10 / 05 / 2020

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Rapport de projet PAF

PAF Dungeon

Groupe

GOMEZ Pierre
DUTRA Enzo

Table des matières

Introduction

L'objectif de ce projet est de développer un jeu vidéo sûr en utilisant un langage fonctionnel et une programmation sûre. Dans le cadre de l'UE de PAF, nous avons utilisé le langage Haskell avec le resolver `lts-12.26`.

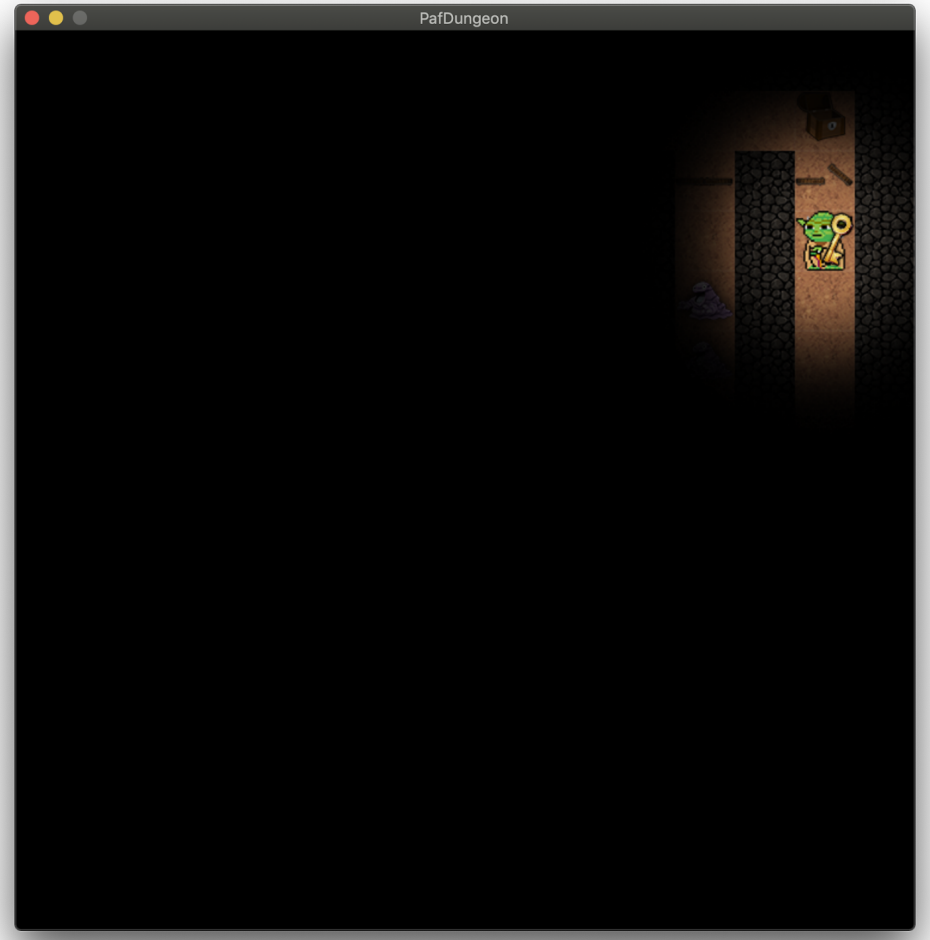
Ce projet a été construit de manière sûre, c'est à dire qu'à chaque type est associé un ou plusieurs invariants et à chaque fonction des pré-conditions et post-conditions.
Nous avons mis en place des tests basés sur les propriétés via QuickCheck.

Nous allons, dans ce document, faire une succincte présentation du jeu développé, des règles du jeu, ainsi que du gameplay. Nous détaillerons les propositions implémentées avec les invariants de types et les proposition d'opérations.
Après avoir détaillé les tests, nous présenterons le fonctionnement du jeu de manière générale ainsi que certains points intéressants.
Nous parlerons également des extensions implémentées, puis nous finirons par quelques utilisations des spécificités du langage utilisé ainsi que des principes de la programmation fonctionnelle.

Présentation du jeu

Le jeu se nomme “PAF Dungeon”, il s’agit d’un dungeon crawler à la 3e personne constitué d’un niveau de 15x15 cases.

