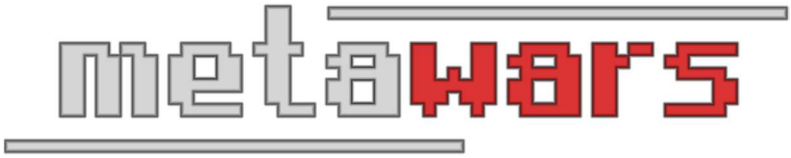
DUBOIS Julien
en collaboration avec
NOIRET Colin
NENY Gabriel

DOSSIER ISN

Informatique et Sciences du Numérique
Enseignement de spécialité



Encadrement pédagogique

M.LEVEQUE

Année scolaire 2018-2019 Lycée Mme de Staël – Montluçon

SOMMAIRE

1 – PRÉSENTATION :

Nature du projet :

Metawars est un jeu vu du dessus dans lequel le joueur doit rester en vie le plus longtemps possible face à des hordes d'ennemis. Il pourra s'aider de bonus très avantageux qui apparenteront de manière aléatoire sur la carte de jeu.

Pourquoi ce jeu ?

Créer un jeu était l'occasion d'en apprendre beaucoup sur la programmation, mais aussi sur l'organisation et le travail en équipe. Cela fut très intéressant de découvrir les différentes étapes de développement d'un jeu vidéo. Ce jeu nous permet d'aborder de façon simple des concepts de programmation complexes telle que la programmation orienté objet (POO).

Règles du jeu :

Dans **Metawars**, l'utilisateur incarne un personnage qu'il peut contrôler avec les touches du clavier (Z, Q, S, D) et la souris. Des zombies apparaissent sur la carte et se dirigent vers le joueur. Muni de ses deux pistolets le joueur peut viser les zombies en déplaçant la souris et leur tirer dessus en cliquant sur le bouton gauche de celle-ci. De plus des bonus apportant de multiples avantages apparaîtront aléatoirement sur la carte au cours de la partie.

Des graphismes !
Le jeu **Metawars** avait besoin d'une interface graphique dédiée ; nous ne pouvions pas jouer à ce jeu à partir de la console Python.
Après de mures réflexions quant au module que nous allions utiliser, notre choix s'est porté sur Pygame (celui-ci étant sans doute le compromis parfait entre simplicité et puissance).

2 – CAHIER DES CHARGES :

Afin de rester cohérent dans le développement du jeu et de ne jamais s'éloigner de notre objectif principal, nous avons tout d'abord défini les différentes contraintes que doit respecter notre jeu :

