EPITA - Projet SAE J3D 2025-2026

RAPPORT DE SOUTENANCE

ORBITE ZÉRO

Équipe Relentless Five

Ilian DROUIN - Chef de projet
Samy EL-ALAOUI - Architecture et réseau
Louis MURAIL - IA et audio
Melvyn TCHATCHOU - Interface graphique

Soutenance Technique n°1 - Janvier 2026

Classe B1.1

Sommaire

Table des matières

1. Introduction

Ce rapport présente l'état d'avancement de notre projet ORBITE ZÉRO depuis la validation du Cahier des Spécifications Techniques (CST). Il s'agit d'un jeu vidéo coopératif en Python/PyGame où deux joueurs doivent réparer les 8 canons de défense d'une station spatiale en moins de 10 minutes pour sauver la Terre d'un astéroïde.

Notre équipe, Relentless Five, est composée de quatre étudiants de première année à l'EPITA. Nous avons dû nous adapter car un cinquième membre a quitté le projet en cours de route, mais nous avons maintenu notre motivation et notre organisation. Ce rapport détaille le travail accompli par chaque membre de l'équipe, les difficultés que nous avons rencontrées, et les fonctionnalités que nous avons réussi à implémenter.

Nous tenons à souligner que ce projet représente notre première expérience de développement d'envergure. C'est la première fois que nous travaillons en équipe sur un projet informatique de cette taille, et c'est aussi notre première confrontation avec des problématiques comme le réseau multijoueur ou la gestion de projet collaborative.

2. Rappel du cahier des charges

2.1 Contexte du projet

Dans le cadre de la SAE J3D (première année EPITA), nous devons réaliser un jeu vidéo en Python avec PyGame. Le cahier des charges impose un mode multijoueur en réseau, ce qui constitue le défi principal du projet.

ORBITE ZÉRO est un jeu coopératif se déroulant en l'an 3000. Un astéroïde géant menace la Terre et le seul espoir est d'activer le système de défense AGERIS. Les 8 canons du système sont en panne et deux scientifiques sont envoyés pour les réparer : le Concepteur (créateur du système) et le Pragmatique (expert en réparation). Ils disposent de 10 minutes avant l'impact.

2.2 Objectifs techniques

Les objectifs que nous nous sommes fixés sont les suivants :

- Développer un jeu fonctionnel de bout en bout, sans bugs bloquants
- Implémenter le multijoueur en réseau local
- Créer une ambiance spatiale immersive avec des graphismes soignés
- Créer un code propre, organisé et commenté
- Atteindre une fluidité de 30-60 FPS et une latence réseau inférieure à 100ms

2.3 Périmètre fonctionnel

Notre jeu comprend les éléments suivants, classés par priorité :

Fonctionnalités obligatoires :

- Menu principal avec navigation clavier/souris
- Système de création et de connexion aux parties
- Déplacements des joueurs (ZQSD/flèches)
- Timer de 10 minutes
- Écrans de victoire et défaite

Fonctionnalités souhaitables :

- 8 canons à réparer avec affichage de leur état
- IA des ennemis avec pathfinding A*
- Audio (musique de fond et effets sonores)
- Barre de vie des joueurs

3. Répartition des tâches au sein de l'équipe

Dès le début du projet, nous avons établi une répartition claire des responsabilités. Chaque membre de l'équipe s'est vu attribuer un domaine principal, tout en participant aux tâches transversales comme les tests et la documentation.

3.1 Organisation générale

Membre	Rôle	Responsabilités principales
Ilian DROUIN	Chef de projet	Coordination, planification, gameplay, main.py, structures de données
Samy EL-ALAOUI	Dev principal	Multi joueur, réseau
Louis MURAIL	Développeur	IA ennemis, algorithme A*, Gestionnaire audio
Melvyn TCHATCHOU	Développeur	Interface utilisateur

4. Contributions individuelles détaillées

Dans cette section, chaque membre de l'équipe présente personnellement son travail et ses contributions au projet.

4.1 Ilian DROUIN - Chef de projet

En ce qui me concerne, j'ai pris en charge la coordination générale du projet et la mise en place des fondations du jeu. Mon rôle principal a été de m'assurer que l'équipe avance dans la bonne direction et que tout le monde sait ce qu'il doit faire.

