



UV 2.1 – Projet Informatique

-

EnstaZ :

Simulation de comportement lors d'une  
apocalypse zombie

Nicolas VEYLON – Sophie TUTON

Groupe 8

Promotion 2018

26 Mai 2015

# Sommaire

Sommaire .....	1
1. Préambule.....	2
2. Idée générale .....	3
3. Description des classes .....	3
3.1 La classe <i>Case</i> .....	3
3.1.1 Les méthodes <i>Modif_pop()</i> et <i>Modif_loot()</i> .....	3
3.1.2 Les méthodes <i>Create_add_PERS()</i> et <i>Create_add_OBJ()</i> .....	4
3.2 La classe <i>Personne</i> .....	4
3.2.1 La méthode <i>Modif_equi()</i> .....	5
3.2.2 La sous-classe <i>Vivant</i> .....	5
3.2.2.1 La méthode <i>observation_decision()</i> .....	5
3.2.2.2 La méthode <i>interet()</i> .....	6
3.2.2.3 La méthode <i>deplacement()</i> .....	6
3.2.2.4 La méthode <i>combat()</i> .....	6
3.2.2.5 La méthode <i>degat()</i> .....	7
3.2.2.6 La méthode <i>tour()</i> .....	7
3.2.3 La sous-classe <i>Zombie</i> .....	7
3.2.3.1 La méthode <i>observation_decision()</i> .....	7
3.2.3.2 La méthode <i>déplacement</i> .....	8
3.2.3.3 La méthode <i>tour()</i> .....	8
3.2.3.4 La méthode <i>degat()</i> .....	8
3.2 La classe <i>Objet</i> .....	8
4. Les fonctions .....	8
4.1 <i>Générer une carte et une population</i> .....	8
4.1.1 <i>Map-generator</i> .....	8
4.1.2 <i>init_PERS</i> .....	8
4.2 Les événements .....	9
4.2.1 <i>Maladie et Soins</i> .....	9
4.2.2 <i>Explosion</i> .....	9
4.2.3 <i>Livraison</i> .....	9
6. L'interface graphique .....	9
6.1 Descriptif général .....	10
6.2 Les cartes .....	10
6.3 Le graphique .....	11
6.4 L'historique .....	12
6.5 Les boutons .....	13
6.5.1 Descriptif des boutons .....	13
6.5.2 <i>Genere_tour</i> .....	13
7. Lancement .....	14

*"These are the end times. There was no hope of survival. This is how they died."*

Project Zomboid.

## 1. Préambule

Le but de ce projet est de modéliser le comportement des habitants d'une ville lors d'une apocalypse zombie. Dans cette simulation, les personnes vivantes tenteront de fuir les zombies tout en subvenant à leur besoin en nourriture. Chaque personne est dotée de caractéristiques différentes (le courage et la force par exemple) selon qu'elle soit médecin, enfant ou encore militaire. Ces caractéristiques ont un impact notable sur le devenir de la personne, car elles sont ancrées dans les mécanismes de prise de décision et de combat.

La partie commence par l'apparition de zombies à l'hôpital de la ville. Les habitants sont bien sûr déjà sur place. A partir de là, les joueurs (vivant ou zombie) joueront à tour de rôle. Un tour est composé d'une phase d'entretien (alimentation, soin), d'une prise de décision sur l'action à faire, et enfin d'un déplacement (interféré ou non par un combat contre un zombie).

La simulation respecte le genre zombie : si un habitant vient à mourir suite à une morsure, il se relèvera zombifié quelques tours plus tard et, en outre, un système d'équipement et de "loot" est mise en place permettant aux survivants d'affronter leur destin. S'ajoute à cela des événements aléatoires visant à rajouter de la dynamique à la simulation.

## 2. Idée générale

Avant de parler des différentes classes, il faut introduire les grandes mécaniques de la simulation. Lors de sa création, le *vivant* ou le *zombie* est ajouté à la liste *spawn*. Cette liste est vidée à chaque début de tour dans *players*, la liste principale des joueurs. Cette liste est un *LoopList*, une liste chaînée circulaire. Lorsqu'un *vivant* meurt, il est supprimé de *players* et remplacé par un *zombie* possédant le même équipement. Lorsqu'un *zombie* est détruit, il est supprimé de *players*.

La carte de jeu, nommé *carte*, est une matrice d'instances *Case* (cf. 3.1).

Les variables *players* et *carte* sont globales et définies dans le script principal. La simulation s'effectue dans ce même script, dans une boucle `for` se présentant ainsi :

- Vidage de *spawn* dans *players*
- Tour du joueur OU élimination du joueur de *players* (dans le cas où il est mort durant le tour d'un autre joueur) OU Passage du tour si c'est un zombie qui ne s'est pas encore relevé.
- Élimination du joueur de *players* s'il est mort durant son tour.
- Changement de joueur. D'où l'intérêt d'une liste chaînée circulaire : on change à chaque fois pour le joueur suivant.

