l'attaquer, c'est à dire lorsqu'il aura basculé en mode « attaque ». Les points de vie enlevés seront en nombre constant, similaire pour tous les monstres « tronc », `2` par exemple.

Pour le moment, vous ne vous préoccuperez pas comment de le personnage principal combat les monstres troncs.

Comportement général Le monstre tronc a des moments d'inaction durant lesquel il ne fait absolument rien (même pas marcher), peu importe son état. Ce moment d'inaction est nul à la création du monstre et sera codé comme un nombre de pas de simulation. Il ne peut dépasser une certaine valeur (par exemple `24`, commune à tous les monstres de ce type).

Le comportement du monstre tronc est le suivant lorsque son temps d'inaction est écoulé :

- s'il est dans l'état inoccupé et qu'aucun déplacement n'est en cours, il se déplace (voir plus bas) puis peut entamer un moment d'inaction dont la durée est tirée aléatoirement ;
- s'il est en mode attaque , il se déplace dans sa direction courante (méthode `move`) ; lorsqu'aucun déplacement n'est en cours et qu'aucun déplacement n'est possible dans sa direction (il a atteint la cible de son attaque), il s'apprête à sombrer dans le sommeil après l'attaque en passant à l'état correspondant (en effet après la mise à jour des acteurs la méthode qui gère les interactions sera automatiquement invoquée et l'attaque du monstre pourra avoir lieu, la prochaine fois que sa méthode `update` sera invoquée, l'attaque aura déjà eu lieu et il peut dormir) ;
- s'il est en train de sombrer dans le sommeil, il entre dans un temps d'inaction d'une durée aléatoire comprise entre deux constantes `MIN_SLEEPING_DURATION` et `MAX_SLEEPING_DURATION` et bascule dans l'état endormi ;
- s'il est endormi (et que son temps d'inaction est écoulé), il bascule dans l'état en train de se réveiller ;
- et enfin, s'il est en train de se réveiller, il bascule à nouveau dans l'état inoccupé lorsque l'animation en cours a fini de s'exécuter.

A sa mort, le monstre tronc déposera une pièce à la place qu'il occupait (la pièce doit apparaître après l'animation liée à la mort).

Spécification du mode de déplacement Le déplacement du monstre tronc se fait comme celui du crâne volant sauf qu'il ne « vole » pas (c'est à dire qu'il ne pourra pas entrer dans les zones permettant d'être survolées)). Le monstre tronc ne suivra pas le personnage principal lorsqu'il change d'aire.

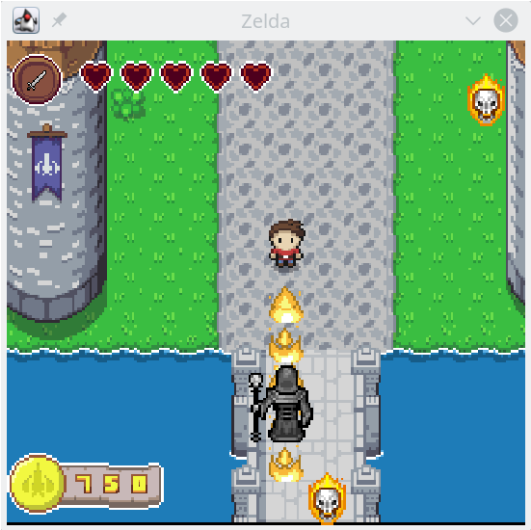


Fig. 5 : Seigneur des ténèbres invoquant des flammes magiques

4.1.4 Tâches

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

Pour tester vos développements, vous ajouterez un contrôle dans l'aire `RouteChateau` permettant d'ajouter un `LogMonster` en `(9,9)` au moyen de la touche `L`.

