Documentation du projet Khet

Moteur graphique :

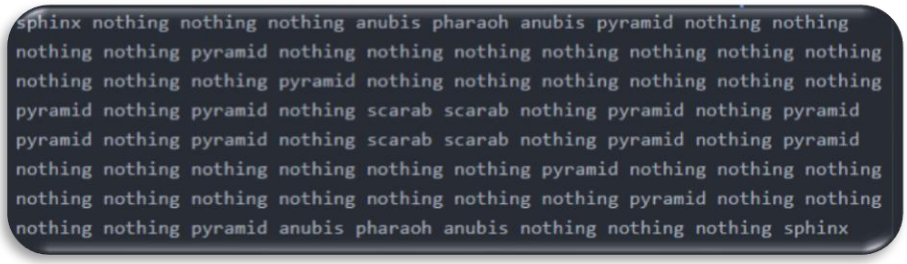
Nous avons choisi pour le projet, d'utiliser le moteur graphique Pygame associé au langage Python, cela n'était pas notre premier choix puisque nous avions opté tout d'abord pour SDL en c++. Après quelques semaines nous nous sommes rendus compte que cette option n'était pas la bonne de part le nombre d'exceptions importantes que l'on peut remarquer dans la librairie SDL. Le langage Python a donc été notre second choix, couplé avec Pygame et ses fonctions de classe nous avions tout ce qu'il nous fallait pour ce projet. De plus Pygame nous permet de créer du contenu multiplateforme (Windows, MacOS, Linux).

Choix algorithmiques :

• Structure du plateau de jeu :

Nous avons opté pour 3 définitions différentes afin d'afficher et de charger correctement notre plateau de jeu.

La première « generate_list_name » : nous avons stocké tous les noms de chaque pièce dans cette map avec leur emplacement respectif, quand on a une case vide elle est marquée comme « nothing » dans la map.



On peut donc voir ici un exemple avec le map classique du jeu.

Dans notre code on a donc cette fonction :

```
def generate_list_name(self):
    """Generate a name's pawn's list"""
    with open('source/map/map_{}'.format(self.name_map), 'r') as my_file_name:
        self.list_name = my_file_name.read().splitlines()
        for i in range (8):
            self.list_name[i] = self.list_name[i].split(" ")
        return self.list_name
```

Dans cette fonction, tout d'abord nous ouvrons la map qui se situe dans le dossier /source/map, en y accédant en spécifiant le chemin d'accès de celle-ci en paramètre (comme nous avons plusieurs map le nom se glisse entre les accolades). On lit le fichier avec le paramètre « r ».
Ensuite toutes les lignes sont stockés dans une liste.
Et enfin on a une boucle qui permet d'itérer sur toutes les lignes de la liste self.list_name et le « split » nous sert à différencier les noms de pièces grâce aux espaces.
On retourne à la fin le nom de la liste.

La seconde « generate_list_owner » : nous avons stocké ici l'appartenance des joueurs à une case et donc à un pion. Les « player_one_fix » et « player_two_fix » nous servent pour les cases qui appartiennent exclusivement au joueur 1 ou 2. Les « player_one » sont les cases appartenant au joueur 1 en début de partie et même chose pour le joueur 2. Les « neutral » sont donc les cases vides, celles qui n'ont pas d'appartenance particulière.

Exemple pour la map classique :

```
player_one_fix player_two_fix neutral neutral player_one player_one player_one player_one player_one_fix player_two_fix
player_one_fix neutral player_one neutral neutral neutral neutral neutral neutral player_two_fix
player_one_fix neutral neutral player_two neutral neutral neutral neutral neutral player_two_fix
player_one_fix neutral player_two neutral player_one player_one neutral player_one neutral player_two_fix
player_one_fix neutral player_two neutral player_two player_two neutral player_one neutral player_two_fix
player_one_fix neutral neutral neutral neutral neutral player_one neutral neutral player_two_fix
player_one_fix neutral neutral neutral neutral neutral neutral player_two neutral player_two_fix
player_one_fix player_two_fix player_two player_two player_two player_two neutral neutral player_one_fix player_two_fix
```

Dans notre code on a cette fonction :

```
def generate_list_owner(self):
    """Generate a name's box's list"""
    with open('source/map/box_{}'.format(self.name_map), 'r') as my_file_name:
        self.list_box_owner = my_file_name.read().splitlines()
        for i in range (8):
            self.list_box_owner[i] = self.list_box_owner[i].split(" ")
        return self.list_box_owner
```

Le principe de cette fonction est identique à la fonction précédente.