## 3. Description des classes

### 3.1 La classe *Case*

Les instances *Case* sont les éléments de la matrice qui compose la carte de jeu. Une *case* n'a pas de dimensions précises, mais pour donner un ordre de grandeur, elle représente très grossièrement un carré de 15 mètres de côté. C'est sur ces *cases* qu'évoluent les différents joueurs. Cette classe possède alors naturellement les méthodes permettant de modifier son contenu. Ces attributs sont les suivants :

- Position : le tuple (x,y) qui est identique à sa position dans la matrice *carte*.
- Population : liste contenant les instances des différents joueurs présents sur cette *case*.
- Loot : liste contenant les instances des différents objets pouvant être ramassés par les joueurs se situant sur la *case*.

#### 3.1.1 Les méthodes *Modif\_pop()* et *Modif\_loot()*

Cette méthode simple permet de supprimer ou d'ajouter un ou plusieurs joueurs d'une case. Elle prend un tableau *numpy* en argument, sous la forme `array([[instance, 0], [instance, 1]...])`. Un 1 signifie à *ajouter*, et 0 à *retirer*. Dans tel ou tel cas, *Modif\_pop()* (resp. *Modif\_loot()*) utilisera naturellement *append()* ou *remove()* sur *self.\_population* (resp. *self.\_loot*).

### 3.1.2 Les méthodes `Create_add_PERS()` et `Create_add_OBJ()`

Ces méthodes créent des instances *Personne* et *Objet* et utilisent *Modif\_pop()* et *Modif\_loot()* pour les ajouter sur la *case*. Pour éviter de faire autant de *if* que de type de personne et de type d'objet dans ces méthodes, et ainsi les factoriser grandement, elles utilisent la « fonction-dictionnaire » *Factory*. Cette fonction permet de créer une instance en rentrant en argument le nom de la classe en chaîne de caractère. Ainsi, les méthodes *Create\_add\_PERS()* et *Create\_add\_OBJ()* se résument à une ligne de code :

```
return self.Modif_pop(np.array([[Factory(class_name)(self),1]]))
```

```
def Factory(class_name):
    classes = {'Nourriture' : Nourriture,\
              'Arme_a_feu' : Arme_a_feu,\
              'Arme_de_melee' : Arme_de_melee,\
              'Medicament' : Medicament,\
              'LambdaV' : LambdaV,\
              'Militaire' : Militaire,\
              'Religieux_ext' : Religieux_ext,\
              'Medecin' : Medecin,\
              'Enfant' : Enfant,\
              'LambdaZ' : LambdaZ,\
              'Tank' : Tank }
    return classes[class_name]
```

### 3.2 La classe *Personne*

Les instances *Personne* sont les joueurs, i.e. les zombies et les vivants. Ces dernières possèdent des attributs et des méthodes très différentes. On notera toutefois que les méthodes *degat()*, *observation\_decision()* et *deplacement()* sont des polymorphismes.