Vous vérifierez alors que :

1. le `LogMonster` se déplace seul en changeant aléatoirement de direction ;
2. qu'il ne peut pas survoler les plans d'eau ;
3. qu'il se dirige droit devant lui en ciblant le personnage principal quand celui-ci entre dans son champs de perception ;
4. qu'il inflige des dégâts au personnage une fois qu'il l'a atteint (lui enlève des points de vie), puis sombre dans le sommeil une fois l'attaque terminée ;
5. qu'il se réveille de lui-même au bout d'un certain temps ;
6. qu'il observe aléatoirement des moments d'inaction ;
7. qu'il est sensible aux dommages de feu infligés par les `FlameSkull` et aux dommages physiques infligés par les bombes ;
8. et qu'il disparaît dans un petit nuage en mourant, laissant apparaître derrière lui une pièce de monnaie que le personnage peut ramasser.

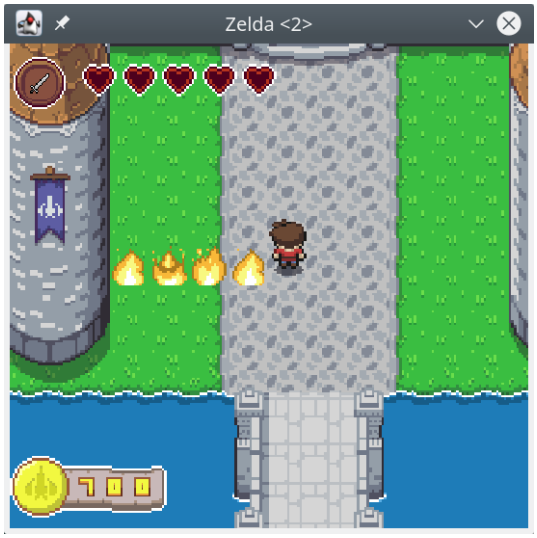


FIG. 6 : Flammes magiques (capables de se propager dans une direction)

4.1.5 « Seigneur des ténèbres »

Le monstre « Seigneur des ténèbres » (`DarkLord`, voir figure 5) est un monstre magicien vulnérable uniquement à la magie.

Il peut être lui aussi dans différents états qui vont conditionner son comportement. Par défaut, il est dans l'état « inoccupé » (« idle »). Autrement il peut être en train : d'attaquer, d'invoquer des créatures/objets occultes, d'invoquer un sort de téléportation ou de se téléporter.

Lorsqu'il se déplace (en marchant ou se téléportant), il aura le représentation graphique/animation spécifique : *"zelda/darkLord*". Lorsqu'il est en mode d'attaque ou en train d'invoquer des sorts ou des créature/objets, il aura la représentation graphique *"zelda/darkLord.spell"*.

Flammes magiques Le `DarkLord` peut lancer des « flammes magiques » (voir figure 6), ou invoquer des crânes volants.

Une flamme magique (`FireSpell`) est un acteur immobile doté d'une orientation et infligeant des dommages de feu à qui passe dessus. Elle est caractérisée par sa force, fixée à la construction, qui conditionne son niveau de nuisance. Elle disparaît d'elle même au bout d'un certain temps. Sa durée de vie, en nombre de pas de simulation, est fixée à sa création au moyen d'une valeur aléatoire tirée entre `MIN_LIFE_TIME` et `MAX_LIFE_TIME` (valant par exemple `120` et `240`). Sa représentation graphique ou ses animations peuvent être faits au moyen de *"zelda/fire"*.

Une fois tous les `PROPAGATION_TIME_FIRE` cycle, et si sa force est strictement positive, le `FireSpell` provoque la création d'un nouvel acteur du même type dans la cellule qui lui fait face, si celle-ci peut être investie. Le nouveau `FireSpell` a une force plus faible d'une unité. l'acteur `FireSpell` est « demandeur » d'interaction de contact. Il accepte toute interaction de contact mais pas les interactions à distance et l'on peut marcher dessus. Il peut interagir avec :

- tous les monstres vulnérable aux dommages de feu (en leur faisant perdre un nombre fixe de points de vie (par exemple `0.5f`) ;
- avec le personnage principal en lui faisant perdre des points de vie (la même valeur que pour les monstres) ;
- avec les touffes d'herbe en les coupant ;
- avec les bombes en les faisant exploser.