La troisième « generate_list_direction » : nous avons stocké ici les directions (north, south, east, west) de chaque pièce au commencement d'une partie.

Exemple pour la map classique :

```
south nothing nothing nothing south south south west nothing nothing
nothing nothing north nothing nothing nothing nothing nothing nothing nothing
nothing nothing nothing east nothing nothing nothing nothing nothing nothing
south nothing north nothing west north nothing west nothing east
west nothing east nothing south east nothing south nothing north
nothing nothing nothing nothing nothing nothing west nothing nothing nothing
nothing nothing nothing nothing nothing nothing nothing south nothing nothing
nothing nothing east north north north nothing nothing nothing north
```

Dans notre code on a cette fonction :

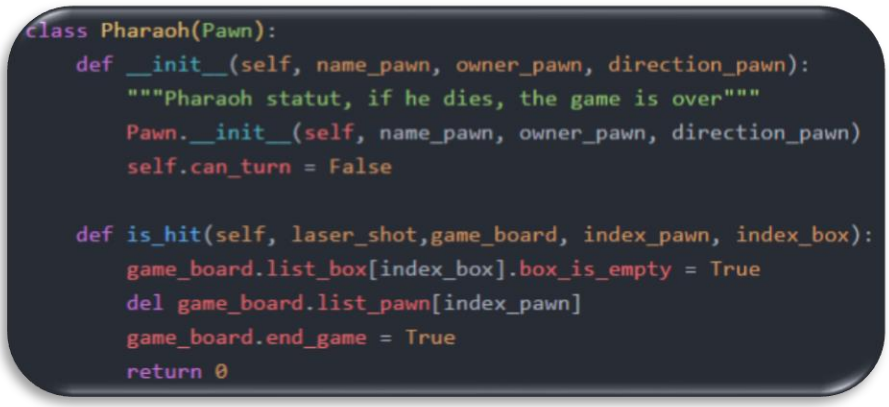
```
def generate_list_direction(self):
    """Generate a 's pawns's list"""
    with open('source/map/direction_{}'.format(self.name_map), 'r') as my_file_direction:
        self.list_direction = my_file_direction.read().splitlines()
        for i in range (8):
            self.list_direction[i] = self.list_direction[i].split(" ")

        return self.list_direction
```

Le principe de cette fonction reste encore une fois le même que vu précédemment.

• Structure des pièces :

Nous avons opté pour l'utilisation de l'objet en Python pour gérer les pièces, de part l'existence des classes et de l'héritage cela nous paraissait le plus astucieux. Pour cela nous avons donc mis chaque pièce dans une classe et nous avons ajouté à celle-ci des fonctions spécifiques à chacune en fonction de leur rôle dans le jeu.

Pour le « Pharaoh » on a cette classe :



```
class Pharaoh(Pawn):
    def __init__(self, name_pawn, owner_pawn, direction_pawn):
        """Pharaoh statut, if he dies, the game is over"""
        Pawn.__init__(self, name_pawn, owner_pawn, direction_pawn)
        self.can_turn = False

    def is_hit(self, laser_shot,game_board, index_pawn, index_box):
        game_board.list_box[index_box].box_is_empty = True
        del game_board.list_pawn[index_pawn]
        game_board.end_game = True
        return 0
```

La fonction « __init__ » sert à initialiser les pions. Quand un pion est créé on rentre les attribus suivant. Tout d'abord le nom ensuite l'appartenance et enfin la direction de celui-ci.

La fonction « is_hit » nous sert simplement à dire que si cette pièce est touchée par le laser alors le jeu s'arrête et c'est la fin de la partie.

