

YaltaChess

Ceci est le rapport qui explique toutes mes démarches, mes recherches pour le codage du jeu en C++.

J'y intègre tous les résultats de mes recherches au fur-et-à-mesure de mes recherches et de l'avancement de mon projet.

Sources

CMake SFML Project Template <https://github.com/SFML/cmake-sfml-project>

Dépôt

Mon dépôt de mon travail: <https://github.com/olfabre/YaltaChess>

Choix initiaux et contraintes

Le codage se fera en C++ en architecture Model Vue Controler. Le jeu aura une interface graphique. Pour intégrer le graphisme, je vais utiliser SFML 3.0.0 qui est une version que je dois compiler et compatible avec mon système d'exploitation. En effet, je code sur mon MacOS Catalina qui est un vieux modèle, je dois systématiquement vérifier la comptabilité des librairies.

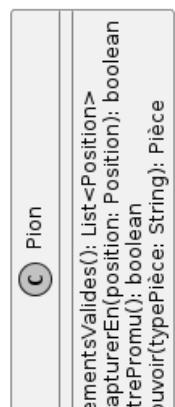
Étape 1: compréhension, réflexion, recherche

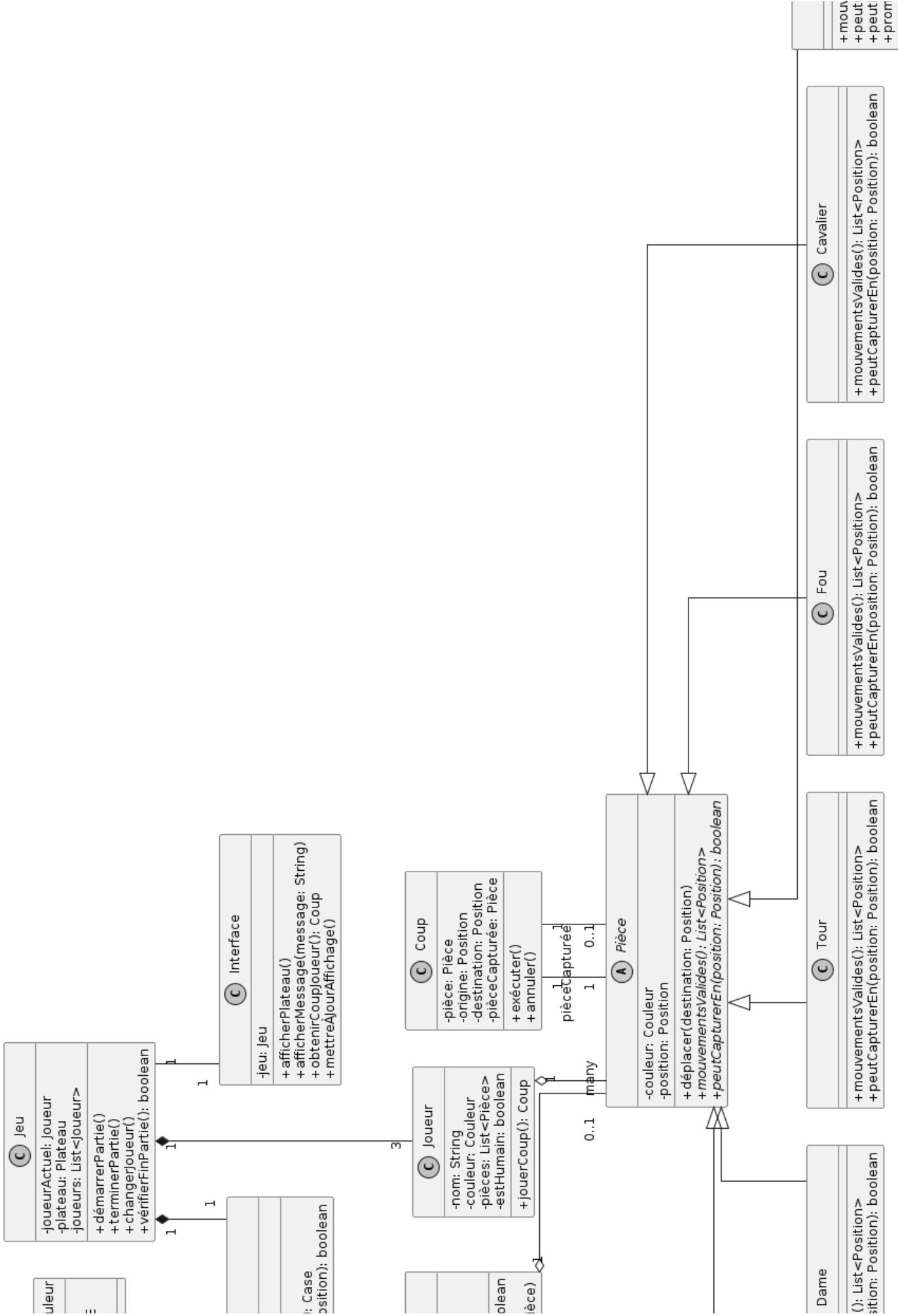
Pour les deux premières séances de TP, j'ai préféré me concentrer sur la recherche d'informations, bien comprendre les technologies que je devrai utiliser et comment les utiliser dans une architecture MVC.

