IUT Henri Poincaré LONGWY

TP Program en langage objet et program temps réel

Étudiant : FIERRO Jasiel

Enseignant : NGUYEN Quang Huy

Compte rendu du projet

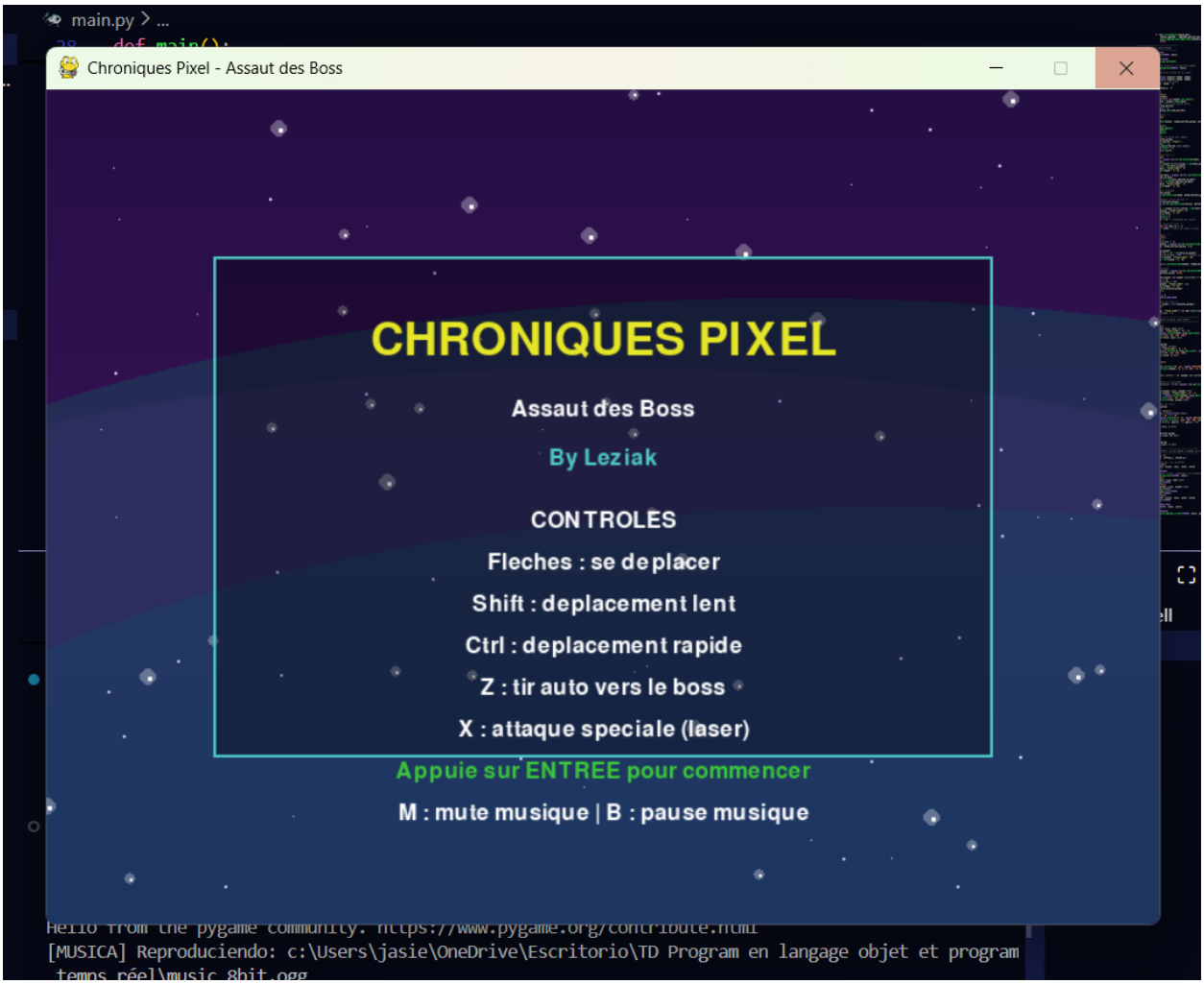
Jeu : “Chroniques Pixel – Assaut des Boss”

