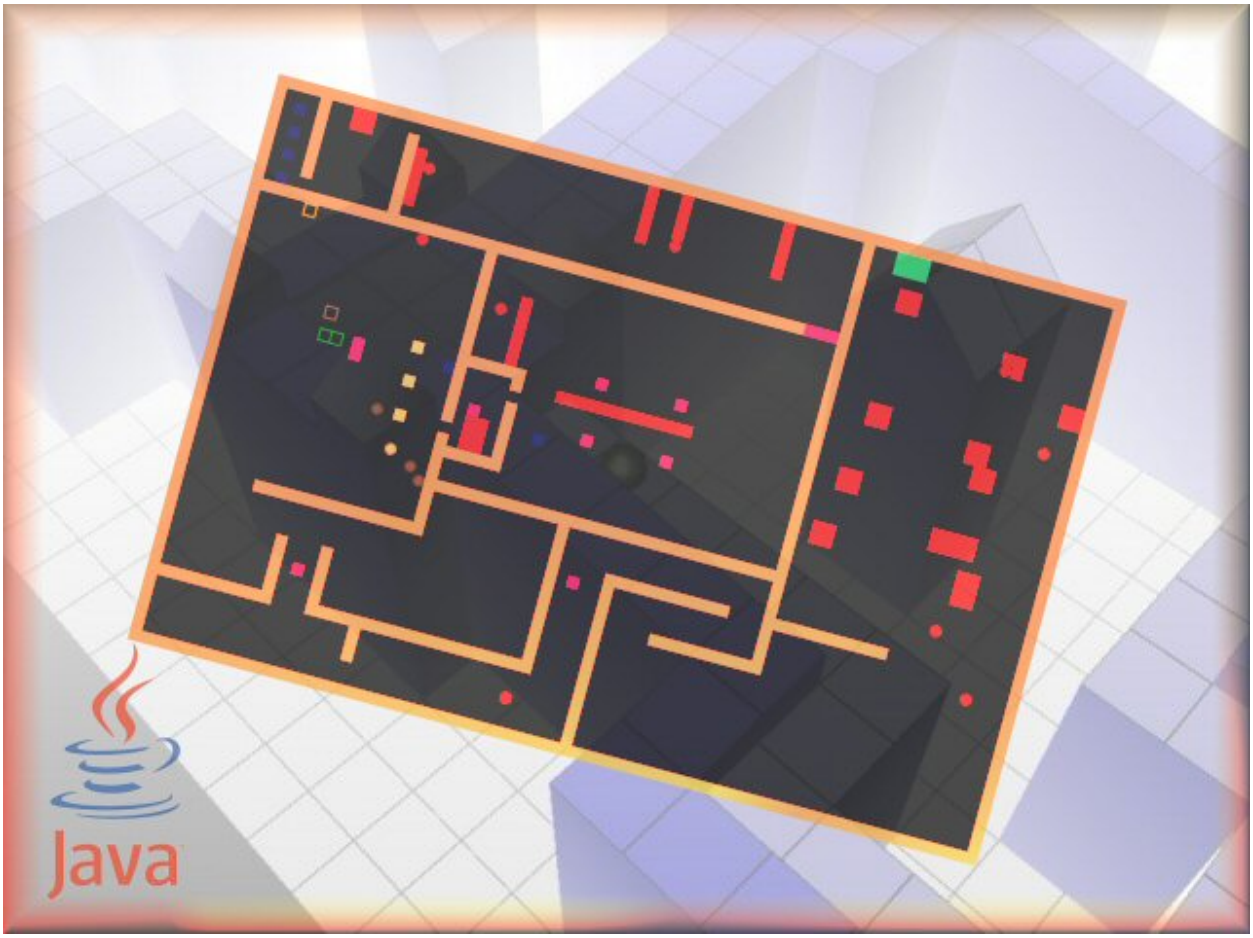
Hubert LACÔTE
Ludovic LEFOULON

LO43

Projet java : Création d'un labyrinthe



Responsable de l'UV : Franck Gechter

SOMMAIRE

Cahier des charges

Sujet

Le but de projet est de réaliser un labyrinthe qu'un personnage, manié par le joueur, devra traverser. Ce labyrinthe sera composé de murs fixes et mobiles, d'ennemis, ainsi que d'un certain nombre d'items placés dans la map par un xml.

Notre héros, qui possède une vitesse et des points de vie prédéfinis, pourra grâce à l'utilisation d'objets (armes, trousses de soin) combattre les ennemies et/ou détruire certains types d'objets.

Contraintes liées au sujet

Notre héros ne peut traverser aucun des murs présents dans le jeu. A l'inverse, les ennemis peuvent traverser les murs intérieurs. Ils (les ennemis) peuvent se déplacer dans toutes les directions verticalement, horizontalement ou encore diagonalement.

