

Rapport de Projet :
Programmation en Python du jeu Virus Killer

Pierre Jacquet, Thomas Blanc

27/11/2017

1 Introduction

Ce rapport de projet, présente le développement d'un petit jeu, nommé "Virus Killer", ressemblant au célèbre jeu "Bomberman".

Le but du projet est de réaliser un programme en langage python, suivant les instructions indiquées. Pour cela, une analyse des consignes et mécaniques du jeu a été réalisée afin de les traduire en concepts logiques appropriés à la programmation, puis des solutions pour implémenter ces concepts au sein du programme ont été élaborées. Enfin, le code Python a été implémenté, et sera détaillé dans la dernière partie de ce document.

2 Analyse

La description du jeu a été transposée en règles et objets plus facilement adaptables à des éléments de programmation.

2.1 Principe du jeu

Virus Killer, (VK pour les fans du jeu), est un jeu au tour par tour, dans lequel le joueur doit éliminer des virus au moyen de "médicaments/bombes" (parfois il faut savoir être radical).

Pour le joueur, une partie repose sur deux actions possibles, se déplacer et poser des bombes, ces deux actions interagissant l'une avec l'autre (il est impossible de placer une bombe après s'être déplacé), elles seront prise en charge par un seul mécanisme.

2.2 Le plateau de jeu

Le plateau de jeu est une grille de 10 lignes par 10 colonnes, en deux dimensions. Cet espace est compté en nombre de cases, pour la longueur de déplacement et le rayon d'action des bombes.

2.3 Les éléments

Les éléments sont tous les objets qui seront placés sur la grille. Ils comprennent le joueur, les virus, les bombes, les ATP, et les murs.

2.3.1 le joueur

Le joueur a un avatar unique placé aléatoirement sur la grille au début du jeu, il est capable de se déplacer verticalement ou horizontalement à chaque tour, d'un nombre de cases choisi par le joueur, et peut poser une bombe de son choix s'il ne s'est pas encore déplacé dans le tour. Il doit pouvoir se déplacer sur les cases contenant l'ATP, mais pas sur les cases contenant les virus ou les murs.

2.3.2 les virus

Les virus sont représentés par des avatars sur la grille, leur seule action possible est de se déplacer dans une des 4 directions, d'un nombre de cases choisi de façon aléatoire par le programme. Ils ne peuvent pas traverser les cases occupées par le joueur, les ATP, les bombes ou les murs. Il y a 4 virus au début de la partie, placés sur des cases aléatoires, si un virus est pris dans l'explosion d'une bombe, il est détruit et enlevé de la partie.

2.3.3 les bombes

Au début du jeu, le joueur a 4 bombes, ces bombes sont définies par leur puissance, qui sont au début égales à 8, 6, 4, et 2. Lorsqu'une bombe est posée par le joueur, sa puissance détermine le nombre de cases que prendra l'explosion (un nombre de cases égal à la moitié de la puissance au dessus de la bombe, et l'autre moitié en dessous.) Lorsqu'une bombe est posée, elle apparaît sur la case du joueur, et elle explose au tour suivant. Dans le chargeur du joueur la bombe utilisée est remplacée par une autre qui prend une valeur de puissance choisie aléatoirement entre 8, 6, 4 et 2. À chaque tour, les bombes n'ayant pas été utilisées voient leur puissance diminuer de 1, et lorsque le joueur passe sur une case contenant de l'ATP, la puissance de deux bombes choisies aléatoirement augmente de 1 point.

2.3.4 les ATP

Les cases contenant de l'ATP augmentent la puissance des bombes du joueur lorsqu'il passe dessus, comme décrit précédemment. Il y en a 8 sur la grille, placées aléatoirement au début de la partie. Lorsqu'un ATP est utilisé par le joueur, la case sur lequel il est deviens une case vide, et une case vide du plateau choisie aléatoirement devient une case ATP.

2.3.5 les parois cellulaires

Les parois cellulaires ont effectivement la fonction de murs, ni le joueur ni les virus ne peuvent traverser les cases de parois cellulaire. Si une bombe explose et qu'une case de paroi se trouve dans le rayon d'explosion, la case de paroi sera détruite, et remplacée par une case vide.

2.4 Les états de victoire et défaites

La condition de victoire est que le joueur détruise tous les virus, elle déclenche un écran de victoire, et le jeu s'arrête. La défaite du joueur surviens lorsque toutes les bombes du joueur ont une puissance de 0, ce qui déclenche un écran de défaite et arrête la partie.

2.5 Initialisation du jeu et Déroulement d'un tour

Lorsque la partie est initialisée, les parois cellulaires, ATP, virus et le joueur sont placés sur la grille, puis le premier tour commence. Chaque tour se déroule dans l'ordre suivant :

- Début du tour.
- Si une bombe a été posée au cours du tour précédent, elle explose.
- Si des ATPs ont été consommé ou détruits, ils sont régénérés comme décrit dans la partie ATP.
- Le joueur peut poser une bombe et se déplacer.
- Les virus se déplacent.
- Fin du tour, début du tour suivant.