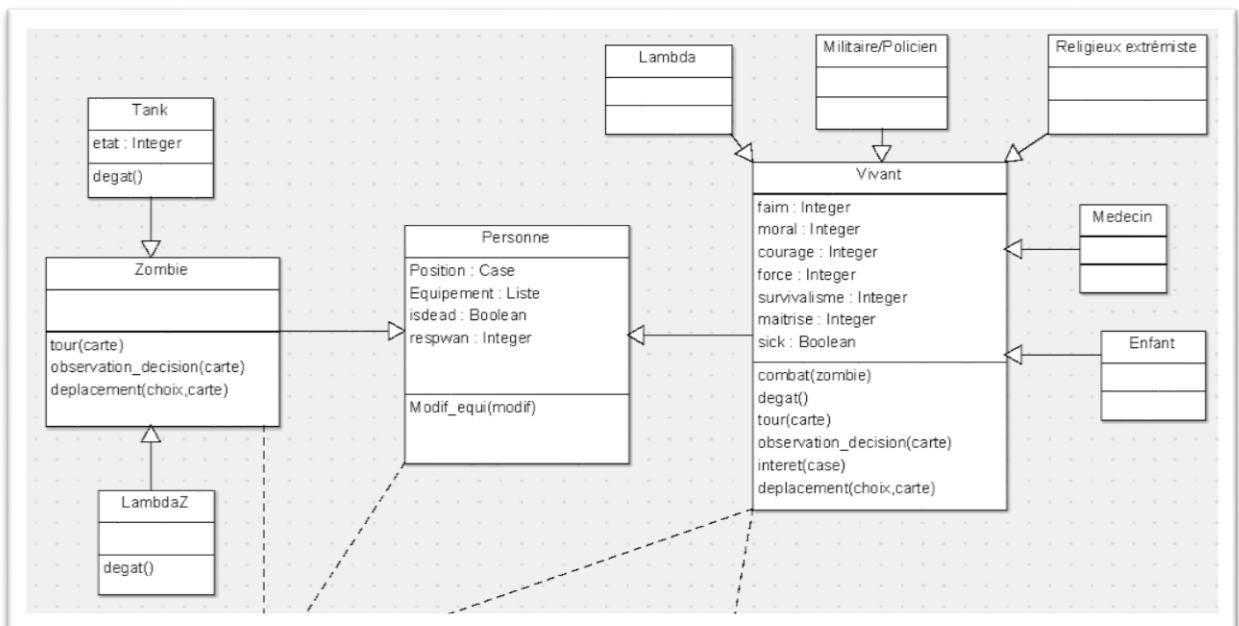


Diagramme de classe de *Personne*

### 3.2.1 La méthode *Modif\_equi()*

Il s'agit d'une méthode permettant de modifier l'inventaire d'une *Personne* en faisant attention au suppression ou ajout sur les *cases*. En effet, si une *personne* récupère un équipement sur une *case*, il est nécessaire pour la simulation que cet équipement soit enlevé de la *case*. Elle gère aussi le dépôt d'*objet* sur une *case* (lors de sa mort par exemple). Pour cela *Modif\_equi* prend pour paramètre un tableau que l'on écrira sous la forme: `np.array([[Obj0,bo],[Obj1,b1],[Obj2,b2]...])` avec *Objx* l'*objet* considéré et *bx* un entier 0 ou 1 ( 0 pour la suppression dans l'inventaire de la *Personne*, et 1 pour l'ajout).

### 3.2.2 La sous-classe *Vivant*

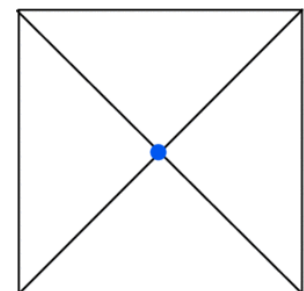
Comme vu précédemment, les vivants possèdent des caractéristiques au cœur des mécanismes de prise de décision et de combat :

- La faim diminue au début du tour du joueur, et celui-ci meurt si elle descend à 0. (cf. *tour()*)
- Le moral joue sur la vitesse de déplacement du joueur, mais surtout sur ses relations avec les autres survivants (cf. *observation\_decision()*). En outre, le moral est modifié si un zombie ou un vivant meurt près du joueur.
- Le courage permet au joueur, selon sa valeur, d'affronter les zombies qui se dressent devant son objectif, ou bien de les fuir. (cf. *observation\_decision()*).
- La force permet au joueur de mieux affronter les zombies et également de mieux les esquiver.
- Le survivalisme ressemble beaucoup à la force. Il est présent lors des phases de combat. Il faut noter qu'il aurait dû servir à la défense de bâtiment, par l'utilisation de barricade, mais cette partie du projet a été avortée faute de temps et d'intérêt. (cf. *combat()*)
- La maîtrise des armes à feu, raccourcie en *maîtrise*, caractérise l'habilité du joueur à tirer sur les zombies et ainsi éviter de risquer sa vie lors d'un combat au corps à corps. (cf. *combat()*)
- *sick* est un booléen qui est Vraie si le joueur est malade. La maladie implique une chute de moral, de courage, de faim et de force. Il peut guérir grâce aux objets *Médicament*.

#### 3.2.2.1 La méthode *observation\_decision()*

Cette méthode représente à elle seule tout le mécanisme de prise de décision des vivants. Chaque *case* dans le rayon d'observation du joueur va être affectée à un entier *I* représentant l'intérêt de la *case*. Cet intérêt *I* est calculée par la méthode *interet()*. Cette méthode retourne simplement l'instance *objet* avec le plus d'intérêt sur la *case* générant le plus grand intérêt *I*.

Cette méthode prend bien sûr en compte les zombies aux alentours. La modélisation est la suivante : On construit la matrice *MZ* donnant le nombre de zombies sur les *cases* voisines de celle du joueur (le joueur étant donc au centre de *MZ*). On s'intéresse ensuite au nombre de zombie sur chaque « cône » composant la matrice (voir figure ci-contre). Si le nombre de zombies sur un « cône » est supérieur à *maxZ* (calculée en fonction du courage du joueur), toutes les *cases* le composant n'ont aucun intérêt. Cela



Découpage de *MZ* en quatre cônes

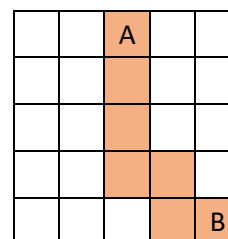
permet de modéliser la fuite des zombies suivant 4 directions, mais également l'affrontement si le joueur est assez courageux pour aller récupérer l'équipement désiré. Dans le cas où le joueur est entouré de zombie et donc que tous les cônes sont désactivés, ou qu'il n'y ait aucun objet à distance, le joueur se dirigera vers le centre de la carte, i.e. le parc, où les parachutages d'équipement se font, où vers le supermarché si sa jauge de faim est faible.

