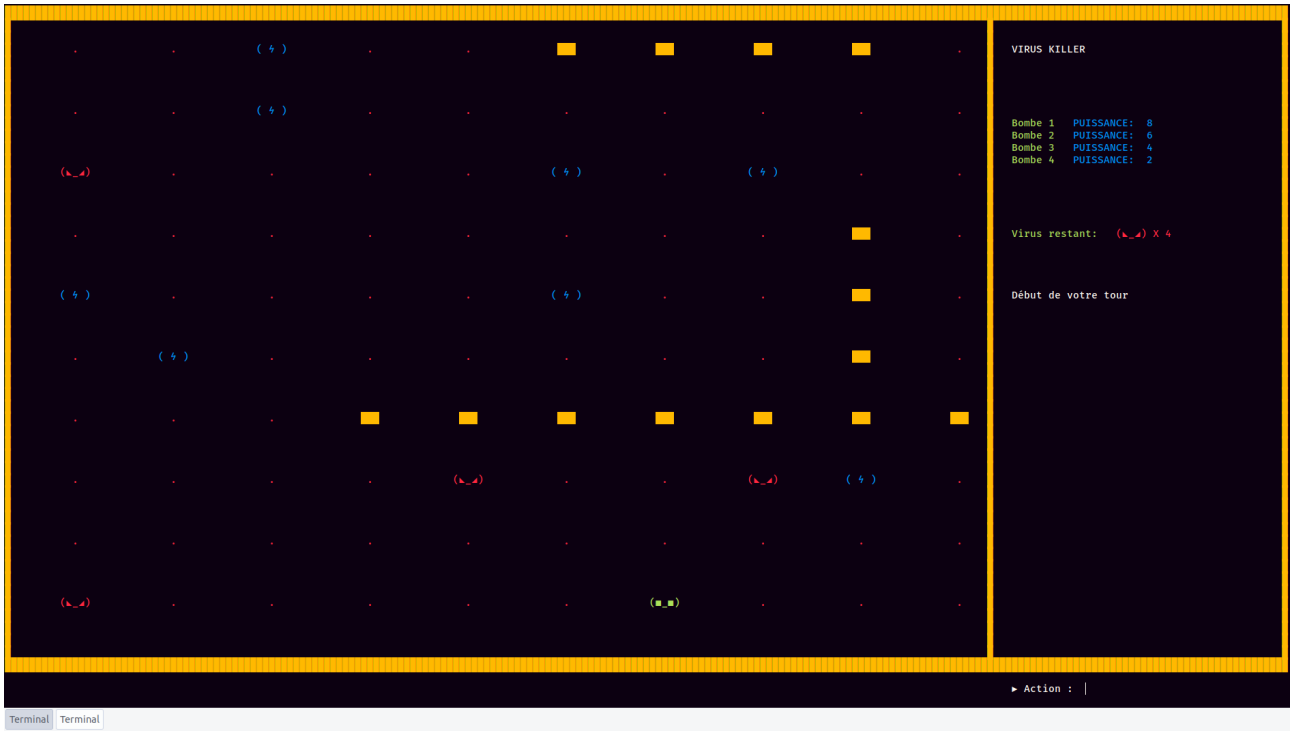
Rapport de Projet
Programmation en Python du jeu Virus Killer

Pierre Jacquet, Thomas Blanc

Université de Bordeaux

27 novembre 2017


VIRUS KILLER
Bombe 1 PUISSANCE: 8
Bombe 2 PUISSANCE: 6
Bombe 3 PUISSANCE: 4
Bombe 4 PUISSANCE: 2
Virus restant: (◣_◢) X 4
Début de votre tour
► Action :
Terminal
Terminal

Sommaire

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. Ce rapport détaillera le processus d'élaboration de ce programme. Tout d'abord, la première partie détaillera le processus d'analyse par lequel les instructions originelles et règles de jeu ont étés traduites en concepts logiques appropriés à la programmation, puis la seconde partie explicitera la conception et les solutions choisies pour implémenter ces concepts au sein du programme. Enfin, la troisième partie détaillera le code et les structures de programmation particulières employées pour sa réalisation.

1 Analyse

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

1.1 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 prises en charge par un seul mécanisme.

1.2 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.

1.3 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.

1.3.1 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 a 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.

1.3.2 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 vides aléatoires, si un virus est pris dans l'explosion d'une bombe, il est détruit et enlevé de la partie.