Voici une capture d'écran du jeu en exécution :



Histoire du jeu :

Vous serez un chevalier jedi historique que sa réputation précède de manière remarquable et que le bas monde connaît sous le nom de “Programmation avancée fonctionnelle” pour ses baffes légendaires (dans le milieu, on le surnomme PAF). Ce chevalier possède un donjon grâce à un héritage de ses illustres ancêtres mais c’est également un scientifique devenu fou après le confinement dû au coronavirus.

Le grand PAF a fait naître des monstres in-vitro dans les laboratoires secrets du LIP6 et les enferme dans son donjon pour leur administrer ses baffes légendaires à 360°.

Pour se venger de cette constante oppression, les monstres ont décidé de cacher les clés de la sortie, de fermer des portes et d’éteindre la lumière afin d’attaquer sournoisement le scientifique.

Fou de rage, vous allez devoir leur infliger une bonne correction et récupérer vos clés pour quitter votre donjon que vous ne contrôlez visiblement plus...

Lancement du jeu

Se placer dans la racine du projet et taper : `stack run`
Attention, afin d'éviter à avoir à télécharger une nouvelle version de GHC, vous devez être sous la version lts-12.26. Il faut également avoir installé SDL2 et SLD2-image.

Règles du jeu

Le joueur démarre sur l'entrée du niveau et l'objectif est d'arriver à la sortie sans mourir.
Il pourra rencontrer sur son chemin des monstres qu'il devra tuer ou des portes fermées qu'il devra ouvrir.
Pour gagner, il faut récupérer dans un coffre la clé qui ouvre les portes pour enfin atteindre la sortie.

Gameplay

Le champ de vision du joueur est relativement réduit (un cercle de 2 cases de rayon).

Le déplacement se fait via les touches suivantes :
- ➢ Z : haut
- ➢ Q : gauche
- ➢ S : bas
- ➢ D : droite

Pour attaquer les monstres aux alentours (8-adjacent)
- ➢ SPACE : attaque

Pour utiliser les portes aux alentours (4-adjacent, la diagonale n'est donc pas comprise)
- ➢ F : utiliser

Pour quitter le jeu :
- ➢ ESC : quitter

Attention : pour que l'input soit correctement pris en compte, n'appuyez pas sur les touches trop brièvement.

Pour récupérer un coffre, il faut passer dessus. Il vous donnera une clé permettant d'ouvrir les portes fermées, ce qui est indispensable pour sortir.

Si vous restez éloigné des monstres, ils vaquent à leur occupations en se déplaçant aléatoirement.
Si vous vous approchez trop d'eux, il ont beaucoup plus de chance de tenter de vous attaquer.
Neutralisez-les rapidement avant qu'ils ne vous tuent (attention ils sont assez puissants).

Propositions implémentées

Invariants de types

Carte

`prop_positiveCoord_inv :: Coord -> Bool`
⇒ Cet invariant vérifie que les coordonnées sont positives

`prop_allCoordsInBounds_inv :: Carte -> Bool`
⇒ Cet invariant vérifie que toutes les coordonnées des cases sont entre les bornes données par la largeur et hauteur de la carte.

`prop_allCoordInCarte_inv :: Carte -> Bool`
⇒ Cet invariant vérifie que toutes les coordonnées avec comme borne la largeur et hauteur correspondent bien à des cases dans la carte.

`prop_entranceExit_inv :: Carte -> Bool`
⇒ Cet invariant vérifie qu'une carte donnée ne contient une seule entrée et une seule sortie.

`prop_surroundedByWalls_inv :: Carte -> Bool`
⇒ Cet invariant vérifie qu'une carte donnée est entièrement entourée par des murs

`prop_doorsFramedByWalls_inv :: Carte -> Bool`
⇒ Cet invariant vérifie que toutes les portes d'une carte donnée sont encadrées par des murs.

`prop_Coord_inv :: Coord -> Bool`
⇒ Cet invariant vérifie que tous les invariants sont respectés pour les coordonnées.

Environnement

`prop_allCoordsPositive_inv :: Envi -> Bool`
⇒ Cet invariant vérifie que toutes les coordonnées de l’environnement sont positives.

`prop_oneUncrossableMobPerCase_inv :: Envi -> Bool`
⇒ Cet invariant vérifie qu'il n'y a qu'une entité infranchissable par case dans l’environnement.

`prop_positiveStats_inv :: Envi -> Bool`
⇒ Cet invariant vérifie que les statistiques de toutes les entités de l'environnement sont positives.