### 3.2.2.2 La méthode *interet()*

Cette méthode prend en argument une *case* et retourne l'intérêt *I* de celle-ci pour le joueur considéré. Cet intérêt *I* est la somme de chaque intérêt des *objets* de la *case* pour le joueur. Pour donner un exemple, un joueur qui a très faim va donner un grand intérêt aux *cases* comportant de la nourriture car l'intérêt des objets *Nourriture* est très élevé. Le calcul prend en compte l'état de l'équipement du joueur (s'il possède ou non les *objets* présents sur la *case*), ses caractéristiques (s'il a faim, s'il est malade) ainsi que l'utilité même de l'objet. Il faut noter que la personne ne peut ramasser de *nourriture* s'il en a déjà le nombre maximal autorisé grâce à cette fonction. L'intérêt de la *nourriture* reste à 0.

### 3.2.2.3 La méthode *deplacement()*

Cette méthode permet au joueur de se déplacer entre les différentes cases. Elle prend en argument l'*objet* qui a été déterminée par la méthode *observation\_decision()*, et dirigera le joueur vers celui-ci. Le plus court chemin sera adopté, selon le schéma ci-contre. Pour que le joueur se déplace d'une case à une autre, la variable d'instance *position* du joueur est modifiée, le joueur est supprimé de la liste *population* de l'ancienne *case*, et ajouté sur celle de la nouvelle.



Déplacement de  
A vers B

Un vivant peut parcourir entre une à quatre *cases* par tour, selon son niveau de moral. A chaque fois qu'il arrive sur une nouvelle case, il combat à tour de rôle chacun des zombies présents sur la case, via la méthode *combat()*.

### 3.2.2.4 La méthode *combat()*

Cette méthode prend en argument un zombie sur la même case que le joueur. Elle reprend les mécanismes de jets de dés propres aux jeux de rôle. Si le joueur possède une arme à feu, il tentera d'abord d'abattre à distance le zombie. Le jet est réussi si  $\text{random.random()} < 0.08 * \text{self._maitrise}$ . Dans ce cas, on appelle la méthode *degat()* du zombie pour le détruire de la case et le combat est terminé (return). Sinon, le joueur procède à un jet d'esquive, qui est réussi si  $\text{random.random()} < 0.04 * \text{self._surv} + 0.03 * \text{self._force}$ . Dans ce cas, le combat est terminé (return). Sinon, le combat se fait au corps à corps et forcément un des deux mourra. Le jet de mêlée est réussi si  $\text{random.random()} < (0.08 * \text{self._force} + 0.06 * \text{self._surv}) / (2 - \text{melee\_arm}^1)$  and  $\text{random.random()} < 0.75^2$ . On appellera simplement la méthode *degat()* sur le perdant du combat.

<sup>1</sup> *melee\_arm* vaut 1 si le joueur possède une arme de mêlée, et 0 sinon.

<sup>2</sup> La deuxième condition à vérifier est le « facteur de Murphy » : même un militaire très bon au combat peut échouer.

#### 3.2.2.5 La méthode *degat()*

Cette méthode permet de tuer le survivant. Il est éliminé de la liste *players*, de la liste *population* de la case où il se situe. Ensuite, un *zombie* (par défaut un *LambdaZ*, cf. 3.2.2) est créé via la méthode *Create\_add\_PERS()* vue plus haut. Ce dernier récupère l'intégralité de l'inventaire du joueur. Pour respecter le genre *Zombie*, ce *LambdaZ* passe trois fois son tour avant de se relever. (cf. 3.2.2)

Lorsqu'un survivant meurt sur une case, tous les autres survivants sur la case baissent en moral. Nous verrons plus tard que c'est le contraire lorsque c'est un *zombie* qui meurt.

#### 3.2.2.6 La méthode *tour()*

Cette méthode est bien entendu la méthode principale des instances *Personne* et permet de générer le tour d'un joueur. Elle se déroule ainsi :

- Diminution de *faim*.
- Alimentation si *faim* est en dessous de 50 et si la *personne* possède de la *nourriture*. Augmentation de 30.
- Si *faim* descend à 0 la *personne* meurt (*degat()*) et le tour se termine (return).
- *observation\_decision()*
- *deplacement()*
- jet de *maladie* (cf. 4.2.1)
- Ramassage de l'objet visé (*choix*)

#### 3.2.3 La sous-classe *Zombie*

La classe *Zombie* correspond à une sous classe de *Personne*, elle hérite donc de toute ses méthodes et variables d'instances. A celles-ci s'ajoute la variable d'instance *respawn*, mise à défaut à 3. Il s'agit d'un compteur. Les *zombies* passent trois fois leur tour avant de se relever. Il existe deux sous classes de *Zombie*: les *Tanks* et les *LambdaZ*. La seule différence est qu'il est nécessaire de toucher 3 fois un *Tank* pour le tuer, alors qu'une seule fois suffit pour achever les *LambdaZ*. Pour se faire, la variable d'instance *état*, par défaut à 4, de la classe *Tank* décrémente à chaque appel de *degat()* et ce jusqu'à 0.

