Acting Shooting Star : Notre version de Crossy Road basé sur le modèle des acteurs

Enzo Picarel, Raphaël Bely, Arno Donias, Thibault Abeille
Encadrants : Vincent Alba, David Renault
ENSEIRB-MATMECA – 2025

6 mai 2025

Table des matières

1 Introduction

Dans le cadre de notre projet de programmation, nous avons choisi de recréer une version personnalisée du jeu *Crossy Road*, un jeu mobile populaire de notre enfance. Ce choix s'est imposé naturellement : d'une part pour son aspect ludique et familier, et d'autre part parce que ses mécaniques simples et répétitives se prêtaient bien à une première implémentation de projet en JavaScript (ou ici, plus précisément en TypeScript).

L'objectif principal du projet était de concevoir un moteur de jeu basé sur le **modèle des acteurs**, dans lequel chaque élément du jeu (joueur, voitures, rivières, etc.) est représenté par un acteur autonome. Chaque acteur devait pouvoir recevoir des messages, réagir en fonction de son état, et potentiellement émettre de nouveaux messages à d'autres acteurs. Une attention particulière a été portée à la **pureté fonctionnelle** de l'implémentation, en évitant les effets de bord et en respectant les principes de la programmation fonctionnelle autant que possible.

Le projet s'inscrivait dans un cadre pédagogique défini, avec des contraintes précises : utilisation de `TypeScript`, affichage dans le *terminal* via le *terminal-kit*, validation du code avec `ESLint`, tests unitaires avec `Jest`, et développement collaboratif via un dépôt `git`. Le code devait également respecter une structure imposée du dépôt.

Parmi les axes de développement retenus, on peut citer la gestion des acteurs et de leurs interactions, la génération dynamique du monde de jeu avec une difficulté croissante, ainsi que des fonctionnalités plus avancées comme le **retour dans le temps**. Enfin, ce projet s'insérait dans une démarche académique plus large, en lien avec un enseignement sur **la programmation fonctionnelle, TypeScript et l'analyse statique de code**.

2 Modèle des acteurs et principes de conception

Notre moteur de jeu repose entièrement sur une architecture fondée sur le **modèle des acteurs**. Chaque acteur est créé à l'aide d'une fonction `make_actor` (définie dans `actor.ts`) à partir d'un nom et d'une position initiale sur l'écran. Un acteur possède :

- une **boîte aux lettres** pour recevoir des messages,
- une **fonction** `update` qui lit ces messages et renvoie une nouvelle version de l'acteur après traitement,
- un ensemble de fonctions génériques (`move`, `collide`, `tick`, etc.) partagées par tous les acteurs.

Les **messages** échangés entre acteurs sont typés par une structure contenant un champ `type` (chaîne de caractères) et un champ `params` (tableau d'arguments). Ces messages représentent des actions à effectuer, et permettent aux acteurs de réagir de manière autonome aux événements du jeu.

Le modèle repose sur une **mise à jour transactionnelle** : lorsqu'un acteur reçoit un `tick`, il ne modifie pas son propre état mais crée une nouvelle version de lui-même en fonction des messages reçus. Cette approche garantit l'absence d'effets de bord et assure un comportement déterministe.

L'ensemble du jeu est piloté par un **runtime** composé de plusieurs boucles d'exécution indépendantes (`setInterval`), telles que `intervalProj`, `tickInterval` ou `carInterval`, qui déclenchent des `tick` à des fréquences différentes selon le type d'acteur. Lorsqu'un tick survient, les acteurs sont parcourus selon l'ordre du tableau dans lequel ils sont stockés, et mis à jour en cascade. Les messages sont envoyés directement d'un acteur à un autre, sans passer par une file globale.

Un mécanisme de sauvegarde de l'état du monde a également été mis en place. Chaque version du monde est enregistrée dans une file, ce qui permet une forme de **retour en arrière**. Le monde courant correspond au dernier élément de cette file ; pour revenir en arrière, il suffit de replacer cet élément par le premier de la file, ce qui revient à restaurer un état antérieur du jeu.

Bien que ce modèle ait offert une approche intuitive et modulaire pour gérer les interactions entre les entités du jeu, il a également introduit une certaine **rigidité**. La recréation fréquente d'acteurs, imposée par la pureté fonctionnelle, a parfois généré des bugs graphiques et une complexité de gestion accrue. Néanmoins, cette architecture s'est révélée pertinente pour développer des comportements réactifs, évitant la prolifération de structures conditionnelles complexes.

3 Architecture du jeu

L'architecture du jeu repose sur la modélisation de toutes les entités visibles sous forme d'acteurs. On distingue plusieurs types d'acteurs :

- **Chicken** : l'acteur principal contrôlé par le joueur ;
- **Tree**, **Water_L**, **Water_R**, **Log_L**, **Log_R**, **Car_L**, **Car_R** : éléments dynamiques ou statiques constituant les obstacles ;
- **Projectile** : acteur créé par le joueur pour détruire certains obstacles ;
- **Empty** : utilisé pour représenter une cellule vide dans le monde.