3 Conception

3.1 Jouabilité

Pour que l'utilisateur est un peu d'intérêt pour le jeu, il faut que les parties soit dynamiques en permettant au joueur d'effectuer rapidement ses actions, sans être gêné par une interface trop encombrée, contenant trop de texte. Il semble préférable d'axer l'interaction joueur - jeu, sur l'utilisation de raccourcis clavier, en reprenant par exemple, les touches directionnelles biens connues des joueurs (français), Z Q S D, ainsi que 1 2 3 4 pour l'utilisation des bombes. Il est donc nécessaire, de détailler l'utilisation de ces touches, dans un écran d'instructions, accessibles avant le lancement d'une partie.

3.2 Interface utilisateur

Étant donné le souhait de rester dans un environnement simple de terminal, l'esthétique du jeu sera travaillé par plusieurs *print* de séquences de texte et caractères ASCII, à différents endroits, risquant certainement, d'alourdir la lecture du code. Cependant cette contrainte est nécessaire, pour éviter l'utilisation de modules externes, qui permettrait de gérer l'interface de manière plus flexible.

3.3 Représentation de la grille dans les données.

Pour la représentation de la grille au niveau des variables, une liste a deux dimensions, (une dimension codant pour les coordonnées de ligne, et l'autre de colonne) a d'abord été envisagée. Il a cependant été décidé d'utiliser une liste de 100 cases, numérotées de 0 à 99. Dans cette liste, les déplacements horizontaux seront traduits en déplacements de +1 ou -1 dans l'index de la case, selon si on va vers la gauche ou la droite, et les déplacements verticaux en déplacement de +10 ou -10 dans l'index de la case, selon si on va vers le haut ou le bas.

3.4 La fonction de mouvement

Pour la façon d'implémenter les commandes de mouvement, il a d'abord été envisagé de demander au joueur de combien de cases il souhaite se déplacer et dans quelle direction, ou de proposer des touches de mouvement déplaçant l'avatar du joueur d'une case à la fois, avec une touche pour terminer la phase de déplacement. La deuxième solution a été retenue pour son côté intuitif.

3.5 Les parois cellulaires et limites de la grille.

Les parois cellulaires sont définies comme un type de case, donc pour empêcher le joueur ou les virus de passer, un simple test conditionnel pour voir si le mouvement va sur une case de paroi a été utilisé.

Pour délimiter les limites de la grille et empêcher les éléments mobiles d'en sortir ou de passer d'un bord au bord opposé, des cases mur entourant la zone de jeu ont été envisagés, mais il a plutôt été décidé de vérifier que le déplacement n'est pas en dehors de la grille et d'interdire les tentatives de déplacement illicites, pour éviter de complexifier les calculs et d'avoir une grille trop grande.

3.6 Représentation des bombes

Le chargeur de bombes du joueur a été représenté sous forme de dictionnaire, avec le nom de chaque bombe pour clé, et des variables correspondant à la puissance et la position de la bombe associées à ces clés. Cette représentation a été choisie car elle permet de conserver toutes les données utile à une bombe dans une seule variable.

3.7

4 Réalisation

Virus Killer a été codé en python 3 (version 3.5.3). Nous avons choisi cette version de python disponible sur les ordinateurs du CREMI, car elle permet de changer les caractères en fin de ligne, lors d'un print de manière simple.

Le jeu a été conçu pour un terminal en plein écran. L'affichage du jeu dans étant relativement grand il peut être nécessaire selon les écrans d'ajuster la taille de police du terminal. La manipulation est simple, il suffit de changer le profil du terminal (accessible par cliquer droit sur ce dernier, en changeant les paramètres d'affichage et notamment de police). Cela permet une meilleure lisibilité des composants du jeu.

Un menu a été créé permettant d'accéder au jeu en lui-même, aux instructions ou à la sortie du programme.

Le programme est disponible sur le serveur interne du CREMI:

```
/net/stockage/bioinfo_2017_M1/Pierre_Thomas/viruskiller.py
```

et également sur Github:

```
https://github.com/pierrejacquet/biovirusgameproject
```

4.1 Note sur l'utilisation de variables à des fins graphiques

Il a été décidé par souci de lisibilité, de placer dans un autre fichier (`viruskillerdrawing.py`), toutes les variables (la majorité étant de type *string*) dont l'utilisation est essentiellement "graphique". Il y a notamment, les dessins en ASCII, mais également les balises de couleur. Ces variables sont utilisées tout au long du programme pour colorer les lignes affichées par le terminal.

Ces variables de couleurs sont très utilisées dans le programme mais n'ont qu'une utilité graphique. Les tabulations "\t" permettent un espacement égal entre les éléments. Enfin la ligne : `"os.system("clear")"` permet de vider le terminal de tous les éléments précédemment affichés.

