RAPPORT SUR LE PROJET FINAL DE CONCEPTION DE LOGICIEL

Olivier Burnel
Quentin Dumont
Alexis Pestel

L1 Informatique Groupe 2B
Chargé de TP : Sébastien Gamblin

UNIVERSITÉ
CAEN
NORMANDIE

https://www.unicaen.fr

INTRODUCTION

Différents sujets étaient proposés pour le projet final de conception de logiciel, et notre choix s'est porté sur la réalisation d'un jeu de type Puzzle Quest. Le développement du jeu nous paraissait assez simple à première vue, mais nous n'imaginions pas tout ce à quoi il fallait penser pour pouvoir rendre un jeu fonctionnel. Pour introduire ce rapport nous allons rappeler les grandes lignes directrices de notre sujet : un Puzzle Quest demande d'associer des items identiques, pour former des combinaisons qui auront des effets différents suivant les items combinés. L'objectif dans une partie de Puzzle Quest est de vaincre un adversaire contre lequel nous jouons. Ainsi, la formation de combinaisons d'items nous permet d'attaquer, de nous défendre et d'effectuer d'autres actions que nous détaillerons plus tard. Pour fabriquer un vrai Puzzle Quest, il faut aussi créer un périple, avec comme étapes des niveaux dont la difficulté et l'environnement varient. Voilà pour les grandes lignes de notre sujet. Nous avons par la suite morcelé le jeu en différentes fonctionnalités à programmer. Ce rapport va détailler la création et la mise en place de toutes les parties du jeu, de ses débuts en console jusqu'à son rendu fonctionnel en interface graphique.

Les initiales de l'étudiant sont marquées en rouge à côté de chaque titre de partie qu'il a travaillé.

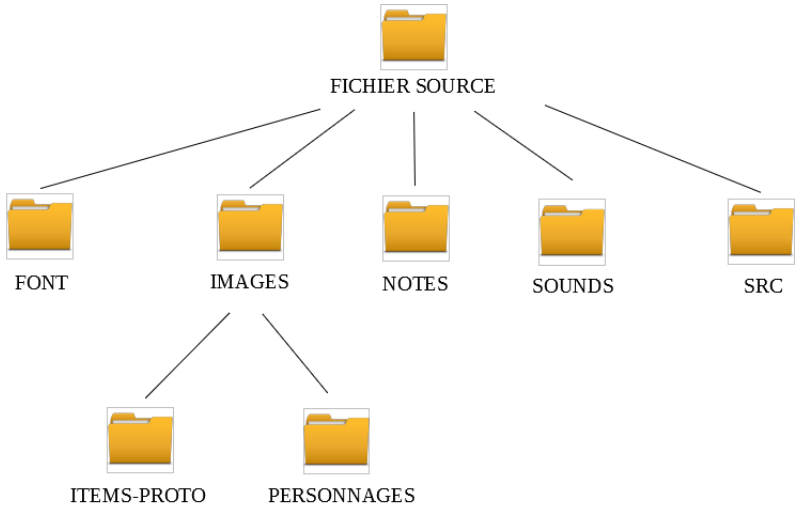
Sommaire

1 ARCHITECTURE DE L'APPLICATION

Avant toute chose, pour correctement travailler sur notre projet, il nous faut instaurer un environnement de travail sain et organisé. Nous verrons dans cette partie comment nous nous sommes globalement organisés dans notre travail.

1.1 Arborescence des fichiers

Une base d'un projet est l'organisation des fichiers. Dans cette partie, nous allons vous expliquer comment nous avons organisé ceux-là.



Organisation de nos dossiers.

Comme vous pouvez le voir ci-dessus, notre fichier source est organisé en **cinq sous-dossiers**, tous aussi importants les uns que les autres. Nous pouvons y trouver :

- Le dossier "*FONT*" contenant la police d'écriture principale de notre jeu.
- Le dossier "*IMAGES*" contenant les images utiles à l'affichage des menus et de l'arrière plan, ainsi que deux autres sous-dossiers :
  - LE dossier "'*ITEMS-PROTO*" contenant les images que nous avons utilisé pour nos items de grille ;
  - Le dossier "*PERSONNAGES*" contenant les images des personnages utilisés dans le jeu, comme le chien dans l'avion, le requin ou le doritos.
- Le dossier "*NOTES*" contenant nos toutes premières idées, celles à la base du jeu ;
- Le dossier "*SOUNDS*" contenant tous les sons et bruitages utilisés dans le jeu, comme la musique de fond ;
- Et enfin le dossier "*SRC*" contenant l'intégralité du code de notre jeu.

1.2 Organisation de notre code

Organiser nos fichiers est certes très important, mais il nous a aussi fallu organiser notre code. De ce fait, une personne qui n'y connaît pas grand chose ou simplement quelqu'un qui ne connaît pas notre méthode de travail y comprend tout de même quelque chose. Dans notre dossier "*SRC*", nous pouvons y trouver six fichiers :

- Le fichier *entity.py* contenant la classe *Entity()* et toutes les méthodes relatives à ces dernières ;
- Le fichier *grille.py* contenant la classe *Grid()* nous permettant de correctement organiser, faire fonctionner et afficher les grilles des joueurs et de l'ordinateur ;
- Le fichier *itemPygame.py* gérant les items présents sur la grille en interface graphique ;
- Le fichier *items.py* contenant la classe *Items()* gérant en globalité les items des grilles ;
- Le fichier *level.py* contenant la classe *Level()* nous permettant de gérer la progression du joueur et la gestion des niveaux ;
- Et enfin le fichier *main.py*, fichier principal du jeu contenant la boucle de jeu ainsi que toutes les classes d'affichages et de pages.

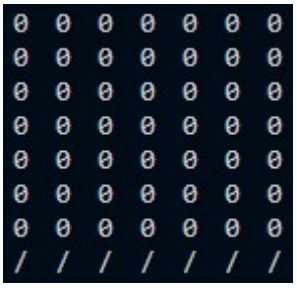
Comme vous avez pu le lire, chaque fichier contient une classe importante et de ce fait cela rend nos fichiers peu dépendants entre eux et donc assez modulaires.

2 DÉVELOPPEMENT DE LA GRILLE (Q.D)

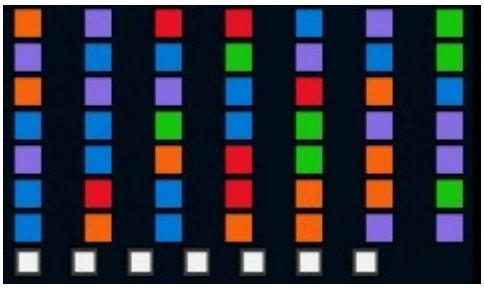
Une partie de Puzzle Quest se joue sur une grille, sur laquelle on peut échanger des items voisins pour former des combinaisons de trois items ou plus. Lorsqu'une combinaison est formée, les items qui la forment sont détruits, ce qui entraîne la chute des items situés au-dessus. Par conséquent, de nouveaux items sont générés en haut de la grille, jusqu'à ce qu'elle soit de nouveau remplie. Nous allons détailler ici tout le bagage de fonctions nécessaires au fonctionnement de cette grille. Je précise que nous nous placerons dans un contexte de développement en console, et qu'il ne sera pas encore question d'implémentation en interface graphique.

2.1 Mise en place

Pour construire notre grille, j'ai créé une classe "Grid" de sorte à pouvoir instancier des objets "Grid" dans la construction du jeu. Notre grille est carrée et fixée en 7x7. On crée sept lignes de zéros qu'on rajoute dans une liste pour remplir la grille de zéros. J'ai eu besoin de rajouter un fond à la grille, en rajoutant une ligne de sept "/" à la fin. Vous découvrirez pourquoi dans la section 2.4 ci-après. En console, nous obtenons alors ceci :