Les murs extérieurs ne peuvent pas bouger tandis que les murs intérieurs se déplacent horinzontalement ou verticalement.

Pendant toute la durée du jeu, notre héros trouvera différents objets modifiant sa vitesse ou ses points de vie.

L'implémentation de toutes les contraintes du sujet se trouvera détaillée dans la partie Interaction avec l'environnement de ce rapport.

Personnalisation

Afin de permettre à notre héros de se défendre dans ce milieu hostile, comme par exemple contre les murs mouvants, nous avons créé un objet pierre qui va les bloquer. Cette option permet ainsi au héros de se faire un passage sûr. Néanmoins, cet avantage (le déplacement des pierres) est aussi accordé aux ennemis afin de rendre le jeu plus intéressant.

Une IA a également été implémentée permettant aux ennemis de poursuivre le héros si il se situe dans leur champ de vision.

Pour combattre les ennemis, notre héros est muni de différentes armes comme un pistolet, un bouclier et des bombes. Les comportements précis seront détaillés ultérieurement, cependant leur utilisation générale est la suivante :

- Gun : permet de frapper un ennemi au corps à corps avec la crosse ou de le blesser à distance avec un balle;
- Shield : permet d'avoir des points de vie de protection contre les ennemis ;
- Bomb : permet de détruire directement toute entité qui rentre en contact avec elle.

En plus des modifications sur les points de vie ou la vitesse, un objet dans la map pourra également modifier la façon dont on commande notre personnage, inversant ainsi les commandes.

Par exemple : en appuyant sur la commande pour faire aller notre personnage à gauche on le fera en réalité aller à droite etc…

Explication de notre programme

Description de l'environnement

Les différentes parties du jeu :

- Les murs extérieurs en rose ;
- Les murs intérieurs en rouge ;
- Notre héros en blanc ;
- Les rochers permettants de bloquer les murs mouvants en magenta;
- Les cristaux en bleu ;
- Les ennemis représentés par un point rouge ;
- La sortie en cyan.

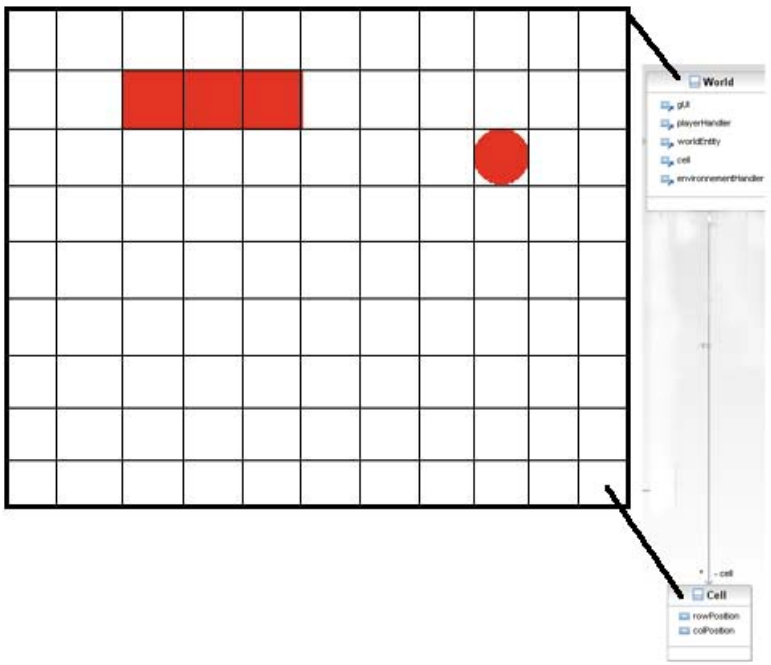
Tous les autres éléments sont des objets permettant d'aider ou d'handicaper notre personnage : armes, trousses de soin ou encore pillules pour aller plus vite ou moins vite.

Comment jouer ?

Le personnage se dirige grâce aux flèches haut, gauche, bas et droite. Dans la mesure où ce mouvement n'est pas un mouvement interdit, c'est-à-dire un mouvement permettant de traverser les murs, par exemple. La touche entrée permet d'utiliser l'arme (on peut tirer en rafale en la maintenant appuyée), et la touche tabulation permet d'en changer.

Concept

Représentation du monde



Généralités sur les systèmes multi-agents

On peut relever cinq problématiques principales lors de la création de systèmes multi-agents.

D'abord, la problématique de l'action : comment un ensemble d'agents peut agir de manière simultanée dans un environnement partagé, et comment cet environnement interagit en retour avec les agents ? Les questions sous-jacentes sont entre-autres celles de la représentation de l'environnement par les agents, de la collaboration entre agents, de la planification multi-agent.