`prop_uniqueIds_inv ::  Envi -> Bool`
⇒ Cet invariant vérifie que toutes les entités de l'environnement ont un identifiant unique.

`prop_Envi_inv :: Envi -> Bool`
⇒ Cet invariant regroupe tous les invariants de l'environnement.

`prop_Entite_inv :: Entite -> Bool`
⇒ Cet invariant vérifie les statistiques de l'entité en fonction de son type.

Modèle

`prop_Modele_inv :: Modele -> Bool`
⇒ Ceci est l'invariant du type modèle.
Il vérifie que les invariants de la carte et de l'environnement qu'il possède sont valides.

Moteur

`prop_Etat_inv :: Etat -> Bool`
⇒ Cet invariant vérifie que le nombre de tours est positif et que l'invariant du modèle qu'il contient est bien valide.

Propositions d'opérations

Carte

getCase
`prop_getCase_pre :: Coord -> Carte -> Bool`
⇒ Vérifie que la coordonnée est bien dans les bornes de la carte et que les coordonnées sont bien une clé dans table associative de la carte

```
prop_getCase_post :: Coord -> Carte -> Bool
```
⇒ Etant donné que cette fonction retourne une valeur, il n'y a pas de modification à évaluer après l'appel

isTraversable

```
prop_isTraversable_pre :: Case -> Int -> Bool
```
⇒ vérifie que le clearance level est bien supérieur à 0

```
prop_isTraversable_post :: Case -> Int -> Bool
```
⇒ Etant donné que cette fonction retourne une valeur, il n'y a pas de modification à évaluer après l'appel

editCase

```
prop_editCase_pre :: Coord -> Case -> Carte -> Bool
```
⇒ Vérifie que les coordonnées sont correctes et qu'elles se trouvent bien dans la carte.

```
prop_editCase_post :: Coord -> Case -> Carte -> Bool
```
⇒ Vérifie qu'il n'y a eu aucun changement en dehors de la case à éditer et que la case a bien été changé par ce que l'on souhaitait.

openDoor

```
prop_openDoor_pre :: Coord -> Carte -> Bool
```
⇒ Vérifie que la case aux coordonnées demandées est bien une porte.
A noter : on ne met pas de précondition sur l'état de la porte avant de l'ouvrir (ouvrir une porte ouverte n'a pas d'effet)

```
prop_openDoor_post :: Coord -> Carte -> Bool
```
⇒ Vérifie qu'il n'y a eu aucun changement en dehors de la porte à ouvrir et que la porte a bien été ouverte comme on le souhaitait.

closeDoor

```
prop_closeDoor_pre :: Coord -> Carte -> Bool
```
⇒ Vérifie que la case aux coordonnées demandées est bien une porte.
A noter : on ne met pas de précondition sur l'état de la porte avant de la fermer (fermer une porte fermée n'a pas d'effet)

```
prop_closeDoor_post :: Coord -> Carte -> Bool
```
⇒ Vérifie qu'il n'y a eu aucun changement en dehors de la porte à fermer et que la porte a bien été fermée comme on le souhaitait.

Environnement

setEntity

```
prop_setEntity_pre :: Entite -> Coord -> Envi -> Bool
```

⇒ Vérifie que les invariants de l'entité et de l'environnement sont bien respectés et que l'entité à ajouter ne se trouve pas déjà dans l'environnement (via son id).

```
prop_setEntity_post :: Entite -> Coord -> Envi -> Bool
```

⇒ Vérifie qu'on retrouve bien l'entité par id aux nouvelles coordonnées.

removePvie

```
prop_removePvie_pre :: Entite -> Int -> Bool
```

⇒ vérifie que l'invariant de l'entité est valide et que les dégâts à infliger sont non nuls.

```
prop_removePvie_post :: Entite -> Int -> Bool
```

⇒ Vérifie que les invariants de l'entité sont toujours valides.

rmEntById

```
prop_rmEntById_pre :: Int -> Envi -> Bool
```

⇒ Vérifie que l'id fourni est correcte et que l'entité à retirer se trouve bien dans l'environnement.

```
prop_rmEntById_post :: Int -> Envi -> Bool
```

⇒ Vérifie que l'id fourni est correcte et que l'entité que l'on a retiré ne se trouve plus dans l'environnement.