##### 3.2.3.1 La méthode *observation\_decision()*

Cette méthode prend la *carte* en paramètre et renvoie la *case*, appelé *choix*, où le *zombie* désire aller. Les *zombies* veulent se rapprocher des *vivants*. Ils vont évaluer la distance le séparant de chaque *vivant* se trouvant dans leur champ de vision. Suite à cela ils prendront comme cible la *case* la plus proche contenant un *vivant*. Dans le cas où il n'y a pas de *vivant* dans leur champs de vision, les *zombies* ont une certaine probabilité d'errer, i.e. de se déplacer aléatoirement d'une *case* autour de leur position actuelle.



### 3.2.3.2 La méthode déplacement

Cette méthode est presque identique à la méthode déplacement de la classe *Vivant*, la seule différence est que les *zombies* ne se déplacent que d'une *case*. Elle appelle d'ailleurs la méthode *combat()* de la classe *Vivant*.

### 3.2.3.3 La méthode tour()

Seuls l'observation, la décision et le déplacement composent le tour des *zombies*. La méthode *tour()* est donc constituée du calcul de *choix* (case but) via *observation\_decision()* et du déplacement résultant via *deplacement()*.

### 3.2.3.4 La méthode degat()

Cette méthode permet de tuer les *LambdaZ* et les *Tanks* lorsque leur variable *etat* est inférieur ou égal à 1. Lorsque *etat* est strictement supérieure à 1, cette méthode décrémente *etat*. Tuer consiste à modifier la variable *isdead* en *True*, à augmenter le moral des *vivants* se trouvant sur la même *case*, à supprimer le *zombie* de la liste *players*, et à déverser son inventaire sur la *case*.

## 3.2 La classe Objet

La classe *Objet* possède une variable d'instance de type *Case* nommée *pos*. Elle possède trois méthodes : une surcharge *\_\_str\_\_*, un getter et un setter. Cette classe possède 3 sous-classes: *Nourriture*, *Arme*, et *Medicament*. *Arme* possède deux sous-classe: *Arme\_a\_feu* et *Arme\_de\_melee*. En réalité aucune fonction supplémentaire ou variable n'est rajoutée d'une classe mère à une cours fille.

## 4. Les fonctions

### 4.1 Générer une carte et une population

#### 4.1.1 Map-generator.

La fonction *Map\_generator* de *enstaZ.py*, de paramètre *n* entier naturel, renvoie la carte utilisée pour la simulation. Elle commence par créer la carte initiale contenant les instances *cases*. Des instances d'objets telles que *Nourriture*, *Arme\_de\_melee* et *Arme\_a\_feu* sont ensuite rajoutées aléatoirement dans la variable d'instance *loot* de chaque *case*. Cependant certaines zones (ou ensembles de *Cases*) auront une plus forte probabilité de se voir attribuer un type d'objets, ce sont les maisons. D'autres zones où des quantités de types d'objets existent forcément sont présentes : il s'agit d'un supermarché (*Nourriture*), d'un hôpital(*Medicament*) et d'une armurerie(*Arme\_a\_feu*).

#### 4.1.2. init\_PERS.

La fonction *init\_PERS* de *enstaZ.py* prend en paramètre obligatoire *carte* (générée au préalable par *Map\_generator*), et de paramètres facultatifs le nombre de zombie voulue (5 par défaut), et le nombre de vivants voulue (15 par défauts). Les zombies sont répartis aléatoirement dans l'hôpital,

et il y a 1 chances sur 10 qu'un zombie soit un *Tank*. Les vivants, eux, sont répartis aléatoirement sur toute la carte, les différentes catégories de survivants (*médecin, militaire, etc*) étant équiprobables.

## 4.2 Les événements

Quatre événements existent, ils sont déclenchés aléatoirement durant la simulation :

- *maladie*,
- *soin*,
- *explosion*,
- *livraison*.

### 4.2.1 Maladie et Soins

*Maladie* et *Soin* ne font que modifier les variables d'instances de l'objet *Personne* se trouvant en paramètre de la fonction.

### 4.2.2 Explosion

*Explosion* consiste à supprimer les instances (*Personne* et *Objet*) se trouvant dans les variables des objets *Cases* considérés. Pour une explosion trois paramètres peuvent être donnés: la carte (obligatoire), le rayon (facultatif), et la position (facultative). Les paramètres facultatif sont définis aléatoirement lorsqu'ils n'ont pas été définis en paramètre. Le principe de cette fonction est le suivant: toute *personne* se trouvant dans un rayon donné de la *case* repérée par *position* est tuée, et les objets sont supprimés.

La présence d'une explosion dans un tour est représentée par une étoile sur le graphique se trouvant à l'onglet "Evolution des populations".

### 4.2.3 Livraison

Il s'agit là du dépôt d'objet sur la *case* "centrale" de la carte : le parc. Trois paramètres sont possibles: *carte* (obligatoire), *strType* (facultatif), et *quantite* (facultatif). On entend par *strType* le nom de la classe des objets à déposer, et par *quantite* la quantité d'objet à déposer. Si ces valeurs ne sont pas indiquées par l'utilisateur alors ils sont définis aléatoirement. Ils sont ensuite ajoutés sur la *case* dont il est question à l'aide de la méthode *Create\_add\_OBJ* de la classe *Case*.