1. Présentation générale

Ce projet est un jeu de tir 2D en pixel-art réalisé avec la bibliothèque **Pygame**. Le joueur contrôle un petit vaisseau/personnage vert qui doit affronter un boss principal et ses sbires (minions) dans un décor spatial animé. Le jeu propose :

- Un fond animé avec étoiles et nébuleuses.
- Un système de **boss** avec plusieurs patterns d'attaque.
- Des **minions** qui descendent depuis le boss.
- Des tirs normaux, tir spécial (laser), et un système de **power-ups / améliorations** entre les niveaux.
- Une interface complète : écran titre, pause, HUD, écran de game over, écran de sélection d'amélioration.
- De la **musique 8-bit** avec gestion du mute et de la pause.

L'architecture du projet est modulaire : chaque fichier a une responsabilité précise.



2. Organisation des fichiers

- config.py : constantes globales du jeu (taille de fenêtre, couleurs, polices, états, chemin de base, volume de la musique).
- music.py : initialisation et contrôle de la musique (lecture, mute, pause).
- background.py : création et animation du fond spatial (dégradé, planète, nébuleuses, étoiles avec parallax).
- upgrades.py : définition des différentes améliorations possibles pour le joueur.
- ui.py : toutes les fonctions d'interface graphique (titre, pause, HUD, écran de game over, écran d'améliorations).
- entities.py : définition de toutes les entités du jeu : Joueur, Boss, balles, minions, attaque spéciale.
- main.py : point d'entrée du jeu. Contient la boucle principale, la gestion des états, des entrées clavier/souris, des collisions et de la progression de niveaux.

3. Détail par fichier

3.1. Fichier config.py – Configuration globale

Rôle : centraliser toutes les constantes utilisées dans le jeu.

Principales parties :

- Importations :
- import pygame
- import os

pygame pour tout le côté graphique/son, os pour gérer les chemins de fichiers.

- Pré-initialisation du mixer :
- try:
- pygame.mixer.pre_init(44100, -16, 2, 512)
- except Exception:
- pass
- pygame.init()

On prépare le module son (pre_init) avant pygame.init() pour éviter des décalages ou bugs audio. Le try/except évite un crash si cela échoue.

- Taille de la fenêtre & FPS :
- WIDTH, HEIGHT = 800, 600
- FPS = 60

Le jeu tourne en 800×600 pixels, à 60 images par seconde.

- Polices :
- GOTHIC_FONT_NAME = "Old English Text MT"
- FONT = pygame.font.SysFont(GOTHIC_FONT_NAME, 24)
- BIG_FONT = pygame.font.SysFont(GOTHIC_FONT_NAME, 48)

On utilise une police de style gothique si elle est disponible. Sinon, Pygame tombe sur une police système par défaut.

- Définition des couleurs (en RGB) :
- BLACK = (0, 0, 0)
- WHITE = (255, 255, 255)
- RED = (200, 60, 60)

- ...

Ces constantes facilitent le dessin d'éléments (texte, barres de vie, etc.).

- États du jeu :
- STATE_TITLE = "TITLE"
- STATE_PLAYING = "PLAYING"
- STATE_PAUSED = "PAUSED"
- STATE_GAME_OVER = "GAME_OVER"
- STATE_UPGRADE = "UPGRADE"

Ces chaînes sont utilisées dans main.py pour savoir ce qu'on doit afficher et quelle logique appliquer.

- Dossier de base et volume de la musique :
- BASE_DIR = os.path.dirname(os.path.abspath(__file__))
- MUSIC_VOLUME = 0.6

BASE_DIR pointe vers le dossier où se trouve le script, ce qui permet de charger des fichiers de musique sans problème de chemin. MUSIC_VOLUME définit le volume global de la musique (60 %).

3.2. Fichier music.py – Gestion de la musique

Rôle : encapsuler toute la logique liée à la musique de fond.

- Import et constantes :
- import os
- import pygame
- from config import BASE_DIR, MUSIC_VOLUME

On importe le chemin de base et le volume déclarés dans config.py.

- Variables d'état de la musique :
- music_muted = False
- music_paused = False

Permet de savoir si la musique est coupée ou simplement en pause.