Interactions Le `DarkLord` ne peut interagir qu'à distance et s'il est vivant. Son champs d'action est constitué de toutes les cellules de son voisinage dans un « rayon » fixé par une constante propre à tous les monstres de ce type (`3` par exemple). Il ne s'agira pas d'un rayon circulaire mais du carré entourant la position du monstre.

Le `DarkLord` n'interagit qu'avec les acteurs de type `ARPGPlayer`. L'interaction consiste à se préparer à invoquer un sort de téléportation (il bascule dans ce mode), s'il n'est pas déjà en train de se téléporter.

Pour le moment, vous ne vous préoccuperez pas non plus de comment le personnage principal combat ce terrible ennemi.

Comportement général Tous les `n` cycles de simulation (`n` étant un entier tiré au hasard entre `MIN_SPELL_WAIT_DURATION` et `MAX_SPELL_WAIT_DURATION`), le `DarkLord` essaye de se placer dans des conditions favorables pour lancer ses attaques et sorts.

Il applique pour cela la stratégie suivante :

- un nombre est tiré au hasard et s'il dépasse une certaine valeur, le `DarkLord` bascule en mode attaque sinon il bascule en mode invocation de créatures/objets occultes ;
- il cherche ensuite parmi les orientations possibles qu'il peut prendre (en les tirant au hasard), laquelle permettrait de placer une flamme magique en face de lui ;
- il s'oriente selon la direction ainsi trouvée (sinon, il garde son ancienne orientation).

Le `DarkLord` a des moments d'inaction, mais seulement lorsqu'il est dans l'état inoccupé.

Une fois qu'il a tenté de se mettre en posture favorable pour ses attaques, son comportement est le suivant :

- en mode inoccupé et s'il n'est pas dans un temps d'inaction, il se déplace de la même manière que le monstre tronc ;
- en mode attaque, il envoie une flamme magique puis rebascule en mode inoccupé ;
- en mode « invocation de créature », il provoque l'apparition d'un crâne volant puis rebascule en mode inoccupé ;
- en mode « invocation de sort de téléportation », si aucun déplacement n'est en cours il prend l'apparence voulue (graphisme « attaques et invocation de sorts ») et bascule en « mode téléportation » ;

- en mode « téléportation », il choisit au hasard des coordonnées `x` et `y` comprises entre `-TELEPORTATION_RADIUS` et `TELEPORTATION_RADIUS`, deux constantes identiques pour tous les `DarkLord`; et s'y rend s'il est possible d'y entrer. Il répète la tentative jusqu'à ce qu'il lui soit possible de le faire (ou qu'il a dépassé un nombre maximal de tentatives). Il rebascule alors dans l'état inoccupé.

A sa mort, le `DarkLord` dépose une `CastleKey` qui pourra être ramassée par le personnage pour ouvrir la porte du château.

4.1.6 Tâches

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Pour tester vos développements, vous ajouterez un « seigneur des ténèbres » dans l'aire `RouteChateau`, par exemple en `(9,12)`. Vous vérifierez alors :

1. que le `DarkLord` se déplace seul et se téléporte dans des endroits aléatoirement choisis lorsque le personnage principal s'approche de lui ;
2. qu'il observe parfois des temps d'inaction ;
3. qu'il s'oriente spontanément dans des directions lui permettant de créer des objets en face de lui ;
4. qu'il invoque tantôt des crânes volants, tantôt des flammes magiques qui apparaîssent devant lui ; les flammes se propagent en lignes droites et disparaîssent spontanément au bout d'un certain temps ;
5. que les dommages physiques (bombes) et par le feu (crânes volants et flammes magiques) n'ont pas d'impact sur lui ;
6. que les flammes magiques font perdre des points de vie au personnage principal et aux monstres tronc lorsqu'ils passent dessus ; qu'elles font exploser des bombes ou coupent l'herbe se trouvant sur leur chemin ;
7. et qu'elles ne sont pas sensibles aux crânes volants.