Ensuite la problématique de l'agent et de sa relation au monde, qui est représenté par le modèle cognitif dont dispose l'agent. L'individu d'une société multi-agent doit être capable de mettre en œuvre les actions qui répondent au mieux à ses objectifs. Cette capacité à la décision est liée à un "état mental" qui reflète les perceptions, les représentations, les croyances et un certain nombre de paramètres "psychiques" (désirs, tendances...) de l'agent. La problématique de l'individu et de sa relation au monde couvre aussi la notion d'engagement de l'agent vis-à-vis d'un agent tiers.

Les systèmes multi-agents passent aussi par l'étude de la nature des interactions, comme source de possibilités d'une part et de contraintes d'autre part. La problématique de l'interaction s'intéresse aux moyens de l'interaction (quel langage ? quel support ?), et à l'analyse et la conception des formes d'interactions entre agents. Les notions de collaboration et coopération (en prenant coopération comme collaboration + coordination d'actions + résolution de conflits) sont ici centrales.

On peut évoquer ensuite la problématique de l'adaptation en terme d'adaptation individuelle ou apprentissage d'une part et d'adaptation collective ou évolution d'autre part.

En reprenant les problématiques précédentes, on peut décrire quelques éléments de l'architecture d'un système multi-agent.

Les agents doivent être dotés de systèmes de décisions et de planification à plusieurs niveaux. Les théories de la décision sont un domaine à part entière d'étude à ce sujet. Dans la catégorie des interactions avec l'environnement, un autre problème récurrent des systèmes d'agents est celui du pathfinding.

Les agents doivent être dotés d'un modèle cognitif : là aussi, plusieurs modèles existent, l'un des plus classiques étant le modèle BDI (Beliefs-Desires-Intentions). Il considère d'une part l'ensemble de croyances (Beliefs) de l'agent sur son environnement, qui sont le résultat de ses connaissances et de ses perceptions, et d'autre part un ensemble d'objectifs (Desires). En croisant ces deux ensembles, on obtient un nouvel ensemble d'intentions (Intentions) qui peuvent ensuite se traduire directement en actions.

Adaptation pour le projet

L'ensemble du sujet s'inspire d'un système multi-agents extrêmement simplifié. Celui-ci étant caractérisé par 3 composants principales : le Kernel (le monde : World), les agents (ici les ennemis, les murs ainsi que tous les objets qui se déplacent), le/les scheduler où les agents s'éxécutent en parallèle pour atteindre leur(s) objectif(s).

- Kernel : entité qui créé les différents agents et qui les ordonnance via le scheduler;
- Agents : entité possédant un état interne,un objectif et qui est autonome;
- Scheduler : entité qui va décider à quel moment doit s'éxécuter un agent.

Un système multi-agents est composé d'entités appelées agent qui sont indépendantes les unes des autres et qui possèdent un objectif. Dans notre cas et si l'on prend le cas des ennemis, leur but est clairement de détruire le héros. Les ennemis n'ont pas de tactiques de groupe afin d'attaquer notre héros.

Il est important de savoir que dans ce type de système, le comportement de l'agent n'est pas dicté par l'environnement. Cependant, les décisions de l'agent s'adaptent aux changements de son environnement(ici modélisé par la map). Chaque agent communique avec l'environnement pour savoir, à un instant donné, quelles sont les entités qui l'entourent. En sachant cela, l'agent peut alors savoir où il peut aller et finalement choisir une direction. L'agent n'a donc qu'une perception du monde très limitée. Pour lui le monde se résume à son champ de perception.

Précédemment, il a été dit qu'un agent était autonome et que tous les agents s'exécutaient en parallèle. Néanmoins, le vrai parallélisme n'existe pas. Il faut donc le simuler.

Afin de simuler le parallélisme on met en place un système d'ordonnancement.En effet, à chaque pas de temps (tour de boucle) du Kernel (classe World dans notre projet) on va demander au scheduler de mélanger aléatoirement l'ensemble de nos entités mouvantes. Une fois ce mélange aléatoire effectué on appelle les méthodes live des différents agents.

Dans notre programme, les différents ennemis ne peuvent occuper la même position. Ainsi, si on exécute les différents agents de façon linéraire, le mouvement d'un agent (ennemi) dépendra forcément de celui qui le précède et invalidera la théorie d'autonomie de l'agent. C'est pour contrer ce phénomène que l'on mélange le tableau aléatoirement.

De plus, pour une plus grande fluidité nous avons choisi de supprimer le système de message.