4.2 Début du Programme et initialisation du plateau

Après l'affichage des crédits, le joueur accède au menu permettant de lancer le jeu ou l'affichage des instructions. Celles-ci sont affichées par du texte et des dessins de caractères ASCII.

Le lancement du jeu dans le menu, appelle la fonction ***startgame***. La fonction se décompose en deux parties. La première initialise les différents éléments du plateau:

1. placement des murs - ***initmurs()***
2. placement du joueur - ***initjoueur()***
3. placement des virus - ***initvirus()***
4. placement de l'ATP - ***spawnATP()***
5. affichage de la grille (plateau de jeu) - ***showGameBoard()***

La deuxième partie de la fonction, est une boucle *while* qui définit les tours de jeu et qui continue tant qu'il n'y a ni victoire ni défaite. Elle sera détaillée un peu plus loin.

4.2.1 La grille

Le système de plateau de jeu se base sur l'utilisation d'une grille de 100 éléments (0 à 99).

L’affichage de la grille, par la fonction **showGameBoard()** correspond (outre les ajouts purement esthétiques - comme un dictionnaire contenant des éléments de l’interface) à une boucle *for* de range 100, qui print chaque éléments de la liste *grille* un par un sans revenir à la ligne.

Le retour à la ligne est réalisé tous les 10 éléments (correspond aux valeurs 9, 19, 29, 39 etc...), c’est à dire lorsque:

$$\text{indice de la liste} \% 10 == 9$$

Figure 1: Schéma représentatif de la liste grille affiché dans le terminal - i correspond ici à l’indice de la liste

4.2.2 Génération des murs sur la grille

Le joueur doit choisir parmi 3 niveaux de difficulté. Ce choix conditionne le nombre de murs et la longueur des murs qui vont être immédiatement générés sur la grille. Pour les murs la plupart des paramètres sont générés aléatoirement, afin d’apporter une grande diversité à chaque partie.

Par exemple, si le joueur saisi une difficulté de 3, le nombre de mur est un nombre aléatoire entre 5 et 6, la longueur de ces murs est un nombre de case aléatoire entre 6 et 7.

Il est important de rappeler que bien qu’il y ait des murs au sein de la grille, il n’y a pas de parois cellulaires externes en tant que tel. Il est possible d’aller sur une case orange ou bleue, mais pas de passer de l’une à l’autre.

Chaque mur, se voit assigné une position de "départ" aléatoire sur la grille avec la fonction **randomin-grille(0,99)** (qui génère une position aléatoire entre les deux nombres en paramètres) et un sens (vertical ou horizontal). En fonction du sens du mur chaque portion de mur à partir de la position de départ est testée pour s’assurer que la position existe dans la grille.

Pour cela la fonction **passemuraille(oldpos,newpos)** est utilisée. Cette fonction prend comme paramètres la dernière position validée (oldpos) et la nouvelle position testée (newpos). La fonction renvoie "False" si la nouvelle position testée est bien comprise entre 0 et 99 et si lorsque la dernière position valide se trouve sur une bordure, la nouvelle position testée ne correspond pas à une position se trouvant à l’opposé de la grille. Ceci permet par exemple qu’un mur horizontal qui a comme position de départ 21 et une longueur de 3, s’étende vers la gauche, à la position 20, mais ne continue pas à la position 19 car elle correspond à l’autre coté de la grille. Au contraire, dans l’impossibilité d’aller plus vers la gauche, le mur s’étend vers la droite.

La fonction **passemuraille()** sera utilisée à plusieurs reprises dans la suite du code afin notamment de valider les déplacements des personnages.

4.2.3 Placement du joueur et des virus

Le joueur est placé sur la grille à une position aléatoire grâce à la fonction **randomin-grille(0,99)**. La fonction **initvirus()** génère 4 index de grille différent tant que la position générée n’est pas vide puis place les virus (remplace la valeur dans la liste par la string correspondant au virus).

4.2.4 Placement des molécules d’ATP

La fonction **spawnATP()** s’occupe à la fois de l’initialisation des ATP et de leur rechargement sur la grille après que le joueur en ait ramassé. D’abord le nombre d’ATP présent dans la grille est compté grâce à la fonction **countATP()** qui parcourt toute la liste *grille* en incrémentant le nombre d’ATP compté. Tant que le nombre d’ATP est inférieur à 8, un ATP est généré aléatoirement sur la grille.