Coordination et planification

En tant que chef de projet, j'ai organisé nos réunions et créé un planning pour nous aider à respecter les échéances. Au début c'était un peu difficile parce qu'on ne savait pas trop comment s'organiser, mais on a fini par trouver notre rythme.

J'ai mis en place un document partagé où on note les tâches à faire et qui fait quoi.

J'ai aussi rédigé une bonne partie du cahier des charges fonctionnel (CDSF) et du cahier des spécifications techniques (CST). L'objectif était que tout le monde comprenne bien ce qu'on devait faire avant de commencer à coder.

Fichier de configuration (config.py)

J'ai créé le fichier config.py qui centralise toutes les constantes du jeu. L'idée c'est que si on veut changer une couleur ou autre, on le fait à un seul endroit :

Extrait de config.py :

```
# Dimensions de la fenêtre
LARGEUR = 1280
HAUTEUR = 720

# Couleurs du jeu (thème spatial)
COULEUR_FOND = (10, 14, 26)       # Bleu très foncé
COULEUR_TEXTE = (255, 255, 255)   # Blanc
COULEUR_ACCENT = (230, 126, 34)   # Orange (pour les boutons)
COULEUR_ACCENT_HOVER = (211, 84, 0)   # Orange foncé au survol

# Configuration réseau
PORT_SERVEUR = 5555
TAILLE_BUFFER = 4096

# Paramètres de gameplay
TEMPS_PARTIE = 600  # 10 minutes en secondes
NOMBRE_CANONS = 8
VITESSE_JOUEUR = 5
```

Ce fichier est importé par tous les autres modules. C'est pratique parce que quand on a voulu changer les couleurs pour avoir un thème plus spatial, j'ai juste modifié ce fichier et ça s'est appliqué partout.

Structures de données

J'ai conçu les classes Joueur et Partie qui représentent les données principales du jeu. L'idée c'est d'avoir des objets qu'on peut facilement convertir en JSON pour les envoyer sur le réseau :

Classe Joueur (structures_donnees.py) :

```
class Joueur:
    """Représente un joueur dans la partie"""

    def __init__(self, pseudo, role, x=0, y=0):
        self.pseudo = pseudo
        self.role = role  # "Concepteur" ou "Pragmatique"
        self.x = x
        self.y = y
        self.vie = 100
        self.en_action = False

    def to_dict(self):
        """Convertit le joueur en dictionnaire pour JSON"""
        return {
            "pseudo": self.pseudo,
            "role": self.role,
            "x": self.x,
            "y": self.y,
            "vie": self.vie,
            "en_action": self.en_action
        }

    @classmethod
    def from_dict(cls, data):
        """Crée un joueur à partir d'un dictionnaire"""
        joueur = cls(data["pseudo"], data["role"], data["x"], data["y"])
        joueur.vie = data["vie"]
        joueur.en_action = data["en_action"]
        return joueur
```

Les méthodes to_dict() et from_dict() sont très importantes pour le réseau. Quand on envoie l'état d'un joueur à l'autre machine, on le convertit en dictionnaire, puis en JSON. De l'autre côté, on fait l'inverse.

Classe Partie (structures_donnees.py) :

```
class Partie:
    """Représente une partie en cours"""

    def __init__(self):
        self.joueurs = []  # Liste des joueurs (max 2)
        self.canons = [False] * 8  # 8 canons, False = cassé
        self.temps_restant = 600  # 10 minutes
        self.en_cours = False

    def ajouter_joueur(self, joueur):
        if len(self.joueurs) < 2:
            self.joueurs.append(joueur)
            return True
        return False

    def reparer_canon(self, index):
        if 0 <= index < 8 and not self.canons[index]:
            self.canons[index] = True
            return True
        return False

    def victoire(self):
        return all(self.canons)  # Tous les canons réparés
```

Point d'entrée du jeu (main.py)

J'ai aussi travaillé sur main.py, le fichier principal qui lance le jeu. C'est lui qui initialise PyGame, crée la fenêtre et lance le jeu :

Extrait de main.py :

```
import pygame
from config import LARGEUR, HAUTEUR, COULEUR_FOND
from interface import MenuPrincipal

def main():
    # Initialisation de PyGame
    pygame.init()
    ecran = pygame.display.set_mode((LARGEUR, HAUTEUR))
    pygame.display.set_caption("ORBITE ZÉRO")
    horloge = pygame.time.Clock()

    # Lancement du menu principal
    menu = MenuPrincipal(ecran)
    menu.afficher()

    # Boucle principale du jeu
    en_cours = True
    while en_cours:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                en_cours = False

        ecran.fill(COULEUR_FOND)
        pygame.display.flip()
        horloge.tick(60)  # 60 FPS max

    pygame.quit()

if __name__ == "__main__":
    main()
```

4.2 Samy EL-ALAOUI - Multi joueurs et réseau

En ce qui me concerne, j'ai pris en charge tout le module réseau. C'est clairement la partie la plus technique du projet et j'ai passé beaucoup de temps à comprendre comment ça fonctionne.