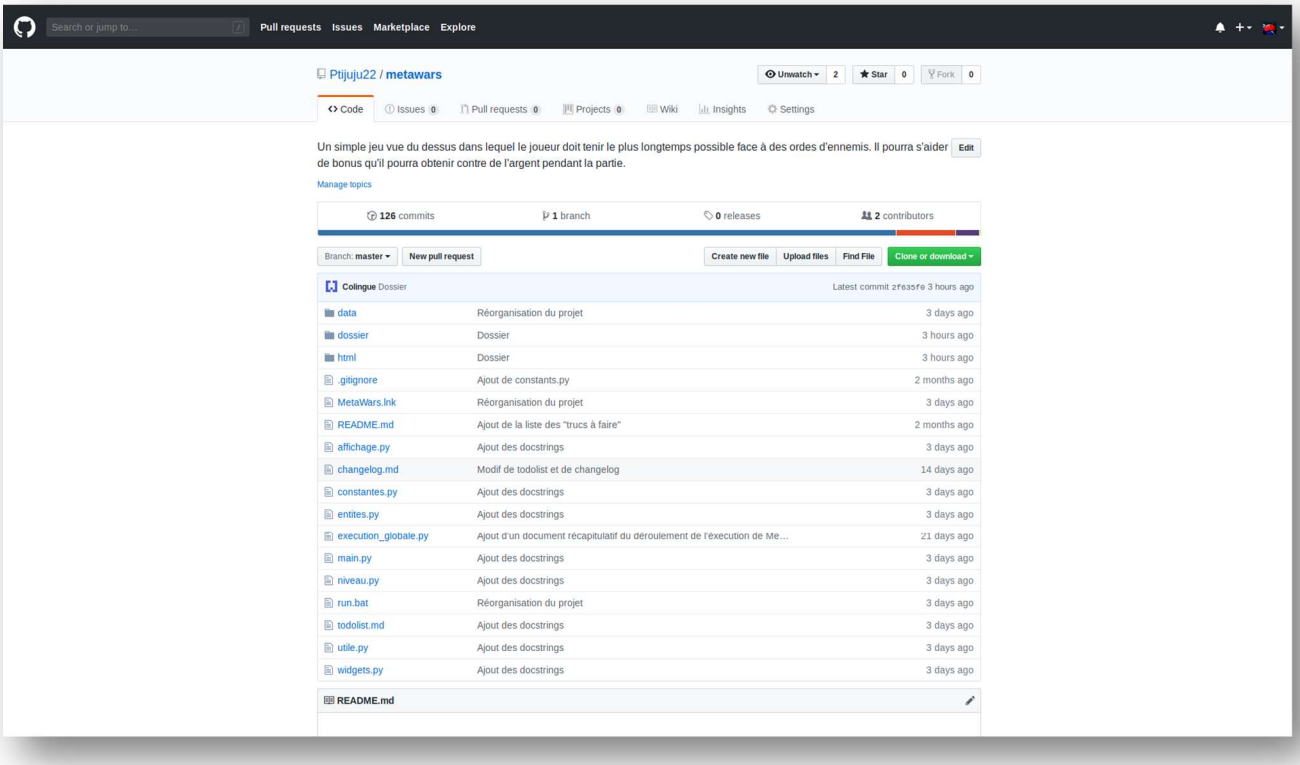
- Organiser le projet par classes
- Créer une interface graphique à l'aide de Pygame
- Gérer les entrées clavier et déplacements de la souris
- Contrôler le temps sans dépendre de la vitesse du CPU de la machine
- Ajouter de l'aléatoire (apparition des entités, tirs, ...)
- Optimiser l'accès aux ressources (ex : images)
- Dessiner des interfaces intuitives grâce aux Widgets

3 – DÉMARCHE COLLABORATIVE :

Les clés pour travailler en équipe de manière efficace sont l'organisation et la communication.

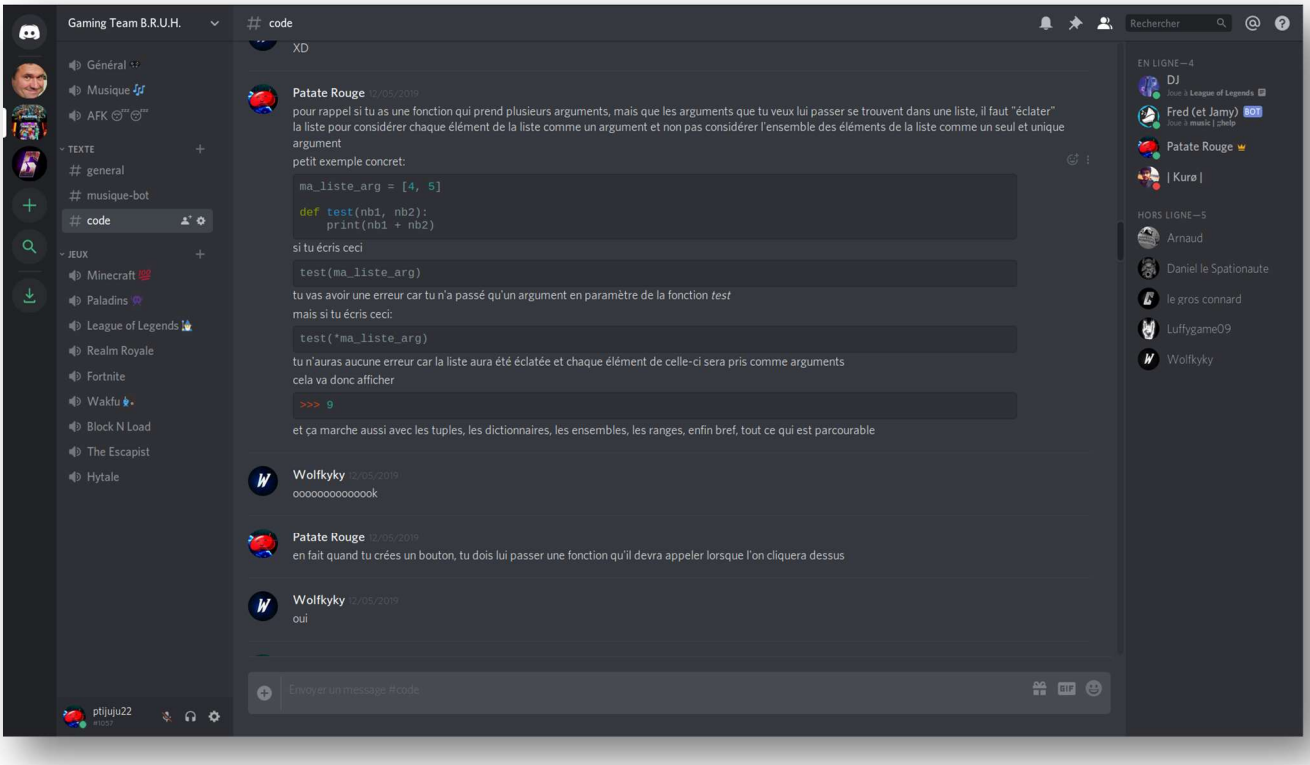
Organiser :