bougeById

```
prop_bougeById_pre :: Int -> Coord -> Envi -> Carte -> Bool
```

⇒ vérifie que les coordonnées auxquelles on veut placer notre entité sont bien dans la carte, qu'elles sont bien dans l'environnement, que l'id donné est correcte et qu'il correspond bien à une entitée.

```
prop_bougeById_post :: Int -> Coord -> Envi -> Carte -> Bool
```

⇒ Vérifie que l'entité n'a pas été dupliquée et qu'elle n'a pas été supprimée pendant son déplacement.

setAttackingTrue

```
prop_setAttackingTrue_pre :: Entite -> Envi -> Coord -> Bool
```

⇒ Vérifie que les invariants de tous les arguments de la fonction sont bien respectés.

```
prop_setAttackingTrue_post :: Entite -> Envi -> Coord -> Bool
```
⇒ vérifie que l'entité est bien en mode attaque.

cleanUpEntities

```
prop_cleanUpEntities_pre  :: Envi -> Bool
```
⇒ Vérifie que les invariants de l'environnement à nettoyer sont bien respectés.

```
prop_cleanUpEntities_post  :: Envi -> Bool
```
⇒ Vérifie qu'aucune entité qui peut attaquer n'est en mode attaque (avant chaque tour on sort toutes les entités du mode attaque et celles qui attaqueront re-entreront en mode attaque).

openChest

```
pre_openChest_pre :: Int -> Envi -> Bool
```
⇒ Vérifie que l'id fourni est valide et que les invariants de l'environnement sont bien valides.

```
pre_openChest_post :: Int -> Envi -> Bool
```
⇒ Vérifie que le coffre que l'on souhaitait ouvrir est désormais bien ouvert.

giveKeyToPlayer

```
prop_giveKeyToPlayer_pre :: Envi -> Bool
```
⇒ Vérifie que les invariants de l'environnement sont bien valides.

```
prop_giveKeyToPlayer_post :: Envi -> Bool
```
⇒ Vérifie que le joueur possède bien une clé.

Modèle

bouge

```
prop_bouge_pre :: Modele -> Entite -> Coord -> Bool
```
⇒ Vérifie que les pré-conditions de bougeById sont bien vérifiées car cette fonction sert de pont avec cette dernière afin de simplifier le code dans le reste du fichier.

```
prop_bouge_post :: Modele -> Entite -> Coord -> Bool
```
⇒ Vérifie que les post-conditions de bougeById sont bien vérifiées car cette fonction sert de pont avec cette dernière afin de simplifier le code dans le reste du fichier.

prévoir

`prop_prevoir_pre :: Entite -> Envi -> Bool`

⇒ vérifie que les invariants de l'entité ainsi que ceux de l'environnement fournies sont bien valides.

A noter: la fonction n'apporte aucune modification qu'il faudrait vérifier à l'aide d'un post

gameStep

`prop_gameStep_pre :: RealFrac a => Modele -> Keyboard -> a -> Bool`

⇒ Vérifie que les invariants du modèle sont bien respectés.

A noter: coder une post-condition pour cette fonction reviendrait quasiment à recoder tout le fichier, ce qui serait relativement peu pertinent.

Tests

Pour exécuter les tests, se placer dans la racine du projet et taper **`stack test`**. Les tests basés sur les propriétés (property-based testing) seront exécutés sur un set de données prédéfini.

Tests basés sur les propriétés

Pour les tests basés sur les propriétés, nous avons mis en place des invariants pour tous les types créés que nous avons testés pour une carte et un environnement donné.

A chaque module est associé un fichier HSpec contenant les tests pour chaque invariants du module et finissant par un test qui regroupe l'entièreté des invariants.

Rapport :

Structure du jeu

Le jeu se décompose en 8 modules et un fichier Main.
Le fichier Main.hs s'occupe de charger les différents sprites, et gérer la boucle du jeu où le moteur est appelé et les sprites sont affichés.

