Space Game

Objectif :

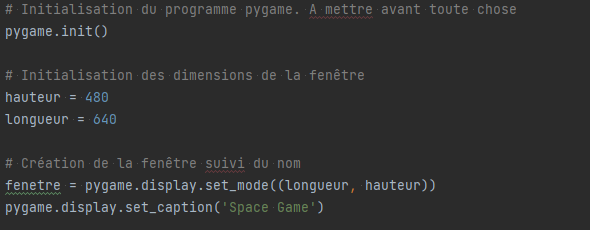
Créer un jeu où l’on contrôle un vaisseau qui évite des astéroïdes jusqu’à atteindre une plateforme d’atterrissage.

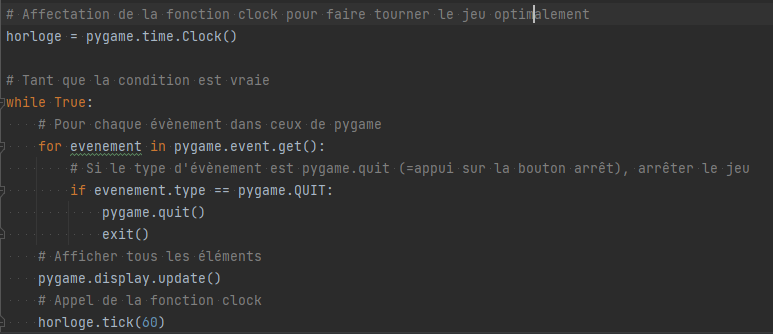
Compétences :

* Savoir créer une fenêtre
* Être capable de créer une classe Joueur, une classe Asteroids suivi de leurs attributs (OOP)
* Créer un sol
* Mettre en place un score par temps
* Créer les conditions de réussites

1. Créer une fenêtre







1. Afficher le background

Ouvrir l’éditeur d’images GIMP :