Avant de commencer à véritablement travailler sur notre projet, nous nous sommes donc tournés vers un *service web d'hébergement et de gestion de développement de logiciels* très célèbre, **Github**. Ce service permet à chacun des membres de l'équipe de modifier un projet centralisé et de prendre conscience des implémentations faites par les autres collaborateurs. Github est un service en ligne accessible grâce au logiciel en ligne de commande **Git**. De plus, tous les membres de l'équipe utilisent l'éditeur de code **Sublime Text 3**, qui intègre une extension (appelée **Sublime Merge)** ajoutant une interface pour gérer les projets git, ce qui facilite grandement son utilisation.



Communiquer :

Coté communication, nous utilisons le logiciel de chat texte et vocal **Discord**. Bien que destiné à la base aux joueurs, Discord permet l'intégration de code (notamment Python) embellie par une coloration syntaxique digne des plus grands éditeurs de code.



4 – RÉPARTITION DU TRAVAIL :

Afin d'être le plus efficace possible, il a fallu répartir les tâches. Cependant, afin que tous les membres du groupe comprennent l'entièreté du code et tous les concepts de programmation rencontrés, nous avons décidé de créer une *liste des trucs à faire* (fichier todolist.md à la racine du projet) listant toutes les fonctions et méthodes que nous envisagions d'ajouter, de modifier ou de supprimer. Chacun peut travailler sur une fonction précise, sans gêner les autres, en modifiant n'importe quelle partie du code. Notre code est donc très modulaire (réparti en 6 fonctions et 75 méthodes dans 10 classes différentes) ce qui permet, en plus de répartir plus efficacement le travail entre nous, de repérer les bugs plus rapidement en les isolants.
Pour ma part, de part mes connaissances déjà avancées du langage Python, j'ai surtout veillé à ce que mes deux *collègues* comprennent l'intégralité du programme tout en les guidant afin de toujours garder un code clair et organisé. J'ai développé les grandes lignes du jeu, son architecture et certaines fonctions un peu complexes que je me suis bien sûr empressé d'expliquer à mes camarades.

5 – FONCTIONNEMENT :

Le code du jeu est divisé en plusieurs fichiers de code Python :

main.py est le script principal du projet. C'est lui qu'il faut lancer pour faire fonctionner le jeu. Les autres fichiers n'abritent que des classes et ne font rien s'ils sont exécutés seuls. Ce fichier est composé de 2 fonctions : *lancer_jeu()* (fonction principale du jeu) et *lancer_partie()* (permet de créer un terrain jouable sur l'affichage donné).

affichage.py propose une classe dont le rôle est de créer et modifier la fenêtre pour dessiner le niveau de jeu et l'interface. Cette classe se charge également de charger et de distribuer les images aux niveau et entités (voir plus bas). C'est aussi ici que l'on va gérer les événements comme le déplacement du personnage.

entites.py contient 5 classes représentant ce que l'on appelle des entités. Une entité est un objet défini par sa position, sa taille, un angle de rotation et une vitesse. Les entités sont des objets indépendants qui peuvent se déplacer sur la carte de jeu. Le joueur, les ennemis, les bonus ou encore les tirs sont des entités. Ils héritent tous d'une classe *abstraite* de base : Entite. Cette dernière défini le comportement de base de toutes les entités du jeu.