Fonction start_music()

1. Réinitialise les drapeaux :
2. music_muted = False
3. music_paused = False

4. Initialise le mixer si besoin :
5. if not pygame.mixer.get_init():
6. pygame.mixer.init()
7. Liste de chemins possibles :
8. music_paths = [
9. os.path.join(BASE_DIR, "music_8bit.ogg"),
10. os.path.join(BASE_DIR, "music_8bit.mp3"),
11.]

On essaie d'abord .ogg, puis .mp3.

12. Boucle sur les chemins :
    - os.path.exists(path) vérifie si le fichier existe.
    - pygame.mixer.music.load(path) charge la musique.
    - set_volume(MUSIC_VOLUME) règle le volume.
    - play(-1) lance la musique en boucle infinie.
      Si aucun fichier n'est trouvé, on affiche un message dans la console.

Fonction toggle_mute_music()

- Vérifie que le mixer est initialisé et qu'il y a quelque chose en train de jouer.
- Si music_muted est False, met le volume à 0.0.
- Sinon, remet le volume à MUSIC_VOLUME.
  Ce n'est **pas** une pause : la musique continue de tourner, mais sans son.

Fonction toggle_pause_music()

- Si music_paused est False :
- pygame.mixer.music.pause()
- music_paused = True
- Sinon :
- pygame.mixer.music.unpause()
- music_paused = False

Ici la musique est vraiment figée dans le temps.

3.3. Fichier background.py – Fond animé

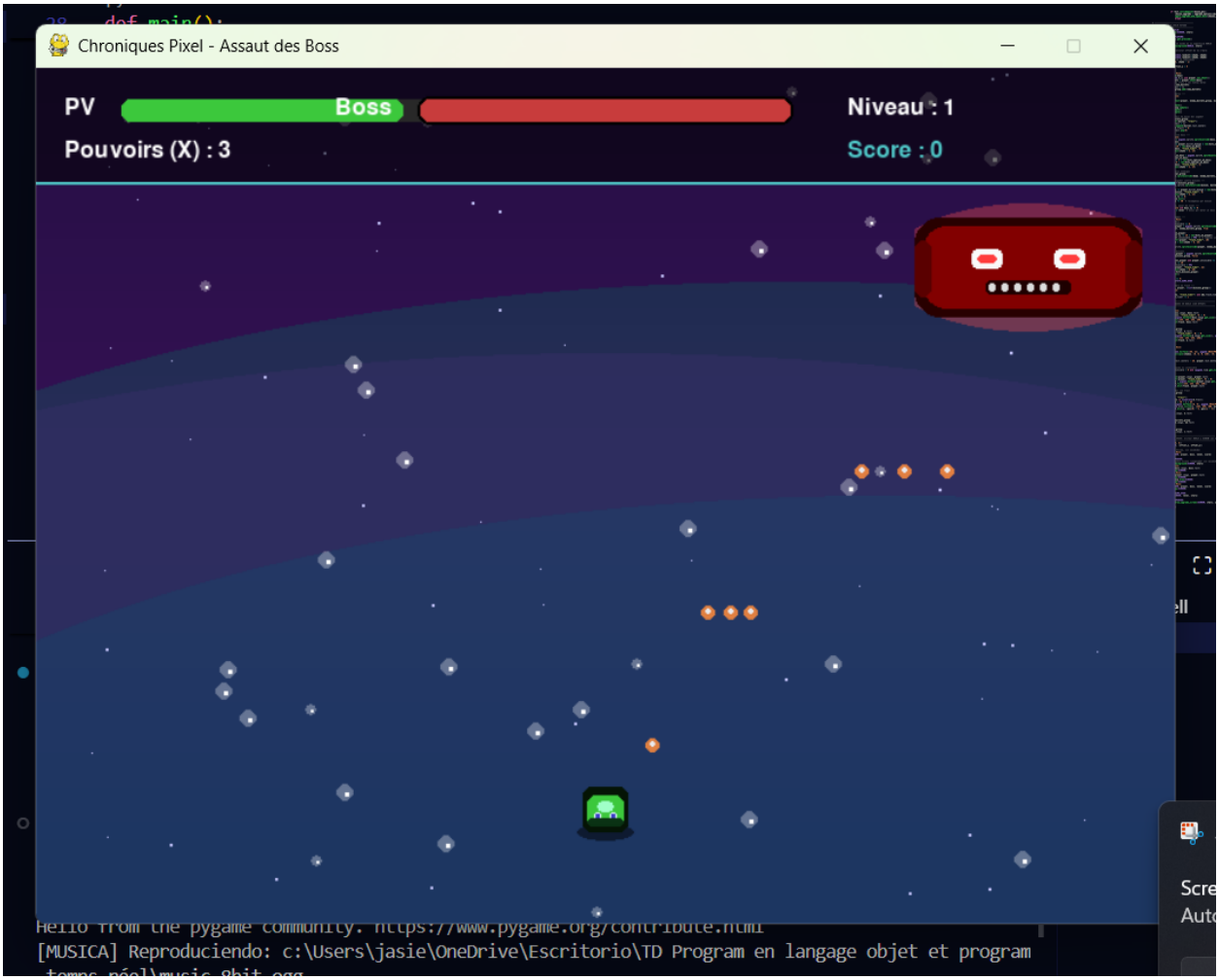
Rôle : créer un fond dynamique avec un effet de galaxie et des étoiles en parallax.