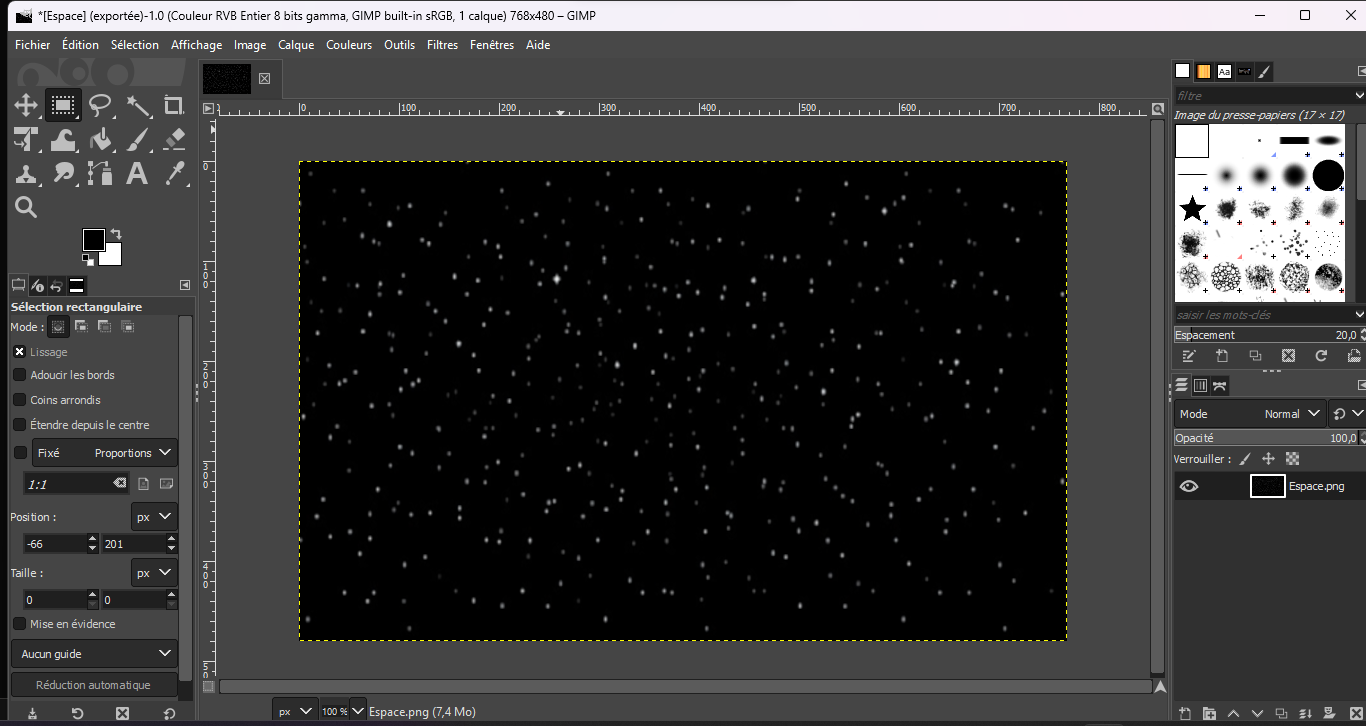
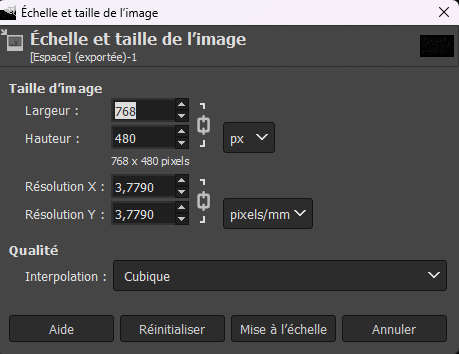
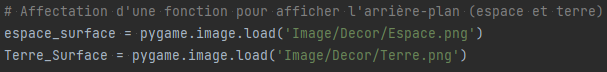
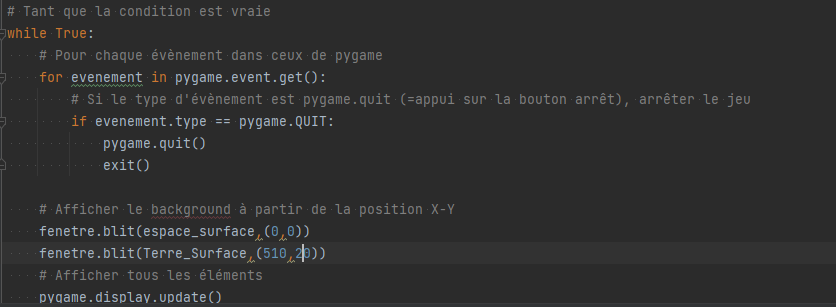
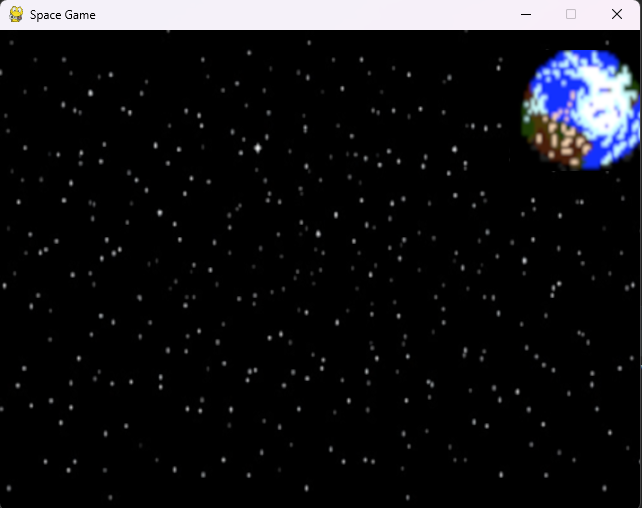


Image > Echelle et Taille de l’Image



Revenir sur PyCharm et écrire ces lignes afin insérer les surfaces régulières nécessaires pour afficher l’arrière-plan (ici l’espace et la terre)

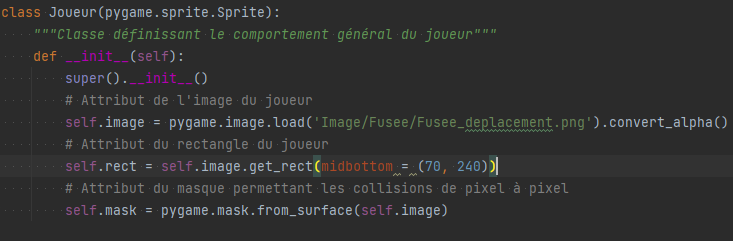
Voici le résultat :



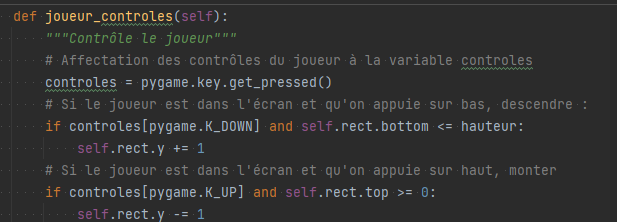
1. Créer les classes Joueurs et Obstacles suivi de leurs méthodes

L’utilité de l’OOP est de simplifier le code et le rendre plus lisible.