niveau.py définit une classe représentant le niveau de jeu. Ce dernier stock les métadonnées de la partie en cours comme le nombre de pièces amassées, le temps écoulé ou encore l'image de fond de la carte de jeu. Il centralise également l'ensemble des entités de la partie dans une liste, ce qui permet de toutes les actualiser simplement grâce à ***Niveau.****actualise()*.

widgets.py regroupe des classes permettant de créer les éléments graphiques de l'interface de jeu (texte, bouton, image, …). Contrairement à Tkinter, Pygame ne propose aucun widget de manière native. Il a donc fallu créer de *faux* boutons et textes afin d'afficher certaines informations essentielles au jeu dans la fenêtre. Un bouton par exemple affiche simplement une image différente en fonction de la position de la souris et de l'état des boutons pour créer un effet d'enfoncement lors du clic sur celui-ci.

utile.py propose des fonctions utilitaires qui ne dépendent d'aucune classe. Par exemple, on peut y trouver une fonction *arreter()* qui permet d'arrêter le programme à n'importe quel endroit du code.

constantes.py stock toutes les données numériques et chaînes de caractères qui ne changent pas en cours de partie. Utiliser des constantes permet de centraliser ces données dans un seul fichier. Ainsi, modifier une valeur dans ce fichier change le comportement de tous les objets qui utilise la constante associée. Cela permet d'éviter les oublis en cas de modification d'une constante utilisée à plusieurs endroits dans le code.

Notre programme a également été créé de façon à s'adapter à un maximum de situations :

Il fonctionne même s'il manque des images : L'architecture de notre programme est faite de façon à ce que chaque objet aillant besoin d'une image passe par la méthode ***Affichage.****obtenir_image()*. Cette méthode vérifie alors que l'image demandée existe et a bel et bien été chargée puis la renvoie. Dans le cas contraire, elle crée une image noire de remplacement qu'elle retourne à la place de l'image demandée. Cela permet au jeu de continuer à fonctionner même avec des images manquantes.

Il fonctionne sur Windows, Linux et MacOS : Les systèmes d'exploitation Linux et MacOS ne représentent pas les chemins de fichier de la même façon que Windows. En effet, alors que Windows utilise des antislashs « \ » pour séparer les dossiers des sous-dossiers et fichiers, Linux et MacOS se servent de simple slashs « / ». En utilisant la fonction ***os.****path.join()*, nous permettons à notre programme de s'adapter aux différents systèmes (Gabriel et Colin utilisant Windows, et moi Linux, nous n'avions pas vraiment le choix).

Il s'adapte à la vitesse du processeur : En fonction du PC sur lequel s'exécute le programme ainsi que des différentes tâches que le processeur doit gérer, il se peut que ce dernier accélère ou ralentisse. Pour palier à ce problème, nous avons décidé de calculer en permanence le temps écoulé entre chaque boucle de jeu afin d'augmenter ou de diminuer les déplacements et rotations des personnages du jeu en conséquence. Ainsi, les mouvements des personnages restent uniformes même en cas de ralentissement du processeur.

Il ne surcharge pas la mémoire : Python étant par nature très gourmand en ressources, il a fallu faire un maximum d'économies. Pour ce faire, certaines entités telles que les tirs ou les bonus sont supprimées après un certain temps. De plus, les images ne sont chargées qu'une seule fois, lors du démarrage du jeu, car l'accès au disque dur est

très lent et ralentit considérablement le jeu. Une simple copie d'une image chargée permet d'en récupérer rapidement une nouvelle sans avoir à recharger celle-ci une nouvelle fois depuis le disque dur.

6 – DOCUMENTATION :

Pour ce qui est de la documentation, mes connaissances seules ne suffisaient pas. Nous nous sommes bien évidemment servis de la documentation officielle du langage Python https://org.python.org/fr/3 et de Pygame https://www.pygame.org/docs/. On a également navigué sur des forums tel que celui d'Open Classroom https://openclassrooms.com/forum/ et le très célèbre Stack Overflow https://stackoverflow.com/. Ce n'est pas grand-chose mais suffisant pour arriver à bout de notre projet.