Fonction init_stars(num_far=55, num_near=35)

- Crée une liste de “stars” (étoiles).
- Chaque étoile est un petit tableau : [x, y, speed, size, layer]
    - layer = 0 : étoiles lointaines, petites, lentes.
    - layer = 1 : étoiles proches, plus grosses, plus rapides.
- On utilise random.randint pour la position et random.uniform pour la vitesse, afin d’avoir un ciel un peu aléatoire.

Fonction update_and_draw_background(surface, stars)

1. **Dégradé vertical** :
    - Pour chaque ligne de pixel i de 0 à HEIGHT, on calcule un mélange entre trois couleurs (top_color, mid_color, bottom_color).
    - On dessine une ligne horizontale (pygame.draw.line) qui donne l’illusion de ciel de galaxie.
2. **Planète et nébuleuses** :
    - pygame.Surface(..., pygame.SRCALPHA) permet de dessiner avec transparence.
    - On dessine de grandes ellipses colorées qui sont blitées par-dessus le fond pour donner un effet de planète et de nuages spatiaux.
3. **Mouvement des étoiles** :
    - Pour chaque étoile :
        - On augmente y de speed.
        - Si l’étoile sort de l’écran (y > HEIGHT), on la remet en haut avec nouvelle vitesse/taille.
    - On met à jour la liste stars avec les nouvelles coordonnées.
4. **Dessin des étoiles** :
    - Couleur légèrement différente entre near/far.
    - Pour les étoiles proches (layer == 1), on crée un petit “glow” :
        - Une surface transparente.
        - Un cercle avec un peu d’alpha (transparence).
    - On dessine enfin un petit rectangle (ou pixel) à la position de l’étoile.



3.4. Fichier upgrades.py – Système d'améliorations

Rôle : définir les différentes améliorations que le joueur peut choisir entre les niveaux.

- Fonction principale :
- def get_upgrade_options(player):

Elle renvoie une liste de **3 upgrades** aléatoires.

- À l'intérieur, on définit plusieurs fonctions locales, chacune modifiant un attribut du player :
  - up_fire_rate(p) : diminue shoot_cooldown_max → le joueur tire plus vite (cooldown plus court).
  - up_bullet_speed(p) : augmente bullet_speed → balles plus rapides.
  - up_move_speed(p) : augmente base_speed et fast_speed → déplacement plus fluide et rapide.
  - up_hp(p) : +20 de PV max, et on remet la vie actuelle au max.
  - up_special_charge(p) : ajoute 1 charge au pouvoir spécial X.

- Chaque amélioration est un **dictionnaire** :
- {
- "name": "Cadence de tir",
- "desc": "Tirs plus rapides (Z)",
- "apply": up_fire_rate,
- }

apply est une fonction qu'on appelle plus tard dans main.py quand le joueur clique sur une carte.

- random.sample(all_upgrades, 3) choisit 3 upgrades différentes à proposer à chaque inter-niveau.

3.5. Fichier ui.py – Interface utilisateur

Rôle : tout ce qui concerne l'affichage de texte, HUD, écrans spéciaux.

Importe plusieurs constantes de config.py et la fonction update_and_draw_background pour réutiliser le fond animé.