4.3 Déroulement d’un tour de jeu

La deuxième partie de la fonction **startgame()** est constituée d’une boucle *while* qui définit un tour de jeu. Tant que les conditions de victoire ou de défaite ne sont pas réunies la boucle continue (ces conditions seront détaillées plus loin). Un tour se déroule de la manière suivante:

1. déplacement aléatoire des virus - ***randommovevirus()***
2. une boucle while permet au joueur de poser des bombes ou de se déplacer, tant qu'il n'a pas utilisé la touche espace - ***actionjoueur()***
3. explosion des bombes éventuellement posées - ***boom()***
4. la liste *virus* est mise à jour pour prendre en compte les virus mort - ***constatdesmorts()***
5. rechargement les molécules d'ATP sur la carte - ***spawnATP()***
6. recréation d'une bombe de valeur aléatoire si une bombe a été posée - ***reloadbombe()***.
7. décrémente la puissance de deux bombes aléatoires - ***bombemolle()***
8. vérifie les conditions de victoire - ***win()***
9. vérifie les conditions de défaite - ***rip()***

4.3.1 Déplacement des virus

Pour chaque virus, la fonction ***dirpossiblevirus(virus,numvirus)*** qui prend comme paramètres la liste des positions des virus et la position du virus concerné, renvoie une liste des directions (0,1,2,3) dans lesquelles le virus peut se déplacer sans être immédiatement bloqué. La direction est alors tirée aléatoirement grâce à `random.choice()` ciblant la liste. La fonction calcule ensuite depuis la position du virus, la distance des bords de la grille dans cette direction.

Si un virus est à la position 62 dans la grille, il peut parcourir $\text{int}(62/10) = 6$ cases vers le haut, ou encore $62\%10 = 2$ cases vers la gauche. La distance qu'il parcourt alors est tirée aléatoirement entre 1 et cette distance max. Le virus avance alors "pas à pas" dans une boucle qui appelle la fonction ***movevirus(virus,numvirus,dirvalue)*** pour chaque "pas" qu'il a à parcourir. Cette fonction vérifie que le déplacement est possible (le virus ne peut traverser un mur, une paroi, un joueur, ou de l'ATP) en regardant si la position cible correspond à la valeur "*casevide*" dans la liste.

4.3.2 Les actions du joueur

Le fonctionnement central des actions réalisables par le joueur, repose sur la liste *mouvement*, qui contient 4 éléments. Le premier est la position effective du joueur (*oldpos*), le deuxième est la position à tester (*newpos*), le troisième est le type de déplacement (pas de déplacement, déplacement horizontal ou vertical) et enfin le dernier est une valeur (0 ou 1) qui indique si le joueur souhaite continuer à faire quelque chose ou non. Les actions réalisables par le joueur passent par la fonction ***keyinput(mouvement)***, qui modifie le contenu de la liste *mouvement* en fonction de l'input du joueur et renvoie cette liste ainsi que la valeur de l'input. Tant que le joueur ne saisie pas un symbole correct, la demande de saisie est renouvelée.

La fonction ***actionjoueur()***, gère en fonction de ce que le joueur a saisi, l'action à effectuer. Si, le joueur à saisi un espace, il indique qu'il souhaite finir son tour, `keyinput()` a renvoyé la liste *mouvement*, avec `mouvement[3] = 0`, le programme sort de la boucle.

Si la valeur de déplacement à tester (`mouvement[2]`) est différente de la position du joueur, il s'agit d'un déplacement, la fonction vérifie que le déplacement n'est pas interdit avec ***passemuraille()*** et procède au déplacement. Pour cela, la valeur dans la liste *grille* à l'index *oldpos* est remplacé par *casevide* et la valeur dans la liste *grille* à l'index *newpos* est remplacé par *casejoueur*. Si la nouvelle position correspond à une *caseATP*, la fonction ***boostbombe(bombeloader)*** est appelée, prenant comme paramètre le dictionnaire *bombeloader* qui contient la position et la puissance des bombes, et incrémente aléatoirement de 1, la puissance de deux bombes différentes.

Si la valeur de déplacement à tester (`mouvement[2]`) est identique à la position du joueur, il ne s'agit pas d'un déplacement, mais d'une tentative de poser une bombe. La fonction vérifie que le joueur n'a effectué aucun mouvement au préalable (`mouvement[3] == "n"`) et remplace la valeur de la liste *grille* à l'index (*newpos*) par *casejouurbomb*. De plus la fonction lit la valeur de l'input du joueur et en fonction du type de bombe sélectionné (1,2,3 ou 4), renseigne dans le dictionnaire *bombeloader* la position de la bombe correspondante sur la liste *grille*.

Tant que le joueur n'a pas appuyé sur espace lors de l'input la fonction ***keyinput()*** est rappelée.

4.3.3 Explosion des bombes

L'explosion des bombes, s'effectue grâce à **boom()**. Au début de la fonction, une boucle *for* parcourt rapidement les clés du dictionnaire *bombeloader*, regarde si une bombe a une valeur de position différente de "n" et si tel est le cas stock sa position et sa puissance dans une liste *bombeactive*. Dans le cas où cette liste n'est pas vide, (une bombe est donc bien active), le rayon d'explosion est calculé. Le rayon correspond à la puissance de la bombe divisé par deux et arrondi au supérieur.

L'explosion, à partir de la bombe, se fait vers le haut d'une longueur égale au rayon et vers le bas également. Grâce à une boucle *while*, chaque position d'explosion est testée pour s'assurer qu'elle ne sort pas de la liste *grille*. Dans un premier temps, la liste *grille* est remplacée par des étoiles aux positions d'explosions, puis par *casevide*.