7 – PROLONGEMENT POSSIBLE :

Comme beaucoup, nous avons sûrement eu les yeux plus gros que le ventre. En effet, lors de la réalisation du cahier des charges, nous avions décidé d'ajouter les fonctionnalités ci-dessous :

- Mode multijoueur en ligne (module **socket**)
- Cartes de jeu personnalisables (notamment avec des obstacles)
- Système de transaction pour obtenir des bonus
- Un système audio (pygame permet de le faire)

Peut-être que nous continuerons à développer ce projet après le bac pour implémenter ces fonctionnalités…

8 – BILAN PERSONNEL :

Assister à la création d'un jeu de A à Z est toujours quelque chose de très excitant. On s'organise, on se documente, on partage beaucoup et on apprend des autres. Même avec un niveau avancé, on continue à en apprendre tous les jours notamment en ce qui concerne la gestion d'un projet **en équipe**. Travailler seul, ça n'a rien à voir : On a une idée que l'on suit du début à la fin. En équipe, les choses sont beaucoup plus dynamiques, chacun apporte ses idées, des idées qui peuvent être très différentes des nôtres mais qui restent constructives et font la force des projets à plusieurs.

Pour moi, c'était un peu la première fois que je travaillais en groupe sur un projet en langage Python. J'ai pris beaucoup plus de plaisir à travailler ainsi et j'espère pouvoir continuer à programmer en équipe comme ici, pourquoi pas avec les mêmes personnes.

9 – ANNEXE :

Le code source de notre projet est visible à l'adresse suivante : https://github.com/Ptijuju22/metawars, mais voici quelques fonctions et méthodes au fonctionnement interressant.

Le jeu démarre grâce à cette fonction :

```
16  def lancer_jeu():
17      """ Fonction principale du jeu (à ne lancer qu'une seule fois) """
18
19      # on creer un nouvel "affichage" (fenetre)
20      affichage = Affichage()
21      # on charge l'ensemble des images du jeu
22      affichage.charge_images()
23      # on crée les widgets du niveau
24      affichage.creer_widgets_menu(lancer_partie)
25
26      # on creer un niveau de jeu
27      niveau_menu = Niveau(affichage)
28      # le niveau et les entités recupères les images dont elles ont besoin
29      niveau_menu.charge_image()
30
31      # on fait avancer le joueur pour éviter qu'il soit immobile
32      niveau_menu.joueur.droite()
33
34      # cette variable retient le temps (en seconde) du dernier tick de jeu
35      temps_precedent = time.time()
36
37      while True:
38          # cette variable stocke le temps écoulé depuis le dernier tick de jeu
39          temps_ecoule = time.time() - temps_precedent
40          temps_precedent = time.time()
41
42          # on gère les evenements utilisateurs (clic, appui sur une touche, etc)
43          affichage.actualise_evenements(niveau_menu, False)
44          # on actualise le niveau et les entités qu'il contient
45          niveau_menu.actualise(temps_ecoule)
46          # pour éviter que le joueur ne meurt, on reinitialise sa vie en permanence
47          niveau_menu.joueur.vie = constantes.VIE_JOUEUR
48          # on redessine la fenetre pour afficher de nouveau le niveau
49          affichage.actualise(niveau_menu, False)
```

Cette fonction commence par créer un nouvel affichage. Elle fait en sorte que l'affichage charge les images du jeu puis crée l'interface du menu (bouton « Jouer », bouton « Quitter », image de titre). Le niveau affiché en fond du menu est ensuite créé.

Pour rendre le menu, un peu plus dynamique, on fait bouger le joueur vers la droite indéfiniment. Puis, une boucle infinie se lance et réactualise en permanence le niveau et l'affichage en calculant à chaque fois le temps écoulé depuis le dernier tour de boucle. On fait également en sorte que la vie du joueur reste toujours au maximum afin qu'il ne meure pas et continue à animer le menu du jeu.

Les ennemis et les bonus apparaissent grâce à celle-ci :

```
50      def fait_apparaitre(self, temps):
51          """ Fait parfois apparaitre un bonus et/ou un ennemi.
52
53              <temps> (float): Le temps écoulé depuis la dernière actualisation. """
54
55          # on pioche un nombre aléatoire
56          nb = random.random()
57
58          # si le nombre pioché est inférieur au temps écoulé divisé
59          # par la fréquence moyenne d'apparition...
60          if nb <= temps / constantes.FREQUENCE_APPARITION_ENNEMI:
61              # on crée un ennemi
62              self.cree_ennemi()
63
64          # pareil pour les bonus
65          nb = random.random()
66
67          if nb <= temps / constantes.FREQUENCE_APPARITION_BONUS:
68              self.cree_bonus()
```

Cette méthode appartenant à la classe **Niveau** sélectionné à chaque fois un nombre aléatoire compris entre 0 et 1. Si ce nombre est inférieur à au temps écoulé depuis le dernier tour de boucle divisé par la fréquence d'apparition alors un nouvel ennemi (ou nouveau bonus) est créé.