1.3.3 les bombes

Au début du jeu, le joueur a 4 bombes, ces bombes sont définies par leur puissance, lesquelles sont au début égales a 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. A 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.

1.3.4 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 laquelle il est devient une case vide, et une case vide du plateau choisie aléatoirement devient une case ATP.

1.3.5 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 cellulaires. 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.

1.4 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 survient lorsque toutes les bombes du joueur ont une puissance de 0, ce qui déclenche un écran de défaite et arrête la partie.

1.5 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 consommés 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.

2 Conception

2.1 Jouabilité

Pour que l'utilisateur ait un peu d'intérêt pour le jeu, il faut que les parties soient 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, accessible avant le lancement d'une partie.

2.2 Interface utilisateur

Étant donné le souhait de rester dans un environnement simple de terminal, l'esthétique du jeu sera travaillée par plusieurs *print* de séquences de textes 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 permettraient de gérer l'interface de manière plus flexible.

2.3 Représentation de la grille dans les données.

Pour la représentation de la grille au niveau des variables, une liste à 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.

2.4 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.

2.5 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 envisagées, 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.

2.6 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 associés à ces clés. Cette représentation a été choisie car elle permet de conserver toutes les données utiles à une bombe dans une seule variable.

2.7 Niveaux de difficulté

L'implémentation de 3 niveaux de difficulté a été décidée pour offrir différentes façons de construire la grille en augmentant le nombre de cases paroi cellulaire.

3 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 le terminal étant relativement grand il peut être nécessaire selon les écrans d'ajuster la taille de police. La manipulation est simple, il suffit de changer le profil du terminal (accessible par clique 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

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

Par souci de lisibilité, toutes les variables (la majorité étant de type *string*) dont l'utilisation est essentiellement "graphique" sont placées dans un autre fichier (viruskillerdrawing.py). 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.

3.2 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.

3.2.1 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 :