Dans un premier temps, après la création de la classe joueur, créer la méthode \_\_init\_\_. Elle regroupe tous les attributs (caractéristiques) du joueur : l’image du joueur sur l’écran, le rectangle permettant de le positionner par le milieu et masque pour les collisions de pixel à pixel. Super().\_\_init\_\_() aide Python à faire des connexions entre les classes enfants et parents.

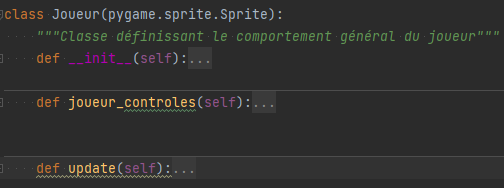


La méthode joueur\_controles permet de créer les conditions nécessaires pour déplacer le joueur : il ne doit pas sortir de l’écran. Dans notre jeu, il ne peut se déplacer que de haut en bas.



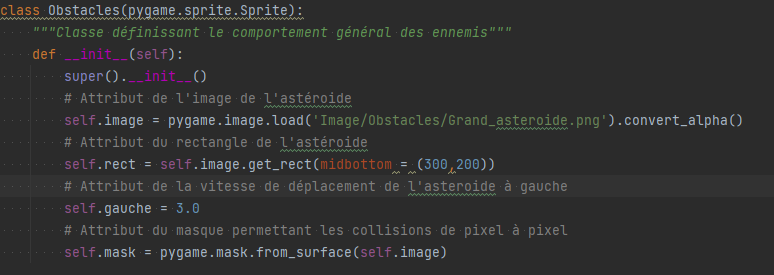
La méthode update permet de grouper toutes les methodes en une seule pour des raisons de lisibilité (éviter d’utiliser trop d’appel de méthodes). A mettre après toutes les méthodes.

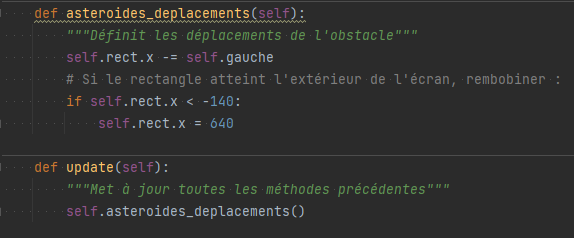


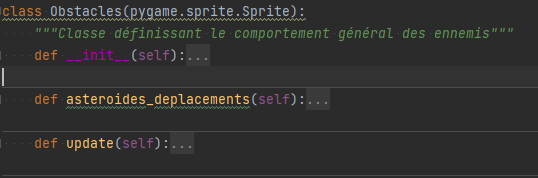


*Structure globale de la classe Joueur*

L’image a la même structure globale : une méthode résumant ses caractéristiques, des méthodes qui définissent son comportement propre et une méthode permettant d’appeler ces dernières.

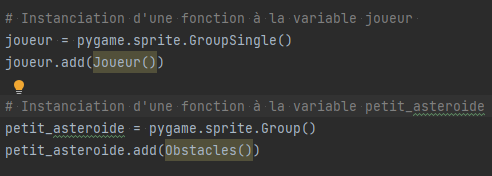




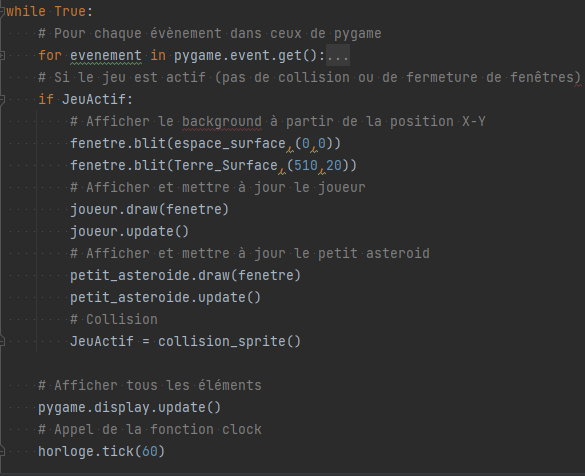


*Structure globale de la classe obstacle*

Une fois ceci fait, affecter les fonctions à la variable joueur. Pygame.sprite.GroupSingle() n’affecte qu’un sprite (celui du joueur) tandis que Pygame.spriteGroup() affecte plusieurs spirte (ceux des ennemis)



Dans la boucle True, appeler les fonctions.