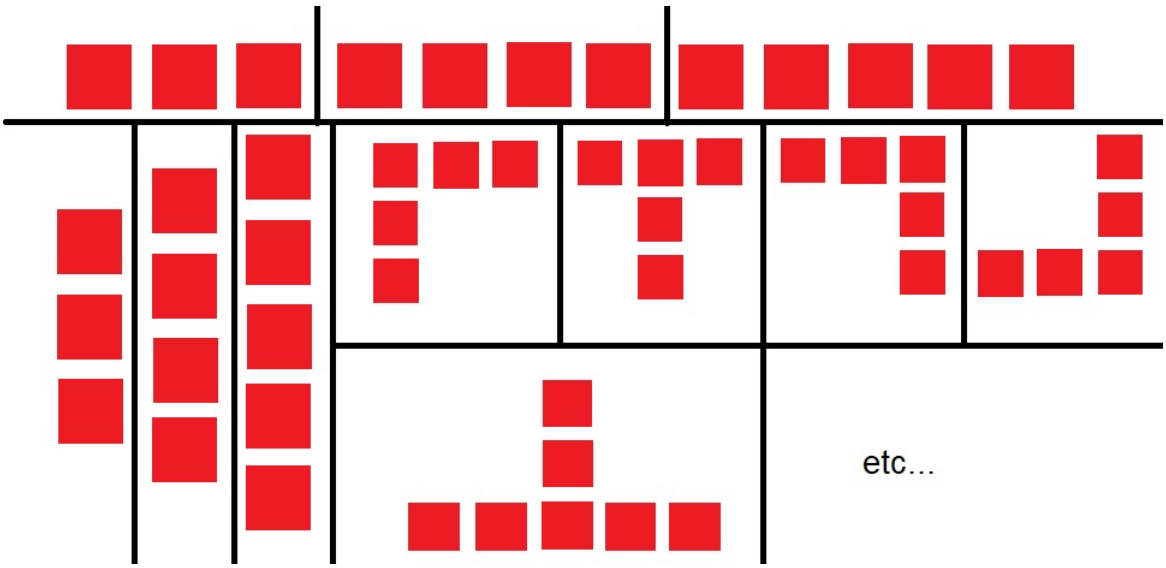
Cette grille est destinée à être remplie par des items de différentes couleurs. On crée donc une classe "Item" pour pouvoir instancier des objets "Item" dans la grille. On leur donne un attribut "color" et des attributs "coordX" et "coordY" pour pouvoir stocker en mémoire leur position, et les cibler avec les différentes fonctions. Plutôt que de les afficher avec leur nom de couleur en terminal, j'ai décidé d'utiliser des caractères spéciaux unicode pour mieux comprendre la grille. Les items apparaissent alignés et les couleurs sautent aux yeux, ce qui permet de mieux voir les combinaisons possibles. Grâce à ces caractères il serait facile de jouer au jeu en console.



Les carrés blancs représentent le fond de grille.

2.2 Détection des combinaisons

Une grille de Puzzle Quest doit être capable de détecter quand une combinaison est formée en elle. Définissons d'abord ce qu'est une combinaison : nous avons choisi de détecter les alignements de 3, 4 et 5 items. Ces alignements peuvent être en ligne et en colonne, et il faut également prendre en compte le fait qu'ils peuvent se produire simultanément. On doit alors être capable de détecter toutes ces figures :



La liste est non-exhaustive mais on peut facilement déduire les figures restantes.

Pour détecter toutes ces possibilités de combinaisons, j'ai d'abord programmé deux fonctions de base : "combo_ligne" et "combo_colonne". Ces deux fonctions détectent des alignements d'items allant de 3 à 5 d'affilée, en ligne et en colonne comme leur nom l'indique. Ces fonctions parcourent la grille et se posent cette succession de questions sur chaque item :

- Existe-t-il un alignement de 5 items ?
- De 4 items ?
- De 3 items ?

Ces fonctions retournent les coordonnées des items formant la première combinaison trouvée dans la grille. Sur combo_ligne, on incrémente l'index des colonnes (ci-dessous j) pour détecter un alignement horizontal. Sur combo_colonne, on incrémente l'index des lignes (ci-dessous i) pour détecter un alignement vertical. Si elles ne trouvent rien, elles renvoient simplement le booléen "False".

```
elif (j+2 < self.largeur) and (self.grid[i][j]) == (self.grid[i][j+1]) and (self.grid[i][j]) == (self.grid[i][j+2]) :
    return [[i, j], [i,j+1], [i,j+2]]
```

Dernière condition de la fonction combo_ligne : recherche d'alignement horizontal de 3 items

À noter que nous comparons les items de la grille à partir de leur attribut "color". Or, les items que nous instancions sont des objets, qui sont obligatoirement différents de part leur adresse en mémoire. On ne peut pas non plus considérer que la grille n'est remplie que d'items puisqu'ils peuvent être détruits/remplacés par un zéro, qui n'aura pas par conséquent d'attribut "color". Pour contrer cela, nous avons redéfini l'équivalence entre objets de type "Item".

```
def __eq__(self, other):
    if isinstance(other, Item) :
        return self.color == other.color
```

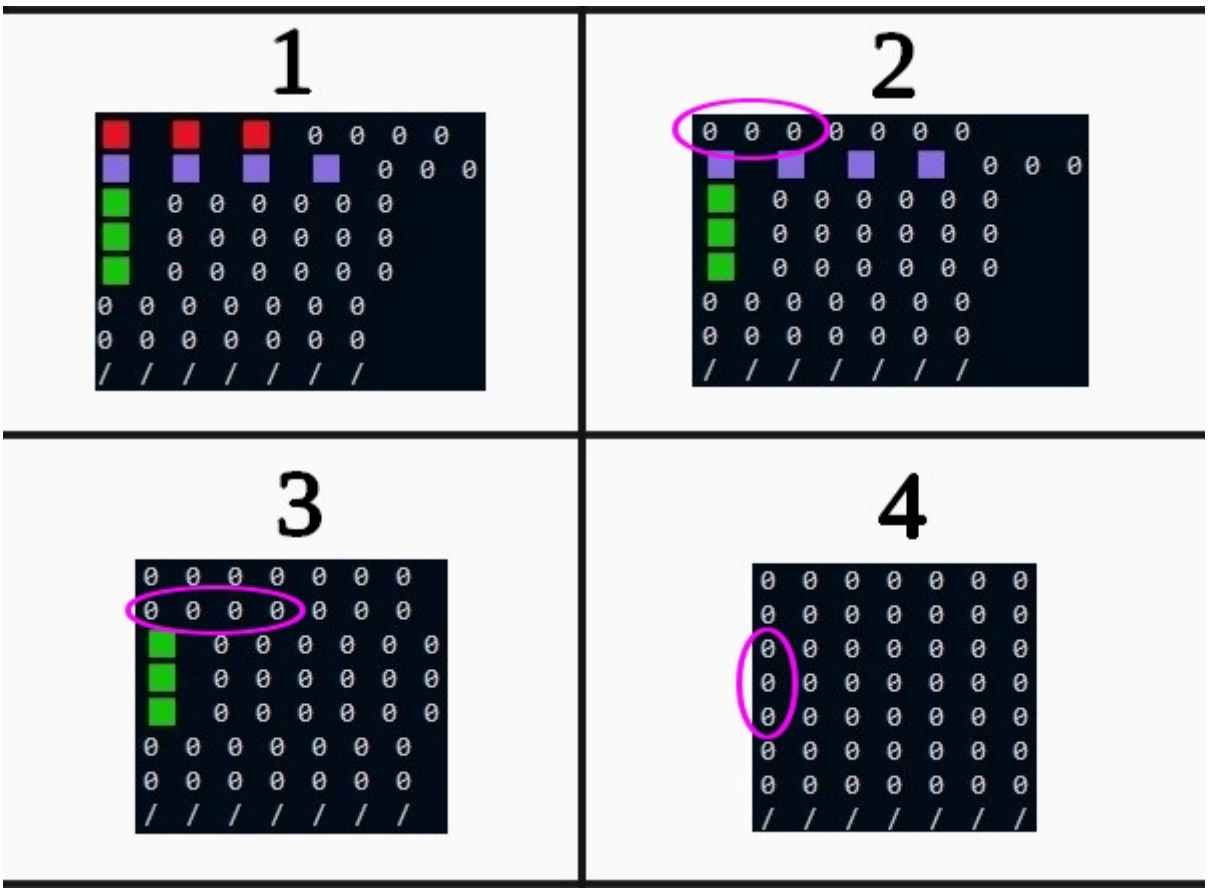
Redéfinition de l'équivalence entre objets "Item"

Grâce à ce que nous venons de voir, nous sommes capable de détecter des alignements en colonne et en ligne, suivant la couleur des items. J'ai fait le choix de programmer un troisième détecteur qui s'occupe des lignes croisées de colonnes. cette fonction s'appelle "is_L". Ce choix s'explique par le fait que la grille est pensée pour détecter et détruire les combinaisons au fur et à mesure. En effet on ne compile pas toutes les coordonnées de toutes les combinaisons pour détruire les items d'un seul coup : différencier leur couleur donc les effets produits par les combinaisons serait plus compliqué. On peut détecter des lignes, des colonnes, maintenant il nous faut détecter les alignements en forme de L et T. "is_L" teste si on a un alignement en colonne et un autre en ligne. Elle regarde ensuite si les deux combinaisons ont des coordonnées d'item communes ; si c'est le cas, alors elle renvoie les coordonnées des items formant le L (ou le T), sinon elle renvoie "False".

2.3 Destruction des combinaisons

Une combinaison détectée doit être détruite, autrement dit, les items formant la combinaison doivent être supprimés. Dans notre cas, ils seront remplacés par un zéro. J'ai écrit une fonction "destroy" qui est capable d'analyser la grille et de détruire toutes les combinaisons qu'elle trouve. J'utilise pour cela les fonctions de détections dont j'ai décrit le fonctionnement juste avant. La fonction "destroy" appelle successivement les trois détecteurs. Tant qu'un des détecteurs trouve une combinaison, on cible les items de la combinaison grâce à leurs coordonnées, puis on leur donne la valeur zéro. Clarifions tout cela dans un exemple illustré.

Mettons nous dans la situation suivante :



1. Nous avons une ligne de trois items rouges, une autre de quatre items violets et enfin une colonne de trois items verts.