Comportement du Kernel - classe World (dans notre programme qui joue aussi le role d'environnement)

```
Public void run(){
            long timeLoopBegin, timeHandlers, timeToSleep;
            while(this.gUI != null && !shouldStop && isHeroeAlive &&
!this.playerHandler.hasHeroeWon()){
                  this.waitIfPaused();

                  timeLoopBegin = System.currentTimeMillis();

                  isHeroeAlive = this.playerHandler.makeLiveHeroe();
                  this.environnementHandler.schedule();
                  this.deleteEntitiesToDelete();
                  this.scene.repaint();

                  timeHandlers = System.currentTimeMillis() - timeLoopBegin;
                  timeToSleep = World.LOOP_SLEEP - timeHandlers;

                  this.waitJustNecessaryTime(timeToSleep);
            }
            endTheGame();
      }
```

Comportement du scheduler (pour les entités animées autre que le héros)

```
public void schedule(){

            Collections.shuffle(this.tabAnimatedEntity);
            Iterator<AnimatedEntity> it = tabAnimatedEntity.iterator();
            boolean isCurrentEntityAlive = false;

            while(this.world != null && it.hasNext()){
                  AnimatedEntity entity = it.next();

                  isCurrentEntityAlive = entity.live();
                  if (!isCurrentEntityAlive)
                        this.addEntityToDelete(entity);
            }


            if(!this.toDelete.isEmpty())
                  this.deleteAnimatedEntity();

      }
```

Comportement général d'un agent

```
Public boolean live(){
            dealWithPosition(null);
            if (this.isAlive()){
                  --this.countDown;
                  if(this.countDown <= 0){
                        this.countDown = this.getConstCountDown();
                        Direction dir = makeMeMove();
                        if(dir != null)
                              this.updatePosition(this.getPositions(), dir);
                  }
                  return true;
            }
             else {
                  this.removeMe();
                  return false;
            }}
```

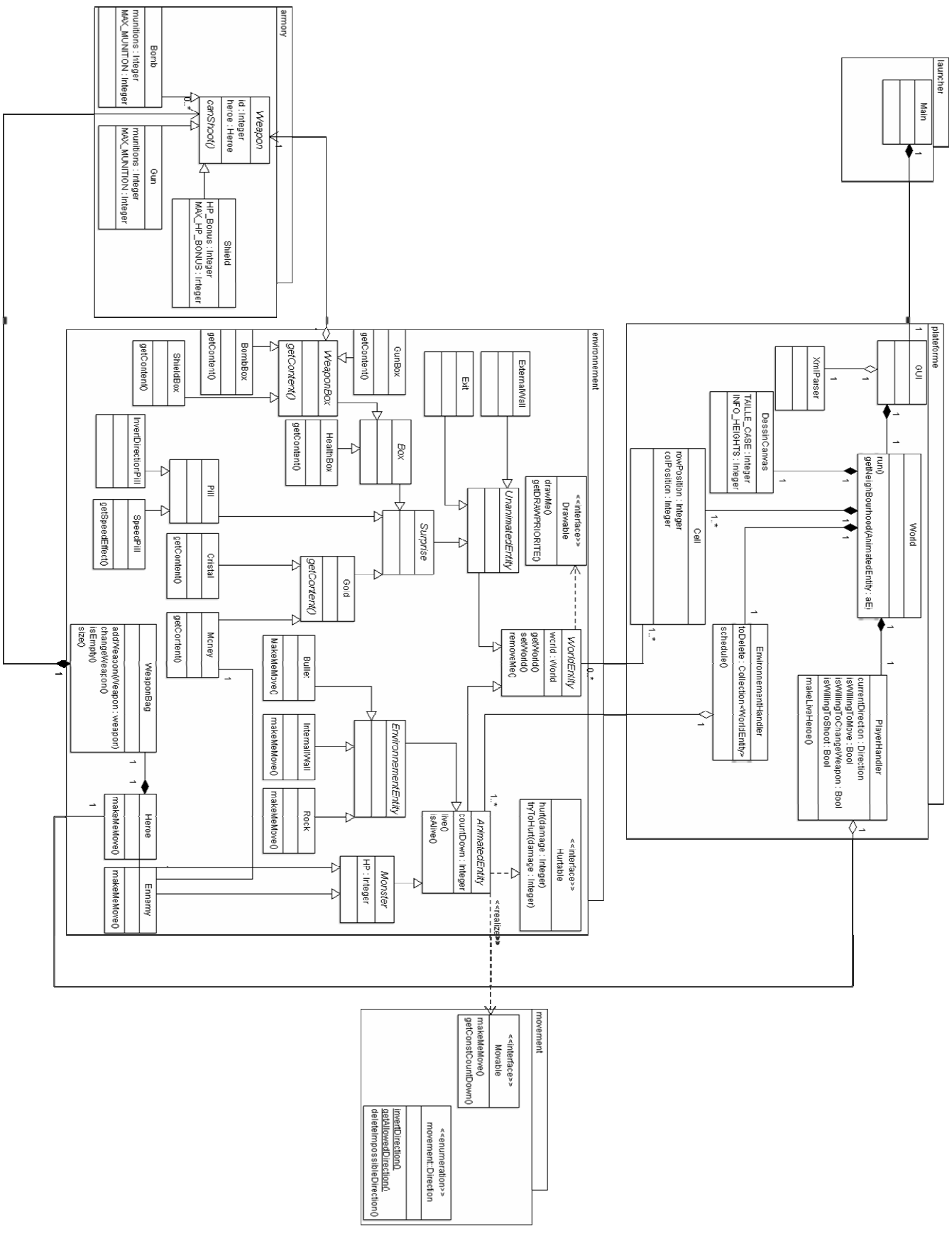
Le comportement général d'un agent se spécialise ensuite pour les ennemis qui ont la possibilité d'attaquer :

```
public boolean live(){
            if (super.live()){
                  WorldEntity opponent = chooseWhichToAttack();
                  if (opponent != null)
                        this.attack(opponent);
                  return true;
            } else
                  return false;
      }
```