$$indice\ de\ la\ liste\ \%\ 10 == 9$$

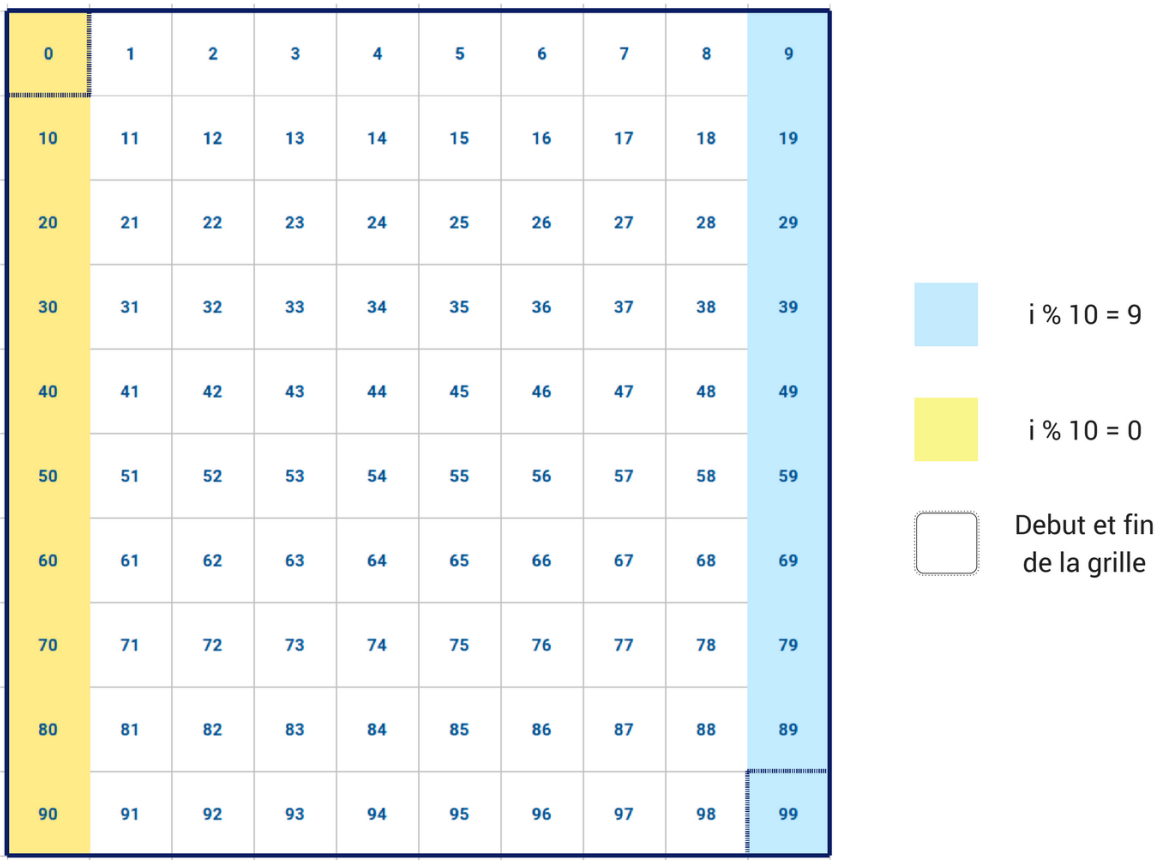


FIGURE 1 – Schéma représentatif de la liste grille affichée dans le terminal - i correspond ici à l'indice de la liste

3.2.2 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 seront 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 saisit une difficulté de 3, le nombre de murs est un nombre aléatoire entre 5 et 6, la longueur de ces murs est un nombre de cases 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 telles. 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 ***randomingrille(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.

3.2.3 Placement du joueur et des virus

Le joueur est placé sur la grille à une position aléatoire grâce à la fonction ***randomingrille(0,99)***. La fonction ***initvirus()*** génère 4 index de grille différents 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).

3.2.4 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.

3.3 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 morts - ***constatdesmorts()***
5. rechargement des 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()***

3.3.1 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 $int(62/10) = 6$ 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.

3.3.2 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 saisit pas un symbole correct, la demande de saisie est renouvelée.

actionjoueur(), gère en fonction de ce que le joueur a saisi, l'action à effectuer. Si le joueur a 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 remplacée par *casevide* et la valeur dans la liste *grille* à l'index *newpos* est remplacée 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 *casejoueurbomb*. De plus la fonction lit la valeur de l'*input* du joueur et en fonction du type de bombe selectionné (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.

3.3.3 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 à une valeur de position différente de "n" et si tel est le cas stocke 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 faible puissance (dont le rayon est inférieur ou égal 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".

3.3.4 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 en fin de tours. Cela revient à ne décrémenter que les bombes qui n'ont pas été utilisées.

3.4 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 prend 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*.

Conclusion

Le programme présenté, permet de jouer au jeu Virus Killer et répond 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 adaptées 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 faible valeur 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 la 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.

Annexe

viruskiller.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import os
import sys
import random
import time
from viruskillerdrawing import *

os.system('clear')

# grille: index de 0 à 99, represente dans le terminal par des points rouge.
grille = [casevide] * 100
virus = []
ATP = []

# bombe: [position,puissance] ; "n" pour none cad n'apparaissant pas sur la grille
bombeloader = {
    "bombe1": ["n", 8],
    "bombe2": ["n", 6],
    "bombe3": ["n", 4],
    "bombe4": ["n", 2]
}

# mouvement du joueur [ancienne position, nouvelle position, type de déplacement, continuer le déplacement]
# "n": ne s'est pas encore déplacé pendant le tour | "v" déplacé verticalement | "h" déplacé horizontalement
mouvement = [0, 0, "n", 1]

# message: utilisé parfois à certains moment pendant le tour de jeu sur l'interface utilisateur
message = "Tour de jeu" + JAUNE + "\t\t\t\t\t " + BLANC


# sautdeligne - fonction de "style" pour le print de la grille dans le terminal
def sautdeligne():
    print(JAUNE + " \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t " +
          "\t\t\t\t\t\t " + BLANC)
    print(JAUNE + " \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t " +
          "\t\t\t\t\t\t " + BLANC)
    print(JAUNE + " \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t " +
          "\t\t\t\t\t\t " + BLANC)
    print(JAUNE + " \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t " +
          "\t\t\t\t\t\t " + BLANC)