2. "destroy" va d'abord appeler "is_L" : tant que des L ou des T sont dans la grille, elle remplace les items qui composent ces combinaisons par zéro. Ici, il n'y a pas de L ou de T, donc "is_L" retourne "False". "destroy" passe alors aux lignes avec "combo_ligne" : la première combinaison en ligne est la rouge, ses items sont donc remplacés par des zéros.

3. "destroy" effectue un nouveau test avec "combo_ligne", qui détecte cette fois-ci la ligne violette et remplace ses items par des zéros.

4. Sur son itération suivante "combo_ligne" retourne "False" puisqu'il n'y a plus de lignes. "destroy" appelle "combo_colonne" qui détecte la colonne verte, puis elle la remplace elle aussi par des zéros.

"destroy" ne fait pas que détruire les items : elle compte également le nombre d'items détruits par couleur. Un dictionnaire est initialisé dans la fonction, celui-ci a pour clés les couleurs que peuvent avoir les items. Quand une combinaison est détectée, on va chercher la couleur d'un des items de la combinaison. Cela nous permet de savoir dans quelle clé du dictionnaire nous devons stocker la valeur représentant le nombre d'items de la combinaison. Après avoir activé tous les détecteurs, "destroy" retourne le dictionnaire. Cette fonctionnalité va nous être très utile pour rendre fonctionnel le système des entités (voir 3.3).

```
def put_zero(self, liste) :
    value = 0
    key = self.grid[liste[0][0]][liste[0][1]].color
    for i in range(len(liste)) :
        self.grid[liste[i][0]][liste[i][1]] = 0
        value += 1
    return key, value
```

La fonction ci-dessus retourne la couleur de la combinaison et le nombre d'items compris dedans. On récupère ces valeurs, puis on remplit le dictionnaire comme ceci :

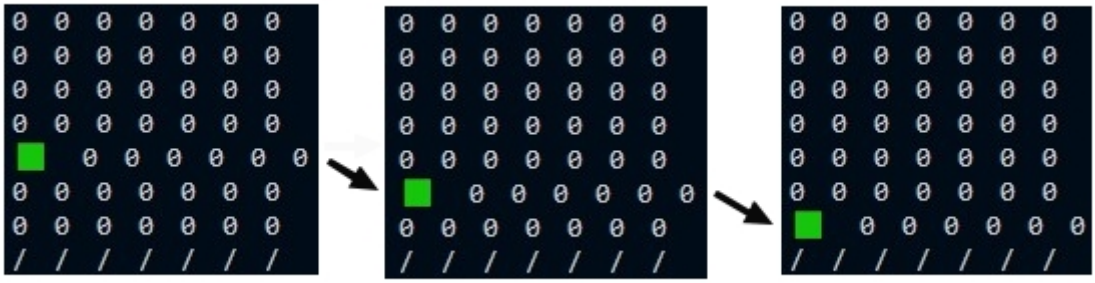
```
dict = {"🟥" : 0,"🟧" : 0,"🟩" : 0,"🟦" : 0,"🟪" : 0}

while self.is_L() != False :
    liste_a_detruire = self.is_L()
    key, value = self.put_zero(liste_a_detruire)
    dict[key] = value
```

2.4 Gravité artificielle

Il nous manque un élément primordial pour que la grille soit fonctionnelle, qui n'est autre que le décalage des items vers le fond de la grille. Autrement dit, c'est une forme de gravité artificielle appliquée sur les items. Dans la section 2.1 nous avons rajouté une ligne de "/" après toutes les lignes de "0". J'ai fait ce choix pour pouvoir programmer une fonction simple qui va donner vie à une forme de gravité : "decalage_item". Cette fonction parcourt toutes les positions de la grille, et si c'est un item, elle teste la condition suivante :

Est-ce que la case en dessous de celle de l'item est un zéro ?

- Si oui, alors la case en dessous de l'item prend la valeur de celle de l'item, et la case de l'item prend la valeur 0. En d'autres termes, on échange les valeurs des deux cases.
- Si non, il n'y a pas d'échange et l'item reste à sa place.



Ce fonctionnement permet de *faire descendre* au maximum les items de la grille, à chaque appel de "decalage_item". La ligne de "/" permet d'arrêter la chute des items, puisqu'elle est remplie de caractères différents de 0. Nous avons alors construit un semblant de gravité dans la grille.

2.5 Remplissage procédural

Jusqu'à présent nous avons placé manuellement des items dans la grille pour illustrer le fonctionnement des méthodes de l'objet "Grid". Il nous faut maintenant générer aléatoirement des items de couleurs différentes. Pour cela j'ai écrit une fonction "feed" qui fait appel à une méthode de l'objet "Item" : "setRandomColor". "setRandomColor" renvoie une couleur aléatoire dans une liste de couleurs. On cible ensuite la première ligne de la grille, et pour chaque position dans la ligne, on place un item ayant ces attributs :

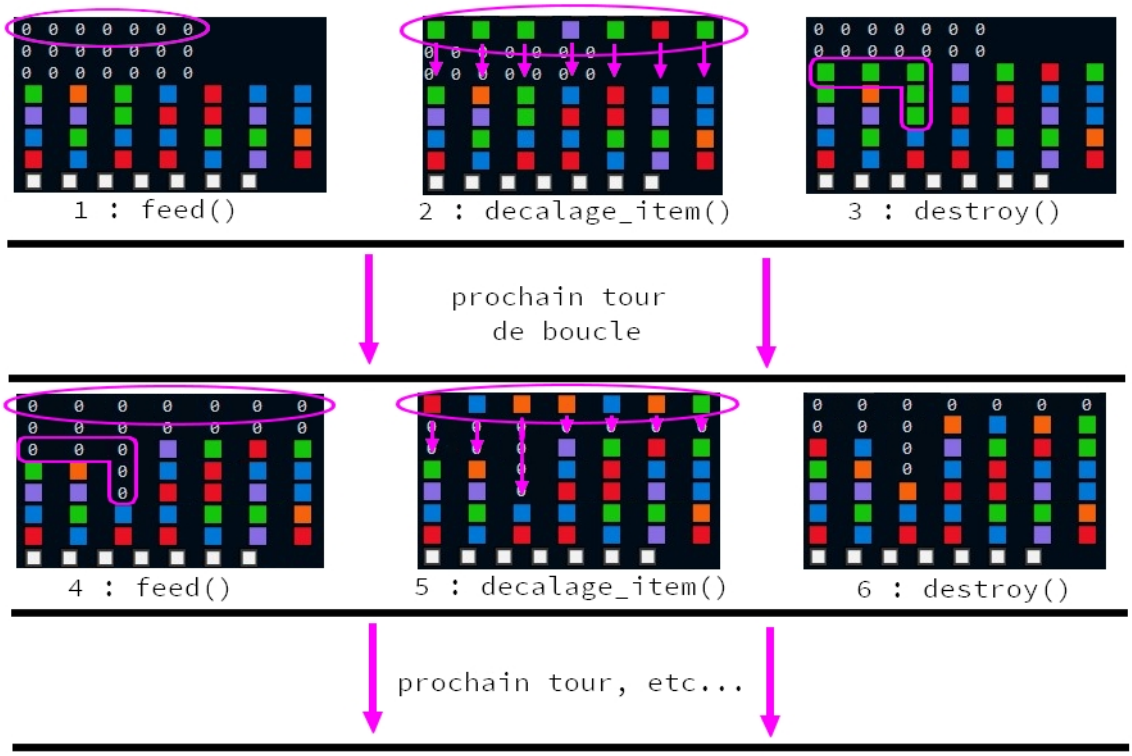
– "color" : couleur renvoyée par "setRandomColor"
– "coordX" : coordonnée correspondante à sa position dans la ligne
– "coordY" : coordonnée égale à 0 puisque nous sommes sur la première ligne

Nous disposons des trois fonctions qui vont nous permettre de remplir la grille, de façon à ce que le joueur puisse commencer une partie (donc une grille sans aucune combinaison). J'ai eu besoin de créer une petite fonction "is_pleine" qui teste si la grille est remplie d'items. Ce détecteur permet de mettre en place la boucle qui suit :

Tant que la grille n'est pas pleine, le tour de boucle suivant s'effectue :

1. Génération d'une ligne d'items (ou un item là où il n'y a qu'un zéro) en haut de la grille (appel de la fonction "feed")
2. Application de la gravité artificielle (appel de la fonction "decalage_item")
3. Détection et destruction des éventuelles combinaisons (appel de la fonction "destroy")

Ci-dessous vous trouverez un exemple de remplissage de la grille en console, qui met en évidence l'appel des différentes fonctions :



2.6 Échange d'items

Il ne manque plus que la notion d'interaction pour que notre grille soit opérationnelle en console. Au départ j'ai créé une fonction "switch" qui prenait en paramètres les coordonnées des deux items que l'on voulait échanger de place. Cette fonction vérifiait si une fois échangés, les items formaient une combinaison. Si oui, "switch" échangeait les deux items. Cependant nous avons par la suite changé ce fonctionnement car il était trop laborieux de changer tout le temps de place les objets "Item", particulièrement en interface graphique. Nous reparlerons du fonctionnement définitif de "switch" plus loin dans ce rapport (5.4). Nous en avons terminé avec le développement de la grille, et nous pouvons désormais passer au développement des entités.

```
def switch_en_console(self) :
    '''Fonction d'échange de deux items'''
    i = int(input("ligne du premier objet à déplacer : "))
    j = int(input("colonne du premier objet à déplacer : "))
    i2 = int(input("ligne du deuxième objet à déplacer : "))
    j2 = int(input("colonne du deuxième objet à déplacer : "))

    tmp = self.grid[i][j]
    self.grid[i][j] = self.grid[i2][j2]
    self.grid[i2][j2] = tmp

    if (not self.combo_ligne()) and (not self.combo_colonne()) :

        tmp = self.grid[i][j]
        self.grid[i][j] = self.grid[i2][j2]
        self.grid[i2][j2] = tmp
        return self.switch_en_console()
```

Pour échanger les items, on les cible avec les coordonnées renseignées par l'utilisateur, et on utilise une variable temporaire.