Avant de réussir a faire quelque chose, j'ai du beaucoup aller sur internet pour comprendre, et trouver des exemples de code. ça a été très difficile car je n'ai pas de notion en réseau du tout. Même pour tester c'était pas facile, car il fallait 2 PC pour simuler les 2 joueurs.

Module réseau (reseau.py)

Le module réseau a été le plus difficile à développer. J'ai dû apprendre les sockets TCP et le threading pour que le jeu ne se bloque pas pendant les échanges réseau. Voici le serveur :

Classe ServeurJeu (reseau.py) :

```
import socket
import threading
import json
from config import PORT_SERVEUR, TAILLE_BUFFER

class ServeurJeu:
    """Serveur TCP pour héberger une partie"""

    def __init__(self):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.client = None
        self.messages_recus = []  # File des messages reçus
        self.connecte = False

    def demarrer(self):
        """Lance le serveur et attend une connexion"""
        ip = socket.gethostbyname(socket.gethostname())
        self.socket.bind((ip, PORT_SERVEUR))
        self.socket.listen(1)
        print(f"Serveur en attente sur {ip}:{PORT_SERVEUR}")

        # Attente de connexion dans un thread séparé
        thread = threading.Thread(target=self._attendre_client)
        thread.daemon = True
        thread.start()

    def _attendre_client(self):
        """Attend qu'un client se connecte"""
        self.client, adresse = self.socket.accept()
        print(f"Client connecté depuis {adresse}")
        self.connecte = True
        self._ecouter()  # Lance l'écoute des messages

    def _ecouter(self):
        """Écoute les messages entrants"""
        while self.connecte:
            try:
                data = self.client.recv(TAILLE_BUFFER)
                if data:
                    message = json.loads(data.decode())
                    self.messages_recus.append(message)
            except:
                self.connecte = False
```

Le threading était vraiment compliqué à comprendre au début. Le problème c'est que si on attend un message réseau dans le thread principal, le jeu se fige et ça donne l'impression qu'il a planté. Avec le thread séparé, l'écoute se fait en arrière-plan.

4.3 Louis MURAIL - IA et audio

En ce qui me concerne, j'ai pris en charge la partie intelligence artificielle des ennemis et le système audio du jeu. J'ai fait beaucoup de recherches sur l'algorithme A* et j'ai préparé le gestionnaire de sons.

Recherches sur l'algorithme A*

Pour que les ennemis puissent trouver leur chemin vers les joueurs en évitant les obstacles, j'ai étudié l'algorithme A* (A-star). C'est un algorithme de pathfinding très utilisé dans les jeux vidéo.

J'ai trouvé un super tutoriel sur le site Red Blob Games qui explique très bien le fonctionnement. L'idée c'est d'explorer les cases autour de la position actuelle en privilégiant celles qui semblent les plus proches de la destination. Voici les bases que j'ai comprises :

Base de l'algorithme A* (recherches) :

```
import heapq

def heuristique(a, b):
    """Distance de Manhattan entre deux points"""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star(grille, depart, arrivee):
    """Trouve le chemin le plus court avec A*"""
    # File de priorité : (coût estimé, coût actuel, position, chemin)
    file = [(0, 0, depart, [depart])]
    visites = set()

    while file:
        _, cout, position, chemin = heapq.heappop(file)

        if position == arrivee:
            return chemin  # Chemin trouvé !

        if position in visites:
            continue
        visites.add(position)

        # Explorer les voisins (haut, bas, gauche, droite)
        for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            voisin = (position[0] + dx, position[1] + dy)
            if est_valide(grille, voisin) and voisin not in visites:
                nouveau_cout = cout + 1
                estimation = nouveau_cout + heuristique(voisin, arrivee)
                heapq.heappush(file, (estimation, nouveau_cout, voisin, chemin +
[voisin]))

    return None  # Pas de chemin trouvé
```

Je n'ai pas encore implémenté tout ça dans le jeu, c'est prévu pour la soutenance 2. Mais j'ai bien compris le principe et j'ai fait des tests sur papier pour vérifier que ça fonctionne.