L'affichage est actualisé par cette méthode :

```python
    def actualise(self, niveau, en_partie):
        """ Efface la fenêtre, redessine le terrain, les entités, puis les widgets en les actualisant.

            <niveau> (niveau.Niveau): Le niveau à afficher
            <en_partie> (bool): Si True, actualise les textes affichant les stats du joueur """

        # On rend tous les pixels de la fenetre blanc
        self.fenetre.fill((255, 255, 255))

        # on affiche le fond du niveau
        self.affiche_carte(niveau)

        # on affiche les entités (dont le joueur)
        self.affiche_entite(niveau.joueur)

        for entite in niveau.entites:
            self.affiche_entite(entite)

        # si on est en partie, on acalise le score
        if en_partie:
            # on actualise le score en fonction de celui du niveau
            self.actualise_scores(niveau)

        # on redessine les widgets
        self.affiche_widgets()

        # On actualise l'écran
        pygame.display.update()
```

On commence tout d'abord par remplir la fenêtre par du blanc puis on affiche le fond du niveau. Ensuite on dessine le joueur et les autres entités du niveau. Si on est en partie, on actualise le score (car le menu principal ne dispose pas de score). On redessine les widgets (textes, boutons et images) et enfin on applique les changements sur la fenêtre à l'aide de ***pygame.****display.update()*.

<u>Les évènements clavier et souris sont interceptés ici :</u>

```python
    def actualise_evenements(self, niveau, en_partie):
        """ Lit les évenements du clavier et de la souris et exécute les fonctions associées
            à certaines touches (ex: Appui sur la touche Z -> le joueur monte).

            <niveau> (niveau.Niveau): Le niveau à actualiser en fonction des actions utilisateur.
            <en_partie> (bool): si True, fait bouger et tirer le joueur, sinon le joueur ne réagit pas. """

        # on parcourt l'ensemble des evenements utilisateurs (clic, appui sur une touche, etc)
        for evenement in pygame.event.get():

            # si l'utilisateur a cliqué sur la croix rouge de la fenêtre
            # on arrete le jeu
            if evenement.type == pygame.QUIT:
                utile.arreter()

            # sinon si on est en partie (et pas dans le menu principal)
            elif en_partie:
                # si l'utilisateur appui sur une touche du clavier...
                if evenement.type == pygame.KEYDOWN:
                    # en fonction de la touche appuyée, on appelle la fonction
                    # commandant le déplacement correspondant
                    if evenement.key == pygame.K_w:
                        niveau.joueur.haut()

                    if evenement.key == pygame.K_s:
                        niveau.joueur.bas()

                    if evenement.key == pygame.K_a:
                        niveau.joueur.gauche()

                    if evenement.key == pygame.K_d:
                        niveau.joueur.droite()

                    if evenement.key == pygame.K_ESCAPE:
                        utile.arreter()

                # si l'utilisateur relache une touche du clavier...
                elif evenement.type == pygame.KEYUP:
                    # en fonction de la touche appuyée, on appelle la fonction
```

Pygame place automatiquement les évènements qui n'ont pas encore été gérés dans une liste que l'on peut parcourir à l'aide d'une simple boucle **for**. Chaque évènement possède un attribut *type* qui nous permet de savoir s'il s'agit d'un appui sur une touche, d'une touche relâchée, d'un clic de souris, etc… La méthode exécutée dépend de l'évènement reçu. On peut cependant noter que le joueur « répond » aux évènements uniquement si on est en partie (il faut éviter que l'utilisateur fasse bouger le joueur du menu principal).