Les différents modules et leurs rôles sont les suivants:

- **SpriteMap** : Permet de charger des sprites à partir de fichiers.
- **Sprite** : Permet de positionner des sprites.
- **TextureMap** : Permet le rendu des textures sur une fenêtre.
- **Keyboard** : Permet la capture d'événements du clavier.
- **Carte** : Définit les types "Case", "Coord", et le type "Carte" qui est une map (Coord,Case), ainsi que des fonctions permettant la création d'une carte, sa lecture et d'effectuer des actions dessus.
- **Environnement** : Définit les types "Entite" et "Envi" qui est une map (Coord,[Entite]) ainsi que des fonctions permettant la création d'un environnement, sa lecture et d'effectuer des actions dessus.
- **Modele** : Définit le type "Modele" composé de la carte actuelle, de l'environnement actuel, du générateur aléatoire actuel, du journal de tour et de l'état du clavier.
- **Moteur** : Définit le type "Etat" qui est l'état du jeu, soit Gagné, soit Perdu, soit un tour qui contient le modèle du tour.

Carte

Pour pouvoir utiliser correctement la map contenue dans la carte, il faut que les clés de la map soientt une instance de la typeclass Ord, dans notre cas les clés sont de type Coord donc nous avons instancié Ord pour Coord. On défini ici en quoi une coordonnée est plus petite qu'une autre.

```haskell
-- définition de la manière d'ordonner des coordonnées
instance Ord Coord where
   compare c1 c2
       | (cy c1) < (cy c2) = LT
       | ((cy c1) == (cy c2)) && ((cx c1) < (cx c2)) = LT
       | ((cy c1) == (cy c2)) && ((cx c1) == (cx c2)) = EQ
       | otherwise = GT
```

Pour pouvoir créer une Carte à partir d'une String nous avons instancié Read pour Carte
À cette fin nous avons également créé un type de case Undefined permettant de faire échouer la précondition prop_createCarte_pre si des caractères inconnus sont lus.

De même pour afficher une carte dans le terminal, il faut qu'elle instancie Show

```
instance Show Carte where
   show = toString
instance ToString Carte where
   toString c = foldl (\accstr cur ->
       accstr ++ (toStringCarteAux (cartel c) cur) ) "" (listFromCarte c)
```

Nous avons créé la proposition noChangesExceptAtCoord

```
--vérifie si deux carte sont identiques, sauf a la coordonnee donnee
noChangesExceptAtCoord :: Carte -> Coord ->  Carte -> Bool
noChangesExceptAtCoord carte coord post_fonction =
       foldl (\boolAcc ((co1, ca1) , (co2, ca2)) ->
           boolAcc && (if ((co1 /= coord) && (ca1 == ca2)) then True else False ) )
       True (zip (listFromCarte carte) (listFromCarte post_fonction))
```

Cette proposition permet de vérifier que deux cartes sont identiques case par case, à l'exception de la coordonnée fournie. On l'utilise dans beaucoup de post-conditions afin de s'assurer que les autres cases de la carte n'ont pas changé lors d'une opération sur une case.

Environnement

L'environnement maintient une map dont les clés sont les coordonnées et les valeurs sont une liste d'entités à cette coordonnée.
Une fonction très utilisée dans le jeu est trouveId qui permet de récupérer une entité et ses coordonnées à partir de son identifiant

```
trouveId :: Int -> Envi -> Maybe (Coord, Entite)
```

Une pré-condition indispensable à cette opération est que les identifiants soient uniques dans tout l'environnement

Modèle

Le modèle représente l'état d'un tour de jeu, avec l'image courante de la carte et de l'environnement.
C'est lui qui s'occupe à chaque tour de faire exécuter une action à chaque entité de l'environnement.
Nous n'appelons pas une méthode "tour" exécutant une action pour chaque entité, nous avons, à la place, une fonction récursive travaillant sur la liste des entités de l'environnement en modifiant le modèle au fur et à mesure.

```
-- fonction récursive permettant à chaque entitée d'accomplir une tâche
gameStepAux :: Modele -> Keyboard -> [Entite] -> Modele
gameStepAux modele kbd (entity:entities) = case E.trouveId (E.idn entity) (envi modele) of
 Just (c, E.Joueur _ _ _ _ _ _) ->
      gameStepAux (handlePlayerActions modele entity kbd c) kbd entities
 Just (c, E.Monstre _ _ _ _ _) ->
      gameStepAux (decider (prevoir entity (envi modele)) modele entity) kbd entities
 Just (c, E.Coffre _ _ False) ->
      gameStepAux (handleChest c entity modele) kbd entities
 _ -> gameStepAux modele kbd entities
gameStepAux modele kbd [] = modele
```