3 DÉVELOPPEMENT DES ENTITÉS

Avec la partie vue précédemment, nous sommes en capacité de faire interagir le joueur sur notre grille. Il peut échanger des items et former des combinaisons. Le joueur entraîne alors la destruction des items qui composent la combinaison et l'apparition de nouveaux items. Cependant nous ne devons pas simplement développer un jeu de "match-3" comme *Candy Crush* par exemple. Autrement dit la fin d'une partie ne sera pas définie par un objectif d'items à détruire. Un Puzzle Quest oppose deux entités : le personnage du joueur et le personnage de l'ordinateur. La fin d'une partie se caractérise alors par la mise en échec de l'adversaire. Cette contrainte implique la mise en place de deux entités, ainsi que le développement d'actions basiques de jeux de rôles. Nous allons détailler tout cela juste ici.

3.1 Mise en place (O.B)

Pour instancier des entités dans le jeu (nous en aurons besoin de deux, une pour le joueur, une pour l'ordinateur), j'ai créé une classe *Entity()* dans le fichier *entity.py*. Comme dans un jeu de rôle, nous devons mettre en place des caractéristiques de points de vie et d'armure sur chaque entité. Nous avons aussi décidé d'ajouter une notion d'attaque chargée. Les attributs suivants sont alors nécessaires à chaque entité :

- "player" : booléen qui renseigne la nature de l'entité (joueur ou IA)
- "life" et "lifeMax" : "life" contient la vie de l'entité et "lifeMax" est la limite maximale que "life" peut atteindre.
- "armor" et "armorMax" : même principe que pour "life" et "lifeMax" mais cette fois-ci pour le bouclier.
- "charged" et "chargedMax" : idem, pour la charge d'attaque puissante.

3.2 Effets des combinaisons (O.B)

Nous avons choisi plusieurs actions que les entités sont capables de faire, grâce aux combinaisons formées sur leur grille respective :

- Elle peuvent se protéger :

  – soit en formant des combinaisons **vertes**, qui donnent une quantité de **points de vie**,

  – soit en formant des combinaisons **bleues**, qui donnent une quantité de **bouclier**. Ce bouclier va réduire ou absorber totalement les attaques de base adverses.

- Elles peuvent attaquer :

  – Une combinaison **rouge** inflige des **dégâts de base** à l'entité adverse, qui sont diminués par le bouclier adverse s'il existe.

  – Une combinaison **violette** inflige un **empoisonnement** à l'entité adverse : cela lui retire directement des points de vie, et ce sur plusieurs tours de jeu.

- Enfin, nous avons intégré une action bonus :

  – La combinaison **jaune** donne des **points de charge**. Quand cette charge est remplie de moitié, les dégâts infligés par une combinaison rouge sont multipliés par 2. Quand la charge est pleine, ces dégâts sont multipliés par 3.

3.2.1 Système de dégâts subits

La fonction suivante *"apply damage"* sert à infliger une quantité de dégâts au choix, elle prend donc en compte la vie et le bouclier de l'entité en attaquant en priorité le bouclier.

```
def apply_damage(self, val) :
        if val > 0 :
            if self.armor == 0 :
                if (self.life - val) > 0 :
                    self.life -= val
                else :
                    self.life = 0
            else :
                if self.armor > val :
                    self.armor -= val
                else :
                    val -= self.armor
                    self.armor = 0
                    self.life -= val
```

Fonction *"apply_ damage"*.

3.2.2 Système d'application de bouclier

Cette fonction nommée *"apply_ armor"* a pour utilité de donner une quantité personnalisée de bouclier à l'entité, tout en prenant en compte la valeur maximale que peut prendre le bouclier.

```
def apply_armor(self, val) :
        if val > 0 :
            if (self.armor + val) < self.armorMax :
                self.armor += val
            else :
                self.armor = self.armorMax
```

Fonction *"apply_ armor"*

La valeur du bouclier donnée est simple, pour chaque item bleu détruit la valeur augmente de 1. La fonction *"apply_ armor"* est appelée à condition que le nombre d'items bleus détruits soit supérieur ou égal à 1.

3.2.3 Système d'application du soin

La fonction *"apply_ heal"* permet d'ajouter de la vie au joueur en quantité personnalisé, elle prend donc en compte les points de vies du joueur mais aussi ses points de vies maximum afin de ne pas dépasser cette limite de vie.

```
def apply_heal(self, val) :
    if val > 0 :
        if (self.life + val) < (self.lifeMax) :
            self.life += val
        else :
            self.life = self.lifeMax
```

Fonction *"apply_ heal"*

La valeur du soin est définie par le nombre d'items verts détruits, la fonction *"apply heal"* est appelée à condition qu'au moins un item vert ait été détruit.

3.2.4 Système d'application de la charge

La fonction *"apply_charge"* augmente la valeur de la variable *"charged"* en prenant en compte sa limite définie dans *"chargedMax"*. C'est la valeur de cette variable qui est utilisée pour déterminer si les dégâts seront amplifiés ou non.

```
def apply_charge(self, val) :
        if val > 0 :
            #print("jauge d'attaque chargée actuelle : " + str(self.charged))
            if (self.charged + val) < self.chargedMax :
                self.charged += val
            else :
                self.charged = self.chargedMax
```

Fonction *"apply_charge"*.

Pour chaque item orange détruit, la valeur ajouté à *"charged"* est augmenté de 1, la fonction *"apply_charge"* est appelé à condition qu'au moins un item orange ait été détruit.

3.2.5 Système d'application du poison

La fonction *"apply_poison"* vérifie si le joueur est déjà empoisonné, si c'est le cas, celui-ci voit son temps d'empoisonnement augmenter de un, et s'il n'est pas déjà empoisonné alors son temps d'empoisonnement restant est définit par le nombre d'items violets détruits, et le booléen *"isPoisoned"* prend donc la valeur **True**.

```
def apply_poison(self, val) :
        if val > 0 :
            if self.poisonRows == 0 :
                self.poisonRows = val
                self.isPoisoned = True
            else :
                self.poisonRows += 1
```

Fonction *"apply_poison"*.

Plus le temps d'empoisonnement restant est grand plus les dégâts sont élevés, s'il reste un tour d'empoisonnement alors les dégâts sont égaux à 1.5, tandis que s'il reste 5 tours, cela infligera 7.5 points de dégâts. La formule est la suivante : (1.5 * temps restant).

Si le joueur est empoisonné, la fonction *"update_poison_row"* enlève un tour d'empoisonnement restant au joueur, calcul et applique les dégâts à l'entité. S'il ne l'est pas le booléen *"isPoisoned"* prend donc la valeur **False**.

```
def update_poison_rows(self) :
        if self.poisonRows > 0 :
            self.life -= (1.5 * self.poisonRows)
            self.poisonRows -= 1
        else :
            self.isPoisoned = False
```

Fonction *update_poison_row*

3.3 Application des altérations (Q.D)

Grâce aux dernières fonctions qui ont été développées, nous sommes en mesure de créer deux fonctions très importantes dans le jeu : *update_ defense* et *update_ attack*. Ces deux fonctions sont des méthodes de *Entity()* et mettent en relation l'entité du joueur et l'entité de l'ordinateur, ainsi que leurs grilles respectives. En effet il y a une subtilité à prendre en compte : certaines combinaisons d'items produisent des effets qui influencent l'entité auteure de la combinaison (par exemple le bouclier), tandis que d'autres combinaisons ont une incidence sur l'autre entité (par exemple le poison). J'ai donc procédé de la façon suivante :

1. *update_ defense* prend en paramètre une des deux grilles (on appellera donc *update_ defense* deux fois). Sur cette grille, on appelle la fonction *destroy*, qui renvoie un dictionnaire (voir section 2.3).