Il a été décidé, pour ajouter de l'intérêt aux bombes de faibles puissance (dont le rayon est inférieur ou égale à 2), de réaliser une explosion en croix. Dans ce cas le comportement est le même que précédemment avec un test en plus sur les côtés pour s'assurer que l'explosion latérale est possible. Enfin, une fois l'explosion terminée, les valeurs de position dans le dictionnaire *bombeloader* sont toutes remises à "n" et la puissance de la bombe qui a explosé est remplacée par la string "X".

4.3.4 Virus morts, Rechargement de l'ATP, des bombes et décrémentation des bombes.

La fonction **constatdesmorts()** compare les positions stockées dans la liste *virus* avec les positions dans la liste *grille*. Si à la position testée, la valeur dans la liste *grille* est celle d'une *casevide* alors, la fonction actualise la liste *virus* en remplaçant la position par la string "mort" dans la liste.

La fonction **spawnATP()** est ré-invoquée, puis la fonction **reloadbombe()**, parcourt le dictionnaire *bombeloader* à la recherche d'une bombe dont la puissance a "X" comme valeur. Cette valeur est remplacée par une valeur aléatoire entre (3,5,7,9), valeurs majorées de 1 en prévision de la fonction suivante **bombemolle()** qui décrémente toutes les bombes enfin de tours. Cela revient à ne décrémenter que les bombes qui n'ont pas été utilisées.

4.4 Fin de tour, Victoire ou Défaite

En fin de tour, la variable *victory* est mise à jour en prenant comme valeur ce que retourne la fonction **win()** (0 ou 1) qui prend comme paramètre la liste *virus*. Cette fonction parcourt la liste, et compte le nombre de string "mort". Si cette valeur est égale à 4, *victory* = 1, un message de victoire est affiché à l'écran.

La variable *rip* (rest in peace), prend également comme valeur ce que retourne la fonction **loose(bombeloader)** (0 ou 1) qui prends comme paramètre le dictionnaire *bombeloader*. La fonction parcourt les clés du dictionnaire et regarde si toutes les bombes ont une valeur égale à zero. si c'est le cas, un message de défaite est affiché à l'écran.

La boucle *while* correspondant au tour de jeu, s'arrête donc, si *victoire* = 1 ou si *rip* = 1.

5 Conclusion

Le programme présenté, permet de jouer au jeu Virus Killer et réponds aux consignes du projet. Plusieurs points peuvent être améliorés, à la fois au niveau du ressenti vis à vis du jeu (gameplay) et du code en lui même.

Il pourrait être nécessaire d'uniformiser la langue employée, en commentant le code et en utilisant des variables, en anglais. De même, comme précisé plus haut, les touches Z, Q, S, D, passant par des inputs sont adaptés aux claviers français, car ils s'adaptent à la position des doigts. Ce n'est plus le cas sur des claviers QWERTY notamment. Il serait donc intéressant, de rendre les touches modifiables et de passer non plus par des inputs, mais par des *keylistener* adaptés à Python.

Concernant, les règles du jeu, il semble que l'utilisation des bombes de faibles valeurs de puissance, a peu d'intérêt, ce qui entraîne fréquemment, un abandon de ces bombes par le joueur qui préfère se focaliser sur l'utilisation d'une seule bombe, et sur la génération aléatoire de sa puissance chaque fois qu'il l'a réutilise. C'est pourquoi, l'explosion en croix a été implantée, mais cela reste très certainement insuffisant. On pourrait imaginer, d'autres types d'explosion, ou bien de reporter les mécanismes de défaite, non pas sur la perte de puissance des bombes mais sur une action particulière des virus.

Le développement d'un jeu directement dans la console a l'avantage d'être simple, mais semble vite limité aux mécanismes d'affichages "lignes par lignes".

Cependant le jeu offre déjà quelques possibilités d'amusement, il est relativement modulable permettant ainsi, d'implémenter de nouveaux modes de difficultés ou de changer les éléments de façon non destructrice pour le code.

viruskiller.py

[illegible]

[illegible]