Le Joueur est une entité contrôlable qui réagit aux touches du clavier tandis que les Monstres sont des entités qui choisissent aléatoirement dans une liste pondérée une action à exécuter.
La liste pondérée des actions est contextuelle, en effet, si le monstre détecte que le Joueur est à moins de deux case de distance, sa liste pondéré contiendra beaucoup de chance d'attaque, alors qu'en cas d'absence de joueur, ils ne feront que se déplacer.

```
    case (E.getPlayerCoord env, E.entityCoord entity env) of
      (Just coP, Just coM) -> ((abs ((C.cx coP) - (C.cx coM))) <= 2)
                           && ((abs ((C.cy coP) - (C.cy coM))) <= 2)
      (_, _) -> False )
  then [(1, Haut ),(1, Bas ),(1, Droite ),(1, Gauche ), (2, Rien ), (0, Uti ), (8, Atk )]
  else [(1, Haut ),(1, Bas ),(1, Droite ),(1, Gauche ), (2, Rien ), (0, Uti ), (0, Atk )]
 otherwise -> [(1, Haut ),(1, Bas ),(1, Droite ),(1, Gauche ),(1, Uti ), (0, Atk ), (1,
Rien )]
```

Pour tirer aléatoirement dans cette liste, nous utilisons un couple (seed :: Int, gen StdGen) stocké dans le modèle.
La fonction pickOrder s'occupe de choisir un ordre le plus aléatoirement possible et d'enregistrer une nouvelle seed de générateur pour que la prochaine utilisation soient totalement indépendante.

```
pickOrder :: [Ordre] -> Modele -> (Ordre, Modele)
pickOrder orders modele =
   (orders!!((R.randomRs (0,(length orders) - 1) (snd (gene modele))
         )!!((R.randomRs (1,99999)               (snd (gene modele))
         )!!1))
   , modele { gene = ( (fst (gene modele)) + 1, R.mkStdGen (fst (gene modele))) } )
```

Pour arriver à un ordre "le plus aléatoire possible", nous avons effectué deux "imbrications" de tirages aléatoire. Nous avons en effet remarqué qu'en plus du premier élément d'une liste aléatoire qui tombait systématiquement sur le même nombre, le reste des tirages ne semblait pas réellement aléatoire (on observait donc des patterns très courts et marqués dans les déplacement des mobs).
A partir de 3 "imbrications" de tirages aléatoire (tirage aléatoire d'un index dans une liste aléatoire), les déplacements nous ont semblé plus naturels.

Moteur

Le moteur maintient l'Etat actuel du jeu, c'est à dire s'il est Gagné, Perdu ou, s'il est en cours, le modèle actuel du tour
C'est la fonction etat_tour qui appelle le calcul du nouveau tour du modele (Modele.gamestep) et qui utilise ce nouveau modèle pour décider du nouvel Etat du moteur. Ainsi, si après le calcule du tour, les coordonnées du joueur et de la sortie sont égales, l'état passe à Gagné, mais si les points de vies du joueur sont égaux à 0, l'état passe à Perdu. Autrement l'état est enregistré avec le nouveau modèle

Extensions

Nous avons implémenté un certain nombre d'extensions parmis celles proposées afin d'arriver à un jeu ayant un intérêt ludique.

Combats

Le joueur peut attaquer des monstres. Le fonctionnement des attaques est détaillé plus bas (la fonction "attack").

Lorsque le joueur attaque, il se met en “mode” attaque et son sprite change pour le montrer en train d’attaquer.

Ennemis dangereux

Les monstres ont un comportement variable en fonction de leur distance par rapport au joueur. Ils sont passifs lorsque le joueur est loin et agressifs lorsque le joueur est détecté à proximité. Les attaques des monstres utilisent exactement les mêmes fonctions que celles des joueurs avec également un mode attaque.

Coffre et Clé

Certaines portes peuvent être fermées, et on peut les ouvrir en récupérant une clé dans un coffre. Le coffre est une entité qui s’ouvre lorsqu'un qu’un joueur passe dessus, leur interaction donne la clé (un attribut booléen) au joueur.

Brouillard

Le brouillard réduit le champ de vision du joueur. Il est généré à l’aide d’un sprite qui est une image complètement noir avec un halo transparent au milieu. L’image est centrée sur le joueur et se déplace en même temps que lui.

Fissure

Les murs peuvent avoir des fissures par lesquelles les monstres peuvent passer mais pas le joueur.
La gestion des “droits de passage” des entités se fait à l’aide d’un attribut “clearance” (un entier) qui est vérifié à chaque tentative de déplacement d’une entité pour savoir si il est suffisant pour aller sur un type de case donné.