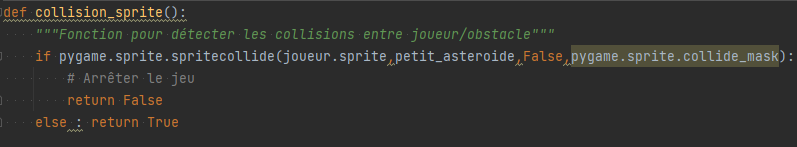


1. Gestion de la collision

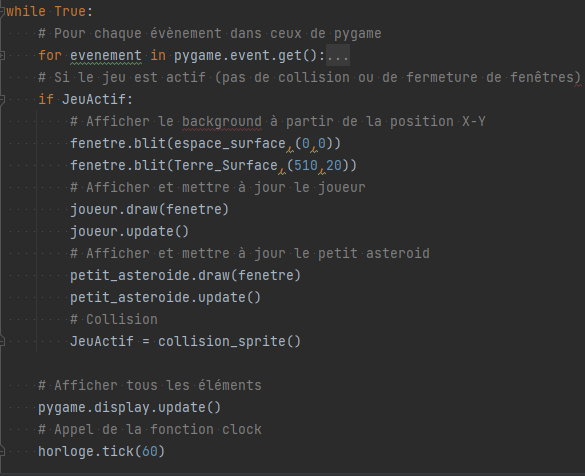
Avant d’écrire la fonction permettant la gestion de la collision, créer la variable permettant de mettre en arrêt et en avant le jeu. Pour ça, nous utiliserons les variables booléennes.



La fonction pygame.sprite.spritecollide() retourne tous les sprites qui ont des collisions. Elle a ici 4 arguments : le sprite du joueur, la classe petit\_asteroide, la valeur booléenne False et pygame.sprite.collide\_mask permettant qu’il y ait une collision masque par masque et non par rectangle. S’il y a une détection de collision, la valeur False sera retourné à une variable.

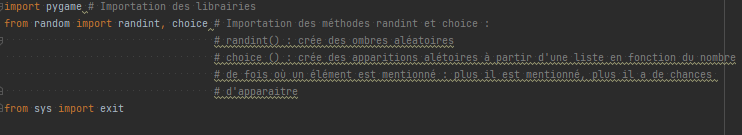


La condition if JeuActif: permet d’afficher le jeu dans des conditions normales quand il fonctionne. Le jeu doit s’arrêter de fonctionner quand il y a collision. Affecter à la variable JeuActif la fonction collision\_sprite, qui aura la valeur False, arrêtant le jeu.



Pour la suite :

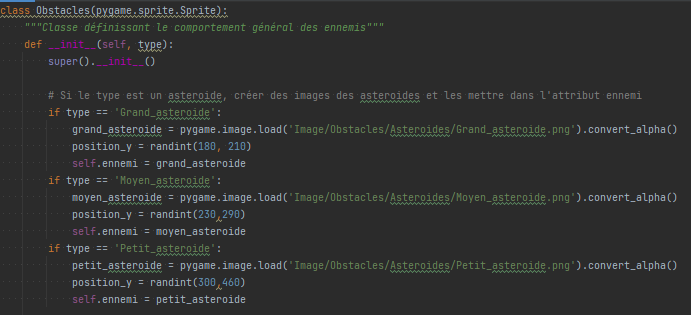
1. Création de plusieurs types d’obstacles



Ajouter les méthodes randint et choice. Randint() nous sera utile pour générer les coordonnées X et Y des obstacles. Choice() quant à lui servira à générer la fréquence d’apparitions des asteroides.

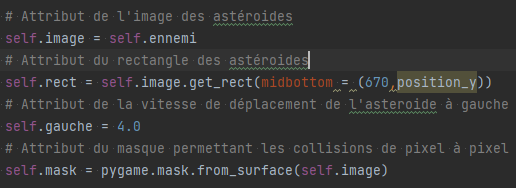
Il faut procéder à des modifications dans la classe Obstacles. Ajouter le paramètre type dans la méthode \_\_init\_\_()

Pour afficher plusieurs asteroides en des endroits différents, il faut créer différentes conditions if. Si le type matche avec le nom de l’image par exemple grand\_asteroide, alors il faut affecter l’image du grand asteroide à une variable (ici grand\_asteroide). Elle sera ensuite affecté à l’attribut ennemi. Une position\_y sera associé uniquement à cet asteroide.