4.2 Batailles

Le seigneur des ténèbres se téléporte dès que le personnage l'approche. Pour parvenir à le vaincre, il faudra donc disposer d'armes l'atteignant à distance. Il vous est donc demandé maintenant d'introduire le concept de projectile dans la maquette.

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 tournure du type `move(MOVE_DURATION/speed)`). La distance maximale pourra être codée comme un entier.

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.

Deux déclinaisons concrètes de projectiles à coder sont décrites ci-dessous.

4.2.2 Flèches

Les flèches (`Arrow`) sont des projectiles que l'on peut représenter graphiquement au moyen de *`"zelda/arrow"`*. Ils ne sont pas animés mais prennent simplement la bonne représentation en fonction de leur orientation.

Il intéragissent avec :

- tous les monstres vulnérables aux dommages physiques (en leur faisant perdre un nombre fixe de points de vie (par exemple `0.5f`) ;
- avec l'herbe en la coupant ;
- avec la bombe en la faisant exploser ;
- et avec les flammes magiques en les éteignant.

Dans chacun de ces cas, sauf celui des flammes, le projectile est stoppé dans sa course dès qu'il a interagi.

4.2.3 Projectiles magiques

Les `MagicWaterProjectile` sont des projectiles que l'on peut représenter graphiquement et animer au moyen de *`"zelda/magicWaterProjectile"`*.

Il interagissent avec :

- tous les monstres vulnérables à la magie (en leur faisant perdre un nombre fixe de points de vie (par exemple 1.f) ;
- et avec les flammes magiques en les éteignant.

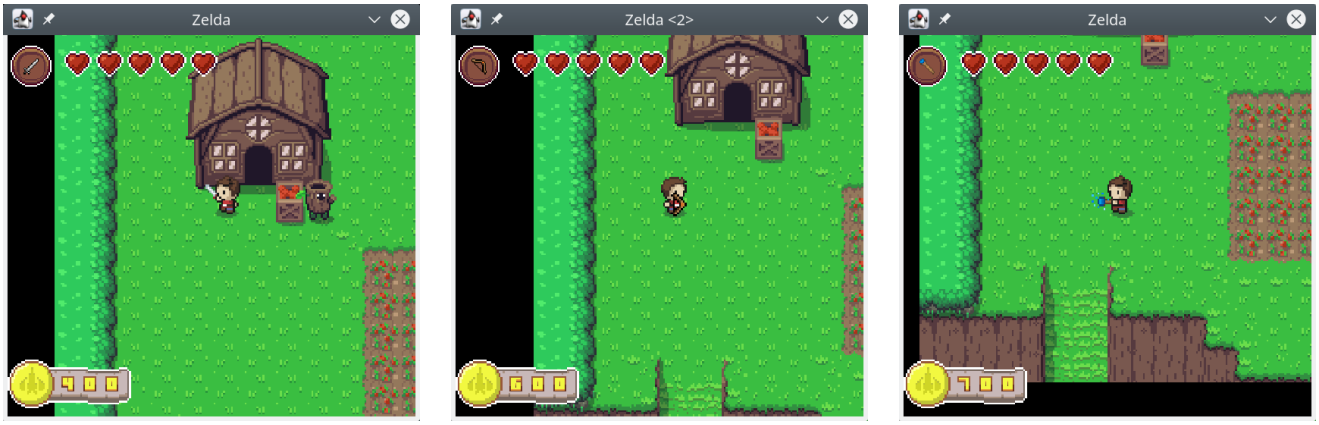


FIG. 7 : Etats « attaque » pour le personnage (avec visuels spécifiques)

Dans le premier cas, le projectile est stoppé dans sa course dès qu'il a interagi.