# histoire sur le menu
def lore():
    print(BLANC + """\n\n       Unité 13,\n
```

```
        C'est le moment de mettre vos nerfs à rude épreuve.                                                                    \\\ /////
        Vos derniers exploits dans la bataille contre ce satané R-HuM, sont célèbres dans le corps entier.                      |     '|
        Plusieurs virus non-identifiés ont fait irruption dans des cellules. Ils sont extrêmement dangereux.                   (| _  _ |)
        En effet de par leurs déplacements totalement aléatoires, ils sont difficilement atteignables et particulièrement irritants.  |` |   |
        Ils sont de plus capables de déclencher des stress cellulaires provoquant une érruption de réactions comme : """
           + BLANC + "\t\t               |  __  |")
    print("\t\t  - " + ROUGE + "NameError Traceback (most recent call last)" +
          BLANC + "\t\t\t\t\t\t\t\t\t\t         >>>___/\_^__/\___<<<")
    print("\t\t  - " + ROUGE + "IndexError: index out of range" + BLANC +
          "\t\t\t\t\t\t\t\t\t\t        /               |||  \\ \n")
    print(
        """       ... vous obligeant à revenir sur le menu et relancer une partie. Mais si les virus sont calmes ça ne devrait pas arriver.\n
       Unité 13... Bonne Chance""")


def credit():

    for i in range(100):
        os.system('clear')
        print(BLEU + thomaspierrot + BLANC+"\n\n\n")
        print(BLEU+"\t\t\t Loading: "+BLANC+str(100*i/100)+" %"+BLANC + " "*i + VERT +"( _ )"+ROUGE +"   ( _ )"+BLANC)
        print(""+BLEU+"\t\t\t_____________________________" + "_"*i+BLANC+"_"*(100-i))
        time.sleep(0.04)

# Explications des règles
def instructions():
    os.system('clear')
    print(regles)
    print(keyboard)
    input("\n\n  Appuyer sur ENTRER pour revenir au menu...")


# Affichage de la grille
# Le principe est le suivant; Dans une boucle for, on print chaque élément de la liste 1 par 1 sans retour à la ligne (,end="").
# Tous les 10 éléments print, c'est à dire les index 9,19,29... définis par ( i % 10 == 9 ) on insert un retour à la ligne.
# Les tabulations (\t) servent à assurer une distance équivalente entre chaque éléments de la GRILLE
# Le reste n'est que des considérations graphiques comme à i == 9 par exemple où l'on print le titre à la fin avant d'effectuer un retour à la ligne
# Ceci permet d'afficher une section à droite de la grille contenant des éléments d'interface utilisateur.
def showGameBoard(grille, message):
    os.system('clear')
    barrehaute = "\n" + (JAUNE + " " + BLANC) * 209
    barrebasse = (JAUNE + " " + BLANC) * 209

    UI = {
        'Titre': '  VIRUS KILLER',
        'BOMBE1': VERT + '  Bombe 1' + BLEU + '   PUISSANCE:  ' + str(bombeloader["bombe1"][1]) + BLANC,
        'BOMBE2': VERT + '  Bombe 2' + BLEU + '   PUISSANCE:  ' + str(bombeloader["bombe2"][1]) + BLANC,
        'BOMBE3': VERT + '  Bombe 3' + BLEU + '   PUISSANCE:  ' + str(bombeloader["bombe3"][1]) + BLANC,
        'BOMBE4': VERT + '  Bombe 4' + BLEU + '   PUISSANCE:  ' + str(bombeloader["bombe4"][1]) + BLANC,
        'ATP': VERT + '',
```

```
96          'VIRUSRESTANT': ROUGE + " ( _ ) X " + str(grille.count(casevirus)) + BLANC,
97      }
98
99      print (barrehaute)
100     print (JAUNE + " \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t " + "\t\t\t\t\t\t " + BLANC)
101     print (JAUNE + " \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t " + "\t\t\t\t\t\t " + BLANC)
102
103     for i in range(100):
104         if i % 10 == 0:
105             print (JAUNE + " " + BLANC, end="")
106         print (grille[i], end='')
107
108         if i == 9:
109             print (JAUNE + " " + BLANC, UI["Titre"], JAUNE + "\t\t\t\t " + BLANC)
110             sautdeligne()
111         if i == 19:
112             print (JAUNE + " " + "\t\t\t\t\t\t " + BLANC)
113             print (JAUNE + " \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t " + BLANC, UI["BOMBE1"], JAUNE + "\t\t\t " + BLANC)
114             print (JAUNE + " \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t " + BLANC, UI["BOMBE2"], JAUNE + "\t\t\t " + BLANC)
115             print (JAUNE + " \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t " + BLANC, UI["BOMBE3"], JAUNE + "\t\t\t " + BLANC)
116             print (JAUNE + " \t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t " + BLANC, UI["BOMBE4"], JAUNE + "\t\t\t " + BLANC)
117         if i == 39:
118             print (JAUNE + " " + VERT, "  Virus restant: ", UI["VIRUSRESTANT"], JAUNE + "\t\t\t " + BLANC)
119             sautdeligne()
120         if i == 49:
121             print (JAUNE + " " + BLANC, "   " + message)
122             sautdeligne()
123         if i % 10 == 9 and i not in [9, 19, 39, 49]:
124             print (JAUNE + " " + "\t\t\t\t\t\t " + BLANC)
125             sautdeligne()
126
127     print (barrebasse)
128
129
130 def initvirus(virus):
131     # Génére une liste de 4 index entre 0 et 99 dans la grille
132     virus = random.sample(range(0, 100), 4)
133     # Refais un aléa si les index généré sont sur des cases non vides
134     while grille[virus[0]] != casevide or grille[virus[1]] != casevide or grille[virus[2]] != casevide or grille[virus[3]] != casevide:
135         virus = random.sample(range(0, 100), 4)
136     # on remplie la grille avec les 4 virus
137     for i in range (len(virus)):
138         grille[virus[i]] = casevirus
139     return virus
140
141 # Renvoie une valeur aléatoire dans la grille à un emplacement casevide;
142 def randomingrille(n1, n2):
143     value = random.randint(n1, n2)
144     while grille[value] != casevide:
145         value = random.randint(n1, n2)
```

