Proposition d'architecture pour Quoridor (C++/SFML)

Équipe: Tarazona Javier, Liang Tianyi

Novembre 2025

Objectifs

Concevoir une application Quoridor en C++ utilisant SFML avec une architecture **MVC** claire, portable **Windows/Linux/macOS**, et livrée sous forme d'exécutable cliquable (Windows : `.exe`, macOS : `.app`, Linux : archive AppImage ou `.tar.gz`).

Vue d'ensemble MVC

- **Modèle** (règles ét état du jeu) : gestion du plateau 9×9, murs, pions, validations des coups, calculs de chemins (A*/BFS), IA (Minimax/Négamax + élagage α–β, heuristiques).
- **Vue** (rendu ét UI) : rendu 2D avec SFML (grille, murs, pions, textes), gestion des ressources (textures, polices, sons), feedback visuel.
- **Contrôleur** (entrée ét boucle de jeu) : gestion clavier/souris, orchestration de la boucle de jeu, machine d'états (menus, partie, pause), intégration IA.

Modules et classes pressenties

M1. **Modèle**
- `Board` : représentation du plateau, murs placés, cases, coordonnées.
- `Rules` : validations complètes (déplacements, sauts, placements de murs, « chemin restant »).
- `State` : état courant (positions des pions, murs restants, joueur actif, historique minimal pour undo/replay optionnel).
- `Pathfinder` : A* et/ou BFS pour vérification d'existence de chemin et estimation de distance.
- `AI` : Minimax/Négamax + élagage α–β, profondeur bornée, génération de coups, heuristiques pluggables ; niveaux Facile/Normal/Difficile.

M2. **Vue**
- `Renderer2D` : encapsulation de SFML (`sf::RenderWindow`, `sf::View`, dessins de grille/pions/murs, HUD).
- `ResourceCache` : chargement pérénisant (RAII) des polices, textures, sons ; clés symboliques ; évite rechargements.
- `UIWidgets` (léger) : surlignage de case, info tour, messages d'erreur.

M3. **Contrôleur**
- `InputHandler` : mapping clavier/souris, états de drag pour murs, clics sur grille.
- `Game` : boucle principale (poll événements, update, render), cadence ; scène de *Partie*.
- `SceneManager/GameState` : machine d'états (Menu, Partie, Pause, Fin).
- `Config` : chargement de paramètres (JSON/TOML simple) : taille fenêtre, assets, niveau IA.

Boucle de jeu et flux de données

B1. Événements SFML → `InputHandler` → intentions de jeu (déplacer/placer mur).

B2. `Rules` valide l'intention contre `State` et `Board` ; en cas d'échec, la Vue affiche un message.

B3. En tour IA : `AI` demande à `Rules`/`Pathfinder` l'évaluation/validité ; choisit un coup.

B4. `State` est mis à jour, `Renderer2D` dessine la nouvelle scène ; boucle continue jusqu'à victoire.

Détails d'implémentation clés

— **Validation des murs** : empêche chevauchements, hors-limites, et bloqueurs de chemin. Vérifie « un chemin au moins » via BFS/A* pour chaque joueur après placement hypothétique.

— **IA configurables** : Facile (profondeur faible, randomisation plus forte), Normal (équilibré), Difficile (profondeur accrue, moins de hasard). Heuristique combine distance propre *vs.* adversaire et valeur des murs.

— **Rendu** : calques séparés (grille, murs, pions, HUD). Mise à l'échelle selon taille fenêtre ; police vectorielle lisible.

— **Ressources** : `ResourceCache` garde à vie : polices, textures ; chemin `assets/`. Gestion d'erreurs avec messages clairs.

— **Tests** : tests unitaires sur `Rules`, `Pathfinder`, et cas limites de placement ; tests IA sur validité des coups générés.

Structure du dépôt

```
ENSTQuoridor/
|-- CMakeLists.txt
|-- cmake/                    # Scripts CMake (FindSFML.cmake si besoin, helpers)
|-- src/
|   |-- app/                  # main(), initialisation fenêtre, SceneManager
|   |-- controller/           # InputHandler, Game, GameState/Scene
|   |-- model/                # Board, Rules, State, Pathfinder, AI
|   `-- view/                 # Renderer2D, UIWidgets, ResourceCache
|-- include/                  # En-têtes publics (si séparation)
|-- assets/                   # textures/, fonts/, sounds/
|-- shaders/                  # (optionnel)
|-- tests/                    # tests unitaires (CTest / Catch2)
|-- docs/
|   |-- exigences/
|   `-- architecture/
|-- scripts/                  # scripts build/packaging
|-- packaging/                # CPack config, icônes, entitlements (macOS)
|-- .github/workflows/ci.yml  # (optionnel) CI multi-plateforme
`-- README.md
```

Build multi-plateforme

Outil : CMake (>= 3.20), C++20, SFML (~2.6). Dépendances via vcpkg ou Conan, ou SFML installé localement.