[illegible]

```

148 # Renvoie true si la valeur testée sort de la grille ou revient par l'autre coté
149 def passerunraille(olddpos, newpos):
150     if (newpos < 0 or newpos > 99 or (olddpos % 10 == 9 and newpos % 10 == 0) or (olddpos % 10 == 0 and newpos % 10 == 9)):
151         return True
152     else:
153         return False
154
155 # Initialisation des murs, choix de la difficulté (nombre et longueur des murs), sens aléatoire.
156 def initmurs():
157     os.system("clear")
158     # Choix de la difficulté génère des valeurs différentes pour le nombre de murs et leurs longueurs.
159     print("\n\t A quel niveau de difficulté voulez vous jouer ?\n\n\t " + BLEU + "Rhume: (1)" + ORANGE + "Gastro: (2)" + ROUGE + "Chikungunya: (3)" + BLANC)
160     inputdifficulty = input("\n\n\t Niveau: ")
161     if inputdifficulty == "1":
162         nbmur = random.randint(3, 4)
163         longmur = random.randint(2, 3)
164         elif inputdifficulty == "2":
165             nbmur = random.randint(4, 5)
166             longmur = random.randint(4, 5)
167         elif inputdifficulty == "3":
168             nbmur = random.randint(5, 6)
169             longmur = random.randint(6, 7)
170         else:
171             initmurs()
172
173 # Boucle while pour générer tous les murs (while mur <= nbmur)
174 mur = 1
175 while mur <= nbmur:
176     # On génère en suite soit un mur vertical (sensmur=1) soit un mur horizontal (sensmur=2)
177     sensmur = random.randint(1, 2)
178     # On tire une position au hasard sur la grille pourvu que la place soit disponible (avec la fonction randomingrille)
179     startingpoint = randomingrille(0, 99)
180     grille[startingpoint] = casemur
181     # Pour les verticaux on remplit vers le haut. Si l'on sort de la grille (fonction passerunraille), on commence à remplir vers le bas.
182     # On a donc besoin d'avoir des compteurs indépendant pour haut et bas.
183     if sensmur == 1:
184         haut = 0
185         bas = 0
186         for brique in range(1, longmur):
187             oldpos = startingpoint - 10 * haut
188             newpos = startingpoint - 10 * (haut + 1)
189             if passerunraille(olddpos, newpos) is False:
190                 grille[newpos] = casemur
191                 haut = haut + 1
192
193     else:
194         oldpos = startingpoint + 10 * bas
195         newpos = startingpoint + 10 * (bas + 1)
196         if passerunraille(olddpos, newpos) is False:
197             grille[newpos] = casemur

```

```

198     bas = bas + 1
199
200     # Pour les horizontaux on remplit vers la gauche... si l'on sort de la grille, on commence à remplir vers la droite.
201     # On a donc besoin d'avoir des compteurs indépendants pour gauche et droite.
202     else:
203         gauche = 0
204         droite = 0
205         for brique in range(1, longmur):
206             oldpos = startingpoint - 1 * gauche
207             newpos = startingpoint - 1 * (gauche + 1)
208             if passeruraille(oldpos, newpos) is False:
209                 grille[newpos] = casemur
210                 gauche = gauche + 1
211
212
213     else:
214         oldpos = startingpoint + 1 * droite
215         newpos = startingpoint + 1 * (droite + 1)
216         if passeruraille(oldpos, newpos) is False:
217             grille[newpos] = casemur
218             droite = droite + 1
219
220     mur += 1
221
222 # Initialisation du joueur
223 def initjoueur(mouvement):
224     joueur = randomgrille(0, 99)
225     mouvement[1] = joueur
226     grille[joueur] = casejoueur
227     return mouvement
228
229 # Permet de compter combien d'ATP est présent dans la grille
230 def countATP():
231     nbATP = 0
232     for i in range(len(grille)):
233         if grille[i] == caseATP:
234             nbATP = nbATP + 1
235     return nbATP
236
237 # Permet de générer de l'ATP sur un emplacement vide de la grille
238 def randomATP():
239     randomATPpos = randomgrille(0, 99)
240     return randomATPpos
241
242 # Génère de l'ATP tant qu'il y a moins de 8 molécules dans la grille.
243 def spawnATP():
244     nbATP = countATP()
245     while nbATP < 8:
246         newATP = randomATP()
247         grille[newATP] = caseATP
248         nbATP = countATP()
249     showGameBoard(grille, message)

```

[illegible]


```

298 mouvement = [oldpos, newpos, voh, continuer]
299 return mouvement, inputkey
300
301 # Fonction de déplacement du joueur et de posage de bombe - interprete la fonction keyinput.
302 def actionjoueur(mouvement, bombeloader):
303     testmouvement, inputkey = keyinput(mouvement)
304     oldpos = testmouvement[0]
305     newpos = testmouvement[1]
306     voh = testmouvement[2]
307     continuer = testmouvement[3]
308
309     if continuer == 0:
310         mouvement = [oldpos, newpos, voh, continuer]
311         return mouvement, bombeloader
312
313     elif newpos != oldpos:
314         if passemuraille(oldpos, newpos) == True or grille[newpos] not in [casevide, caseATP]: # Cas où le déplacement n'est pas autorisé
315             continuer = 1
316             message = "Vous ne pouvez pas aller là" + JAUNE + "\t\t" + BLANC
317         elif grille[newpos] == casevide: # Cas où le déplacement est autorisé et va sur une case vide
318             if (grille[oldpos] == casejoueurbomb): # Cas où à l'ancienne position le joueur avait amorcé la bombe
319                 grille[oldpos] = casebomb
320             else: # Cas standard où l'ancienne position devient vide après déplacement du joueur
321                 grille[oldpos] = casevide
322                 grille[newpos] = casejoueur
323             mouvement = [oldpos, newpos, voh, continuer]
324             message = "Vous avancez" + JAUNE + "\t\t\t" + BLANC
325         elif grille[newpos] == caseATP: # Cas où le déplacement est autorisé et va sur une case d'ATP
326             if grille[oldpos] == casejoueurbomb:
327                 grille[oldpos] = casebomb
328             else:
329                 grille[oldpos] = casevide
330                 grille[newpos] = casejoueur
331             mouvement = [oldpos, newpos, voh, continuer]
332             message = "Vous avancez" + JAUNE + "\t\t\t" + BLANC
333             boostbombe(bombeloader)
334
335     # Si la nouvelle position testée est identique à l'ancienne il s'agit d'une tentative d'amorçage de bombe
336     # (la condition de fin de tour (continuer=0) où le joueur reste à la même position ayant été préalablement testé avec le premier if).
337     elif (newpos == oldpos and voh != "n"): # Cas où la bombe ne peut être posé car le joueur s'est déplacé
338         grille[newpos] = casejoueur
339         message = "Bombes inaccessibles après déplacement" + JAUNE + "\t" + BLANC
340
341     elif (newpos == oldpos and voh == "n"): # Cas où la bombe peut être posé
342         grille[newpos] = casejoueurbomb
343         tabulationdumessage = JAUNE + "\t\t" + BLANC
344         print ("INPUT=", inputkey)
345         if inputkey == "1":
346             bombeloader["bombel"][0] = newpos
347             message = "Vous venez de déposer la bombe 1" + tabulationdumessage