```
    return value

# Renvoie true si la valeur testée sort de la grille ou reviens par l'autre coté
def passemuraille(oldpos, newpos):
    if (newpos < 0 or newpos > 99 or (oldpos % 10 == 9 and newpos % 10 == 0) or (oldpos % 10 == 0 and newpos % 10 == 9)):
        return True
    else:
        return False

# Initialisation des murs, choix de la difficulté (nombre et longueur des murs), sens aléatoire.
def initmurs():
    os.system("clear")
    # Choix de la difficulté génère des valeurs différente pour le nombre de murs et leurs longueurs.
    print("\n\t A quel niveau de difficulté voulez vous jouer ?\n\n\t " + BLEU + "Rhume: (1)" + ORANGE + "   Gastro: (2)" + ROUGE + "   Chikungunya: (3) " + BLANC)
    inputdifficulty = input("\n\n\t Niveau: ")
    if inputdifficulty == "1":
        nbmur = random.randint(3, 4)
        longmur = random.randint(2, 3)
    elif inputdifficulty == "2":
        nbmur = random.randint(4, 5)
        longmur = random.randint(4, 5)
    elif inputdifficulty == "3":
        nbmur = random.randint(5, 6)
        longmur = random.randint(6, 7)
    else:
        initmurs()

    # Boucle while pour générer tous les murs (while mur <= nbmur)
    mur = 1
    while mur <= nbmur:
        # On génère en suite soit un mur vertical (sensmur=1) soit un mur horizontal (sensmur=2)
        sensmur = random.randint(1, 2)
        # On tire une position au hasard sur la grille pourvu que la place soit disponible (avec la fonction randomingrille)
        startingpoint = randomingrille(0, 99)
        grille[startingpoint] = casemur
        # Pour les verticaux on remplie vers le haut. Si l'on sort de la grille (fonction passemuraille), on commence à remplir vers le bas.
        # On a donc besoin d'avoir des compteurs indépendant pour haut et bas.
        if sensmur == 1:
            haut = 0
            bas = 0
            for brique in range(1, longmur):
                oldpos = startingpoint - 10 * haut
                newpos = startingpoint - 10 * (haut + 1)
                if passemuraille(oldpos, newpos) is False:
                    grille[newpos] = casemur
                    haut = haut + 1

                else:
                    oldpos = startingpoint + 10 * bas
                    newpos = startingpoint + 10 * (bas + 1)
```

```
                    if passemuraille(oldpos, newpos) is False:
                        grille[newpos] = casemur
                        bas = bas + 1

        # Pour les horizontaux on remplie vers la gauche... si l'on sort de la grille, on commence à remplir vers la droite.
        # On a donc besoin d'avoir des compteurs indépendant pour gauche et droite.
        else:
            gauche = 0
            droite = 0
            for brique in range(1, longmur):
                oldpos = startingpoint - 1 * gauche
                newpos = startingpoint - 1 * (gauche + 1)
                if passemuraille(oldpos, newpos) is False:
                    grille[newpos] = casemur
                    gauche = gauche + 1

                else:
                    oldpos = startingpoint + 1 * droite
                    newpos = startingpoint + 1 * (droite + 1)
                    if passemuraille(oldpos, newpos) is False:
                        grille[newpos] = casemur
                        droite = droite + 1
        mur += 1