Gestionnaire audio (audio.py)

Pour l'audio, j'ai créé une classe qui gère la musique de fond et les effets sonores. PyGame a un module mixer qui permet de jouer des sons facilement :

Classe GestionnaireAudio (audio.py) :

```
import pygame
import os

class GestionnaireAudio:
    """Gère tous les sons et musiques du jeu"""

    def __init__(self):
        pygame.mixer.init()
        self.volume_musique = 0.7
        self.volume_effets = 0.9
        self.sons = {}  # Dictionnaire des effets sonores

    def charger_son(self, nom, chemin):
        """Charge un effet sonore"""
        if os.path.exists(chemin):
            self.sons[nom] = pygame.mixer.Sound(chemin)
            self.sons[nom].set_volume(self.volume_effets)

    def jouer_son(self, nom):
        """Joue un effet sonore"""
        if nom in self.sons:
            self.sons[nom].play()

    def jouer_musique(self, chemin, boucle=True):
        """Lance une musique de fond"""
        if os.path.exists(chemin):
            pygame.mixer.music.load(chemin)
            pygame.mixer.music.set_volume(self.volume_musique)
            pygame.mixer.music.play(-1 if boucle else 0)

    def set_volume_musique(self, volume):
        """Modifie le volume de la musique (0.0 à 1.0)"""
        self.volume_musique = max(0.0, min(1.0, volume))
        pygame.mixer.music.set_volume(self.volume_musique)
```

Le code est prêt mais on n'a pas encore trouvé les fichiers audio. Je vais chercher sur OpenGameArt et Freesound des musiques et effets sonores libres de droits qui correspondent à notre thème spatial.

4.4 Melvyn TCHATCHOU - Interface graphique

En ce qui me concerne, j'ai pris en charge toute la partie interface utilisateur du jeu : les menus, les boutons, les champs de texte, et aussi le site web de présentation. J'ai voulu créer une ambiance spatiale cohérente.

Je me suis beaucoup aidé de la documentation sur internet avec des exemples de programme que j'ai pu trouver pour m'inspirer.

Composants d'interface (interface.py)

J'ai créé des classes qui seront les mêmes pour tous les éléments d'interface. Ça permet d'avoir des boutons et des champs de texte qui fonctionnent partout de la même façon :

Classe Bouton (interface.py) :

```
class Bouton:
    """Un bouton cliquable avec effet de survol"""

    def __init__(self, x, y, largeur, hauteur, texte, couleur, couleur_hover):
        self.rect = pygame.Rect(x, y, largeur, hauteur)
        self.texte = texte
        self.couleur = couleur
        self.couleur_hover = couleur_hover
        self.survol = False
        self.selectionne = False  # Pour navigation clavier

    def dessiner(self, ecran, police):
        # Couleur selon l'état
        couleur_actuelle = self.couleur_hover if (self.survol or
self.selectionne) else self.couleur

        # Dessiner le rectangle du bouton
        pygame.draw.rect(ecran, couleur_actuelle, self.rect, border_radius=8)

        # Dessiner le texte centré
        surface_texte = police.render(self.texte, True, (255, 255, 255))
        pos_texte = surface_texte.get_rect(center=self.rect.center)
        ecran.blit(surface_texte, pos_texte)

    def verifier_survol(self, pos_souris):
        self.survol = self.rect.collidepoint(pos_souris)
        return self.survol

    def est_clique(self, pos_souris, clic):
        return clic and self.rect.collidepoint(pos_souris)
```

J'ai aussi créé une classe ChampTexte pour que les joueurs puissent entrer leur pseudo et l'adresse IP :