Tous les acteurs partagent un ensemble de fonctions communes, comme `tick`, `move` ou `update`. Une fonction `collide` était initialement prévue mais n'est finalement pas utilisée. La détection des collisions se fait par simple comparaison des positions avec l'acteur principal. Si un acteur occupe la même case que le joueur, la partie est terminée.

L'identité d'un acteur est déterminée à sa création par la fonction `make_actor`, à partir d'un `nom` et d'une `position`. Ces éléments sont suffisants pour distinguer les comportements spécifiques.

Le monde est représenté comme une structure contenant un tableau de lignes (`Line[]`), chacune de ces lignes étant composée d'acteurs. La progression du jeu se fait par le défilement automatique de ces lignes, simulant l'avancement du joueur. Ce système permet de maintenir une tension continue : si le joueur avance rapidement, le monde suit son rythme ; s'il reste en bas de l'écran, le défilement automatique reprend.

Les lignes sont générées dynamiquement via la fonction `tick_line`, à chaque avancée du monde. Elles sont construites à partir de motifs d'obstacles (2 à 4 cases consécutives), ce qui permet de représenter visuellement des camions, rangées d'arbres ou bûches flottantes. La probabilité d'apparition des obstacles augmente progressivement, suivant une logique de difficulté croissante. Les motifs sont choisis et combinés aléatoirement, ce qui rend peu probable la répétition exacte de deux lignes.

Aucun système ne garantit qu'un chemin praticable existe toujours, mais la rareté des impasses et la possibilité pour le joueur de tirer des projectiles destructeurs offrent un équilibre entre hasard et jouabilité. Les projectiles, comme tous les autres éléments mobiles, sont des acteurs à part entière. Le déplacement dans le monde se fait par recréation successive d'acteurs dans de nouvelles positions, selon une logique purement fonctionnelle.

Enfin, toutes les collisions avec des obstacles sont létales : la gestion des règles est donc simple mais rigoureuse, renforçant l'aspect arcade et punitif du jeu.

4 Implémentation technique

Le projet est organisé selon une structure claire et modulaire. Le répertoire principal contient les fichiers de configuration nécessaires pour l'utilisation de `ESLint`, `Jest`, ainsi qu'un `Makefile` destiné à automatiser les différentes tâches de compilation, d'exécution et de tests.

Organisation des fichiers

- **src**/ : contient les fichiers source principaux :
  - `actor.ts` : implémente les fonctions de création et de gestion des acteurs ;
  - `world.ts` : contient la logique de simulation du monde, les ticks, le moteur du jeu et l'affichage terminal.
- **test**/ : contient les tests unitaires, notamment `actor.test.ts` qui vérifie le bon fonctionnement des fonctions définies dans `actor.ts`.
- **dist**/ : contient les fichiers JavaScript générés automatiquement à la compilation.

Technologies et outils utilisés

Le développement repose sur plusieurs outils modernes :
- `TypeScript` pour garantir un typage statique des données ;
- `ESLint` pour assurer une qualité de code homogène (indentation, style, bonne utilisation des structures) ;
- `Jest` pour l'écriture et l'exécution des tests unitaires ;
- `terminal-kit` pour l'affichage dynamique dans le terminal.

La configuration TypeScript (`tsconfig.json`) et le `Makefile` étaient fournis dans le sujet. Le Makefile propose notamment une commande `make run` qui compile le projet avec `tsc` puis exécute le jeu via Node.js :

```
npx tsc
node dist/src/world.js
```

Le fichier principal d'exécution est `world.ts`, qui gère l'ensemble de la boucle de jeu.

Le jeu est entièrement jouable depuis un terminal, sans interface graphique additionnelle. L'affichage s'adapte dynamiquement à la taille de l'écran du terminal, ce qui permet une bonne ergonomie malgré l'usage de caractères ASCII.

Éléments clés du code

Trois éléments de l'implémentation méritent une attention particulière :
- **Le système de tick** : chaque type d'acteur est mis à jour à une fréquence spécifique via des fonctions `setInterval`. Cela permet de différencier la cadence de déplacement des voitures, bûches, projectiles, etc.
- **La génération d'acteurs** : la fonction `make_actor` crée des entités selon leur type, leur position et leur comportement à partir de modèles réutilisables.
- **La génération de lignes** : des motifs prédéfinis (pattern) permettent de générer dynamiquement des obstacles selon une probabilité qui augmente avec la progression du joueur. Cela assure un équilibre entre imprévisibilité et jouabilité.

5 Affichage et interactions

— Affichage ASCII dans le terminal
— Contraintes liées à `terminal-kit`
— Possibilités d'interactions (touches clavier, etc.)

6 Difficultés rencontrées et solutions apportées

— Collisions sans effets de bord
— Identification des acteurs et passage de tick
— Débogage et visualisation du comportement

7 Évaluation

— Fonctionnalité et stabilité
— Performances en terminal
— Extensibilité du moteur de jeu

8 Conclusion et perspectives

Résumé des apports du projet, limites actuelles, et pistes d'amélioration (graphismes, IA, multijoueur, temps réel).

Références

— Documentation de `terminal-kit` : `https://github.com/cronvel/terminal-kit`
— Tutoriel sur le modèle des acteurs
— Documentation ESLint / Jest / TypeScript