Pour « Anubis » on a cette classe :

```
class Anubis(Pawn):
    """Create Anubis's pawn"""
    def __init__(self, name_pawn, owner_pawn, direction_pawn):
        Pawn.__init__(self, name_pawn, owner_pawn, direction_pawn)
        self.accept_exchange_position = True

    def is_hit(self, laser_shot,game_board, index_pawn, index_box):
        index_laser_shot = laser_shot.list_direction_possible.index(laser_shot.direction_laser)
        if self.direction_pawn == laser_shot.list_direction_possible[((index_laser_shot + 2) % 4)]:
            pass
        else:
            game_board.list_box[index_box].box_is_empty = True
            del game_board.list_pawn[index_pawn]

        return index_box
```

La fonction « __init__ » est la même que précédemment et elle restera la même pour toutes les autres classes.

La fonction « is_hit » dit que si Anubis est touché mais qu'il est face au laser alors il reste sur le terrain et n'est pas détruit sinon il est supprimé.

Pour le « Scarab » nous avons :

```python
class Scarab(Pawn):
    def __init__(self, name_pawn, owner_pawn, direction_pawn):
        Pawn.__init__(self, name_pawn, owner_pawn, direction_pawn)
        self.can_exchange_position = True
        self.return_laser = True

    def is_hit(self, laser_shot,game_board, index_pawn, index_box):
        index_laser_shot = laser_shot.list_direction_possible.index(laser_shot.direction_laser)

        if self.direction_pawn == 'north' or self.direction_pawn == 'south':
            if laser_shot.direction_laser == 'north':
                index_box = index_box + 1
                laser_shot.direction_laser = 'east'

            elif laser_shot.direction_laser == 'south':
                index_box = index_box - 1
                laser_shot.direction_laser = 'west'

            elif laser_shot.direction_laser == 'west':
                index_box = index_box + 10
                laser_shot.direction_laser = 'south'

            elif laser_shot.direction_laser == 'east':
                index_box = index_box - 10
                laser_shot.direction_laser = 'north'
```

La fonction « is_hit » prend en compte les reflets du laser face au scarab dans chaque direction. Dans cette image on prend en compte quand le pion est dirigé vers le nord et le sud (est et ouest sont fait de la même façon).
Si le laser va en direction du Nord alors l'index de notre pion est implémenté de 1 ce qui signifie dans le tableau qu'il va vers la case suivante de droite donc dans la direction « east ».
Sinon si le laser va en direction du Sud, l'index du pion est décrémenté de 1 ce qui signifie que dans un tableau c'est comme si on allait sur la case de gauche d'où la direction du laser qui est « west ».
Sinon si le laser va en direction de l'Ouest alors l'index du pion est incrémenté de 10 car on veut aller à la case en dessous de celui si et comme il y a 10 cases par lignes c'est pour cela que l'on incrémente de cette façon. On va donc à la case du dessous d'où le pourquoi la direction du laser est « south ».
Sinon si le laser va en direction de l'Est alors l'index du pion est décrémenté de 10 (même principe que le « elif » du dessus mais à l'inverse) le laser va donc aller sur la case au-dessus de notre pion d'où la direction « north ».

Pour la « Pyramid » :

```
class Pyramid(Pawn):
    def __init__(self, name_pawn, owner_pawn, direction_pawn):
        Pawn.__init__(self, name_pawn, owner_pawn, direction_pawn)
        self.accept_exchange_position = True
        self.return_laser = True


    def is_hit(self, laser_shot,game_board, index_pawn, index_box):
        index_laser_shot = laser_shot.list_direction_possible.index(laser_shot.direction_laser)

        if self.direction_pawn == laser_shot.list_direction_possible[(index_laser_shot + 3) % 4]\
        or self.direction_pawn == laser_shot.list_direction_possible[(index_laser_shot) % 4]:

            if self.direction_pawn == 'south':
                if laser_shot.direction_laser == 'west':
                    index_box = index_box - 10
                    laser_shot.direction_laser = 'north'

                elif laser_shot.direction_laser == 'south':
                    index_box = index_box + 1
                    laser_shot.direction_laser = 'east'
```

La fonction « is_hit » :
Si la direction du pion est égale à la direction du laser opposé (exemple pion direction nord et laser direction sud) alors on peut avoir un reflet ou quand les directions sont les mêmes (exemple pion = nord et laser = nord) alors on peut faire un reflet (évidement ces reflets se font en fonction du nom et du positionnement que l'on a donné à nos images).
Si la direction du pion est sud alors soit le laser à une direction Ouest et dans ce cas il renvoie avec le même principe que le « Scarab » le laser en direction du Nord.
Sinon si la direction du laser est Sud donc le reflet envoie le laser vers l'Est.