# Initialisation du joueur
def initjoueur(mouvement):
    joueur = randomingrille(0, 99)
    mouvement[1] = joueur
    grille[joueur] = casejoueur
    return mouvement

# Permet de compter combien d'ATP est présent dans la grille
def countATP():
    nbATP = 0
    for i in range(len(grille)):
        if grille[i] == caseATP:
            nbATP = nbATP + 1
    return nbATP

# Permet de générer de l'ATP sur un emplacement vide de la grille
def randomATP():
    randomATPpos = randomingrille(0, 99)
    return randomATPpos

# Génère de l'ATP tant qu'il y a moins de 8 molecules dans la grille.
def spawnATP():
    nbATP = countATP()
    while nbATP < 8:
        newATP = randomATP()
        grille[newATP] = caseATP
```

```
246             nbATP = countATP()
247         showGameBoard(grille, message)
248         return grille
249 
250     # Gestion des touches utilisés par le joueur pendant son tour de jeu.
251     def keyinput(mouvement):
252         message="Début de votre tour" + JAUNE + "\t\t\t\t " + BLANC
253         showGameBoard(grille, message)
254         # newpos et oldpos sont utilisés pour le déplacement, oldpos est conservé si la direction testée (newpos) ne réponds pas aux critères de déplacement
255         newpos = mouvement[0]
256         oldpos = mouvement[1]
257         voh = mouvement[2]           # vertical ou horizontal ou none
258         continuer = 1  # continuer le déplacement tant que continuer = 1
259 
260         inputkey = input("\n\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t      Action :  ")
261         while inputkey not in ["z","q","s","d","1","2","3","4"," "]:
262             inputkey = input("\n\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t      Selectionnez la bonne touche :  ")
263 
264         # aller vers le haut (z) si on s'est déplacé verticalement (v) ou pas encore déplacé (n)
265         if inputkey == "z" and (voh == "v" or voh == "n"):
266             newpos = oldpos - 10    # Pour aller vers le haut on enlève 10 à l'index.
267             voh = "v"
268 
269         # aller vers la gauche (q) si on s'est déplacé horizontalement (h) ou pas encore déplacé (n)
270         elif inputkey == "q" and (voh == "h" or voh == "n"):
271             newpos = oldpos - 1
272             voh = "h"
273 
274         # aller vers le bas (s) si on s'est déplacé verticalement (v) ou pas encore déplacé (n)
275         elif inputkey == "s" and (voh == "v" or voh == "n"):
276             newpos = oldpos + 10
277             voh = "v"
278 
279         # aller vers la droite (d) si on s'est déplacé horizontalement (h) ou pas encore déplacé (n)
280         elif inputkey == "d" and (voh == "h" or voh == "n"):
281             newpos = oldpos + 1
282             voh = "h"
283 
284         # poser une bombe si on ne s'est pas déplacé
285         elif (inputkey in ["1", "2", "3", "4"] and voh == "n"):
286             newpos = oldpos
287             voh = "n"
288 
289         # empêche la pose de bombe si on s'est déplacé
290         elif (inputkey in ["1", "2", "3", "4"] and voh != "n"):
291             newpos = oldpos
292 
293         # demande de fin de tour (touche espace)
294         elif (inputkey == " "):
295             newpos = oldpos
```

```
296             continuer = 0
297
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]                       # oldpos= position actuelle
305         newpos = testmouvement[1]                       # newpos = position a tester
306         voh = testmouvement[2]                          # voh = type de deplacement
307         continuer = testmouvement[3]                    # continuer = 0 ou 1
308
309         if continuer == 0:                              # Cas où il y a eu touche espace: fin de tour
310             mouvement = [oldpos, newpos, voh, continuer]
311             return mouvement, bombeloader
312
313         elif newpos != oldpos:                          # Si la nouvelle position testée est différente de l'ancienne il s'agit d'un déplacement.
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\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\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\t " + BLANC
333                 boostbombe(bombeloader)                 # On boost deux bombes au hasard
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 inaccessible 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":
```

```
            bombeloader["bombe1"][0] = newpos
            message = "Vous venez de déposer la bombe 1" + tabulationdumessage
        elif inputkey == "2":
            bombeloader["bombe2"][0] = newpos
            message = "Vous venez de déposer la bombe 2" + tabulationdumessage
        elif inputkey == "3":
            bombeloader["bombe3"][0] = newpos
            message = "Vous venez de déposer la bombe 3" + tabulationdumessage
        elif inputkey == "4":
            bombeloader["bombe4"][0] = newpos
            message = "Vous venez de déposer la bombe 4" + tabulationdumessage
    showGameBoard(grille, message)
    return mouvement, bombeloader