2. *update_ defense* donne ensuite les valeurs du dictionnaire correspondant au bouclier, au soin et à la charge (donc uniquement les modifications qui influencent l'auteur de la combinaison) aux fonctions qui vont appliquer ces modifications. *update_ defense* se nomme ainsi parce qu'elle n'applique que les altérations qui concernent l'émetteur de la combinaison. Arrivés à ce point, nous avons mis-à-jour notre quantité de vie, de bouclier et de charge. Il nous reste encore à infliger des dégâts à l'adversaire. *update_ defense* renvoie donc les valeurs du dictionnaire correspondant au poison et à l'attaque. Attention toutefois pour l'attaque de base : on doit envoyer directement le nombre de dégâts infligés, donc vérifier si la charge d'attaque puissante est remplie de moitié, pleine ou pas encore assez remplie.

```
damage = dico["🟥"]
poison = dico["🟪"]

if damage > 0 :
    if self.charged == self.chargedMax :
        damage = dico["🟥"] * 3
        self.charged = 0
    elif self.charged >= self.chargedMax/2 :
        damage = dico["🟥"] * 2
        self.charged = 0
```

Extrait de la fonction *update_ defense*, avec la mise à jour des dégâts infligés suivant l'état de la charge.

3. *update_ attack* prend en paramètres les valeurs retournées par *update_ defense*. Grâce à ces valeurs, elle applique le poison et l'attaque à l'entité sur laquelle cette méthode est appelée. C'est donc ici qu'il faut prêter plus d'attention : pour pouvoir mettre en oeuvre cela dans le jeu, *update_ defense* doit être appelée par la première entité (exemple le joueur) et *update_ attack* (qui prend en paramètres les valeurs retournées par *update_ defense* doit être appelée par la deuxième entité (exemple l'IA) !

```
damageIA, poisonIA = joueur.update_defense(gridJ)
ordi.update_attack(damageIA, poisonIA)
```

Fonctionnement du transfert de valeurs après une combinaison effectuée par le **joueur**.

Pour terminer l'application des altérations, il ne nous manque plus qu'une fonction qui met à jour le nombre de tours d'empoisonnement restants : *update_poison*. Elle fait simplement appel à *update_poison_ row*, fonction détaillée dans la section précédente.

3.4 Intelligence Artificielle (Q.D)

Nous avons à présent le bagage nécessaire pour lier les objets *Grid* aux objets *Entity*. Cependant c'est bien le fait de trouver une combinaison qui a des conséquences sur le personnage du joueur ou sur le personnage de l'ordinateur. Or, l'ordinateur est pour l'instant incapable de trouver une combinaison et d'échanger des items sur sa propre grille. Pour que l'ordinateur joue sur sa propre grille, il faut lui développer une intelligence artificielle. Nous avons pensé l'intelligence artificielle de l'ordinateur sur plusieurs niveaux de difficulté.

3.4.1 Niveau facile

Au niveau facile, l'ordinateur fait appel à la fonction *lister_switch*. Cette dernière va lister tous les coups possibles dans la grille. Pour chaque item de la grille :

1. Elle échange l'item avec son voisin de droite.
2. Si une combinaison est formée, elle ajoute les tuples de coordonnées des deux items dans une liste.
3. Elle annule l'échange, puis fait un nouvel échange de l'item avec cette fois-ci son voisin d'en-dessous.
4. Si une combinaison est formée, elle ajoute une sous-liste contenant les tuples de coordonnées des deux items dans la liste des coups possibles.
5. Elle annule l'échange.

lister_switch nous retourne finalement la liste de tous les coups possibles, il ne nous reste plus qu'à en choisir un aléatoirement et à procéder à l'échange. C'est *switch_random* qui s'en occupe.

```
def switch_random(self, listeRandom) :
    print("----------")
    switch_choisi = randint(0, len(listeRandom)-1)

    i,j = listeRandom[switch_choisi][0]
    i2, j2 = listeRandom[switch_choisi][1]

    itemDown = self.grid[i][j]
    itemUp = self.grid[i2][j2]

    print("coup choisi : ", [i,j], [i2,j2])
    switch(itemDown, itemUp, (18,18))
```

On tire au sort un entier entre 0 et le nombre de coups possibles, puis on retrouve les items à échanger grâce au contenu de la sous-liste choisie aléatoirement.

3.4.2 Niveaux "normal" et "difficile"

Ces deux niveaux reposent sur le même fonctionnement. En effet au lieu de lister tous les coups sans distinction et d'en tirer un au sort, nous allons d'abord les trier. J'ai développé la fonction *dico_switch*, qui va procéder de la même manière avec les items (échange avec celui de droite puis celui du dessous) sauf qu'il va calculer le nombre d'items présents dans la combinaison détectée. J'ai écrit *calcul_combo* pour éviter trop de répétition de code.

```
def calcul_combo(self) :
    combo = 0
    comboC = 0
    comboL = 0
    if self.combo_ligne() != False :
        comboL = len(self.combo_ligne())
    if self.combo_colonne() != False :
        comboC = len(self.combo_colonne())
    if self.is_L() != False :
        combo = comboL + comboC - 1
    else :
        combo = comboC + comboL
    return combo
```

Si une combinaison en forme de L ou de T est détectée il faut penser à retirer 1 à la variable "combo", puisqu'un item est commun à la ligne et à la colonne.

Ensuite, on procède à un tri : les combinaisons de trois items sont rangées dans la "liste3", les combinaisons de quatre items dans "liste4", et celles de cinq items ou plus dans "liste5". Une fois tous les items analysés, la fonction renvoie un dictionnaire avec les trois listes. Ce dernier a pour clés "combo3", "combo4" et "combo5".

Finalement, on utilise soit *switch_ normal* soit *switch_ hard* en fonction de la difficulté qu'on souhaite. Ces deux fonctions appellent *dico_ switch* pour trier les coups possibles, puis elles priorisent leur tirage au sort. Prenons le cas de *switch_ normal* :

1. Si la liste des combinaisons de quatre items n'est pas vide, alors on tire au sort un des/le coup(s) de la liste.
2. Si elle est vide, on tente avec la liste des combinaisons de cinq items.
3. Si elle aussi est vide, alors on "pioche" dans celle qui rassemble les combinaisons de trois items.

```
def switch_ia_hard(self) :
    '''IA Mode difficile : priorité aux co
    dico = self.dico_switch()
    if len(dico['combo5']) >= 1 :
        self.switch_random(dico['combo5'])
    elif len(dico['combo4']) >= 1 :
        self.switch_random(dico['combo4'])
    else :
        self.switch_random(dico['combo3'])
```

Autre exemple avec *switch_ hard* : priorité aux combos de 5, puis de 4 et enfin de 3.

3.5 Déroulement d'une partie (Q.D)

Nous disposons désormais de tous les éléments nécessaires
pour jouer une partie en console, alors allons-y !

1. Initialisation : Premièrement, nous avons besoin d'instancier deux grilles (*Grid*) et deux entités (*Entity*) : pour rappel, le joueur et l'ordinateur jouent sur leur grille respective. Les deux entités partent avec un nombre de points de vie de base. On remplit les grilles d'items avec la fonction *remplir_grille* dont le fonctionnement a été détaillé dans la section 2.5

2. Le joueur échange deux items voisins pour former une combinaison avec la fonction *switch*. La combinaison est détruite et provoque des altérations. On appelle *update_defense* sur l'entité du joueur en stockant ses valeurs de retour, puis on appelle *update_attack* sur l'entité de l'ordinateur, en passant en paramètres les valeurs de retour de *update_defense*. Les items au-dessus de la combinaison détruite tombent et de nouveaux items apparaissent.

3. L'ordinateur échange deux items sur sa grille grâce à son intelligence artificielle. Cette fois-ci on appelle *update_defense* sur l'entité de l'ordinateur puis on appelle *update_attack* sur l'entité du joueur. Les items au-dessus de la combinaison détruite tombent et de nouveaux items apparaissent.

4. Ces échanges d'items se poursuivent tour-à-tour, et la partie se termine lorsque les points de vie d'une des deux entités arrivent à zéro.

4 FONDATIONS DE L'INTERFACE GRAPHIQUE (A.P)

Afin de rendre harmonieuse l'utilisation des aspects techniques précédemment cités, il nous fallait utiliser un moteur de jeu : notre choix s'est dirigé vers Pygame, une bibliothèque Python qui s'adapte parfaitement à ce que l'on souhaite faire. Interface graphique, gestion des effets sonores, détection et création d'événements... Tout y est pour que l'on puisse correctement développer notre jeu, dans de bonnes conditions.

4.1 Navigation

Dans tous les jeux, un menu de navigation est un élément indispensable. Alors, voyons comment et par quelles techniques nous avons réussi à inclure celui-ci dans notre jeu.

4.1.1 Première version

Au début, avant d'utiliser un moyen beaucoup plus facile et complet pour afficher les différentes pages, nous utilisions une boucle infinie pour afficher le menu principal, avant la boucle de jeu. Une technique peu conseillée car peu adaptative et complétive. Le procédé était simple : tant que le joueur ne cliquait pas sur le bouton jouer au milieu, rien ne se passait. Cependant, dès que celui-ci était appuyé le joueur pouvait directement jouer, sans passer par un menu de navigation complet (qui était impossible à ajouter en utilisant ce procédé).

```
while menu :

    fenetre.blit(fond, (0,0))
    fenetre.blit(bouton_start.image, (435,100))

    for event in pygame.event.get():
        if event.type == pygame.QUIT :
            sys.exit()

        if event.type == pygame.MOUSEBUTTONDOWN:
            if bouton_start.rect.collidepoint(pygame.mouse.get_pos()) :
                menu = False
```

Extrait de l'ancien code permettant l'affichage d'un menu simple.