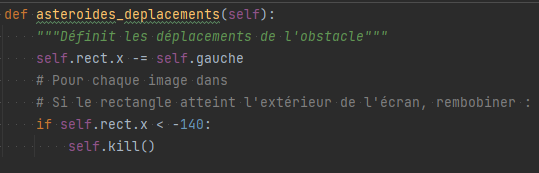


Une fois cela fait, affecter self.ennemi à la variable self.image. Dans self.rect, choisir comme coordonnées horizontales 670 afin que l’image soit le plus à droite de l’image possible et comme coordonnées verticales, la position y des différents asteroides.

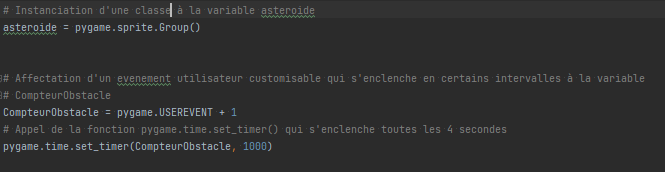
La vitesse a ici été augmenté afin de rendre le jeu moins facile et redondant.



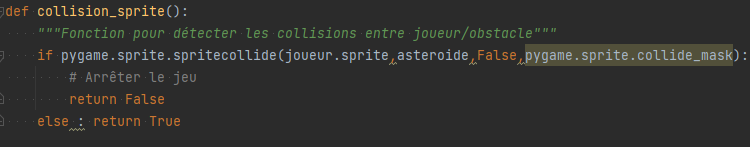
Dans cette fonction, la méthode kill() efface complètement l’obstacle quand il est un peu trop à droite.



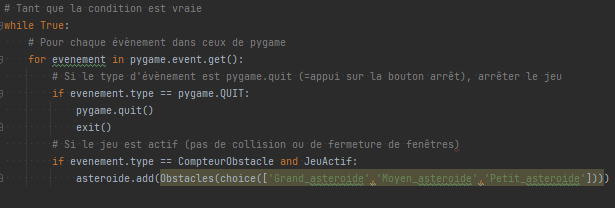
Dans la liste des variables dites globales, le nom du groupe d’obstacles a été changé en asteroide à la place de petit\_asteroid pour éviter toute confusion. La variable CompteurObstacle permet d’avoir un évènement qui s’enclenche pendant une durée définie par nous-mêmes (ici 1 seconde).



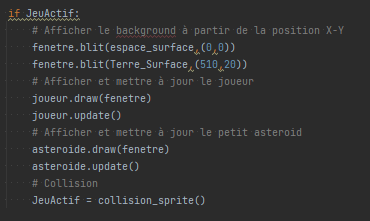
Comme nous avons modifié le nom de la variable instanciant la classe plus haut, nous devons le faire pour collision\_sprite()



La fonction asteroid.add() permet d’ajouter les obstacles suivant un ordre d’apparition décidé par la méthode choice()



Comme plus haut, le nom de la variable instanciée a été changé.



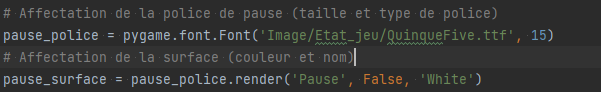
Bilan :



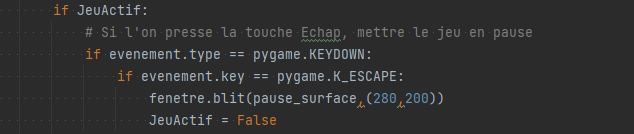
1. Conditions de pause

Pour créer les poses, faut déjà initialiser les variables qui grouperont les caractéristiques du texte (police, taille). Ensuite, il faudra crée la surface du texte, nécessaire pour l’affichage sur la fenêtre.

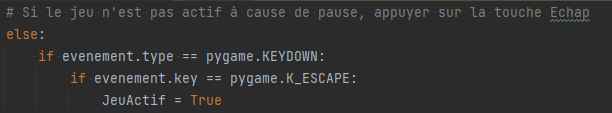
QuinqueFive.ttf est un fichier regroupant un type précis de police disponible dans un dossier préalablement crée.