<u>Le chargement des images par les entités :</u>

```python
    def __charge_image__(self, chemin_image):
        """ Méthode interne utilisée pour simplifier le chargement et le
            redimensionnement d'une image donnée.

            <chemin_image> (str): Le chemin de l'image. """

        affichage = self.niveau.affichage
        taille_pixel_x = int(self.taille[0] * constantes.ZOOM)
        taille_pixel_y = int(self.taille[1] * constantes.ZOOM)

        self.image = affichage.obtenir_image(chemin_image)
        self.image = pygame.transform.scale(self.image, (taille_pixel_x, taille_pixel_y))

    def charge_image(self):
        """ Appelé pour charger l'image de l'entité. Ne fait rien par défaut. """
        pass
```

Les entités ont besoin d'images pour être affichée sur la fenêtre. Lorsqu'une entité est créée, elle charge les images dont elle a besoin en utilisant la méthode héritée de la classe *Entite*, *__charge_image__*. Cette méthode récupère une copie d'une image chargée par l'affichage grâce à ***Affichage.****obtenir_image()* puis la redimensionne à la taille voulue. Cette est généralement utilisée par *charge_image()* et ne doit pas être redéfinie par les autres entités. Ces dernières ne redéfinissent que la méthode *charge_image()*.

Gestion des collisions :

```python
    def collisionne(self, entite):
        """ Renvoie True si l'entité collisionne avec celle donnée, sinon False.

            <entite> (entites.Entite): L'entité à tester. """

        # on renvoie True si une entité déborde sur une autre
        if self.position[0] + self.taille[0] / 2 > entite.position[0] - entite.taille[0] / 2:
            if self.position[0] - self.taille[0] / 2 < entite.position[0] + entite.taille[0] / 2:
                if self.position[1] + self.taille[1] / 2 > entite.position[1] - entite.taille[1] / 2:
                    if self.position[1] - self.taille[1] / 2 < entite.position[1] + entite.taille[1] / 2:
                        return True
        return False
```

Cette méthode renvoie **True** si l'entité entre en collision avec *entite*. Sinon, elle renvoie **False**. On considère que les entités sont des rectangles centrés sur leur position (d'où le ***self.****taille[x] / 2*).

Un peu de trigonométrie dans les déplacements :

```python
    def bouge(self, temps):
        """ Change la position de l'entité en fonction de sa vitesse et son angle de rotation.

            <temps> (float): Le temps écoulé en seconde depuis la dernière actualisation. """

        # Un peu de trigonométrie...
        self.position[0] += self.vitesse * math.cos(self.angle) * temps
        # on soustrait la position car les coordonées de l'écran en pixels sont inversées
        # elles vont de haut en bas au lieu d'aller de bas en haut
        self.position[1] -= self.vitesse * math.sin(self.angle) * temps
```

Les entités bougent grâce à cette méthode présente dans la classe *Entite*. On utilise ici des fonctions trigonométriques afin de convertir l'angle de rotation de l'entité en coordonnées comprises entre -1 et 1. Il suffit ensuite de multiplier la valeur obtenue par le temps écoulé depuis le dernier tour de boucle et par la vitesse de l'entité pour obtenir la distance parcourue par celle-ci en *temps* secondes.

Et dans la rotation du joueur :

```python
    def regarde_position(self, dx, dy):
        """ Tourne le joueur de façon à ce qu'il regarde en direction de (dx, dy).

            <dx> (float): La distance horizontale entre le point à regarder et le joueur (positif
                si le point se trouve à droite du joueur, sinon négatif).
            <dy> (float): La distance verticale entre le point à regarder et le joueur (positif
                si le point se trouve en bas du joueur, sinon négatif). """

        # on calcule la distance entre le joueur et cette position à l'aide de Pythagore
        d = math.sqrt(dx ** 2 + dy ** 2)

        # si la souris est à une distance de 0 du joueur, on ne peut pas définir d'angle
        if d != 0:
            # on détermine un angle possible à l'aide de Arc cosinus
            angle = math.acos(dx / d)

            # on détermine si on doit prendre la valeur négative ou positive de cette angle
            if dy >= 0:
                self.angle = -angle
            else:
                self.angle = angle
```

A l'inverse, à partir de la distance (x, y) entre un point (le curseur de la souris par exemple) et le joueur, il est possible de déterminer l'angle que ce dernier doit avoir afin qu'il « regarde » en direction de ce même point. On a également fait en sorte de vérifier que la distance entre ce point et le joueur ne soit pas nulle, ce qui provoquerait une erreur.