Le code récupérait une variable menu précédemment déclarée et dont la valeur était le booléen *True*, et lorsque le joueur appuyait sur le bouton cette variable changeait de valeur en *False* affichant directement le jeu. Une technique simple qui nous a bien aidé mais qui n'était ni fiable ni durable. Désormais, le code n'est plus du tout comme cela, mais bien plus complet et compréhensible.

4.1.2 Création des classes

Dans notre nouvelle version du code, nous avons principalement utilisé des classes pour correctement organiser nos menus et que l'on ne se perde pas dans notre code. Chaque classe présente dans le fichier *main.py* est utile pour un menu distinct : la classe *Start()* pour la page d'accueil, la classe *Game()* pour l'affichage de la grille... et d'autres.

```
class Options() :
    def __init__(self) :
        self.groupSprites = pygame.sprite.Group()
        self.groupSprites.add(Button(250, 50, "../Images/volume_on.png", 'sound_off', 160, 160))
        self.groupSprites.add(Button(560, 550, "../Images/panneauMP.png", 'to_menu', 188, 116))

    def react(self, event) :
        if event.type == pygame.MOUSEBUTTONDOWN :
            for sprite in self.groupSprites :
                sprite.set_triggered(event.pos)

    def draw(self, fenetre) :
        self.groupSprites.update()
        self.groupSprites.draw(fenetre)
```

La classe *Options()* présente dans le fichier *main.py*.

Toutes ces classes ont différentes méthodes en commun, comme la méthode *react()* qui réagit aux événements provoqués par le joueur, ou encore la classe *draw()* pour afficher les boutons et les décorations. À propos des boutons et des décorations, leur création, leur assignation d'événement et leur affichage sont eux aussi gérés par une classe. Chaque bouton ou décoration créés par le biais des classes sont attribués à un élément appelé sprite, une méthode adaptée pour afficher des objets sur l'interface graphique. Concentrons nous sur la classe *Button()*, classe qui détecte le clic sur un sprite préalablement placé sur l'interface graphique. Dans celle-ci nous y retrouvons deux méthodes :

- La méthode *set_ triggered()* qui active le bouton quand il est cliqué ;
- La méthode *update()* qui change de page selon l'événement (*variable evenement*) assigné au bouton cliqué.

```
def update(self):
    global isWorld2
    global isMuted

    if self.triggered:

        if self.evenement == 'to_menu' :
            pygame.event.post(pygame.event.Event(MENU_PRINCIPAL))

        if self.evenement == 'to_diff1' :
            isWorld2 = False
            pygame.event.post(pygame.event.Event(CHOIX_DIFF1))
```

Extrait de la méthode *update()* de la classe *Button()*.

Afin de changer de page, nous utilisons des événements préalablement créés plus haut dans le fichier *main.py*. Ces événements sont ensuite assignés dans la boucle de jeu à une classe de page comme *Start()* ou *Options()*.

4.1.3 Gestion des événements

La bibliothèque Pygame nous permet de créer nos propres événements et de les appeler sous conditions spéciales que nous même pouvons définir. Ceux-là nous permettent donc, comme nous pouvons le voir sur l'image ci-dessus, de déclencher des événements selon des conditions données. Dans notre cas, chaque bouton a un intitulé d'événement différent selon sa fonction, et celui-ci active ou non le changement de page.

```
NEW_GAME = pygame.USEREVENT + 1
RESET_GAME = pygame.USEREVENT + 2
MENU_PRINCIPAL = pygame.USEREVENT + 3
OPTIONS = pygame.USEREVENT + 4
```

Exemple de création des événements.

4.2 Gestion de l'affichage

Nous venons de voir les techniques utilisées pour permettre l'affichage de plusieurs pages, cependant maintenant il nous faut directement les afficher dans l'interface graphique.

4.2.1 Boucle de jeu

Dans la boucle de jeu principale, sous la forme « *while True :* », nous avons spécifié chaque événement et ce que ceux-là provoquaient (*Activation de la musique, changement de page...*). De cette manière, il est pour nous très facile de rajouter plus de menus, plus de boutons, plus de conditions, sans nous compliquer la tâche.

```
while True:

    for event in pygame.event.get():

        if event.type == pygame.QUIT:
            sys.exit()

        elif event.type == MENU_PRINCIPAL :
            if isWorld2 :
                background = backgroundM2
            else :
                background = backgroundM1

            if isGameEnded and not isMuted :
                channelMusic.play(musiqueSTA,999)

            isGameEnded = False
            frame = Menu()
            fenetre.blit(background, (140, 0))
            fenetre.blit(fond, (0, 0))
```

Début de la boucle de jeu, où l'on peut voir un changement de page (*frame = Menu()*).

4.2.2 Variable d'affichage

Pour nous permettre d'afficher toutes ces informations directement dans l'interface graphique, nous utilisons une seule et unique variable *frame*, qui selon les événements variera entre plusieurs classes. Dans le code ci-dessus, la valeur par défaut de la variable *frame* est la classe *Start()*, soit la première frame que l'on veut voir affichée lorsque l'on lance le jeu. Quand une touche ou la souris est appuyée, la variable *frame* change de valeur et la classe *Menu()* lui est attribuée. Selon les configurations et les boutons cliqués, la variable *frame* changera de valeur pour laisser la nouvelle page s'afficher.

Pour concrètement afficher les différentes pages, nous avons dit précédemment que chacune des classes de pages ont des méthodes en commun, dont la méthode *draw()* qui affiche donc en dur les boutons et décorations dans l'interface graphique.

5 JEU EN INTERFACE GRAPHIQUE (Q.D)

Après avoir posé les bases de notre développement en interface graphique, nous allons pouvoir parler du fonctionnement du jeu dans celle-ci. En effet en interface, nous ne disposons pas d'un terminal nous permettant de communiquer directement avec le jeu, il a donc fallu repenser l'interaction avec l'application.

5.1 Mise en place

En nous conformant à l'organisation mise en place dans la section 4, il est naturel de créer une classe *Game()* pour créer une nouvelle partie. C'est dans cette classe que va se dérouler le jeu. Il y a cependant beaucoup d'autres choses à faire pour pouvoir commencer à travailler en interface graphique. En effet, nos objets n'ont pour l'instant aucune représentation graphique. Il faut donc leur rajouter des attributs. C'est ce que nous faisons par exemple avec les grilles et les entités, en leur rajoutant une image. Tous les visuels ont été faits maison (décors et items : Q.D, personnages : O.B et A.P)

```
doriTaille = (288/2,288/2)
doritos1 = pygame.transform.scale(pygame.image.load("../Images/Personnages/Doritos/Doritos-0.png"), doriTaille)
doritos2 = pygame.transform.scale(pygame.image.load("../Images/Personnages/Doritos/Doritos-1.png"), doriTaille)
doritos3 = pygame.transform.scale(pygame.image.load("../Images/Personnages/Doritos/Doritos-4.png"), doriTaille)
doritos4 = pygame.transform.scale(pygame.image.load("../Images/Personnages/Doritos/Doritos-5.png"), doriTaille)
spritesDoritos = [doritos1, doritos2, doritos2, doritos2, doritos3, doritos4, doritos4, doritos4]
```

Exemple de chargement d'images pour l'apparence du joueur (doritos)

Une fois cela fait nous sommes rapidement en mesure d'afficher un arrière-plan, la grille du joueur, la grille de l'ordinateur et une image fixe de nos personnages, en renseignant quelques coordonnées. Nous avons une base visuelle sur laquelle travailler.

5.2 Affichage des items

Les éléments principaux de notre jeu sont les items, qui vont être amenés à bouger. Avant cela, il faut leur donner une image. Lors du remplissage de notre grille, on construit des items de type *ItemPygame* qui associent à leur nom (un des caractères unicode de couleur) une image. Par exemple, l'item de nom 'orange' va avoir une image de pièce jaune, l'item de nom 'rouge' va avoir une image d'épée rouge, etc... C'est la méthode *setSprite* de la classe *ItemPygame* qui s'en occupe.

```
DICO_IMAGE = {
    "rouge" : pygame.image.load("../Images/items-proto/damageSmall.png").convert_alpha(),
    "orange" : pygame.image.load("../Images/items-proto/chargedSmall.png").convert_alpha(),
    "vert" : pygame.image.load("../Images/items-proto/healSmall.png").convert_alpha(),
    "bleu" : pygame.image.load("../Images/items-proto/armorSmall.png").convert_alpha(),
    "violet" : pygame.image.load("../Images/items-proto/poisonSmall.png").convert_alpha(),
}
```

On précharge les images d'items pour éviter de les charger à chaque construction d'items, par souci de performance. On utilise le suffixe *convert_ alpha* pour que l'image gère la transparence.