```

```

348         elif inputkey == "2":
349             bombeloader["bombe2"][0] = newpos
350             message = "Vous venez de déposer la bombe 2" + tabulationdumessage
351             elif inputkey == "3":
352                 bombeloader["bombe3"][0] = newpos
353                 message = "Vous venez de déposer la bombe 3" + tabulationdumessage
354                 elif inputkey == "4":
355                     bombeloader["bombe4"][0] = newpos
356                     message = "Vous venez de déposer la bombe 4" + tabulationdumessage
357             showGameBoard(grille, message)
358             return mouvement, bombeloader
359
360     # Renvoie pour un virus les directions possibles
361     def dirpossiblevirus(virus, numvirus):
362         posvirus = virus[numvirus]
363         dirpossible = []
364         if posvirus > 9:
365             if grille[posvirus - 10] == casevide:
366                 dirpossible.append(0)
367         if posvirus < 90:
368             if grille[posvirus + 10] == casevide:
369                 dirpossible.append(1)
370         if posvirus > 0:
371             if grille[posvirus - 1] == casevide and posvirus % 10 != 0: # le virus ne peut aller vers la gauche si il est contre la paroi
372                 dirpossible.append(2)
373         if posvirus < 99:
374             if grille[posvirus + 1] == casevide and posvirus % 10 != 9: # le virus ne peut aller vers la droite si il est contre la paroi
375                 dirpossible.append(3)
376         if dirpossible == []: # Dans le cas où les conditions ne sont pas remplies le virus essaie d'aller vers la gauche (sinon une erreur est renvoyé)
377             dirpossible.append(1)
378             return dirpossible
379
380     # Appel la fonction de déplacement aléatoire des virus en fonction de la direction possible
381     def randommovevirus(virus):
382         for numvirus in range(len(virus)):
383             if virus[numvirus] != "mort":
384                 oldvirus = virus[numvirus]
385                 dirpossible = dirpossiblevirus(virus, numvirus) # choix des directions où le virus ne sera pas bloqué
386                 direction = random.choice(dirpossible)
387                 if direction == 0: # haut
388                     dirvalue = -10
389                     maxdistance = int(oldvirus / 10)
390                     elif direction == 1: # bas
391                         dirvalue = 10
392                         maxdistance = 10 - int(oldvirus / 10)
393                     elif direction == 2: # gauche
394                         dirvalue = -1
395                         maxdistance = oldvirus % 10
396                     elif direction == 3: # droite
397                         dirvalue = +1