Fonction utilitaire draw_text(surface, text, font, color, x, y, center=False)

- Rend le texte avec font.render.
- Si center=True, le centre du texte sera placé en (x, y).
- Sinon, (x, y) est le coin supérieur gauche (topleft).

Fonction draw_health_bar(...)

- Calcule le ratio de vie :
- ratio = max(hp, 0) / max_hp
- Dessine un rectangle gris (fond) puis un rectangle coloré (barre de vie) proportionnel au ratio.

Écran titre : draw_title_screen(surface, stars)

- Met à jour le fond animé.
- Crée un panneau central semi-transparent (surface avec alpha) pour afficher le titre.
- Effet de "fade-in" :
- alpha = min(255, int(pygame.time.get_ticks() * 0.15))
- panel.set_alpha(alpha)
- Affiche :
    - Titre du jeu (grande police).
    - Sous-titre, auteur, contrôles.

- Message “Appuie sur ENTREE pour commencer”.
- Rappel des touches M/B pour la musique.

Écran de pause : draw_pause_overlay(surface)

- Dessine un grand overlay sombre sur tout l’écran.
- Puis un panneau central avec bordure cyan.
- Texte : “PAUSE”, “R : reprendre…”, “T : retour à l’écran titre”, “M/B : musique”.

Écran de game over : draw_game_over(surface, level, stars)

- Fond animé, panneau rouge, texte “GAME OVER”.
- Affiche le niveau atteint :
- f"Tu es arrive au niveau {level}"

HUD : draw_hud(surface, player, boss, level, score)

- Bandeau translucide en haut de l’écran.
- Informations :
  - PV du joueur (barre).
  - Nombre de pouvoirs spéciaux (Pouvoirs (X)).
  - Niveau actuel.
  - Score.
- Si un boss existe, affiche aussi une barre de vie au centre.

Écran d’amélioration : draw_upgrade_screen(surface, stars, upgrade_options, mouse_pos)

- Fond animé.
- Titre “AMELIORATION” + explication.
- Crée **3 cartes** pour les upgrades :
  - Si la souris survole une carte (rect.collidepoint(mouse_pos)), on change la couleur de fond et la largeur de la bordure.
- Utilise une fonction interne wrap_text(text, font, max_width) pour couper la description en plusieurs lignes qui tiennent dans la carte (on teste la largeur avec font.size).
- Renvoie la liste de rectangles cards pour que main.py puisse savoir sur quelle carte le joueur a cliqué.

3.6. Fichier entities.py – Entités du jeu

Rôle : définir toutes les classes qui représentent des éléments du jeu (joueur, boss, balles, etc.). Toutes héritent de pygame.sprite.Sprite.

3.6.1. Classe Player

- Constructeur :
  - Crée une surface 32×32 avec transparence.
  - _draw_player_sprite() dessine le sprite (contours, yeux bleus, torse vert, ombres…).
  - Position initiale : centre en bas de l'écran (center=(WIDTH // 2, HEIGHT - 80)).
- Attributs de déplacement :
  - base_speed, slow_speed, fast_speed.
  - Servent à ajuster la vitesse en fonction de Shift / Ctrl.
- Attributs de vie et tir :
  - max_hp, hp.
  - shoot_cooldown, shoot_cooldown_max (cadence de tir).
  - special_cooldown, special_charges (pouvoir X).
- Améliorations :
  - bullet_speed, bullet_damage.
  - fire_mode :
    - 0 : tir normal auto-aim vers le boss.
    - 1 : tir triple en éventail.
    - 2 : tir normal + deux balles en "wave".
- invincible : nombre de frames où le joueur ne peut pas être touché après un coup.

Méthode update(self, keys)

- Lit les touches fléchées pour construire un vecteur (vx, vy).
- Normalise ce vecteur pour avoir la même vitesse en diagonale.
- Choisit la vitesse selon :
  - Shift → slow_speed
  - Ctrl → fast_speed
  - Sinon → base_speed
- Mets à jour la position et la limite à la fenêtre.
- Décrémente les cooldowns (shoot_cooldown, special_cooldown, invincible).