Utilisation des armes par le personnage La méthode qui gère l'utilisation des équipements de `ARPGPlayer` 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'arc, de l'épée et du bâton magique.

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 couper de l'herbe).

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

La transition d'un état à l'autre est conditionnée par l'appel à la méthode qui gère l'utilisation des équipements (invocable par la barre d'espace) et bien sûr par l'équipement sélectionné. Par exemple, le personnage transite à l'état « attaquant avec son arc », lorsque le joueur a sélectionne l'arc comme équipement courant et qu'il a manifesté le souhait d'utiliser cet équipement en appuyant sur la « barre d'espace ».

Le passage à un des modes d'attaque ne doit pouvoir se faire que si le personnage était au préalable en mode inoccupé (il ne peut pas passer directement du monde attaquant avec son épée au mode attaquant avec son arc par exemple). Ce passage doit se traduire par un visuel spécifique : *`"zelda/player.sword"`*, *`"zelda/player.bow"`* ou *`"zelda/player.staff_water"`* (voir la figure 7).

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 barre d'espace le personnage retourne à l'état inoccupé dès que l'interaction demandée s'est terminée ; l'interation comprenant le temps d'animation de l'attaque à l'épée).
- Une attaque à l'arc se termine lorsque le personnage a essayé de lancer une flèche : il doit pour cela disposer d'un arc. L'essai se solde par l'envoi d'un flèche dans le cas où

il en a une. La flèche tirée doit alors disparaître de l'inventaire. S'il n'y a pas de flèche on verra simplement le personnage lever son arc mais il ne se passera rien.

- une attaque au bâton magique se termine lorsque le personnage a envoyé un `MagicWaterProjectile`.

Il vous est demandé de modifier la méthode de mise à jour de `ARPGPlayer` de sorte à intégrer la transition vers de nouveaux états (et conformément aux descriptions ci-dessus). Le `ARPGPlayer` 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) ; il continue aussi à pouvoir interagir à distance sur demande avec la touche `E`;
- ne puisse plus couper l'herbe par simple interaction à distance, mais en utilisant son épée ;
- qu'il puisse faire exploser les bombes en les attaquant à l'épée ;
- et qu'il puisse infliger des dommages physiques à tous les monstres vulnérables à ce type d'attaque.

4.2.4 Tâches

Il vous est demandé de coder les concepts décrits précédemment conformément aux spécifications et contraintes données. Pour tester vos développements, vous doterez le personnage principal de tous les équipements codés (épée, bâton magique, arc, plusieurs flèches etc.).

Vous vérifierez alors que :

1. le personnage principal continue à pouvoir se déplacer sous contrôle du clavier lorsqu'il est inoccupé ;
2. qu'il peut transiter à la demande dans différents modes d'attaque après sélection des équipements arc, bâton magique ou épée ;
3. que l'utilisation des équipements choisis se fait via la barre d'espace et que cela se traduit par le visuel approprié ;
4. que le personnage principal peut désormais couper l'herbe avec son épée ou en tirant des flèches dessus (et plus via un simple contrôle à distance) ;
5. qu'il peut lancer des flèches (quand il a un arc et au moins une flèche) ;
6. qu'il peut lancer des projectiles magiques en utilisant le bâton magique ;
7. qu'il peut détruire les monstres troncs en leur infligeant des dommages physique (soit en tirant des flèches soit en les frappant de son épée) ;

8. qu'il peut infliger des dommages physiques ou magiques aux crânes volants (bâton magique, flèches, épée) ;
9. qu'il peut infliger des dommages magiques aux seigneurs des ténèbres (bâton magique), et uniquement des dommages magiques ;
10. qu'il peut éteindre les flammes magiques au moyen des projectiles magiques ;
11. que le seigneur des ténèbres lâche un clé en mourant après l'affichage du petit nuage accompagnant la disparition des monstres ;
12. que les flèches disparaissent proprement de l'inventaire après avoir été utilisées ;
13. et que les projectiles lancés détruisent ce qui se trouvent sur leur passage tant qu'ils n'ont pas arrêté leur course et qu'ils disparaissent d'eux même une fois la course terminée.