Les ennemis s'arrêtent lorsqu'ils s'approchent trop près du joueur :

```
345     def est_trop_pret(self):
346         """ Renvoie True si l'ennemi est trop prêt du joueur, sinon False. """
347 
348         dx = self.position[0] - self.niveau.joueur.position[0]
349         dy = self.position[1] - self.niveau.joueur.position[1]
350         d = math.sqrt(dx ** 2 + dy ** 2)
351 
352         if d <= constantes.ZONE_AUTOUR_JOUEUR:
353             return True
354         else:
355             return False
```

Simplement à l'aide du théorème de Pythagore, on calcule la distance entre le joueur et l'ennemi en question puis on renvoie **True** si cette distance est inférieure ou égale à la distance minimale entre un ennemi et le joueur. Sinon on renvoie **False**.

Les tirs ne peuvent toucher que des ennemis du tireur :

```
471         # si le tireur est un joueur
472         if type(self.tireur) == Joueur:
473             # on teste pour toutes les entités du niveau
474             for entite in self.niveau.entites:
475                 # si le tir touche une entité
476                 if self.collisionne(entite):
477                     # et si est un ennemi (et pas un bonus ou encore un autre tir)
478                     if type(entite) == Ennemi:
479                         # alors on appelle la méthode touche()
480                         self.touche(entite)
481         else:
482             # sinon, c'est que le tir vient d'un ennemi
483             # si le tir touche le joueur
484             if self.collisionne(self.niveau.joueur):
485                 # alors on appelle la méthode touche()
486                 self.touche(self.niveau.joueur)
```

Si le tireur est un joueur, le tir vérifie s'il entre en collision avec un ou plusieurs ennemis. Si c'est le cas, il appelle la méthode *touche()*. Si le tireur est un ennemi, il vérifie simplement s'il touche le joueur ou non.

Les boutons changent d'état en fonction de la position de la souris :

```python
    def actualise_evenement(self, evenement):
        """ Détecte le survol et le clic de la souris pour changer l'état du bouton et
            exécuter l'action définie.

            <evenement> (pygame.event.Event): L'évènement à tester. """

        x, y = self.obtenir_position_reelle()
        w, h = self.taille

        # si on repère le clic de la souris
        if evenement.type == pygame.MOUSEBUTTONDOWN:
            # si le clic de la souris est faite sur le widget (et pas en dehors)
            if self.est_dans_widget(evenement.pos):
                # en fonction du bouton cliqué, on change l'état du bouton
                if evenement.button == 1:
                    self.etat = "clic_gauche"
                elif evenement.button == 2:
                    self.etat = "clic_central"
                elif evenement.button == 3:
                    self.etat = "clic_droit"

        # si le bouton de la souris se relève
        elif evenement.type == pygame.MOUSEBUTTONUP:
            # si la souris se trouve sur le bouton, on le met en état de 'survol'
            if self.est_dans_widget(evenement.pos):
                # si le bouton relaché est le clic gauche, on execute l'action associée au bouton
                if evenement.button == 1:
                    self.action(*self.arguments_action)

                self.etat = "survol"
            else:
                # sinon, on le remet dans l'état 'normal'
                self.etat = "normal"

        # si la souris se déplace, on regarde si elle survole le bouton
        elif evenement.type == pygame.MOUSEMOTION:
            # si la souris se trouve sur le bouton, on le met en état de 'survol'
            if self.est_dans_widget(evenement.pos):
```

Cette méthode change l'état du bouton afin d'afficher une image adaptée en fonction de la position de la souris et des boutons enfoncés. De plus, si c'est le clic gauche qui est enfoncé, on exécute la fonction associée au bouton.

<u>Des fonctions très utiles :</u>

```python
def radian_en_degres(angle):
    """ Convertit un angle en radian, en degré.

        <angle> (float): L'angle en radian à convertir. """
    return angle * 180 / math.pi


def degres_en_radian(angle):
    """ Convertit un angle en degré, en radian.

        <angle> (float): L'angle en degré à convertir. """
    return angle * math.pi / 180
```

Le fichier *utile.py* contient des fonctions utilitaires qui ne dépendent d'aucune classe. C'est le cas de celles ci-dessus qui permettent comme leur nom l'indique de convertir des angles en degrés en radians et inversement. Pygame fonctionne avec des degrés mais le reste de notre programme utilise des radians (car plus simple à traiter). De telles fonctions permettent d'éviter de stupides erreurs de calcul.