L'avantage de cette méthode est que chaque agent est éxécuté après un autre. Il n'y aura pas de problème de synchronisation. De plus, chaque objet mouvant n'est pas un thread, on peut donc ajouter un nombre conséquent d'entités sans avoir à craindre pour la fluidité a priori.

Le jeu

UML


launcher
Main
plateforme
GUI
XmlParser
DessinCanvas
TAILLE_CASE : Integer
INFO_HEIGHTS : Integer
World
run()
getNeighBourhood(AnimatedEntity : aE)
Cell
rowPosition : Integer
colPosition : Integer
EnvironnementHandler
toDelete : Collection<WorldEntity>
schedule()
PlayerHandler
currentDirection : Direction
isWillingToMove : Bool
isWillingToChangeWeapon : Bool
isWillingToShoot : Bool
makeLiveHeroe()
armory
Bomb
munitions : Integer
MAX_MUNITION : Integer
Gun
munitions : Integer
MAX_MUNITION : Integer
Weapon
id : Integer
heroe : Herse
canShoot()
Shield
HP_Bonus : Integer
MAX_HP_BONUS : Integer
environement
GunBox
getContent()
Ext
ExternalWall
BombBox
getContent()
ShieldBox
getContent()
WeaponBox
getContent()
HealthBox
getContent()
Box
InvertDirectionPill
SpeedPill
getSpeedEffect()
Pill
Cristal
getContent()
Gold
getContent()
Money
getContent()
Surprise
UnanimatedEntity
<<interface>>
Drawable
drawMe()
getDRAWPRIORITE()
WorldEntity
world : World
getWorld()
setWorld()
removeMe()
Bullet
makeMeMove()
InternalWall
makeMeMove()
Rock
makeMeMove()
EnvironnementEntity
AnimatedEntity
courtDown : Integer
live()
isAlive()
<<interface>>
Hurtable
hurt(damage : Integer)
tryToHurt(damage : Integer)
Monster
HP : Integer
WeaponBag
addWeapon(Weapon : weapon)
changeWeapon()
isEmpty()
size()
Herse
makeMeMove()
Ennemy
makeMeMove()
<<realize>>
movement
<<interface>>
Movable
makeMeMove()
getConstCountDown()
<<enumeration>>
movement.Direction
invertDirection()
getAllowedDirection()
deleteImpossibleDirection()

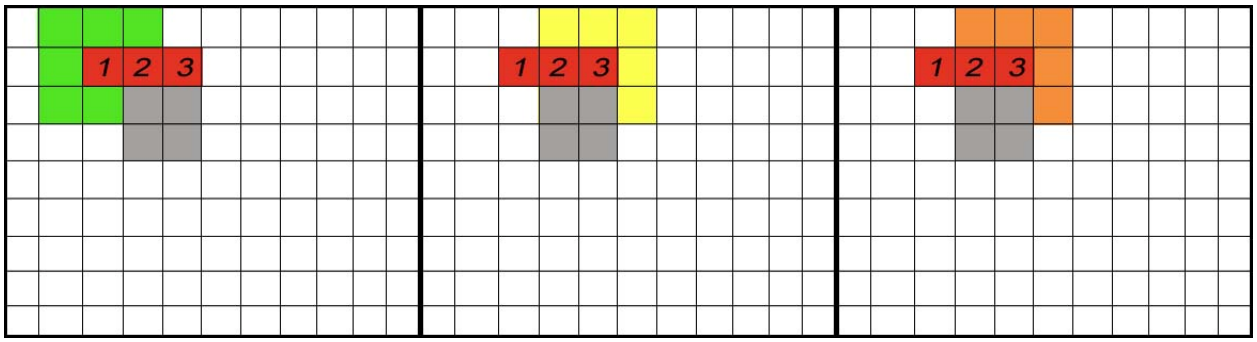
Détection d’objet (voisinage)

Afin de savoir où peut se déplacer notre entité mouvante, il convient de savoir ce qui l’entoure. Sachant cela, l’entité décidera si elle peut bouger ou non vers cette direction. Chaque entité a bien sur un champ de visibilité différent des autres. Par exemple, les murs ont une visibilité plus petite que les ennemis.