Evidement il n'y a pas que le Sud mais pour le reste des directions c'est exactement le même principe que cela.

```
else:
    game_board.list_box[index_box].box_is_empty = True
    del game_board.list_pawn[index_pawn]
    return 0
return index_box
```

Sinon la pièce a reçu le laser d'un coté où elle ne peut pas renvoyer dans elle est supprimée.

Pour le « Sphinx » :

```
class Sphinx(Pawn):
    def __init__(self, name_pawn, owner_pawn, direction_pawn):
        Pawn.__init__(self, name_pawn, owner_pawn, direction_pawn)
        self.can_move = False
        self.can_shoot = True

    def is_hit(self, laser_shot,game_board, index_pawn, index_box):
        return index_box
```

La fonction « __init__ » est ici différente mais seulement à la fin de celle-ci car le Sphinx est un pion qui ne peut pas bouger d'où le « False » et c'est le seul pion qui peut tirer le laser.

La fonction « is_hit » le Sphinx ne peut pas mourir donc on retourne juste index de la box.

• Structure du laser :

Pour la structure du laser nous avons pris la même voie que pour les pions c'est-à-dire, nous avons utilisé une classe et des héritages pour celui-ci.

Nous avons donc une classe « Laser », on a créé dans celle-ci le tir automatique du laser dans les différentes orientations possibles.

Pour notre classe « Laser » nous avons donc deux fonctions :

```
class Laser:
    def __init__(self, direction_laser):
        self.direction_laser = direction_laser
        self.list_direction_possible = ["north", "east", "south", "west"]

    def automatic_shot(self, game_board, surface, index_shooter, next_index_box):
        box_X = int()
        box_Y = int()
        next_position = CELL_SIZE[0] + CELL_SPACING[0]
        if self.direction_laser == 'east':
            if game_board.list_box[next_index_box].box_is_empty:
                position_X  = game_board.list_pawn[index_shooter]._get_pos_X() + CELL_SIZE[0]
                position_Y = game_board.list_pawn[index_shooter]._get_pos_Y() + CELL_SIZE[0] // 2
                while game_board.list_box[next_index_box].box_is_empty:
                    Map.draw_map( game_board, surface, COLOR_LIST, False, 0, 0)
                    pygame.draw.line(surface, COLOR_LIST[1], (position_X, position_Y) , (position_X + next_position + CELL_SPACING[0], position_Y), 4)
                    next_index_box += 1
                    pygame.display.update()
                    if position_X > BOARD_TOPLEFT[0] + (CELL_SIZE[0] + CELL_SPACING[0]) * 8:
                        break

                    position_X += next_position
                    box_X = game_board.list_box[next_index_box]._get_pos_X()
                    box_Y = game_board.list_box[next_index_box]._get_pos_Y()

            else:
                position_X  = game_board.list_pawn[index_shooter]._get_pos_X() + CELL_SIZE[0]
                position_Y = game_board.list_pawn[index_shooter]._get_pos_Y() + CELL_SIZE[0] // 2
                pygame.draw.line(surface, COLOR_LIST[1], (position_X, position_Y) , (position_X, position_Y - CELL_SPACING[1]), 4)
                pygame.display.update()
                box_X = game_board.list_box[next_index_box]._get_pos_X()
                box_Y = game_board.list_box[next_index_box]._get_pos_Y()
```

La fonction « __init__ » :

Dans cette fonction, nous avons créé la direction du laser et ensuite une liste qui contient toutes les orientations possibles.

La fonction « automatic_shot » :

Pour cette fonction nous allons développer pour la direction du laser « East », évidement les autres directions sont structurées de la même façon.

Tout d'abord on initialise plusieurs paramètres, la box_X et box_Y qui nous donne l'emplacement d'une box en X et en Y et la position suivante qui est égale à la largeur d'une cellule plus l'espacement entre elle et une autre.

Ensuite si dans la liste box, on a le prochain index de la box qui est vide alors on continue.

On initialise la position en X et en Y et nous avons créé une boucle « while » qui nous dit que tant que quand on regarde dans la liste box et que le prochain index est égal à une case vide alors on continue d'effectuer ce qu'il y a dedans.