Packaging et exécutables cliquables

Stratégie de distribution conçue

Pour garantir une expérience utilisateur optimale (*télécharger et double-cliquer*), le projet produira trois artefacts autonomes, un par plateforme :

1. **Windows** : `Quoridor-1.0-Windows.zip`
2. **macOS** : `Quoridor-1.0-macOS.dmg` (ou `.zip`)
3. **Linux** : `Quoridor-1.0-Linux-x86_64.AppImage` (ou `.tar.gz`)

Windows — Archive ZIP portable

Contenu : `quoridor.exe`, DLL SFML (`sfml-system-2.dll`, `sfml-window-2.dll`, `sfml-graphics-2.dll`, `sfml-audio-2.dll`, et dépendances tierces si nécessaires : `openal32.dll`, `freetype.dll`), dossier `assets/`.

Workflow utilisateur :

1. Télécharger `Quoridor-1.0-Windows.zip`.
2. Extraire le contenu où souhaité (Bureau, Documents, etc.).
3. Double-cliquer sur `quoridor.exe`.

Build : Enlace dinámico SFML. Copier automatiquement les DLL via script CMake ou post-build. CPack génère le `.zip` final.

Avantages : Aucun installeur requis ; portable ; compatible Windows 10+.

macOS — Bundle application (.app)

Contenu : `Quoridor.app` (bundle macOS) avec frameworks SFML embarqués, `assets/` dans `Contents/Resources/`.

Workflow utilisateur :

1. Télécharger `Quoridor-1.0-macOS.dmg`.
2. Ouvrir le `.dmg` et glisser-déposer `Quoridor.app` vers `/Applications` (ou autre dossier).
3. Double-cliquer sur `Quoridor.app`.

Build : Propriété CMake `MACOSX_BUNDLE` ; utiliser `install_name_tool` ou `BundleUtilities` pour fixer les chemins des dylibs SFML. CPack génère le `.dmg` (générateur `DragNDrop`).

Avantages : Expérience native macOS ; tout autoconténu ; distribution via `.dmg` ou `.zip`.

Linux — AppImage (autonome)

Contenu : Fichier unique exécutable `Quoridor-x86_64.AppImage` contenant le binaire, bibliothèques SFML, et `assets/`.

Workflow utilisateur :

1. Télécharger `Quoridor-1.0-Linux-x86_64.AppImage`.
2. Rendre exécutable : `chmod +x Quoridor-1.0-Linux-x86_64.AppImage`.
3. Double-cliquer (ou exécuter via terminal : `./Quoridor-1.0-Linux-x86_64.AppImage`).

Build : Utiliser `linuxdeploy` + plugin SFML/Qt pour packager dépendances. L'AppImage encapsule tout.

Avantages : Un seul fichier exécutable ; fonctionne sur presque toutes les distributions modernes ; aucune installation SFML requise.

Alternative Linux : Archive `.tar.gz` avec binaire, bibliothèques SFML locales (`lib/`), `assets/`, et script launcher ajustant `LD_LIBRARY_PATH`. Moins élégant mais viable si AppImage pose problème.

Outillage CPack

CMake CPack permet de générer automatiquement les artefacts :

— **Windows** : générateur `ZIP` (ou `NSIS` pour installeur optionnel).

— **macOS** : générateur `DragNDrop` (`.dmg`) ou `ZIP`.

— **Linux** : générateur `TGZ` ; AppImage via outil externe post-CPack.

Configuration CPack dans `CMakeLists.txt` : nom de paquet, version, icônes, générateurs spécifiques par OS.

Critères d'acceptation

— Lancement par double-clic : Windows `quoridor.exe`, macOS `Quoridor.app`, Linux binaire ou AppImage.

— Fonctionnement identique sur les trois OS (rendu, règles, IA).

— Arborescence d'installation contient `assets/` et les dépendances nécessaires.

Évolutivité et maintenance

— Heuristiques IA interchangeables ; profondeurs paramétrables ; modes jeu supplémentaires via `SceneManager`.

— `ResourceCache` centralise les chargements ; ajout d'actifs sans impact code.

— Découplage strict MVC : tests ciblés sur `Rules`/`Pathfinder` sans fenêtre.