Dans la fonction ci-dessous on peut observer la demande à l’entité World du voisinage de l’entité aE. En effet, l’entité aE sait qu’elle est dans le World mais c’est World qui a accès à la map et donc à la localisation des autres entités. Afin d’éviter une perte de temps pour rechercher les positions de l’entité aE, celle-ci a connaissance de ça/ses positions.

```
public HashSet<WorldEntity> getNeighBourhood(final AnimatedEntity aE){
            LinkedList<Position> positions = aE.getPositions();
            int animatedEntityVisibility = aE.getVisibility();
            HashSet<Position> positionsToExplore =
this.getRegionOfInterest(positions, animatedEntityVisibility);
            HashSet<WorldEntity> wEntitiesTab =
this.fetchWorldEntitiesAround(aE, positionsToExplore);
            return wEntitiesTab;
}
```

Cette fonction est elle-même divisée en plusieurs fonctions qui pour chaque cellule occupée par l’entité va chercher le contenu de ces voisines.



Remarque : Comme on le voit l’entité grise va être pointée 3 fois par le voisinage de 3 cellules différentes. Du fait du choix du HashSet pour stocker les différentes entités voisines à notre entité aE, on n’aura pas ce problème de doublons.

Une fois les entités obtenues, on cherche leur direction si les objets détectés font partie d’un panel d’entités bloquantes pour l’objet.

En ce qui concerne les murs on exclura systématiquement les directions diagonales.On a alors l’ensemble des directions interdites qui vont nous permettre de générer les directions autorisées.

Vitesse

La vitesse dans notre programme est vue comme un compteur qui à chaque pas de temps est décrémenté. Une fois le compteur arrivé à zéro, on appelle la fonction permettant de faire vivre l'entité. Quand le compteur est à sa zéro, il est remis à sa valeur initiale et on recommence.

```
Public boolean live(){
            dealWithPosition(null);
            if (this.isAlive()){
                  --this.countDown; //le compteur de vitesse
                  if(this.countDown <= 0){
                        this.countDown = this.getConstCountDown();
                        Direction dir = makeMeMove();
                        if(dir != null)
                              this.updatePosition(this.getPositions(), dir);
                  }
                  return true;
            } else {
                  this.removeMe();
                  return false;
            }
      }
```

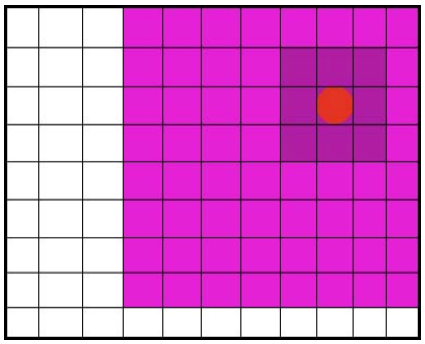
Détails

Comportement de l’ennemi

Déplacement

Nous avons implémenté un comportement agressif pour l’ennemi. Dès qu’il trouve le héros dans son champ de vision il se dirige directement vers lui.

Une optimisation aurait pu consister à faire 2 types de comportements, un ennemi agressif et un fuyard.



Ici on visualise les 2 types de vision de l’ennemi.Tout d’abord en violet foncé la vision pour le déplacement. La zone en violet claire, quant à elle, symbolise le champ de vision de l’ennemi afin qu’il puisse voir le héros.

Afin de savoir quelles sont les différentes directions interdites on va tester toutes les directions possibles de la zone violette et tester si il y a un objet bloquant. Cette technique est légérement différente de celle cité précédemment au vue de la différence de déplacement de l’entité. En effet, comme l’ennemi peut se déplacer en diagonale, la direction HAUT peut être interdite sans que le soit la direction HAUT-GAUCHE.

Interaction avec les objets de l’environnement

Chaque ennemi à une vitesse commune à tous les ennemis mais possède des points de vie et des points d’attaque aléatoires. Ils ne peuvent prendre les trousses de soin et les trésors. Néanmoins, ils peuvent interagir avec les pierres.