4.3 Validation de l'étape 3

Pour valider cette étape, toutes les vérifications des sections 2.7, 3.2.5, 3.3.1, 3.4.1, 4.1.2, 4.1.4, 4.1.6 et 4.2.4 doivent avoir été effectuées.

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

5 Extensions (étape 4)

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

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

Vous pouvez coder vos extensions dans le jeu `ARPG` ou dans un nouveau jeu utilisant la logistique que vous avez mises en place dans les étapes précédentes.

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

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

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

Notez qu'il existe dans la maquette pour des aires additionnelles)

5.1 Préparation aux extensions

Dans certaines des parties précédentes, du code a parfois été ajouté uniquement dans le but de faciliter les tests. Ce code n'est pas toujours compatible avec un jeu dans sa version finale. Il s'agit :

- du code permettant d'ajouter des articles « en dur » dans l'inventaire du personnage principal ;
- des contôles permettant de créer des monstres à la demande dans l'aire `RouteChateau`.

Ces parties doivent être préservées pour faciliter la correction. Si elles vous dérangent car non compatibles avec votre jeu, vous procéderez comme suit :

1. définissez dans le paquetage `game.arpg` l'interface suivante :

```
public interface Test {
public final static boolean MODE = true;
}
```

2. faites en sorte que le code de test ne soit exécuté que si `Test.Mode` vaut `true`. Vous pourrez alors mettre dans le cas `false`, votre code personnel censé remplacer le code de test (et il nous suffira lors de la correction d'attribuer les bonnes valeurs à `Test.Mode` selon ce que l'on veut tester/examiner).

3. documentez ce point dans votre `README`.

5.2 Nouveaux acteurs ou extensions du joueur

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

- des personnages vendeurs disposant d'un inventaire et chez qui le personnage principal pourra 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'article de chaque type et de sélectionner un article (par exemple celui que l'on veut acheter) (~5pts sans menu et 10 avec) :

  Notez que le menu associé à un inventaire peut être codé même sans personnage vendeur.

- modélisation d'un système de ressources (or, argent, bois, nourriture, doses de soins etc.) ;(~4pts)

- complexifier la notion d'inventaire pour pouvoir disposer de poche (une poche pour les armes une autre pour la nourriture ou les doses de soin etc.) ;(~2 à 4pts)

- divers acteurs pouvant servir de signaux (orbes, torches, plaques de pression, leviers) ; (~4pts)

- acteurs de décors animés (par exemple chutes d'eau) ; (~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 sortes de personnages avec des modalités de déplacement et de comportement spécifiques ; pouvant être hostiles ou amicaux à l'égard du joueur ; (~4pts/personnage)
- passages permettant de se téléporter vers certains niveaux/endroits du jeu ; (~2pts)
- créer un nouveau mode de déplacement pour votre personnage (en courant, nageant etc.) avec une adaptation adéquate des sprites/animations ; (~3pts)
- créer un personnage suiveur à l'image du Pikachu de Red dans pokémon jaune ; (~5pts)
- 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. (~5pts)
- implémenter une amélioration de la classe `Background` qui serait animée ;(~2pts)
- complexifier les comportements (par exemple doter les personnages de périodes d'immunité ~2pts) ;
- ajouter de nouveaux type de cellules avec des comportements appropriés (eau, glace, feu, etc.) ; (~2pts/cellules)
- implementer un cycle jour/nuit qui pourrait servir de signal ou qui conditionnerait le comportement du personnage (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 ; (~5pts)
- ajouter de nouveaux contrôles avancés (interactions, actions, déplacements, etc.) ; (~2pts/control)
- ajouter des événements aléatoires (décors, sigaux, etc.) ; (~4pts)
- ajouter des dialogues à choix multiples ;(~4pts)
- ajouter un objet `Box` ou `Safe`, dont l'ouverture serait dirigée par un signal aurait pour contenu un ou plusieurs objets ; (~3pts)
- etc.