Tir : can_shoot() et shoot(boss)

- can_shoot() → vrai si shoot_cooldown <= 0.
- shoot(boss) :
    - Réinitialise le cooldown.
    - Calcule un angle vers le boss s'il existe.
    - Selon fire_mode :
        - 0 : une seule Bullet de type "normal".
        - 1 : trois Bullet de type "spread" avec un angle décalé.
        - 2 : une "normal" + deux "wave" qui montent en ondulant.

Pouvoir spécial : can_use_special() et use_special()

- Vérifie qu'il reste des charges et que le cooldown spécial est à zéro.
- Crée un SpecialAttack (grand laser vertical) centré sur self.rect.centerx.
- Ajoute ce laser dans le groupe special_group.
- Décrémente special_charges et met un cooldown de 2 secondes (FPS * 2).

3.6.2. Classe Bullet – Balles du joueur

- Types (kind) :
    - "normal" : tir auto-dirigé vers le boss.
    - "spread" : tir dans un angle fixe (pour tir en éventail).
    - "wave" : tir qui monte droit mais se déplace en sinusoïde (onde).
- _draw_bullet_sprite(kind) dessine un petit projectile circulaire avec une couleur différente selon le type.
- Dans __init__ :
    - On crée la surface et le sprite.
    - On calcule (vx, vy) en fonction du type :
        - wave : vy négatif (vers le haut), vx = 0, plus un paramètre wave_t qui évolue pour la sinusoïde.
        - spread : utilise un angle passé en paramètre pour calculer vx, vy.
        - normal : calcul direction vers le boss via dx, dy et math.hypot.
- update() :
    - Pour wave, on fait monter la balle (rect.y += vy) et on modifie rect.x avec sin pour l'onde.

- Pour les autres, on ajoute simplement vx et vy.
- Si la balle sort de l'écran, on la supprime (self.kill()).

3.6.3. Classe EnemyBullet – Balles du boss

- Types :
  - "normal" : bullet ordinaire.
  - "slow_orb" : orbes lents qui descendent verticalement.
  - "wave" : balles en vague utilisées dans l'attaque radiale.
- _draw_enemy_bullet_sprite(kind) : change la couleur/glow selon le type.
- update() :
  - Pour "wave" : on bouge en Y et on modifie x avec une petite sinusoïde.
  - Pour les autres : on applique simplement vx, vy.
  - Comme pour Bullet, on supprime quand ça sort de l'écran.

3.6.4. Classe SpecialAttack – Laser spécial

- Rectangle vertical de largeur 60 sur toute la hauteur de l'écran.
- Dessin :
  - Une colonne violette semi-transparente.
  - Un cœur blanc plus lumineux au centre.
- Attribut lifetime = 20 frames → après 20 frames, le laser disparaît (kill()).

3.6.5. Classe Minion – Ennemi secondaire

- Petites entités qui descendent depuis le boss.
- Points de vie dépendants du niveau :
- self.max_hp = 20 + level * 5
- Vitesse de descente également liée au level.
- Mouvement :
  - Descente verticale (rect.y += speed_y).
  - Petit mouvement sinusoïdal horizontal avec self.phase.

3.6.6. Classe Boss

- Grosse entité en haut de l'écran (160×90).
- Points de vie :
- self.max_hp = 120 + level * 50

- Mouvement :
    - Va de gauche à droite ; rebondit sur les bords (inversion de speed_x).
    - Légère oscillation verticale avec sin du temps (pygame.time.get_ticks()).
- Phase “enragée” :
    - Quand hp <= max_hp * 0.5 :
        - enraged = True.
        - Vitesse horizontale augmentée.
        - Intervalle de tir plus court.
        - Durée des patterns raccourcie.
        - Sprite redessiné plus agressif (_draw_enraged_sprite()).
- Patterns d’attaque :
    - current_pattern ∈ {0, 1, 2, 3}.
    - pattern_time et pattern_duration gèrent la durée de chaque pattern.
    - Quand pattern_time dépasse la durée → on choisit un nouveau pattern différent de l’actuel.

