RAPPORT DE PROJET : SPACE INVADERS

Auteurs : Fabien Piat (Groupe 35)i

Cours : C Avancé (Licence 3)

Date : Janvier 2026

---

1. Introduction

Ce projet a pour objectif la conception et le développement d'une réplique du jeu d'arcade *Space Invaders* en langage C. L'enjeu principal ne résidait pas uniquement dans la logique du jeu, mais dans la mise en œuvre d'une architecture logicielle rigoureuse, portable et performante.

Le cahier des charges imposait une contrainte technique forte : la capacité de proposer deux modes d'affichage distincts et commutables à la volée : une vue graphique moderne (utilisant la librairie **SDL3**) et une vue textuelle rétro (utilisant la librairie **Ncurses**). Le projet devait également inclure la gestion du son, un système de sauvegarde persistant et une gestion irréprochable de la mémoire.

---

2. Architecture Logicielle

Pour répondre aux exigences de modularité et de maintenabilité, nous avons opté pour le patron de conception **MVC (Modèle-Vue-Contrôleur)**. Cette séparation stricte des responsabilités permet de modifier l'affichage ou les règles du jeu indépendamment l'un de l'autre.

2.1. Le Modèle (src/model.c)

Le modèle est le "cerveau" du jeu. Il encapsule l'état complet de la partie dans une structure centrale GameState.

- **Indépendance Totale :** Le modèle ne contient aucune référence aux librairies graphiques (ni SDL, ni Ncurses). Il manipule des coordonnées abstraites.
- **Logique Métier :** Il gère les déplacements, la détection des collisions (Hitbox AABB), l'intelligence artificielle des ennemis (descente et accélération), et la gestion des scores.
- **Persistance :** Les fonctions de sauvegarde (model_save_slot) sérialisent directement la structure GameState en binaire, permettant une sauvegarde exacte et rapide de l'état du jeu.

2.2. Les Vues (src/view_sdl.c & src/view_ncurses.c)

Les vues sont responsables de la représentation visuelle du modèle. Le projet implémente un polymorphisme statique :

- **Vue Graphique (SDL3) :** Utilise l'accélération matérielle (ou logicielle selon le driver) pour dessiner des sprites. Une particularité de notre implémentation est le **rendu procédural** : les sprites ne sont pas des fichiers images chargés (.png), mais des tableaux de bits définis dans le code et dessinés pixel par pixel (SDL_RenderFillRect), offrant un style "Pixel Art" authentique sans dépendances externes.
- **Vue Textuelle (Ncurses) :** Adapte les coordonnées du modèle (1000x600) à la grille de caractères du terminal (ex: 80x24) via un facteur d'échelle dynamique.

2.3. Le Contrôleur (src/controller.c)

Le contrôleur est le chef d'orchestre. Il contient la boucle principale (while(1)).

- **Gestion du Temps :** Il assure une vitesse de jeu constante indépendamment de la puissance du processeur.
- **Routage des Entrées :** Il interroge la vue active pour obtenir les commandes utilisateur (Clavier/Souris) et les transmet au modèle.
- **Gestion des États :** Il pilote les transitions entre le Menu (Launcher), le Jeu, la Pause et le Game Over.

---

3. Décisions de Conception et Choix Techniques

3.1. Gestion Audio "Native" (Sans SDL_mixer)

Initialement, nous avons envisagé d'utiliser SDL_mixer. Cependant, face aux problèmes de dépendances sur certaines machines (paquets manquants sous WSL), nous avons pris la décision de développer notre **propre moteur audio** basé sur SDL_AudioStream (SDL3 natif).

- **Avantage :** Aucune dépendance externe. Le projet compile partout où SDL3 est présent.
- **Fonctionnement :** Nous chargeons les fichiers WAV bruts, les convertissons au format de la carte son au démarrage, et les injectons directement dans le flux audio.

3.2. Portabilité via 3rdParty

Pour éviter les problèmes de versions de librairies installées sur les machines de l'université ou personnelles, nous avons inclus la librairie SDL3 compilée localement dans le dossier 3rdParty/ et configuré le Makefile avec rpath.

- **Résultat :** L'exécutable cherche la librairie dans son propre dossier, garantissant que le jeu tourne même si SDL3 n'est pas installé sur le système hôte.

3.3. Système de Sauvegarde Binaire

Plutôt que d'utiliser un format texte (JSON/XML) lourd à parser, nous avons choisi la sérialisation binaire brute (fwrite de la struct GameState).

- **Justification :** C'est la méthode la plus performante en C. Elle permet de sauvegarder l'état exact (position au pixel près, timers des explosions) instantanément.

---

4. Difficultés Rencontrées et Solutions

4.1. Compatibilité Binaire (GLIBC)

Problème : L'exécutable compilé sur un PC récent (Ubuntu 22.04) refusait de se lancer sur un PC plus ancien (Debian/WSL) avec une erreur de version GLIBC.

Solution : Nous avons mis en place une procédure de recompilation complète via le Makefile (make clean && make). Nous avons également appris qu'il est préférable de fournir le code source de la librairie tierce plutôt que les binaires seuls.

4.2. Rendu Graphique sous WSL (Windows Subsystem for Linux)

Problème : Sur certaines machines Windows utilisant WSL, l'accélération matérielle (GPU) provoquait des erreurs de segmentation ou des écrans noirs via SDL3.

Solution : Implémentation d'une commande de secours make run-soft qui force la variable d'environnement SDL_RENDER_DRIVER=software. Cela oblige le jeu à utiliser le processeur pour le dessin, garantissant une stabilité à 100% au prix d'une légère baisse de fluidité visuelle (artefacts de grille).

4.3. Latence Audio (Audio Lag)

Problème : Lors des ralentissements du jeu, les sons s'empilaient dans la file d'attente et étaient joués avec plusieurs secondes de retard.

Solution : Création d'un système "Anti-Lag" dans audio.c. Avant de jouer un son, nous vérifions la taille du buffer en attente. Si elle dépasse 0.1 seconde, le nouveau son est ignoré pour préserver la synchronisation visuelle/sonore.

---

5. Tests et Validation

Le projet a subi plusieurs phases de tests :

1. **Tests de Fuites Mémoire (Valgrind) :**
    - Utilisation intensive de valgrind --leak-check=full.
    - **Résultat :** Le projet final ne présente **aucune fuite de mémoire** (0 bytes lost). Toutes les allocations (textures, audio, fenêtres) sont correctement libérées à la fermeture.

2. **Tests de Robustesse (Input Fuzzing) :**
    - Appui simultané sur toutes les touches, changements rapides de vue (SDL <-> Ncurses).
    - Le jeu reste stable et ne plante pas.
3. **Tests de Cross-Compilation :**
    - Validé sur une machine native Linux (Ubuntu).
    - Validé sur Windows via WSL2 (Ubuntu).

---

6. Conclusion

Ce projet nous a permis de consolider nos connaissances en programmation C bas niveau. Nous avons appris à gérer la mémoire manuellement, à structurer un projet complexe via des Makefiles, et à résoudre des problèmes d'incompatibilité système concrets.

Le résultat est un jeu fonctionnel, amusant, et techniquement solide, respectant l'ensemble des contraintes imposées. L'architecture modulaire mise en place permettrait facilement d'ajouter de nouvelles fonctionnalités (nouveaux ennemis, boss, bonus) sans remettre en cause la structure existante.