5.3 Dialogues, pause et fin de jeu

Les jeux de type ARPG permettent habituellement au personnage de dialoguer avec certains acteurs. Si une composante énigme est ajoutée, le jeu peut aussi rapidement devenir injouable sans quelques indications dispensées à bon escient. Introduire la possibilité de recourir à quelques dialogues peut donc être un plus fort agréable. Par ailleurs, mettre en place un

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

Quelques indications vous sont fournies ci-dessous pour aller dans ce sens.

5.3.1 Dialogues (~3 à 5 points)

(nombre de points dépendant de l'effort réalisé pour mettre en place les dialogues).

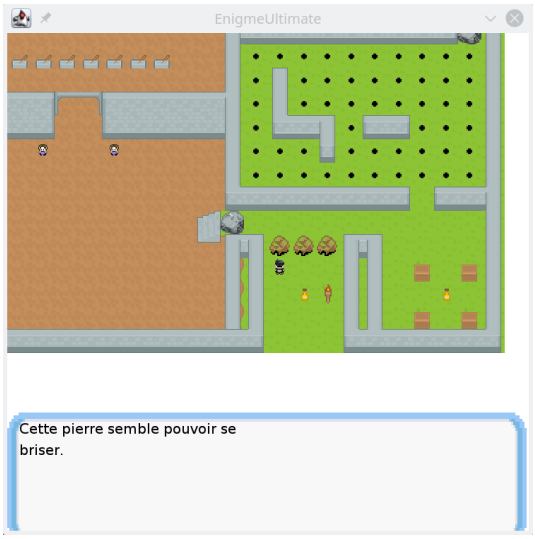
Il est possible d'attacher des textes aux acteurs (un peu comme nous l'avons fait avec `SimpleGhost`). Vous pouvez initier des dialogues dans certaines situations. Par exemple, lorsque le personnage demande une interaction avec un acteur, une indication peut alors s'afficher pour mettre le joueur sur la piste de ce qu'il faut faire pour que l'interaction devienne possible.

Des textes prédéfinis peuvent être stockés dans des fichiers en format `.xml` comme celui fourni dans `res/strings/enigme_fr.xml`. A titre d'exemple :

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

retournera la chaîne de caractères `"utilisez une CastleKey !"`. Le fichier `xml` associé au jeu est défini dans `Play`.

Alternativement, et de façon un peu plus proche de ce qui se fait dans les jeux de type *GameBoy*, vous pouvez exploiter la classe fournie `Dialog` (de `rpg.actor`) et obtenir des visualisations de dialogues ressemblant à ceci :



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

La notion d'aire peut-être exploitée pour introduire la mise en pause des jeux. Sur requête du joueur, le jeu peut basculer en mode pause puis rebasculer en mode jeu. Vous pouvez

également introduire la gestion de la fin de partie (si le personnage a atteint un objectif ou a été battu par exemple).

En réalité, la base que vous avez codée peut être enrichie à l'envie. Vous pouvez aussi laisser parler votre imagination, et essayer vos propres idées. S'il vous vient une idée originale qui vous semble différer dans l'esprit de ce qui est suggéré et que vous souhaitez l'implémenter pour le rendu ou le concours (voir ci-dessous), il faut la faire valider avant de continuer (en envoyant un mail à CS107@epfl.ch).

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 !

5.4 Validation de l'étape 4

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

6 Concours

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 ».[5]

Si vous souhaitez concourir, vous devrez nous envoyer d'ici au **19.12 à 9 :00** un petit "dossier de candidature" par mail à l'adresse **cs107@epfl.ch**. Il s'agira d'une description de votre jeu et des extensions que vous y avez incorporées (sur 2 à 3 pages en format .pdf avec quelques copies d'écran mettant en valeur vos ajouts).

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