Classe ChampTexte (interface.py) :

```
class ChampTexte:
    """Un champ de saisie de texte"""

    def __init__(self, x, y, largeur, hauteur, placeholder=""):
        self.rect = pygame.Rect(x, y, largeur, hauteur)
        self.texte = ""
        self.placeholder = placeholder
        self.actif = False
        self.max_caracteres = 20
```

```
    def gerer_evenement(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN:
            self.actif = self.rect.collidepoint(event.pos)

        if event.type == pygame.KEYDOWN and self.actif:
            if event.key == pygame.K_BACKSPACE:
                self.texte = self.texte[:-1]
            elif event.key == pygame.K_RETURN:
                self.actif = False
            elif len(self.texte) < self.max_caracteres:
                self.texte += event.unicode

    def dessiner(self, ecran, police):
        # Fond du champ
        couleur_fond = (40, 40, 40) if self.actif else (30, 30, 30)
        pygame.draw.rect(ecran, couleur_fond, self.rect, border_radius=5)

        # Bordure
        couleur_bordure = (230, 126, 34) if self.actif else (100, 100, 100)
        pygame.draw.rect(ecran, couleur_bordure, self.rect, 2, border_radius=5)

        # Texte ou placeholder
        affichage = self.texte if self.texte else self.placeholder
        couleur_texte = (255, 255, 255) if self.texte else (150, 150, 150)
        surface = police.render(affichage, True, couleur_texte)
        ecran.blit(surface, (self.rect.x + 10, self.rect.y + 10))
```

Menus du jeu

J'ai créé tous les menus du jeu : menu principal, création de joueur, rejoindre une partie, options, et les écrans de fin. Chaque menu utilise les composants Bouton et ChampTexte que j'ai créés.

J’étais content de voir le menu principal fonctionner avec le fond étoilé animé. J'ai utilisé une technique simple : des petits cercles blancs qui bougent lentement pour simuler des étoiles.

Site web de présentation

J'ai aussi créé le site web de présentation du projet en HTML/CSS. J’ai de la même manière beaucoup regardé sur internet les exemples de site internet et comment créer des CSS. J'ai voulu garder le même thème spatial que dans le jeu, avec un fond étoilé fait uniquement en CSS (pas d'images) :

CSS du fond étoilé (style.css) :

```
body {
    background: linear-gradient(to bottom, #0a0e1a 0%, #1a1f2e 100%);
    min-height: 100vh;
    color: white;
    font-family: Arial, sans-serif;
}

/* Étoiles générées en CSS pur */
.stars {
    position: fixed;
    width: 100%;
    height: 100%;
    background-image:
        radial-gradient(2px 2px at 20px 30px, white, transparent),
        radial-gradient(2px 2px at 40px 70px, rgba(255,255,255,0.8),
transparent),
        radial-gradient(1px 1px at 90px 40px, white, transparent),
```

```
        radial-gradient(2px 2px at 160px 120px, rgba(255,255,255,0.9),
transparent);
    background-repeat: repeat;
    background-size: 200px 200px;
    animation: stars-move 100s linear infinite;
}

@keyframes stars-move {
    from { transform: translateY(0); }
    to { transform: translateY(-200px); }
}
```

5. Fonctionnalités implémentées

Voici les fonctionnalités que nous avons réussi à implémenter jusqu'à cette première soutenance, avec des captures d'écran.

5.1 Menu principal

Le menu principal est la première chose que voit le joueur quand il lance le jeu. Il propose quatre options : Nouvelle Partie, Rejoindre, Options et Quitter. La navigation fonctionne à la fois à la souris (survol et clic) et au clavier (flèches et Entrée).

Figure 1 : Menu principal du jeu

5.2 Création de partie

Quand le joueur clique sur "Nouvelle Partie", il arrive sur l'écran de création où il peut entrer son pseudo et choisir son rôle (Concepteur ou Pragmatique). L'adresse IP du serveur (récupéré par notre programme) s'affiche pour qu'il puisse la communiquer à son partenaire (et ainsi générer la possibilité de jouer à 2).

Figure 2 : Écran de création de partie

5.3 Rejoindre une partie

Le deuxième joueur utilise "Rejoindre" pour se connecter. Il doit entrer l'adresse IP du serveur et choisir son pseudo et son rôle.

Figure 3 : Écran pour rejoindre une partie

Figure 4 : Saisie des informations de connexion

5.4 Connexion réseau

Une fois que le premier joueur a créé la partie, un écran d'attente s'affiche jusqu'à ce que le deuxième joueur se connecte. Quand la connexion est établie, un écran de chargement s'affiche puis la partie commence.

