Documentation Technique

pour le Projet de Jeu Connect Four

Par : **Cédric MAS**
Thomas LEPAGE
David KIZAYILAWOKO

20 octobre 2025

Sommaire

1 Introduction

1.1 Présentation du jeu

Le projet est une implémentation du jeu classique Connect Four (Puissance 4), un jeu de stratégie pour deux joueurs où l'objectif est d'aligner quatre pièces de la même couleur horizontalement, verticalement ou en diagonale sur un plateau de 6 lignes et 7 colonnes. Le jeu est développé en TypeScript avec React pour l'interface utilisateur et utilise une classe Engine pour gérer la logique du jeu. Les joueurs alternent les tours en plaçant des pièces rouges ou jaunes dans les colonnes. Le jeu détecte les victoires, les matchs nuls et permet de réinitialiser la partie.

1.2 Objectifs de la documentation technique

Cette documentation vise à décrire l'architecture technique du projet, les mécaniques de jeu, les choix d'implémentation, les optimisations, la structure du code et les aspects de déploiement. Elle s'appuie sur les fichiers fournis : engine.test.ts (tests unitaires pour l'engine), index.ts (classe Engine), App.test.tsx (tests pour l'interface), App.tsx (composant React principal), ainsi que des exemples de plans (test, cahier des charges, documentation technique) pour structurer le contenu.

2 Architecture du jeu

2.1 Moteur de jeu

Le moteur de jeu est encapsulé dans la classe `Engine` définie dans `index.ts`. Elle gère :

- Le plateau (`board`) : un tableau 2D de cellules (`Cell`) de 6x7, initialisé avec des valeurs nulles.
- Le joueur courant (`currentPlayer`) : alternance entre `Player.Red` (0) et `Player.Yellow` (1).
- Le compteur de tours (`turn`).
- Les méthodes principales : `setPiece` pour placer une pièce dans une colonne, et `checkWin` pour vérifier les alignements de quatre pièces.

Le constructeur permet d'initialiser un nouveau jeu ou de restaurer un état existant.

2.2 Système de rendu

Le rendu est géré par React dans `App.tsx`. Le composant `App` utilise des hooks d'état (`useState`) pour :

- Stocker l'instance de l'engine.
- Gérer l'état du jeu (gagnant, match nul).
- Suivre la colonne survolée pour l'aperçu de pièce.

Le plateau est rendu comme une grille CSS (`grid grid-cols-7`) avec des divs rondes pour les cellules, colorées en fonction de `cell.color` (rouge, jaune ou blanc).

2.3 Gestion des ressources (graphismes, sons, etc.)

Les ressources sont minimales :

- Graphismes : Styles CSS pour les pièces (classes comme `bg-red-500`, `bg-yellow-500` pour les couleurs, `rounded-full` pour la forme).
- Pas de sons ou de ressources externes (images, musiques) ; tout est généré via CSS.
- Composants UI importés depuis `./components/ui/*` (Button, Card, Alert) pour une interface Shadcn-like.

3 Conception du gameplay

3.1 Mécaniques de jeu

- Placement des pièces : Via `setPiece(column, player)`, qui trouve la ligne la plus basse disponible dans la colonne.
- Détection de victoire : `checkWin(cell)` vérifie les alignements dans quatre directions opposées (horizontal, vertical, diagonal, anti-diagonal) en comptant les pièces connectées (au moins 4).

- Match nul : Détecté quand `turn >= 41` (plateau plein sans victoire).
- Alternance des joueurs : Mise à jour de `currentPlayer` après chaque placement valide.

3.2 Systèmes de contrôle

- Entrées utilisateur : Clic sur une cellule ou en-tête de colonne pour placer une pièce (`handleColumnClick`).
- Survol : `handleMouseEnter` et `handleMouseLeave` pour afficher un aperçu de la pièce au-dessus de la colonne (div positionnée absolument).
- Réinitialisation : Bouton « Reset Game » appelant `handleReset` pour recréer une nouvelle `Engine`.