Fonction maybe_shoot(player, enemy_bullets_group, minions_group)

Selon current_pattern :

- **Pattern 0 :** triple tir dirigé vers le joueur
  On calcule un vecteur direction du boss vers le joueur pour trois positions différentes, et on crée trois EnemyBullet.
- **Pattern 1 :** “mur” d’orbes lents
  On place num = 9 balles "slow_orb" régulièrement espacées sur l’axe X, qui descendent verticalement, forçant le joueur à se faufiler.
- **Pattern 2 :** cercle radial tournant
    - num_bullets = 14.
    - On calcule 14 angles répartis sur 360° + un angle de rotation self.spin_angle qui augmente à chaque salve.
    - On crée des EnemyBullet de type "wave" avec ces vecteurs (pattern type danmaku).
- **Pattern 3 :** invocation de minions
    - On crée 3 minions (Minion) avec des décalages horizontaux (-80, 0, +80).
    - Ils descendent ensuite vers le joueur.

3.7. Fichier main.py – Boucle principale du jeu

Rôle : coordonner tout le reste. C'est ici que Pygame est initialisé, que la boucle de jeu tourne, et que tous les états sont gérés.

Initialisation

- pygame.init()
- Création de la fenêtre :
- SCREEN = pygame.display.set_mode((WIDTH, HEIGHT))
- pygame.display.set_caption("Chroniques Pixel - Assaut des Boss")
- CLOCK = pygame.time.Clock() pour contrôler le framerate.
- WORLD = pygame.Surface((WIDTH, HEIGHT)) : surface intermédiaire où on dessine tout le monde, pour pouvoir appliquer le **screen shake** (secousse de caméra) en la blitant avec un offset.
- Lancement de la musique : start_music().
- Variables d'état :
    - state = STATE_TITLE
    - level = 1
    - score = 0
    - shake = 0 : intensité actuelle du screen shake.
    - Groupes de sprites :
        - bullets_group
        - enemy_bullets_group
        - special_group
        - minions_group
    - stars = init_stars() pour le fond.
    - upgrade_options, upgrade_cards pour l'écran d'améliorations.

Fonctions internes

1. start_new_game() :
    - Réinitialise le niveau, le score, le shake.
    - Crée un nouveau Player() et un Boss(level).
    - Vide les groupes de sprites.
2. next_level() :
    - Incrémente level.

- Met à jour le fire_mode du joueur (plus le niveau est élevé, plus le tir devient avancé).
- Génère 3 **upgrades** avec get_upgrade_options(player) et passe l'état du jeu à STATE_UPGRADE.
- Vide les groupes de sprites pour repartir propre au prochain boss.

3. apply_upgrade_and_spawn_boss(chosen_upgrade) :
   - Applique l'upgrade au joueur via chosen_upgrade["apply"](player).
   - Crée un nouveau boss pour le niveau actuel.
   - Remet state = STATE_PLAYING.

Boucle principale while running:

Pour chaque frame :

1. CLOCK.tick(FPS) → limite à 60 FPS.
2. Récupère la position de la souris : mouse_pos = pygame.mouse.get_pos().

Gestion des événements

- pygame.QUIT → running = False.
- Touches globales :
  - M → toggle_mute_music()
  - B → toggle_pause_music()
- Selon l'état :
  - **TITLE** :
    - Si ENTER → start_new_game() et state = STATE_PLAYING.
  - **PLAYING** :
    - P ou ESC → state = STATE_PAUSED.
    - X → si player existe, player.use_special(special_group).
  - **PAUSED** :
    - R → retour au jeu.
    - T → retour à l'écran titre (sans relancer direct la partie).
  - **GAME_OVER** :
    - ENTER → retour à l'écran titre.
  - **UPGRADE** :
    - Clic gauche de la souris :

- On parcourt upgrade_cards et on teste rect.collidepoint(mouse_pos).
- Si clic sur une carte, on récupère l'upgrade correspondante et on appelle apply_upgrade_and_spawn_boss.