Figure 5 : Écran d'attente de connexion

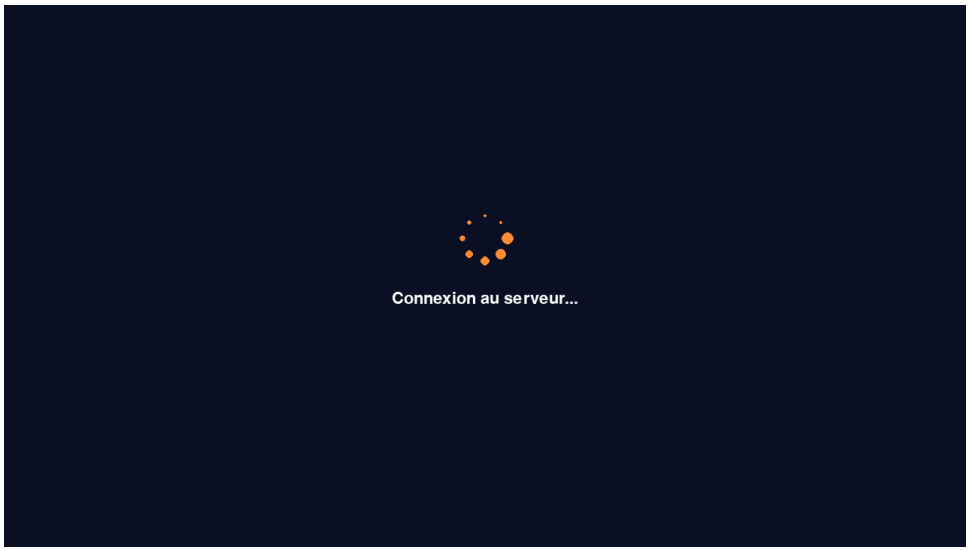

Figure 6 : Écran de chargement

5.5 Menu des options

Le menu des options permet de régler le volume de la musique et des effets sonores avec des sliders. Les contrôles du jeu (ZQSD ou flèches) sont aussi rappelés.

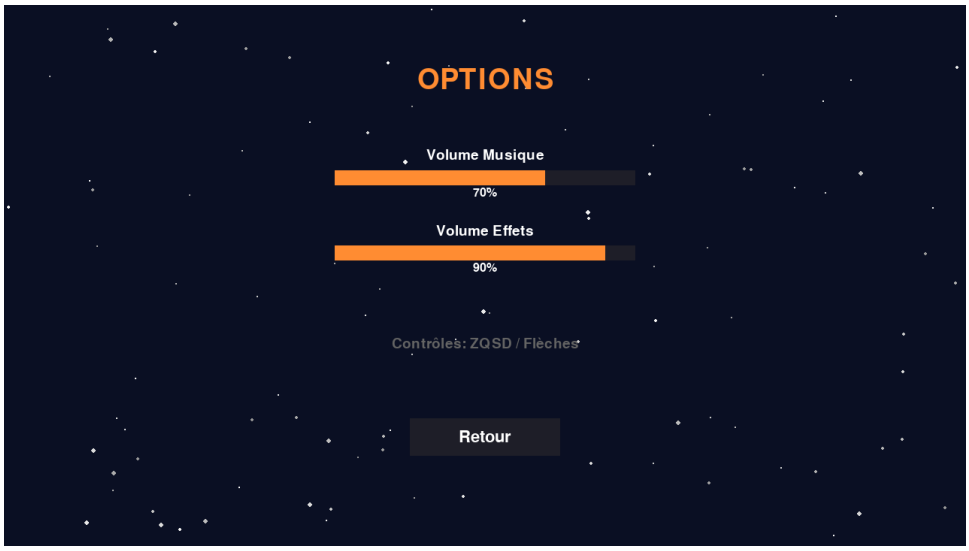

Figure 7 : Menu des options avec réglages audio

5.6 Écrans de fin de partie

Nous avons préparé les écrans de victoire et de défaite qui s'afficheront à la fin de la partie. L'écran de victoire apparaît si les joueurs réparent les 8 canons à temps, l'écran de défaite si le timer arrive à zéro sans que les canons soient réparés.

Figure 8 : Écran de victoire

Figure 9 : Écran de défaite

6. Difficultés rencontrées

Ce projet a été nous avons rencontré plusieurs difficultés.

Voici les principales difficultés que nous avons dû surmonter.