# Renvoie pour un virus les directions possibles
def dirpossiblevirus(virus, numvirus):
    posvirus = virus[numvirus]
    dirpossible = []
    if posvirus > 9:
        if grille[posvirus - 10] == casevide:
            dirpossible.append(0)
    if posvirus < 90:
        if grille[posvirus + 10] == casevide:
            dirpossible.append(1)
    if posvirus > 0:
        if grille[posvirus - 1] == casevide and posvirus % 10 != 0:  # le virus ne peut aller vers la gauche si il est contre la paroi
            dirpossible.append(2)
    if posvirus < 99:
        if grille[posvirus + 1] == casevide and posvirus % 10 != 9:  # le virus ne peut aller vers la droite si il est contre la paroi
            dirpossible.append(3)
    if dirpossible == []:  # Dans le cas où les conditions ne sont pas remplis le virus essaie d'aller vers la gauche (sinon une erreur est renvoyé)
        dirpossible.append(1)
    return dirpossible

# Appel la fonction de déplacement aléatoire des virus en fonction de la direction possible
def randommovevirus(virus):
    for numvirus in range(len(virus)):
        if virus[numvirus] != "mort":
            oldvirpos = virus[numvirus]
            dirpossible = dirpossiblevirus(virus, numvirus)  # choix des directions où le virus ne sera pas bloqué
            direction = random.choice(dirpossible)
            if direction == 0:  # haut
                dirvalue = -10
                maxdistance = int(oldvirpos / 10)
            elif direction == 1:  # bas
                dirvalue = 10
                maxdistance = 10 - int(oldvirpos / 10)
            elif direction == 2:  # gauche
                dirvalue = -1
                maxdistance = oldvirpos % 10
```

```
            elif direction == 3:  # droite
                dirvalue = +1
                maxdistance = 10 - (oldvirpos % 10)
            distance = random.randint(1, maxdistance)  # distance a parcourir
            pas = 0
            while pas < distance:
                movevirus(virus, numvirus, dirvalue)
                pas = pas + 1

# Fonction de déplacement des virus
def movevirus(virus, numvirus, dirvalue):
    oldvirpos = virus[numvirus]
    newposvir = oldvirpos + dirvalue
    message = "Les virus se déplacent" + JAUNE + "\t\t\t " + BLANC
    if (newposvir > 0 and newposvir < 100):
        if passemuraille(oldvirpos, newposvir) is True:
            message = "Le virus tente en vain de passer la paroi" + JAUNE + "\t " + BLANC
            time.sleep(0.3)
        else:
            if grille[newposvir] == casevide:
                grille[oldvirpos] = casevide
                virus[numvirus] = newposvir
                grille[newposvir] = casevirus
                time.sleep(0.3)
            else:
                time.sleep(0.1)
    showGameBoard(grille, message)

# Selectionne deux bombes aléatoirement parmis le chargeur et incrémente de 1 sa puissance
def boostbombe(bombeloader):
    selectrandombombe = random.sample(list(bombeloader.keys()), 2)
    bombeloader[selectrandombombe[0]][1] = bombeloader[selectrandombombe[0]][1] + 1
    bombeloader[selectrandombombe[1]][1] = bombeloader[selectrandombombe[1]][1] + 1
    return bombeloader

# Fonction explosion des bombes
def boom(bombeloader):
    activebombe = []
    for bombe in bombeloader.keys():  # détecte si une bombe a une valeur de position différente de "n"
        if bombeloader[bombe][0] != "n":
            activebombe.append(bombeloader[bombe][0])
            activebombe.append(bombeloader[bombe][1])
            bombeloader[bombe][1] = "X"

    if activebombe != []:
        posbombes = activebombe[0]
        rayon = int(activebombe[1] / 2) + (activebombe[1] % 2 > 0)
        print (rayon)
        for motif in [ROUGE + "\t    \t" + BLANC, casevide]:
            grille[posbombes] = motif
```