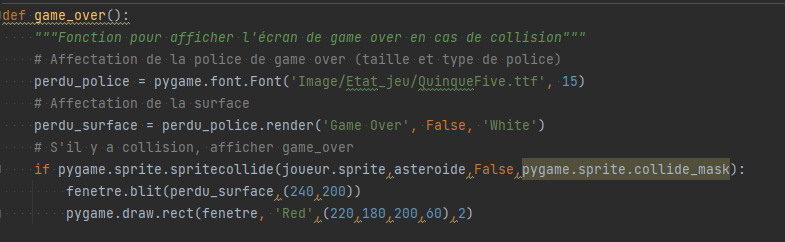
Une fois ceci fait, il faut créer les conditions de pause. Si l’on appuie sur la touche Echap, il faudra que le texte soit affiché à la position 280 en ordonnée et 200 en abscisse.



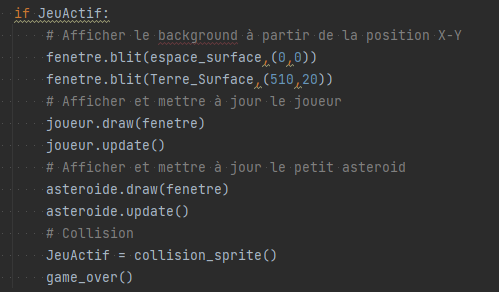
Si le jeu n’est plus actif, alors retaper sur Echap doit nous permettre de rendre le jeu actif.



1. Conditions de Game over

Si la collision a lieu, afficher un message en rouge . 

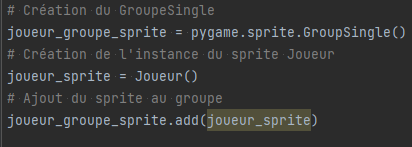
La fonction doit être ensuite appelé dans la boucle while, tant que le jeu est actif.



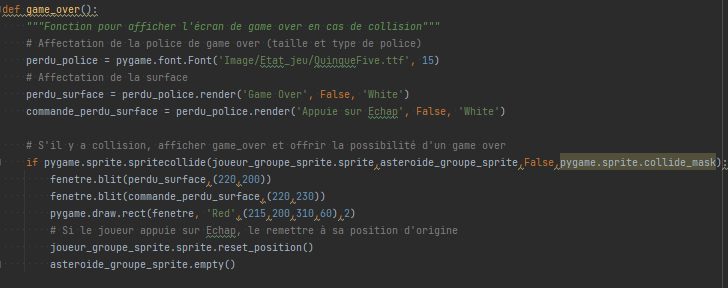
A suivre

* Finir la condition game over
* Appuyer sur R pour démarrer le niveau
* Afficher score
* Nouveau niveau :
* Créer différentes trajectoires (elliptiques et déplacement vers le haut)
* Ajouter de nouveaux obstacles

Pour appeler une méthode à partir d’une classe qui n’a qu’un sprite unique, il faut créer le GroupeSingle. Ensuite créer l’instance qui appelle la classe. Enfin, ajouter dans le GroupeSingle l’instance.

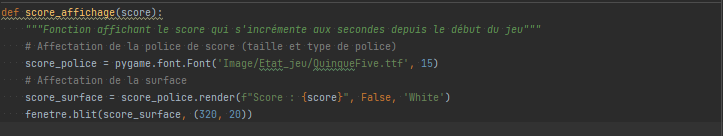


Pour recommencer le niveau, il faudra taper sur la touche « Echap ». La méthode reset\_position() permet de ramener à la position initiale le joueur. La méthode .empty() efface les obstacles de la fenetre.