6.1 Le temps limité avec les cours

La plus grande difficulté a été de trouver du temps pour travailler sur le projet. Entre les cours, les TP, les QCM du lundi matin qu'il faut bien préparer le weekend, c'était compliqué de se coordonner. On a souvent dû se retrouver Ile soir pour avancer.

Au début, on pensait pouvoir avancer plus vite, mais on s'est vite rendu compte que ce serait difficile. On a dû revoir nos objectifs et se concentrer sur les fonctionnalités essentielles pour assurer cette première soutenance.

6.2 Notre manque d'expérience en développement

C'est notre premier gros projet de programmation pour nous 4. Avant ça, on avait fait quelques exercices en cours mais jamais quelque chose d'aussi complet et complexe. On a dû apprendre beaucoup de choses en même temps : PyGame, les sockets, le threading, Git...

Par exemple, au début on ne savait pas du tout comment organiser notre code. On mettait tout dans un seul fichier et c'était vite le bazar. C'est Ilian qui a proposé de séparer en plusieurs modules, et ça a vraiment amélioré les choses.Le fait d'organiser nos développement par thème nous a permis de mieux organiser notre travail.

6.3 La complexité du réseau et du threading

Le module réseau a été de loin le plus difficile. Comprendre comment fonctionnent les sockets TCP, gérer les connexions, et surtout le threading... On a passé des heures à chercher pourquoi le jeu se figeait quand on attendait une connexion.

Le problème c'est que quand on fait socket.accept() ou recv(), le programme attend et ne fait plus rien d'autre. Du coup l'écran ne se rafraîchit plus et ça donne l'impression que le jeu a planté. Il a fallu apprendre le threading pour résoudre ça.

6.4 Git et le travail collaboratif

Git nous a posé pas mal de problèmes au début. On a eu plusieurs fois des fichiers écrasés ou des modifications perdues.

6.5 Les recherches sur internet

Heureusement, internet a été une ressource précieuse. On a trouvé beaucoup de tutoriels et d'exemples de code. Le site Real Python pour les sockets, Red Blob Games pour l'algorithme A*, et plein de vidéos YouTube pour PyGame.

Parfois c'était frustrant de ne pas trouver exactement ce qu'on cherchait, ou de tomber sur des exemples en anglais difficiles à comprendre. Mais au final, ces recherches nous ont beaucoup appris.

6.6 Les satisfactions

Malgré les difficultés, il y a eu des moments vraiment satisfaisants. La première fois qu'on a vu le menu principal s'afficher avec le fond étoilé, c'était vraiment rassurant pour nous. Et quand la connexion réseau a marché pour la première fois entre deux ordis, on était super contents.

Ces avancées nous ont motivés à continuer. C'est vraiment bien de voir que le code qu'on a écrit fonctionne et produit quelque chose de concret.

7. État d'avancement du projet

Voici un récapitulatif de l'état d'avancement de chaque fonctionnalité du projet :

Fonctionnalité	Avancement	Statut
Concept et scénario du jeu	80%	Bien avancé
Menus et interface utilisateur	70%	Bien avancé
Réseau (connexion TCP)	30%	En cours
Audio (musique et effets)	6%	En cours
Gameplay (déplacements, canons)	10%	À faire
IA des ennemis (A*)	5%	À faire

Au total, nous estimons avoir réalisé environ 35% du projet. C'est moins que ce qu'on espérait au départ, mais les fondations sont solides et le reste devrait avancer plus vite maintenant que l'architecture est en place.

8. Perspectives et travaux futurs

Pour la soutenance 2 (prévue en mars 2026), voici les fonctionnalités que nous prévoyons d'implémenter :

8.1 Gameplay de base

- Déplacements des joueurs sur la carte avec ZQSD et détection des collisions
- Placement des 8 canons sur la carte et système de réparation
- Timer fonctionnel de 10 minutes avec affichage à l'écran
- HUD avec vie des joueurs et compteur de canons

> ***HUD signifie "Heads-Up Display" (affichage tête haute en français). Dans un jeu vidéo, c'est l'interface graphique qui s'affiche par-dessus le jeu pendant qu'on joue avec les infos comme les joueurs / le timer / le score ...***

8.2 Synchronisation réseau

- Synchronisation des positions des joueurs en temps réel
- Synchronisation de l'état des canons (réparé/cassé)
- Gestion de la déconnexion propre avec message d'erreur

8.3 Intelligence artificielle

- Implémentation de l'algorithme A*
- Comportements des ennemis : patrouille, poursuite, attaque
- Spawn d'ennemis progressif selon l'avancement de la partie

8.4 Audio

- Recherche et intégration de fichiers audio libres de droits
- Musique d'ambiance spatiale et effets sonores (réparation, dégâts, etc.)