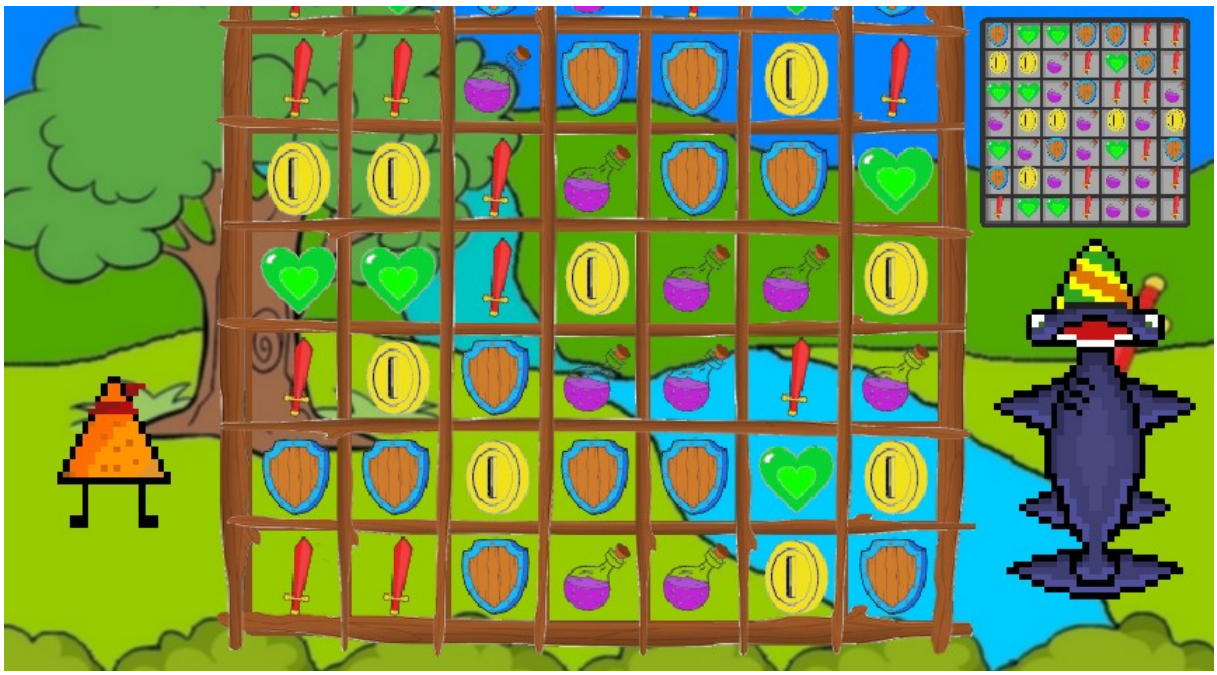
```
def setSprite(self, color, item_size):
    if color == "🟥" :
        load_img = DICO_IMAGE["rouge"]
        load_img = pygame.transform.scale(load_img, item_size)
        self.image = load_img
        self.rect = self.image.get_rect()
        return self.image
```

Extrait de *setSprite* qui affecte ici une image à l'item rouge. Elle lui donne aussi un "rectangle", nous verrons pourquoi dans la section 5.3

Les items ont maintenant une image, il faut désormais trouver un moyen de les afficher durablement. C'est là que les attributs "coordX" et "coordY" de nos items rentrent en jeu. En les multipliant on peut leur donner des coordonnées en interface qui seront proportionnelles, et faire en sorte qu'ils s'affichent correctement. Malheureusement je n'ai pas eu le temps de précalculer les coefficients par rapport aux dimensions de la fenêtre d'affichage, j'ai préféré me concentrer sur d'autres fonctionnalités. J'ai alors fait deux fonctions : une pour afficher les items dans la grille du joueur (les items sont plus grands et plus espacés) et une autre pour afficher les items dans la grille de l'ordinateur.

```
def affiche_grille_joueur(self, fenetre) :
        '''Affichage de la grille du joueur sur Pygame'''
        decalage = 0
        for ligne in self.grid :
            for item in ligne :
                if type(item) == itemPygame.Item_Pygame :
                    coordX = item.coordX * 71 + 405
                    coordY = item.coordY + 124 + decalage
                    item.rect.left = coordX
                    item.rect.top = coordY
                    fenetre.blit(item.image, (coordX, coordY))
            decalage += 70
```

Ici, nous multiplions les coordonnées par un certain nombre pour que l'image de l'item s'affiche correctement, et on incrémente un décalage à chaque ligne d'items. De plus, on met à jour la position du rectangle de chaque image, je vous renvoie encore à la section 5.3 pour comprendre. Les coordonnées sont donc différentes pour l'affichage de la grille de l'ordinateur. En interface (avec les visuels actuels du jeu), nous obtenons ce résultat :



La grande grille est celle du joueur (son personnage est ici le doritos), et la petite est celle de l'ordinateur (son personnage est ici un requin avec un chapeau d'anniversaire)

Nous avons fait le choix de faire apparaître la grille de l'ordinateur pour que le joueur voit ses actions et qu'il puisse éventuellement se préparer à une attaque si il constate que beaucoup d'alignements de potions ou d'épées sont disponibles dans la grille de l'ordinateur par exemple.

5.3 Détection des items

Actuellement nous sommes donc capables d'afficher nos items, mais comment interagir avec eux ? Nous n'avons pas de terminal pour entrer les coordonnées d'items à échanger, nous allons donc passer par la souris.

Pygame est capable de détecter beaucoup d'événements comme le clic et le relâchement de souris, et dispose également d'un module consacré à la souris nommé *pygame.mouse*. Il y a une fonction dans ce module qui va nous être particulièrement utile, il s'agit de collidepoint. En effet grâce à cette fonction, nous allons pouvoir tester si les coordonnées du clic de la souris (obtenues avec *pygame.mouse.get_pos()*) sont situées dans le **rectangle** des items de la grille. Il est très important de différencier le rectangle de l'image. En effet *collidepoint* ne fonctionnera pas si on lui donne une image. C'est donc pour cela que nous avons besoin de définir un rectangle aux items de la grille du joueur, mais aussi de les mettre à jour dans leur affichage. C'est parce qu'ils doivent capter le clic et le relâchement.

```
def calcul_i_j(frame, channel, sound, status):
    position = pygame.mouse.get_pos()
    for ligne in range(len(frame.spritePosition(0).grid)) :
        for colonne in range(len(frame.spritePosition(0).grid[ligne])) :
            objet = frame.spritePosition(0).grid[ligne][colonne]
            if type(objet) == itemPygame.Item_Pygame :
                if (objet.rect.collidepoint(position)) :
                    if status == 'down' : channel.play(sound)
                    return ligne, colonne
    return -1, -1
```

C'est cette fonction qui gère la détection du clic sur un item. Ici *frame.spritePosition(0)* peut perturber, mais ce n'est en fait que l'objet 'Grid' du joueur. Il est en effet ajouté dans les sprites de la classe *Game()*. On va premièrement chercher les coordonnées de la souris, puis on parcourt tous les items de la grille du joueur pour savoir si la souris est dans le rectangle d'un de ces items. Si c'est le cas, elle retourne les coordonnées dans la grille de l'item concerné (pas les coordonnées de son image, je précise).

5.4 Échange en interface

Cette fonction *calcul_i_j* est appelée quand la souris est pressée ainsi que lorsqu'elle est relâchée. L'échange d'items est déclenché lorsque la souris a été pressée sur un item, et qu'elle a été relâchée sur un item voisin. Cela est possible car :

- Au clic, *calcul_i_j* est exécuté et retourne les coordonnées du premier item (i,j)
- Au relâchement, *calcul_i_j* est de nouveau exécuté et si il renvoie des coordonnées égales à (i+1, j) ou (i, j+1) ou (i-1, j) ou (i, j-1) alors c'est que le relâchement s'est fait sur un item voisin.