```
            i = 1
            while i <= rayon:
                if posbombes - 10 * i >= 0:
                    grille[posbombes - 10 * i] = motif
                if posbombes + 10 * i < 99:
                    grille[posbombes + 10 * i] = motif
                if rayon <= 2:  # Explosion en croix pour les bombes de rayons <= 2
                    if posbombes - 1 * i >= 0 and (posbombes - 1 * i) % 10 != 9:
                        grille[posbombes - 1 * i] = motif
                    if posbombes + 1 * i < 99 and (posbombes + 1 * i) % 10 != 0:
                        grille[posbombes + 1 * i] = motif
                i = i + 1
            message = "BOOOOM" + JAUNE + "\t\t\t\t\t " + BLANC
            showGameBoard(grille, message)
            time.sleep(1)
        for item in bombeloader.keys():
            bombeloader[item][0] = "n"
        return bombeloader

 # Fonction de recharge de bombe après explosion
def reloadbombe(bombeloader):
    for slot in bombeloader.keys():
        if bombeloader[slot][1] == "X":
            val = random.sample([3, 5, 7, 9], 1)
            bombeloader[slot][1] = val[0]

# Fonction qui décrémante la puissance des bombes de 1
def bombemolle(bombeloader):
    for slot in bombeloader.keys():
        if bombeloader[slot][1] != 0:  # Les bombes ne peuvent pas avoir une puissance négative
            bombeloader[slot][1] = bombeloader[slot][1] - 1
    showGameBoard(grille, message)
    return bombeloader

# Regarde quels sont les virus qui sont mort (après explosion)
def constatdesmorts(virus):
    for individus in range(0, 4):
        if virus[individus] != "mort":
            if grille[virus[individus]] == casevide:
                virus[individus] = "mort"
    return virus

# Gagné si 4 virus sont mort (nbmort == 4)
def win(virus):
    victory = 0
    nbmort = virus.count("mort")
    if nbmort == 4:
        victory = 1
        os.system('clear')
        print (iconvictory)
```

```
496             time.sleep(3)
497         return victory
498 
499     # Perdu si les 4 bombes sont à zéro de puissance
500     def loose(bombeloader):
501         bombeazero = 0
502         for item in bombeloader.keys():
503             if bombeloader[item][1] <= 0:
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, bombeloader, ATP):
517         initmurs()                                          # Initialise les murs
518         initjoueur(mouvement)                               # Initialise le joeur
519         virus = initvirus(virus)                            # Initialise les virus
520         spawnATP()                                          # Initialise l'ATP
521         showGameBoard(grille, message)                      # Affiche la grille
522         victory = win(virus)                                # Condition de victoire
523         rip = loose(bombeloader)                            # Condition de défaire
524 
525         while victory == 0 and rip == 0:                    # Jouer tant qu'il n'y a ni victoire ni défaite
526             randommovevirus(virus)                          # Mouvement des virus
527             while mouvement[3] == 1:                        # Fonction mouvement tant que mouvement[continuer] == 1
528                 mouvement, bombeloader = actionjoueur(mouvement, bombeloader)
529             mouvement[2] = "n"                              # Réinitialise le type de mouvement du joueur à "n"
530             mouvement[3] = 1                                # Ré-autorise le joueur à bouger au prochain tours
531             boom(bombeloader)                               # Explosion éventuelle des bombes
532             virus = constatdesmorts(virus)                  # Recalcul du nombre de virus encore en vie
533             spawnATP()                                      # Régénère le nombre d'ATP sur la grille
534             reloadbombe(bombeloader)                        # Recrée une bombe de puissance aléatoire si une bombe a été posée
535             bombemolle(bombeloader)                         # Décrémente la puissance des bombes
536             victory = win(virus)                            # Re-vérifie si les conditions de victoire sont réunies (victory = 1 ?)
537             rip = loose(bombeloader)                        # Re-vérifie si les conditions de défaite sont réunies (victory = 1 ?)
538 
539         input('     Appuyer sur entrer pour quitter... ')    # Partie terminée
540         sys.exit()
541 
542 
543     def menu():
544         os.system("clear")
545         print (JAUNE + iconbombe + BLANC)
```

```
    lore()
    time.sleep(1)
    input(ORANGE + "\n\n       Appuyer sur ENTRER pour continuer..." + BLANC)
    os.system("clear")
    print (JAUNE + iconbombe + BLANC)
    print (VIOLET + iconmenu + BLANC)
    selection = input("\n\n\t\t\t\t\t\t\t\t Choix: ")
    if selection == "1":
        startgame(virus, mouvement, grille, message, bombeloader, ATP)
    if selection == "2":
        instructions()
    if selection == "0":
        sys.exit()
    else:
        menu()


# -- MAIN ---
credit()
menu()
```