Une livraison lors d'un tour est représentée par une croix sur le graphique se trouvant dans l'onglet "Evolution des populations".

## 6. L'interface graphique

L'interface a été réalisée grâce à PyQt4. Elle se présente ainsi :

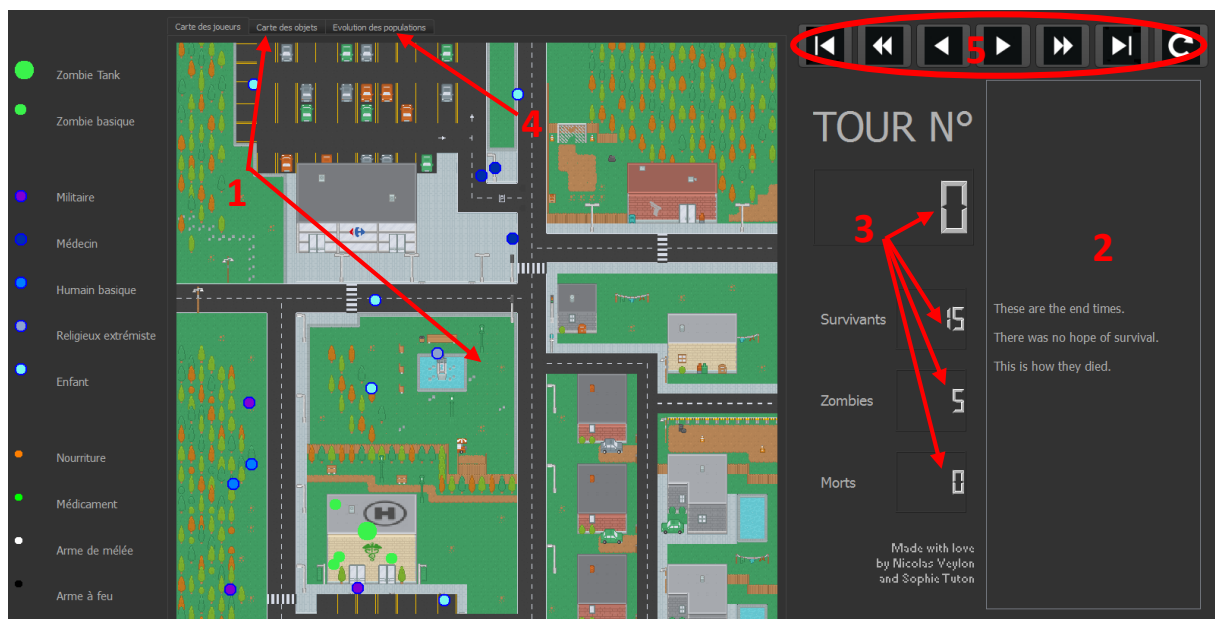
## 6.1 Descriptif général

La majeure partie de l'interface a été élaborée à l'aide de l'outil *QtDesigner*. Le *stylesheet* (habillage de l'interface, thème ou encore «skin») a été récupéré en ligne sur le site de Yasin Uludag<sup>3</sup>.

Cette interface (cf photo suivante) nous donne les informations nécessaires pour comprendre la simulation via :

- Deux cartes **1** où sont représentés les joueurs et les objets (respectivement onglet « Carte des joueurs » et « Carte des objets »).
- Une zone de textes **2** de type label où le tour simulé est détaillé. (Ce label est appelé *historique*)
- Des compteurs de type `LcdNumber` **3** où le tour, le nombre de survivants, de zombies et de morts du sont comptabilisés,
- Un graphique **4** représentant l'évolution du nombre de zombies, de vivants et de morts au cours de la simulation.

Les boutons **5** servent à jouer et visualiser la simulation avec différentes vitesses. Il est également possible de redémarrer une nouvelle simulation à l'aide du dernier bouton. Les images utilisées pour les boutons ont été récupérées sur Internet<sup>4</sup>.



## 6.2 Les cartes

Les cartes des joueurs et des objets sont les pièces maîtresses de cette interface. On les retrouve dans les onglets « Carte des joueurs » et « Carte des objets ». Elles décrivent dynamiquement le déroulement de la simulation. Elles sont composées d'une image en fond d'écran ainsi que des points

<sup>3</sup> Lien vers le stylesheet : <http://www.yasinuludag.com/darkorange.stylesheet>