Logique et dessin par état

État TITLE

- draw_title_screen(SCREEN, stars)

État PLAYING

1. Lecture du clavier : keys = pygame.key.get_pressed().
2. **Fond et screen shake** :
   - On dessine le fond sur WORLD via update_and_draw_background(WORLD, stars).
   - Si shake > 0, on génère un offset_x, offset_y aléatoires entre -shake et +shake.
   - On diminue shake petit à petit.
3. **Mise à jour du joueur** :
   - player.update(keys)
   - Si Z est pressé et player.can_shoot() → on récupère la liste de balles new_bullets = player.shoot(boss) et on les ajoute au bullets_group.
   - On ajoute aussi un champ trail aux balles pour dessiner une trainée lumineuse.
4. **Mise à jour du boss & des tirs ennemis** :
   - boss.update()
   - boss.maybe_shoot(player, enemy_bullets_group, minions_group)
5. **Update de tous les sprites** :
   - bullets_group.update()
   - enemy_bullets_group.update()
   - special_group.update()
   - minions_group.update()
6. **Gestion des traînées des balles du joueur** :
   - Pour chaque bullet, on maintient une petite liste des positions précédentes pour dessiner un effet de "trail".
7. **Collisions sur le boss** :
   - pygame.sprite.spritecollide(boss, bullets_group, True) → balles du joueur qui touchent le boss, on diminue boss.hp et on augmente le score.

- Si le laser spécial (special_group) touche le boss, on applique un autre type de dégâts.

8. **Laser qui nettoie les balles ennemies** :
   - Pour chaque beam dans special_group, on fait spritecollide(beam, enemy_bullets_group, True) → les balles ennemies sont détruites.
9. **Balles du joueur contre minions** :
   - Si un minion est touché, on réduit sa vie, on ajoute du shake, et on augmente le score quand il meurt.
10. **Boss mort → niveau suivant** :
    - Si boss.hp <= 0, on ajoute un bonus de score et on appelle next_level().
11. **Dégâts au joueur** :
    - Si non invincible :
      - Collision avec enemy_bullets_group → on diminue player.hp, on le rend invincible pendant ~1 seconde et on ajoute du shake.
      - Collision avec minions_group → dégâts supplémentaires, minion tué, etc.
    - Si player.hp <= 0 → state = STATE_GAME_OVER.
12. **Timers de flash** :
    - Pour boss, player, et chaque minion, si flash_timer > 0, on le décrémente.
      Ensuite, lors du dessin, si flash_timer > 0, on dessine un overlay blanc semi-transparent pour simuler un “hit”.
13. **Dessin sur WORLD sans offset** :
    - Boss, minions, player (avec ombre et clignotement d’invincibilité), balles, balles ennemies, laser.
    - Effet de trail sur les balles du joueur.
14. **Application du screen shake** :
    - On efface SCREEN et on fait :
    - SCREEN.blit(WORLD, (offset_x, offset_y))
15. **HUD** :
    - draw_hud(SCREEN, player, boss, level, score) affiche PV, pouvoirs, niveau et score.

État PAUSED

- On redessine la scène actuelle (fond, boss, player, balles).
- On dessine le HUD.
- On ajoute draw_pause_overlay(SCREEN) par-dessus.

État GAME_OVER

- draw_game_over(SCREEN, level, stars)

État UPGRADE

- upgrade_cards = draw_upgrade_screen(SCREEN, stars, upgrade_options, mouse_pos)

Enfin, à la fin de chaque frame :

pygame.display.flip()

pour mettre à jour l'écran.

Quand on sort de la boucle :

pygame.quit()

sys.exit()

Conclusion

Ce projet de jeu constitue une réalisation complète et cohérente, combinant programmation orientée objet, gestion d'événements interactifs et création d'éléments graphiques et sonores. L'architecture modulaire mise en place — répartie entre la gestion de la configuration, des entités, du fond animé, de l'interface, des améliorations et de la logique principale — permet une organisation claire et une évolution future facilitée.

Le système de jeu intègre plusieurs mécaniques avancées : patterns d'attaque variés du boss, progression par niveaux, choix d'améliorations, effets visuels (parallax, animations, flashs, screen shake), ainsi qu'un environnement sonore contrôlable par l'utilisateur. L'ensemble offre une expérience dynamique et fluide, démontrant une maîtrise solide des fonctionnalités essentielles de Pygame et des principes fondamentaux du développement de jeux vidéo 2D.

Ce travail met également en avant la capacité à concevoir un projet complet : du design visuel jusqu'à la gestion des interactions, tout en assurant la lisibilité, la modularité et la robustesse du code. Le résultat final est un jeu fonctionnel, attrayant et techniquement abouti, illustrant efficacement la compréhension des outils et des concepts étudiés.