```

```

398         maxdistance = 10 - (oldvirpos % 10)
399         distance = random.randint(1, maxdistance) # distance a parcourir
400         pas = 0
401         while pas < distance:
402             movevirus(virus, numvirus, dirvalue)
403             pas = pas + 1
404
405     # Fonction de déplacement des virus
406     def movevirus(virus, numvirus, dirvalue):
407         oldvirpos = virus[numvirus]
408         newposvir = oldvirpos + dirvalue
409         message = "Les virus se déplacent" + JAUNE + "\t\t\t" + BLANC
410         if (newposvir > 0 and newposvir < 100):
411             if passemuraille(oldvirpos, newposvir) is True:
412                 message = "Le virus tente en vain de passer la paroi" + JAUNE + "\t" + BLANC
413                 time.sleep(0.3)
414             else:
415                 if grille[newposvir] == casevide:
416                     grille[oldvirpos] = casevide
417                     virus[numvirus] = newposvir
418                     grille[newposvir] = casevirus
419                     time.sleep(0.3)
420                 else:
421                     time.sleep(0.1)
422             showGameBoard(grille, message)
423
424     # Sélectionne deux bombes aléatoirement parmi le chargeur et incrémente de 1 sa puissance
425     def boostbombe(bombelader):
426         selectrandombombe = random.sample(list(bombelader.keys()), 2)
427         bombelader[selectrandombombe[0]][1] = bombelader[selectrandombombe[0]][1] + 1
428         bombelader[selectrandombombe[1]][1] = bombelader[selectrandombombe[1]][1] + 1
429         return bombelader
430
431     # Fonction explosion des bombes
432     def boom(bombelader):
433         activebombe = []
434         for bombe in bombelader.keys(): # détecte si une bombe a une valeur de position différente de "n"
435             if bombelader[bombe][0] != "n":
436                 activebombe.append(bombelader[bombe][0])
437                 activebombe.append(bombelader[bombe][1])
438                 bombelader[bombe][1] = "X"
439
440             if activebombe != []:
441                 posbombes = activebombe[0]
442                 rayon = int(activebombe[1] / 2) + (activebombe[1] % 2 > 0)
443                 print (rayon)
444                 for motif in [ROUGE + "\t\t\t" + BLANC, casevide]:
445                     grille[posbombes] = motif
446                     i = 1
447                     while i <= rayon:

```

```

448     if posbombes - 10 * i >= 0:
449         grille[posbombes - 10 * i] = motif
450     if posbombes + 10 * i < 99:
451         grille[posbombes + 10 * i] = motif
452     if rayon <= 2: # Explosion en croix pour les bombes de rayons <= 2
453         if posbombes - 1 * i >= 0 and (posbombes - 1 * i) % 10 != 9:
454             grille[posbombes - 1 * i] = motif
455         if posbombes + 1 * i < 99 and (posbombes + 1 * i) % 10 != 0:
456             grille[posbombes + 1 * i] = motif
457     i = i + 1
458     message = "BOOOOM" + JAUNE + "\t\t\t\t\t" + BLANC
459     showGameBoard(grille, message)
460     time.sleep(1)
461     for item in bombelader.keys():
462         bombelader[item][0] = "n"
463     return bombelader
464
465     # Fonction de recharge de bombe après explosion
466     def reloadbombe(bombelader):
467         for slot in bombelader.keys():
468             if bombelader[slot][1] == "X":
469                 val = random.sample([3, 5, 7, 9], 1)
470                 bombelader[slot][1] = val[0]
471
472     # Fonction qui décrémente la puissance des bombes de 1
473     def bombemolle(bombelader):
474         for slot in bombelader.keys():
475             if bombelader[slot][1] != 0: # Les bombes ne peuvent pas avoir une puissance négative
476                 bombelader[slot][1] = bombelader[slot][1] - 1
477         showGameBoard(grille, message)
478         return bombelader
479
480     # Regarde quels sont les virus qui sont mort (après explosion)
481     def constatedsmorts(virus):
482         for individus in range(0, 4):
483             if virus[individus] != "mort":
484                 if grille[virus[individus]] == casevide:
485                     virus[individus] = "mort"
486         return virus
487
488     # Gagné si 4 virus sont mort (nbmort == 4)
489     def win(virus):
490         victory = 0
491         nbmort = virus.count("mort")
492         if nbmort == 4:
493             victory = 1
494             os.system('clear')
495             print (iconvictory)
496             time.sleep(3)
497             return victory

```

```

498 # Perdu si les 4 bombes sont à zéro de puissance
499 def loose(bombeloder):
500     bombeazero = 0
501     for item in bombeloder.keys():
502         if bombeloder[item][1] <= 0:
503             bombeazero = bombeazero + 1
504     bombeazero = bombeazero + 1
505     rip = 0
506     if bombeazero == 4:
507         rip = 1
508         os.system('clear')
509         print (icondefaite)
510         time.sleep(3)
511         return rip
512
513
514
515 # Fonction tour de jeu
516 def startgame(virus, mouvement, grille, message, bombeloder, ATP):
517     initmurs()
518     initjoueur(mouvement)
519     virus = initvirus(virus)
520     spawnATP()
521     showGameBoard(grille, message)
522     victory = win(virus)
523     rip = loose(bombeloder)
524
525     while victory == 0 and rip == 0:
526         randommovevirus(virus)
527         while mouvement[3] == 1:
528             mouvement, bombeloder = actionjoueur(mouvement, bombeloder)
529             mouvement[2] = "n"
530             mouvement[3] = 1
531             boom(bombeloder)
532             virus = constatdesmorts(virus)
533             spawnATP()
534             reloadbombe(bombeloder)
535             bombemolle(bombeloder)
536             victory = win(virus)
537             rip = loose(bombeloder)
538
539             input(' Appuyer sur enter pour quitter... ') # Partie terminée
540             sys.exit()
541
542
543 def menu():
544     os.system("clear")
545     print (JAUNE + iconbombe + BLANC)
546     lore()
547     time.sleep(1)

```