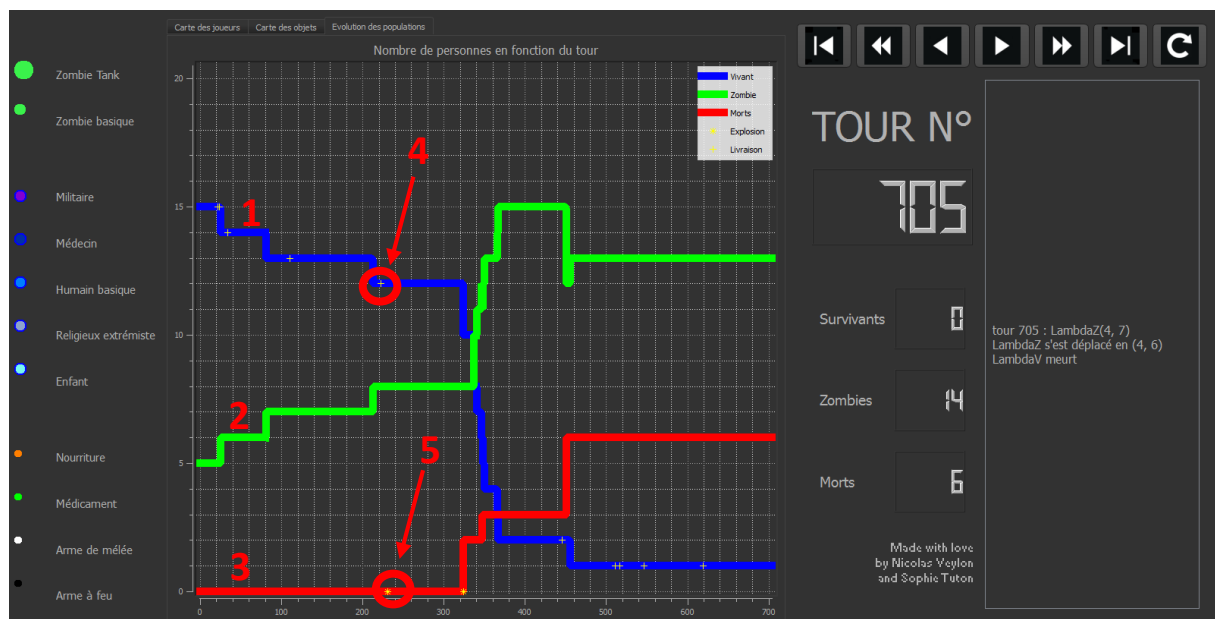
<sup>4</sup> Lien vers les illustrations : <http://www.thinkstockphotos.fr/image/illustration-media-player-sign-black-square-button-icon/174343621>

représentant joueurs et objets. Un code couleur détaillé sur l'interface permet de distinguer différents joueurs (vert pour les zombies, Et différentes nuances de bleu pour les survivants).

- L'image a été créé par nos soins grâce au logiciel *Tiled* et au *spritesheet* du studio *Kenney*<sup>5</sup>. On retrouve le supermarché au nord-ouest, l'armurerie et nord-est, le parc au centre et l'hôpital au sud. Cette image est mise en fond d'écran grâce à l'outil *QPalette*, en lui affectant un *QBrush* contenant l'image. Comme l'image doit tenir parfaitement dans le *QWidget*, il convient de la redimensionner aux dimensions du *QWidget* juste avant d'appliquer la *QPalette*.
- Les points sur les cartes sont créés via les méthodes *tracer()* et *tracer\_objj()*, et grâce à l'outil *QPainter*. Ils diffèrent tous en couleur et contour. Pour afficher les points représentant les joueurs, il suffit de parcourir la liste chaînée *players* contenant les joueurs. Pour chaque joueur, on récupère ses coordonnées (via *instanceJoueur.pos.pos*) et on dessine un cercle grâce au *QPainter* à ces coordonnées (pondérées par la taille en pixels du *QWidget*). Pour afficher les points représentant les objets, on parcourt la variable d'instance *loot* de chaque *case* et on récupère la position à l'aide de la variable d'instance *pos* de *case*. Pour régler le problème des deux points superposés dû à deux joueurs (resp. deux objets) sur la même case, chaque joueur (resp. objet) est affecté de façon permanente à un couple de petits décalages horizontal et vertical. Ces petits décalages (ne pouvant dépasser la taille d'une demi-case en pixel) sont aléatoires. Grâce à cette astuce, chaque joueur se trouve toujours au même endroit sur n'importe quelle case, mais tous les joueurs (resp. objets) sont écartés même s'ils sont sur la même case.

### 6.3 Le graphique

Dans le troisième onglet de l'interface graphique, un *QCurveWidget* permet d'afficher l'évolution des populations le long de la simulation. Il contient 3 courbes (Nombre de vivants **1**, de zombies **2** et de morts **3**) et 2 marqueurs (« + » livraisons **4** sur la courbe des vivants et « \* » explosions **5** sur la courbe de morts).



<sup>5</sup> Lien vers le spritesheet : <http://opengameart.org/content/roguelike-modern-city-pack>

Le tracée du graphique repose sur deux listes `evo_nbV` et `evo_nbZ` dont l'indice correspond au numéro du tour, et dont les éléments correspondent respectivement au nombre de vivants, et au nombre de zombies lors du tour. Ces listes sont initialisées dans `enstaZ.py` à l'aide des variables de classe `Vivant.nbV` et `Zombie.nbZ`.

Il faut tout d'abord initialiser les courbes à l'aide de `make.mcurve()` qui prend en argument les listes des abscisses et des ordonnées, puis on ajoute ces items dans le `QCurveWidget` à l'aide de `curvewidget.plot.add_item`. A chaque nouveau tour généré, on met à jour les *items* à l'aide de la méthode `set_data()` à laquelle on donne en argument la liste des tours (ou seulement de certains tour dans le cas de l'affichage des explosions et livraisons), et la liste du nombre de personnes considérés pendant ces tours (vivants, zombies, mort). La mise à jour de l'item permet directement une mise à jour des courbes.