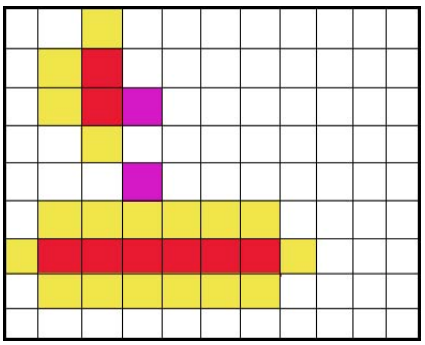
Combat

Pour ce qui est de l’attaque l’ennemi fonctionnne comme le héros. Il va tout d’abord localiser la position du héros si celui-ci se trouve dans son champ de visibilité. Ensuite, il va obtenir une référence sur lui si celui-ci se situe dans son voisinage de déplacement. Enfin, il tentera de l’attaquer.

Comportement des murs

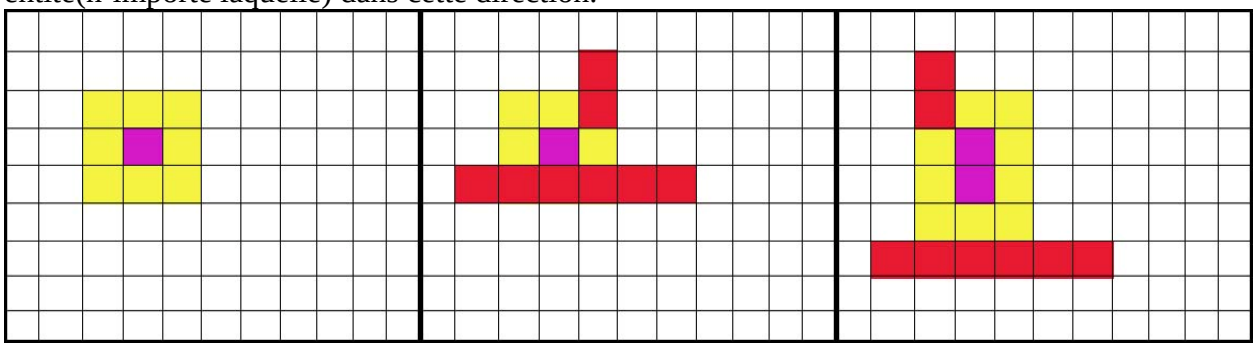
Le mur peut bouger dans les directions horizontales et verticales . Les seules choses qui puissent le bloquer sont des rochers qui lui interdisent d’aller dans la direction sur laquelle ils se situent et les murs extérieurs. Le mur mouvant peut aussi être détruit par une mine.

Le héros, ne pouvant pas traverser les murs, sera poussé par le mur mouvant. Si le héros se situe entre 2 murs il pourra se faire écraser.



Comportement des rochers

Un rocher peut être poussé dans tous les directions sauf si il rencontre une autre entité(n'importe laquelle) dans cette direction.



Les directions possibles pour le(s) rocher(s) sont en jaunes.

Interaction avec l'environnement

Armes

3 types d'armes ont été réalisées : le pistolet, le bouclier et les mines. Chacune ayant un comportement différent.

Le pistolet à la capacité de créer des balles (objet Bullet) qui iront dans la même direction que le héros ou vers l'ennemi si celui-ci se trouve très proche de notre héros. On pourra dès lors toucher les ennemis à distance. Une balle ne peut être évitée et cause un dommage de 3 points de vie. Le stock de balles est limité par le champ munitions. Il est bien sûr possible de tirer en rafale avec le pistolet.

Les mines sont des objets qui une fois posée explosent dès lors qu'un objet les touche. L'objet qui touche la mine est directement détruit. Une optimisation aurait pu consister à faire un certain nombre de dégats avec une mine ou bien de detruire tous les objets dans un rayon d'action. A l'instar des pistolets, les mines ont un champ munition renseignant sur le nombre d'objets utilisables.

Le bouclier sert à protéger notre héros contre les attaques des ennemis. Celui-ci absorbera dans une certaine mesure (un certains nombre de points de vie) des dégâts causés par les attaques. Une fois le bouclier épuisé le héros subira directement les attaques.

Notre héros peut utiliser durant son aventure plusieurs armes celle-ci étant stockée dans un sac. Afin de changer d'armes il suffira au joueur d'appuyer sur la touche tabulation.

Pilules

Les pilules sont elles aussi divisées en 2 parties :

- les pilules qui vont augmenter la vitesse du joueur ou bien la ralentir ;
- les pilules qui inversent les commandes de direction.

Chacun des changements cités est permanent et cumulable.