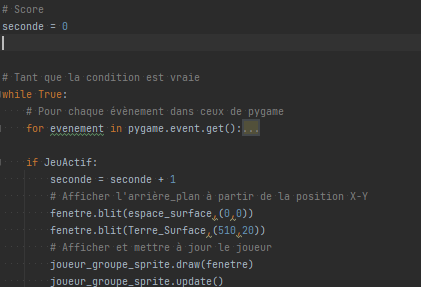


e. Score

Affichage du score



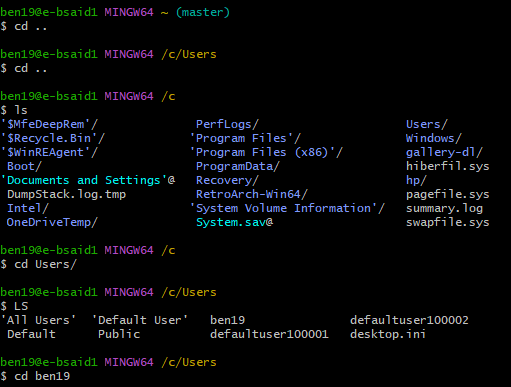
Le score dépend de l’état du jeu. S’il est actif (=pas de game over/pause), le score augmente en fonction du temps resté depuis le début de la partie. S’il est inactif, il faut le remettre à 0

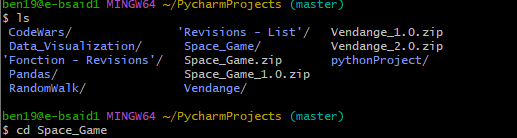




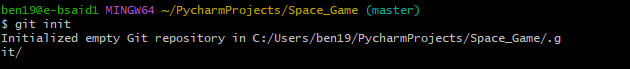
1. Mettre le projet sur Github

Aller vers le projet

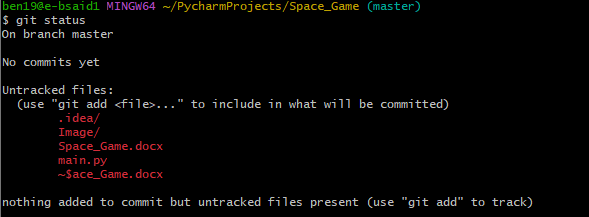




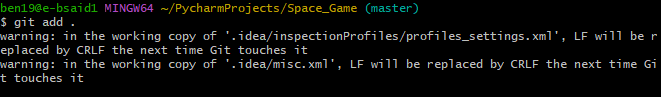
Initialiser le réquisitoire



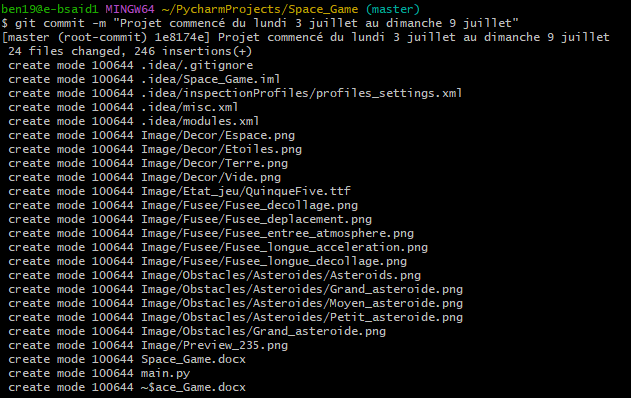
Regarder quels fichiers ne sont pas suivis dans le requisitoire



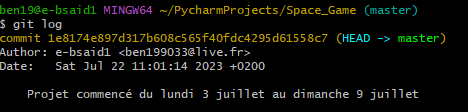
Ajouter les nouveaux changements/fichiers au requisitoire

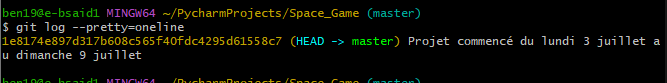


Créer un commit montrant les changements qu’apporte le nouveau fichier/dossier ou sa modification

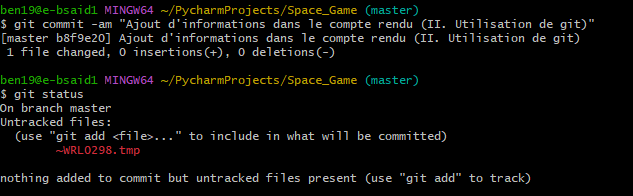


Regarder le compte rendu de tous les changements fait dans le fichier avec l’auteur des changements identifiables avec son ID

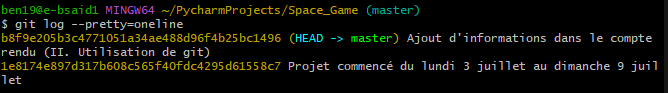




Faire suivre les fichiers modifiés et leur ajouter un commentaire



Retracer toutes les modifications passées



Conclusion :

* Revision sur les notions de OOP
* Decouverte d’une librairie Python
* Revision sur les listes
* Revision sur les conditions if
* Usage de GIMP