## 6.4 L'historique







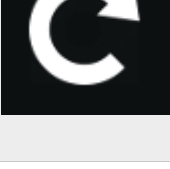
A droite de l'interface graphique se trouvent le label contenant l'historique de la simulation. Celui-ci décrit tout ce qu'il se passe durant la simulation. L'affichage du texte se fait par la méthode `setText()`, exécutée lors de l'appui sur les différents boutons de navigation (cf 6.5 *Les boutons*). La chaîne de caractère à mettre en paramètre de la méthode est récupérée à l'aide de la fonction `search_in_save()` permettant d'afficher ce qu'il se passe lors d'un tour ou entre deux tours fixés en paramètre. Cette fonction va chercher le texte du fichier de sauvegarde, découpe le texte obtenu en une liste dont chaque élément correspond au texte d'un tour (séparation : `\n\n`), puis récupère les éléments des tours qui nous intéressent.

Le fichier de sauvegarde, quant à lui, est rempli à l'aide de la fonction `save` prenant en paramètre une chaîne de caractère et permettant d'ouvrir le fichier de sauvegarde (préalablement créé ou vidé) en mode append, et d'écrire la chaîne de caractère dedans.

## 6.5 Les boutons

### 6.5.1 Descriptif des boutons

7 boutons *QPushButton* permettent de contrôler la simulation.

Bouton	Méthode appelée	Description de la méthode
	<i>toBegin</i>	Cette méthode affiche l'historique du premier tour ainsi que le nombre de vivants, zombies et morts au premier tour.
	<i>moinsdix tours</i>	Cette méthode permet d'afficher l'historique des dix derniers tours ainsi que le nombre de vivants, zombies et morts dix tours auparavant.
	<i>moins un tour</i>	Cette méthode permet d'afficher l'historique du dernier tour ainsi que le nombre de vivants, zombies et morts dix tours du dernier tour
	<i>plus un tour</i>	Cette méthode génère un nouveau tour et l'affiche. Si le tour dernièrement affiché n'est pas le dernier tour généré (i.e. si l'utilisateur est déjà revenu en arrière avec l'un de deux boutons précédents), la méthode affiche le tour suivant grâce aux sauvegardes (le tour a déjà été généré auparavant).
	<i>plus dix tours</i>	Cette méthode génère dix tours et les affiche. Si parmi ces tours il y en a qui ont déjà été générés, il les affiche en utilisant les sauvegarde, et génère les autres.
	<i>simuler</i>	Cette méthode permet de simuler jusqu'à la fin de la partie, i.e. jusqu'à ce qu'il n'y ait plus de zombie ou de survivant.
	<i>reset</i>	Cette méthode permet de commencer une nouvelle partie. Elle réinitialise : -Dans le mécanisme du jeu : la liste des joueurs, les objets et le fichier de sauvegarde. -Dans l'interface : les cartes, les graphes, l'historique et les compteurs.

### 6.5.2 Genere\_tour

Les boutons d'avance génèrent des tours à l'aide de la méthode *genere\_tour* qui a comme paramètre le nombre de tour à générer.

Tant que le nombre de tour n'a pas atteint le nombre de tour définit et que le nombre de Vivant n'est pas nulle:

- Etape 1 : On vide la liste spawn dans la liste chaînée players et on définit le joueur (qui est en fait un *nœud* de la chaîne).
- Etape 2 : On traite les événements explosions et livraisons: ils ont une certaine probabilité d'être actionnée.
- Etape 3 : On vérifie que le joueur n'est pas mort: s'il l'est, on le supprime de players.
  - S'il n'est pas mort, on regarde *respawn*: si sa variable *respawn* n'est pas à 0 alors le joueur ne joue pas et *respawn* est décrémenté.
  - Si il n'est pas mort est que *respawn* est à 0: On exécute le tour du joueur à l'aide de la fonction tour.
- Etape 4 : Si la variable *isdead* du joueur est *True* alors il est mort et on le supprime de la liste des *players*.
- Etape 5 : Si le nombre de Vivants est nul ou que le nombre de Zombie est nul, la simulation est terminée : on exécute un *QtGui.QMessageBox.question*.
- Etape 6 : On met à jour les widgets et les items pour les affichages dans les onglets

## 7. Lancement

Le programme se lance sur une boîte de dialogue permettant de choisir les options de la simulation. Cette boîte de dialogue propose de choisir le nombre de zombies et de survivants initiaux, ainsi que d'activer les interventions militaires (arrivée de militaire et d'un stock de nourriture à un tour définie). Cette boîte de dialogue est un *QDialog*, lancé dans le main du programme, et démarrant l'interface de simulation (*QMainWindow*).

Le programme fonctionne au moins sous Python 3.4 sur Spyder de WinPython.