En ce qui concerne le changement de vitesse il s'opère simplement en changeant la constante du compteur d'action.

```
private static final int CONST_COUNTDOWN = 3;

…

public void changeSpeed(int speedChanging){
            this.setCountDown(Heroe.CONST_COUNTDOWN + speedChanging);
            this.CURRENT_COUNTDOWN += speedChanging;
}

//Live de la classe AnimatedEntity

public boolean live(){
      dealWithPosition(null);
      if (this.isAlive()){
            --this.countDown;
            if(this.countDown <= 0){
                  this.countDown = this.getConstCountDown();
                  Direction dir = makeMeMove();
                  if(dir != null)
                        this.updatePosition(this.getPositions(), dir);
                  }
                  return true;
            } else {
                  this.removeMe();
                  return false;
            }
      }

// Live de la classe Heroe (specialisé de AnimatedEntity puis de Monster)
public boolean live(){
      if (super.live()){
            if (playerHandler.isWillingToChangeWeapon()){
                  this.weapon = this.weaponBag.changeWeapon();
            }

            Direction directionEnnemy = null;
            if (playerHandler.isWillingToShoot() && this.weapon != null &&
this.weapon.canShoot()){
                  if (weapon instanceof Gun){
                        directionEnnemy = getEnnemy();
                        if (directionEnnemy == null)
                              ((Gun)
weapon).shoot(playerHandler.getCurrentDirection());
                        else
                              ((Gun) weapon).shoot(directionEnnemy);
                  }
                  else if(weapon instanceof Bomb){
                        Direction dir =
this.getPlayerHandler().getCurrentDirection();
                        if(this.playerHandler.isWillingToMove())
                              dir = Direction.invertDirection(dir);
                  }
            }
            return true;
      }
      else
            return false;
}
```

Pour ce qui est de l'inversement des directions, il s'opère directement dans le HeroeHandler. Une variable booleen nous permet de savoir dans quel repère on se situe (inversé ou pas). Ainsi,si on est dans le repère inversé toutes les directions fléchées choisies seront inversées.

Cristaux

Les cristaux constituent une façon de collecter de l'argent. Ils sont symbolisés en bleu et contiennent une quantité « d'argent » calculée aléatoirement au lancement du jeu.

Les ennemis ne peuvent les ramasser, ils ne peuvent être détruits par une arme.

Argent

L'argent est sensiblement similaire au cristal à la différence qu'il est relâché aléatoirement par un monstre une fois vaincu.

Système de combat

Durant son périple au sein du labyrinthe, notre héros peut collecter plusieurs armes. Celles-ci sont stockées dans un objet appelé WeaponBag qui peut contenir au maximum 3 armes. On peut changer d'armes en appuyant sur la touche tab.

Si lors du cheminement au sein du labyrinthe, on touche une arme que l'on possède déjà, l'arme dans le weapon bag est automatiquement rechargée dans la limite des munitions maximales. Si elle n'est pas déjà présente et que le stock d'arme limite n'est pas atteint, on ajoute l'arme au stock. Dans les autres cas l'arme n'est pas ajoutée et elle est détruite.

Afin de rendre le jeu plus fluide, certaines améliorations ont été réalisées sur le pistolet et les mines. En effet, si l'ennemi se trouve très prés de notre héros, la balle du pistolet va directement dans sa direction et non pas dans la direction du héros. En ce qui concerne les mines, celles-ci sont toujours placées derrière le héros (dans la direction inverse du mouvement courant) sauf quand le héros ne bouge pas. Dans ce cas là, elles vont se déplacer devant lui.

Chaque monstre, que ce soit le héros ou les ennemis, peut éviter certains types d'attaques. Ainsi lors d'une tentative d'attaque un « dé » est jeté. Si le nombre aléatoire tiré n'est pas compris dans une liste de nombre définie aléatoirement au lancement du jeu, alors l'attaque réussie sinon elle échoue. Ce système n'est néanmoins pas activé pour les balles et les mines. On suppose, en effet, que ces armes ont une efficacité de 100%.

```
protected void attack(WorldEntity opponent) {
            int damage = 0 ;

            if(opponent != null && ((Monster)this).isAlive()){
                  if(this.weapon != null){
                        if((damage = this.weapon.getDamage()) == 0){
                              damage = Heroe.MIN_DAMAGE;
                        }
                  }
                  else
                        damage = Heroe.MIN_DAMAGE;

                  System.out.println("Heroe - J'attaque : " + damage);
                  ((Ennemy)opponent).tryToHurt(damage);
            }
      }

public void tryToHurt(int damage) {
            if(!Heroe.dogde.contains((int)(Math.random()*10)))
                  this.hurt(damage);}
```

Améliorations possibles

A la vue du système et du déroulement du jeu actuel, il pourrait être intéressant de développer les points suivants :

- Nouveaux comportements des ennemis.
- Les mines qui explosent dans un rayon prédéfini.
- La rotation des murs.
- La mise en place de piège.
- Le fait de pouvoir jeter une arme.
- L’apparition aléatoire de trésor(s) et/ou d’ennemi(s).