3.3 Interactions joueur-environnement

- Le joueur interagit avec le plateau via la souris (hover pour preview, clic pour placement).
- Affichage : Texte indiquant le joueur courant, alerte pour victoire ou nul.
- Pas d'interactions complexes comme physique ou environnement dynamique ; c'est un jeu statique en grille.

4 Graphismes et Animation

4.1 Modélisation 3D

Non applicable. Le jeu est en 2D, représenté par une grille de divs CSS sans modélisation 3D.

4.2 Textures et shaders

Non applicable. Les « textures » sont des classes CSS simples pour les couleurs de fond. Pas de shaders (pas de WebGL).

4.3 Animation des personnages et objets

Minimal : Pas d'animations explicites. Le placement de pièce est instantané. L'aperçu sur hover apparaît/disparaît via état React, sans transitions CSS visibles dans le code fourni.

5 Son et Musique

5.1 Effets sonores

Non implémentés. Pas d'effets sonores pour les placements, victoires, etc.

5.2 Musique d'ambiance

Non implémentée.

5.3 Intégration sonore dans le gameplay

Non applicable.

6 Optimisation et Performance

6.1 Gestion des ressources système

Le jeu est léger : Pas de consommation intensive de CPU/GPU. React gère les re-rendus efficacement via états.

6.2 Optimisation du code

- Code modulaire : Logique séparée (Engine pour backend, App pour frontend).
- Évitement des boucles inutiles dans `checkWin` (limité à 4 itérations max par direction).
- Pas de fuites mémoire visibles ; états React bien gérés.

6.3 Tests de performance

Non spécifiés dans les fichiers, mais le jeu étant simple (42 cellules), les performances sont intrinsèquement bonnes pour les navigateurs modernes.

7 Documentation du Code

7.1 Structure du code source

- `index.ts` : Définit enums (`Player`, `Direction`), interfaces (`Cell`), types (`Board`), et la classe `Engine` avec méthodes privées/publiques.
- `engine.test.ts` : Tests Vitest pour `setPiece` (placement, colonnes pleines/invalides) et `checkWin` (directions variées).
- `App.tsx` : Composant React principal avec états, handlers, et rendu JSX.
- `App.test.tsx` : Tests pour rendu initial, interactions (hover, clic, reset), utilisant mocks pour Engine.

7.2 Commentaires et documentation interne

- Commentaires dans `index.ts` : Descriptions pour enums, méthodes (e.g., « Places a piece in the specified column »).
- Pas de JSDoc étendu, mais code clair et auto-descriptif.

7.3 Bonnes pratiques de codage

- Utilisation de TypeScript pour typage strict.
- Immutabilité partielle : Nouvelle instance d'Engine créée pour mises à jour d'état.
- Tests unitaires couvrant cas nominaux et limites (e.g., colonne pleine, victoires partielles).
- Composants React fonctionnels avec hooks.

8 Déploiement et Maintenance

8.1 Configuration requise

- Navigateur moderne (Chrome, Firefox, etc.) supportant React.
- Node.js pour développement/build.
- Pas de serveur requis ; application client-side.

8.2 Procédure d'installation

- Cloner le repository.
- Installer dépendances : `npm install` (Vitest, React, etc.).
- Lancer : `npm run dev` (via Vite, assumé).
- Build : `npm run build` pour production.

8.3 Gestion des mises à jour et correctifs

- Mises à jour via Git : Ajouter fonctionnalités comme IA, multiplayer.
- Correctifs : Basés sur tests failing ; exemple dans `engine.test.ts` pour détections de bugs.

9 Annexes

9.1 Diagrammes UML

Non fournis, mais possible : Diagramme de classe pour Engine (avec méthodes setPiece, checkWin).

9.2 Schémas de base de données

Non applicable (pas de DB).

9.3 Capture d'écrans et illustrations techniques

- Écran principal : Grille 6x7 avec pièces colorées, texte joueur, bouton reset.
- Exemple de test plan (de `Exemple de plan de test.xlsx`) : Tableau avec scénarios, résultats, conformité (adapté pour Connect Four, e.g., « Placement pièce », « Détection victoire horizontale »).
- Code snippets : Extrait de `checkWin` pour logique de directions.