

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

11 mai 2025

Table des matières

1	Introduction	2
2	Modèle des acteurs et principes de conception	2
3	Architecture du jeu	3
4	Implémentation technique	4
5	Affichage et interactions	5
6	Difficultés rencontrées et solutions apportées	5
7	Évaluation	5
8	Conclusion et perspectives	5

1 Introduction

Dans le cadre de notre projet de programmation en JavaScript, nous avons fait le choix de créer notre propre version du jeu Crossy Road. C'est un jeu mobile populaire auquel on a joué durant notre enfance. Ce fut un choix naturel. En effet, son aspect ludique et familier permet une certaine simplicité d'implémentation qui permet de satisfaire les exigences pour un premier projet dans ce langage. Nous verrons par la suite, que, grâce à notre imagination, nous n'avons pas été restreints par la simplicité du jeu.

Le projet est axé principalement sur la conception d'un moteur de jeu basé sur un modèle fonctionnant par interaction des objets définis en tant qu'"acteur". Ces acteurs (qui vont du poulet à la bûche en passant par les voitures) sont indépendants les uns des autres. Chaque acteur devait être capable de recevoir des informations sous forme de messages, puis de traiter ces messages et enfin de réagir en fonction. Pendant l'implémentation, nous avons fait particulièrement attention à avoir un code avec une pureté fonctionnelle maximale, en évitant les effets de bord et en respectant les principes de la programmation fonctionnelle autant que possible.

Le projet était cadré par des contraintes définies en relation avec le cours de programmation fonctionnelle et d'utilisation de *TypeScript*. Les exigences étaient les suivantes :

- affichage dans le terminal même à l'aide de *terminal-kit*
- code validé grâce à l'utilisation d'*ESLint*
- tests unitaires avec *Jest*
- développement collaboratif avec un dépôt git
- code structuré selon les consignes

Durant ce rapport nous aborderons plusieurs thèmes principaux du développement comme la conception des acteurs, l'architecture du jeu, son implémentation, ainsi qu'un retour sur expérience contenant les difficultés rencontrées et nos apprentissages durant le projet.

2 Modèle des acteurs et principes de conception

L'architecture complète du jeu est basée entièrement sur un modèle comportant des acteurs. Ces acteurs font office de **structures abstraites de données**. En effet, leur implémentation est cachée derrière le type **Actor** défini dans le fichier *actor.ts* et généré à l'aide de la fonction *make_actor*. Les acteurs doivent posséder :

- une **mailbox** afin de recevoir des messages ou instructions ;
- une fonction centrale *update* qui lit les messages et renvoie une nouvelle version de l'acteur après avoir effectué les changements propres aux instructions lues ;
- plusieurs fonctions génériques comme *move*, *collide* ou *tick* communes à tous les acteurs de notre jeu ;

Les messages échangés par les acteurs sont typés par une structure composé d'un champ **type** (chaîne de caractères) et un champ **params** (tableau d'arguments). Ces messages permettent la transmission des différentes actions à effectuer, faisant réagir les acteurs de manière **autonome** et **indépendante** durant le déroulement du jeu.

Le modèle repose sur la mise à jour successive et répétée des acteurs : quand l'un d'eux reçoit un "**tick**", il ne modifie pas vraiment son propre état, à la place, il crée une

nouvelle version de lui-même en fonction des messages/instructions reçues.

Le fonctionnement du jeu est dirigé par un **runtime** comportant plusieurs boucles d'exécution indépendantes (**setinterval**) en fonction des acteurs, telles que **intervalProj**, **tickInterval** ou **carInterval**. Ce sont elles qui déclenchent des tick à fréquences différentes selon l'acteur impacté.

Le monde contenant tout les acteurs est également sauvegarder régulièrement. Les versions du mondes sont enregistrées dans une file qui permet alors le **retour en arrière**. Le monde actuel correspond au dernier élément de la file, ainsi afin de revenir en arrière, il faut remplacer cet élément par un élément placé précédemment dans la file, ce qui permet de revenir dans 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 :

```
1 npx tsc
2 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