Utilisation des possibilités offertes par Haskell

Les types sont la plupart du temps définis comme des types algébriques contenant des records ce qui permet de nommer les différents champs d'un type et d'avoir des getters sur ses champs.

La monade Maybe est utilisée dans tous les retours de fonctions dont la réponse n'est pas certaine, récupérer les coordonnées d'une entité par exemple, car on n'est pas certain qu'elle soit effectivement dans l'environnement. Cela permet de mettre l'accent sur la programmation sûr, et de nous forcer à anticiper le cas où l'opération n'est pas possible. C'est une sécurité qui reste présente même sans l'activation systématique des propositions des opérations: un avantage offert par Haskell.

Nous avons, quand nos fonctions nous le permettaient tiré partie de l'eta contraction offerte par Haskell, comme lors de l'instanciation de typeclass Show pour les types algébriques Carte, Envi, Modele et Moteur.

Nous avons également beaucoup utilisé de fonctions de parcours (fold) et de foncteurs (map) en recourant à des lambdas qui nous ont souvent permis de réduire le nombre d'arguments qu'une fonction séparée aurait requis.

La concaténation de listes n'a pas été prépondérante dans notre programme, elle a cependant été fort utile pour la capacité d'attaque des entités.

```haskell
-- Attaque toutes les cases 8-adjacentes
attack :: Modele -> Entite -> Coord -> Modele
attack modele entity c = (\(_, m@(Modele _ env _ _ _) ) -> m { envi = (E.setAttackingTrue
entity env c) } ) (foldl attackAux (50, modele) (case ( -- ici, l'attaque fait 2 de dégâts
 (E.getEntitiesAtCoord (C.Coord (C.cx c) ((C.cy c) + 1)) (envi modele) )
 `mml` (E.getEntitiesAtCoord (C.Coord (C.cx c) ((C.cy c) - 1)) (envi modele) )
 `mml` (E.getEntitiesAtCoord (C.Coord ((C.cx c) + 1) (C.cy c)) (envi modele) )
 `mml` (E.getEntitiesAtCoord (C.Coord ((C.cx c) - 1) (C.cy c)) (envi modele) )
 `mml` (E.getEntitiesAtCoord (C.Coord ((C.cx c) + 1) ((C.cy c) + 1)) (envi modele) )
 `mml` (E.getEntitiesAtCoord (C.Coord ((C.cx c) + 1) ((C.cy c) - 1)) (envi modele) )
 `mml` (E.getEntitiesAtCoord (C.Coord ((C.cx c) - 1) ((C.cy c) + 1)) (envi modele) )
 `mml` (E.getEntitiesAtCoord (C.Coord ((C.cx c) - 1) ((C.cy c) - 1)) (envi modele) )) of
   Just liste -> liste
   Nothing -> []
    ))
```

Ici, on souhaite spécifier quelles cases sont concernées par l'attaque, on va donc utiliser une fonction retournant une liste de toutes les entitées à des coordonnées données et concaténer ces listes pour chaque coordonnées spécifiées.
Cette fonction fournit cependant un résultat de type Maybe, nous avons donc créé une fonction "mml" permettant de concaténer des Maybe de listes que nous avons utilisé en notation infixe :

```haskell
-- Fonction utilitaire permettant de concaténer des Maybe [a]
mml :: Maybe [a] -> Maybe [a] -> Maybe [a] -- merge maybe list
mml a b = case (a,b) of
 (Nothing, Nothing) -> Nothing
 (Nothing, Just b) -> Just b
 (Just a, Nothing) -> Just a
 (Just a, Just b) -> Just (a <> b)
```

Conclusion

Ce projet nous aura permis de programmer pour la première fois un jeu vidéo en programmation fonctionnelle avec l'abstraction qu'elle permet. Il nous aura permis de mettre en pratique un certain nombre de notions abordées en cours ainsi qu'en TME.
Il aura également été l'occasion de nous initier à la pratique de la programmation sûre. Ceci nous a en effet mené à systématiquement penser aux cas extrêmes et quel comportement adopter en cas de données inattendues.

Le fait que le projet consiste en la construction d'un jeu nous a également encouragé à pousser le développement jusqu'au bout en essayant de créer un jeu jouable et intéressant.