A ce moment là, le "switch" en interface a lieu. Sauf qu'on ne change pas vraiment de place les deux items. Cela impliquerait de mettre à jour leurs coordonnées ainsi que leur rectangle. Nous avons choisi de faire plus simple, en échangeant seulement leur attribut "color". Après cette échange nous faisons appel à la fonction *updateSprite* de *ItemPygame* pour mettre à jour l'image de l'item.

```
def switch(itemDown, itemUp, size) :
    itemDown.color, itemUp.color = itemUp.color, itemDown.color
    itemDown.updateSprite(size)
    itemUp.updateSprite(size)
```

On n'échange plus que les attributs "color"

5.5 Visibilité et couches

Pour que l'échange soit visible en interface et qu'il n'y ait pas un empilement d'items incompréhensible, toutes les couches affichées en-dessous des items doivent être réaffichées par dessus, et on réaffiche après tous les items par-dessus. Cet affichage est laborieux et très verbeux, c'est pourquoi j'ai compilé tous ces affichages dans une fonction, qui nous sert assez souvent quand on doit actualiser l'image.

```
def blits(self, gridJ, gridIA) :
    fenetre.blit(self.background, (140,0))
    fenetre.blit(fond, (0,0))
    fenetre.blit(gridJ.image, (gridJ.rect.x, gridJ.rect.y))
    gridJ.affiche_grille_joueur(fenetre)
    joueur.affiche_jauges(fenetre, (230, 110), (130,12))
    ordi.affiche_jauges(fenetre, (930, 110), (130,12))
    fenetre.blit(gridIA.image, (920, 180))
    gridIA.affiche_grille_ia(fenetre)
```

Fonction *blits* de la classe *Game()* qui affiche toutes les couches de base du jeu

On utilise également *blits* dans la fonction principale qui gère les grilles en interface : *update_grilles* du fichier *grille.py*. Cette fonction gère la destruction, la gravité et la génération de nouveaux items (renvoi à 2.5). Il faut donc à chaque fois réafficher toutes les couches. De plus, pour constater toutes les étapes, on actualise avec *pygame.display.flip()* et on laisse du délai avec *pygame.time.delay()*

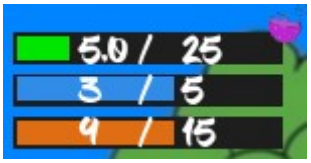
5.6 Affichage de jauges

Pour nous rendre compte de l'état de notre personnage et de notre adversaire, il faut faire exister des jauges. J'ai alors développé la fonction *affiche_jauges* dans la classe *Entity()*. Elle fonctionne en calculant un pourcentage par rapport à la valeur actuelle (de vie par exemple) et la valeur maximale. Ensuite, elle dessine (avec *pygame.draw* un rectangle d'une longueur fixe représentant le maximum, puis elle dessine un autre rectangle par dessus, qui a pour longueur un certain pourcentage de la longueur fixe. Ce pourcentage est défini par la valeur actuelle de vie.

```
#life
pygame.draw.rect(surface, self.fondBarColor, pygame.Rect((x-2, y-2), (longueur+4, hauteur+4)))
pygame.draw.rect(surface, self.lifeColor, pygame.Rect(coord, (lifePercent * longueur, hauteur)))
if self.life < 0 :
    life = font.render('0', 1, (255,255,255))
else :
    life = font.render(str(self.life), 1, (255,255,255))
lifeMax = font.render("/   " + str(self.lifeMax), 1, (255,255,255))
surface.blit(life, (x+30,y-1))
surface.blit(lifeMax, (x+60, y-1))
```

Extrait de la fonction *affiche_jauges*, qui dessine la jauge de vie

J'utilise également le module *font* de Pygame pour pouvoir afficher les nombres par-dessus les jauges, pour plus de clarté. En interface, on obtient ce rendu :



La petite fiole au dessus de la barre de vie indique que l'on est empoisonné.

6 FONCTIONNALITÉS SUPPLÉMENTAIRES

Dans cette section, nous vous décrirons les autres fonctionnalités que nous avons implémenté dans notre jeu, telles que la gestion des niveaux, l'ajout de musique et de sons, ou encore l'animation des personnages.

6.1 Système de progression (A.P)

Sans système de progression et de récompense, le jeu actuel ne serait pas complet. C'est pour cela que nous avons mis en place celui-ci. Trois niveaux pour chaque monde, avec une difficulté différente à chaque fois, et des ennemis qui varient.

6.1.1 Fonctionnement global

Dans l'ensemble de nos fichiers se trouve un fichier *level.py*, gérant les niveaux, leur accès par le joueur et donc la progression.

```
class Level():

    #Initialisation=================================================================
    def __init__(self, progression): #initialiser le manager de niveau
        self.niveau = 1
        self.totalLevel = 6
        self.unlockedLevel = progression
        self.name = ""

    #Tests==========================================================================
    def assertNiveau(self, val) :
        return val <= self.unlockedLevel and self.unlockedLevel <= self.totalLevel
```

Intégralité du fichier *level.py*.

Ce fichier est relativement court, mais ses valeurs internes sont elles très importantes. La valeur *self.unlockedLevel* est le niveau maximum actuel auquel le joueur peut accéder, initialement 1. Celle-ci est incrémentée dans un autre fichier, *entity.py*, par la fonction *end_game()* qui détecte qui du joueur ou de l'ordinateur a terminé le niveau. Si c'est le joueur qui a terminé le niveau, alors la valeur est incrémentée et le niveau suivant est accessible.

```
def end_game(self, surface, level) :
    if (self.life <= 0) :

        if (self.player == False) :
            if (level.unlockedLevel < level.totalLevel and level.niveau == level.unlockedLevel) :
                level.unlockedLevel += 1
            return True

        else : return True

    else : return False
```

Fonction *end_game()* présente dans le fichier *entity.py*.

6.1.2 Application

Maintenant que toutes les bases sont posées, nous pouvons directement gérer la progression. Pour cela, nous nous rendons dans le fichier main.py, plus précisément dans la méthode react() de la classe Game().

```
if joueur.end_game(fenetre, levelManager) :
    pygame.time.delay(1000)
    isGameEnded = True
    pygame.event.post(pygame.event.Event(LOSE))
    return 0
if ordi.end_game(fenetre, levelManager) :
    pygame.time.delay(1000)
    isGameEnded = True
    pygame.event.post(pygame.event.Event(WIN))
    return 0
```

Extrait de la méthode *react()* de la classe *Game()*.

Dans ce code, il nous suffit simplement de détecter qui de l'ordinateur ou du joueur est mort en premier :

- Si l'ordinateur meurt en premier, alors la partie est gagnée par le joueur et le prochain niveau est débloqué pour le joueur ;
- Cependant, si le joueur meurt en premier, alors aucun niveau n'est débloqué pour ce dernier et le niveau actuel est à rejouer et à terminer tant que celui-ci est perdu.

6.2 Musique et sons Q.D

J'ai composé une petite musique pour le jeu, dont on peut gérer le volume dans le menu "Settings" géré par la classe *Options()*. Ce menu dispose d'un bouton mute, un bouton qui augmente le volume et un bouton qui le baisse. J'ai dû utiliser une variable globale "isMuted" pour sauvegarder les changements opérés dans le menu "Settings". J'en ai également utilisé pour sauvegarder le changement de décor lorsque l'on passe au deuxième monde du jeu. J'ai aussi ajouté des sons de claves : un est joué quand on clique sur un bouton ou un item dans le jeu, un autre est joué quand l'échange entre deux items est valide, et un autre est joué lorsque l'échange entre deux items ne produit aucune combinaison.

```
pygame.mixer.init()

musiqueSTA = pygame.mixer.Sound("../Sounds/musiqueSTA.mp3")
buttonPressed = pygame.mixer.Sound("../Sounds/buttonSound.mp3")
switchGood = pygame.mixer.Sound("../Sounds/switchSound.mp3")
switchWrong = pygame.mixer.Sound("../Sounds/noswitchSound.mp3")
victory = pygame.mixer.Sound("../Sounds/victory.mp3")

channelMusic = pygame.mixer.Channel(1)
channelMouse = pygame.mixer.Channel(0)

channelMusic.set_volume(0.5)
channelMouse.set_volume(0.8)

channelMusic.play(musiqueSTA, 999)
```

Pour gérer la musique et ces sons de claves séparément, on doit créer deux "radio" distinctes avec *pygame.mixer.Channel*

6.3 Animation de personnages Q.D

Pour animer les personnages, on charge leurs images que l'on met dans une liste, puis on itère à chaque tour de boucle sur cette liste. On affiche alors à chaque tour de boucle une image différente ; Il ne faut pas oublier de réafficher les couches d'en-dessous, avec une fonction *blits*. Les tours de boucle sont comptés par une variable "compteur" passée en paramètre.

7 MANUEL DE JEU

Bienvenue sur 'Switch Them All'! Vous incarnez un petit Doritos et vous devez rétablir la paix dans deux mondes différents. Le jeu comporte 6 niveaux à débloquer les uns après les autres, et il y a 3 niveaux par monde. Sur votre route vous croiserez des ennemis différents, tantôt pas très fûtés, tantôt relativement intelligents. Nous comptons sur vous pour rétablir la paix dans le monde du Switch!

Dans une partie, cliquez sur un item, maintenez la souris et relâchez sur un item voisin pour les échanger. Il doit obligatoirement se former une combinaison d'au moins 3 items pour qu'elle soit détruite et appliquée, soit à votre petit doritos, soit à votre redoutable ennemi.

- Les **coeurs** remplissent votre **jauge de vie** (en vert)
- Les **boucliers** remplissent votre **jauge d'armure** (en bleu)
- Les **pièces** remplissent votre **jauge de charge** (en gris, puis orange, puis rouge)
- Les **épées** infligent des dégâts à l'adversaire qui peuvent être diminués ou absorbés par son armure.
- Les **potions** infligent un empoisonnement à l'adversaire, qui lui retire des points de vie sur plusieurs tours.

Lorsque votre jauge de charge est **orange**, les épées font 2 fois plus de dégâts, lorsque cette jauge est **rouge**, les épées font 3 fois plus de dégâts. Une attaque lancée alors que la jauge de charge est soit orange soit rouge remet la jauge de charge à zéro.

Vous savez tout, maintenant, à vous de jouer!