9. Conclusion

Ce premier semestre de travail sur ORBITE ZÉRO a été une expérience très enrichissante pour notre équipe. Malgré les difficultés rencontrées, nous avons réussi à poser des fondations pour notre jeu. Même si on ne voit pas grand chose du jeu encore ...

Ce que nous avons appris

Au-delà du code, ce projet nous a appris à travailler en équipe sur un projet informatique. On a découvert l'importance de bien communiquer, de planifier le travail. On a aussi appris à utiliser des outils professionnels comme Git.

Sur le plan technique, on a acquis des compétences en Python, PyGame, programmation réseau et conception d'interface. Ces connaissances nous seront utiles pour la suite de nos études.

Ce qui reste à faire

Il reste encore beaucoup de travail : le gameplay complet, la synchronisation réseau en jeu, l'IA des ennemis, l'audio... Mais maintenant que nos fondations sont en place, on pense pouvoir avancer plus vite.

Nous sommes motivés pour la suite et confiants dans notre capacité à finier un jeu pour la soutenance finale en mai 2026.

Remerciements

Nous tenons à remercier tout ceux qui nous ont ont parfois aidés quand on bloquait sur des problèmes. Et merci à internet pour tous les tutoriels qui nous ont sauvé la vie.

10. Annexes - Captures d'écran supplémentaires

Voici l'ensemble des captures d'écran de notre interface de jeu.

10.1 Simulation de partie

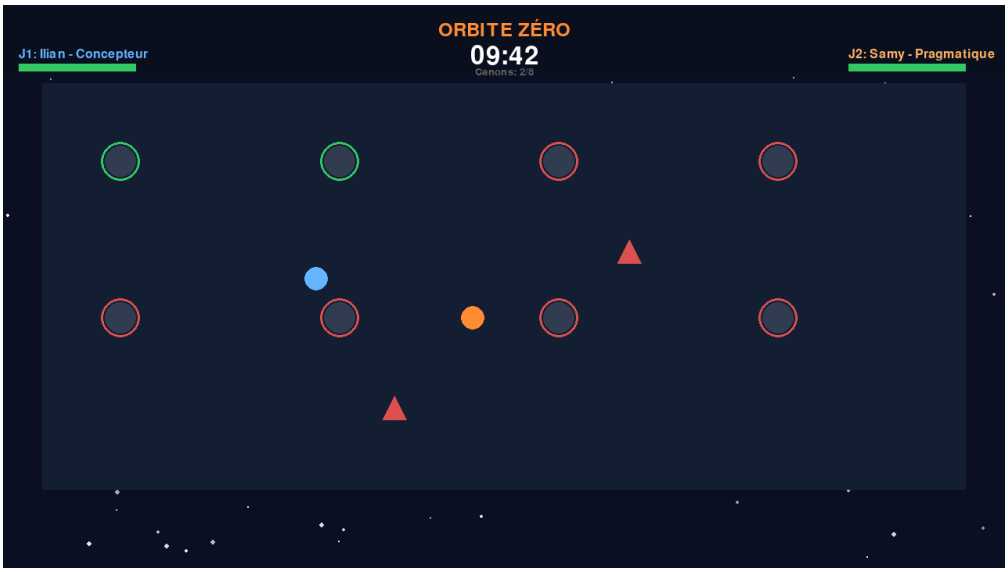

Figure 10 : Simulation de l'écran de jeu (maquette)

Cette capture montre une simulation de ce à quoi ressemblera l'écran de jeu une fois le gameplay implémenté. On y voit les deux joueurs, les canons à réparer et le timer.

10.2 Récapitulatif de l'interface

Notre interface comprend au total 10 écrans différents, tous avec le même thème visuel spatial (fond bleu foncé avec étoiles, couleurs orange et blanc). La navigation est fluide et fonctionne au clavier comme à la souris.

Le code de l'interface représente environ 2500 lignes de Python, réparties entre les composants (Bouton, ChampTexte) et les différents menus ou traitelmens (MenuPrincipal, MenuCreation, MenuRejoindre, MenuOptions, structures de données, main, réseau...).