On va donc dans cette boucle créer le trait qui représente le laser et on implémente de 1 la prochaine position de la box afin que l'on puisse déplacer le laser. Ensuite on met à jour l'affichage.

Enfin si notre position en X (car on veut un déplacement vers l'Est) est inférieure au coté en haut à gauche du tableau de jeu plus la largeur d'une case et l'espacement entre celle-ci et une autre fois 8 (le nombre de ligne au total) alors on arrête la boucle. Ce « if » nous sert donc à arrêter le laser lorsqu'il sort du tableau de jeu.

Enfin on donne à la position de X la nouvelle position qui est appelé « next_position » et l'on initialise à nouveau box_X et box_Y par rapport à la liste des boxes et du prochain index.

Sinon on a un pion sur une case donc on ne va pas continuer le laser il s'arrête donc à cette position.

• Structure du mode Editeur :

Dans le mode éditeur nous devions pouvoir placer les pièces de la façon que l'on voulait pour pouvoir ensuite y jouer et enregistrer cette configuration ainsi que les suivantes pour jouer sur nos propres créations ultérieurement.

On va donc s'intéresser ici aux fonctions qui nous permettent de sauvegarder, de stocké les créations et de charger celle-ci.

La fonction « generate_file_custom_map » :

```
def generate_file_custom_map(self):

    with open('source/map/number_custom_map', 'r') as file:
        self.num_custom_map = file.read().splitlines()

    with open('source/map/custom_pawn_{}'.format(len(self.num_custom_map)), 'wb') as fichier:
        my_pickler = pickle.Pickler(fichier)
        for line_box in range(8):
            for item_box in range(10):
                x = BOARD_TOPLEFT_EDITOR[0] + item_box * (CELL_SIZE[0] + CELL_SPACING[0])
                y = BOARD_TOPLEFT_EDITOR[1] + line_box * (CELL_SIZE[1] + CELL_SPACING[1])
                for em in range(len(self.list_pawn)):
                    pawn_X = self.list_pawn[em]._get_pos_X()
                    pawn_Y = self.list_pawn[em]._get_pos_Y()
                    if pawn_X == x and pawn_Y == y:
                        my_pickler.dump((self.list_pawn[em]))

    with open('source/map/custom_box_{}'.format(len(self.num_custom_map)), 'wb') as file:
        my_pickler = pickle.Pickler(file)
        for line_box in range(8):
            for item_box in range(10):
                x = BOARD_TOPLEFT_EDITOR[0] + item_box * (CELL_SIZE[0] + CELL_SPACING[0])
                y = BOARD_TOPLEFT_EDITOR[1] + line_box * (CELL_SIZE[1] + CELL_SPACING[1])
                for el in range(len(self.list_box_editor)):
                    box_X = self.list_box_editor[el]._get_pos_X()
                    box_Y = self.list_box_editor[el]._get_pos_Y()
                    if box_X == x and box_Y == y:
                        my_pickler.dump((self.list_box_editor[el]))

    with open('source/map/number_custom_map', 'a') as file:
        file.write("Custom_map_{} {}\n".format(len(self.num_custom_map), len(self.list_pawn)))
        file.seek(0)
```

Pour la sauvegarde d'un fichier nous avons utilisé le module pickle qui permet donc d'effectuer cela en format binaire sur n'importe quel objet Python.

Dans notre premier « with » on ouvre tout d'abord le fichier « number_custom_map » qui contient le nombre de map disponible, une map étant une ligne. On sauvegarde ensuite dans « self.num_custom_map » le numéro qui est propre à la map que l'on veut.

Notre second « with » parcours tout d'abord toutes les cases de la map. Après cela on regarde su il y a un pion sur ces cases et s'il y a bien un pion alors on regarde les coordonnées de celui-ci et on donne le propriétaire du pion à notre « my_pickler ».

Le troisième « with » on utilise le même principe que précédemment mais l'on vérifie cette fois-ci tout les boxes du terrain et on récupère à la fin les propriétés de chacune d'entre elles.

Le derniere « with » nous sert à ajouter, tout d'abord, dans le fichier custum le numéro de la map ensuite le nombre de pion sur le terrain et enfin on utilise « /n » pour un retour à la ligne.
La ligne « file.seek(0) » nous est utile pour éviter les occurrences.