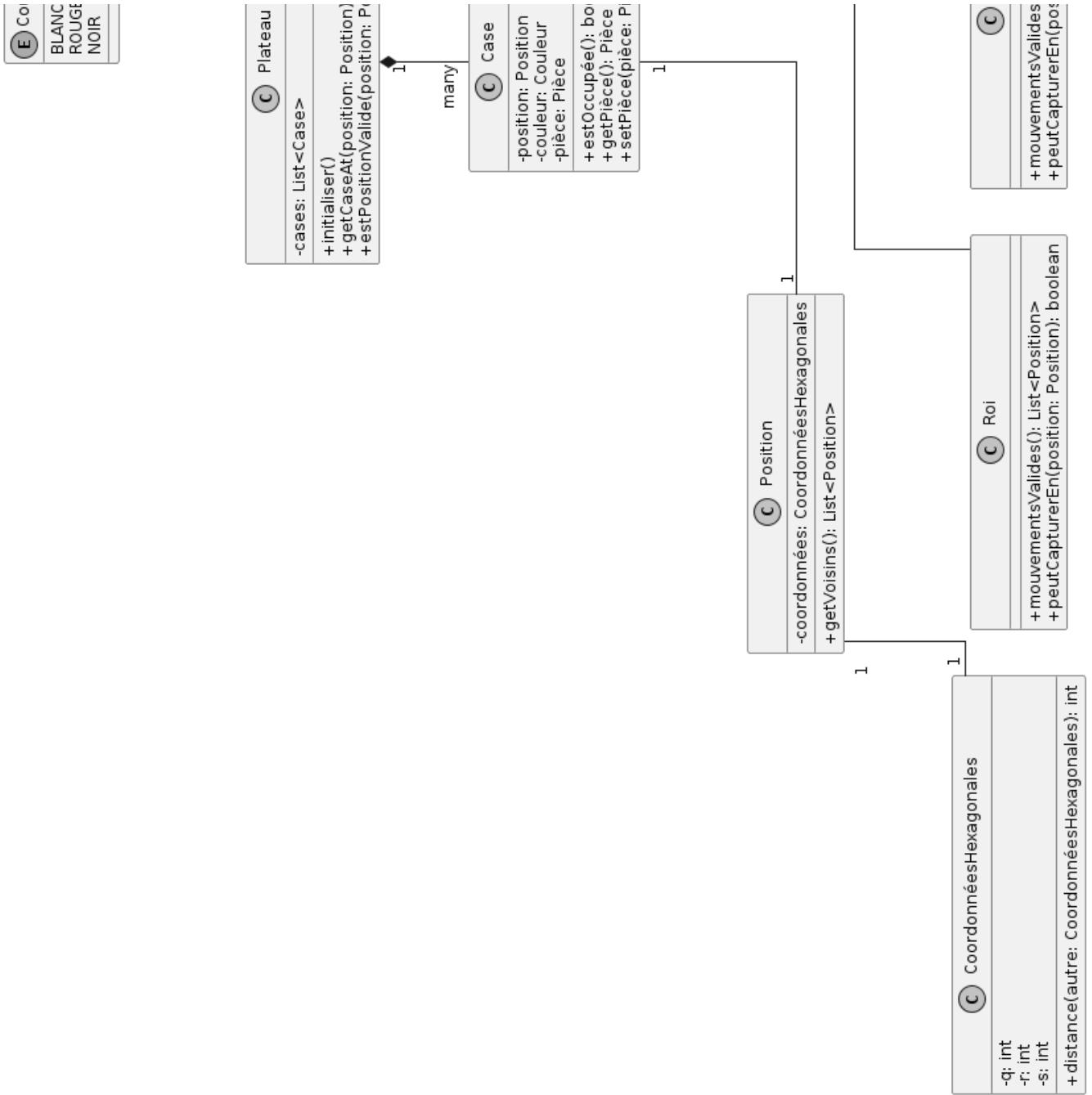
Pour l'instant je ne me concentre pas sur les aspects spécifiques du code: design patterns, algorithmes avancées mais davantage dans la mise en route du projet, l'architecture MVC et les relations entre les entités. Cela devait passer aussi par la conception d'un premier diagramme de classes simple.

Mon principal but est débuter correctement le projet avec une bases solides et une organisation adaptée pour coder proprement en MVC.

Diagramme de classes simple







Étape 2: installation de SFML

```
1 | brew install sfml
```

L'installation a été très longue (1 heure)

Je vérifie l'installation s'est bien passée

```
1 | pkg-config --modversion sfml-system
2 | 3.0.0
```

L'installation de SFML 3 s'est bien passée.

Étape 3: tester SFML

J'ai crée un fichier `CMakePresets.json`

Ce fichier sert à **simplifier la configuration de ton projet avec CMake** en définissant des paramètres prédéfinis. Il permet de centraliser les options importantes pour que tu n'aies pas à les taper manuellement à chaque fois.

```
1  {
2      "version": 3,
3      "configurePresets": [
4          {
5              "name": "default",
6              "description": "default settings",
7              "generator": "Unix Makefiles",
8              "binaryDir": "${workspaceFolder}/build",
9              "cachevariables": {
10                  "CMAKE_BUILD_TYPE": "Debug"
11              }
12          }
13      ]
14 }
```

À quoi ça sert ?

- 📁 `binaryDir` : C'est l'endroit où ton projet va être compilé (au lieu de mélanger les fichiers générés avec ton code source).
- 🐛 `CMAKE_BUILD_TYPE` : Ici, le mode `Debug` est activé, ce qui inclut des informations utiles pour déboguer ton code (comme les symboles de débogage).

j'ai crée un dossier `main` où je vais placer le code principal

Dans ce dossier, j'ai crée un fichier `CmakeLists.txt`

Ce fichier est **le cœur de ton projet** sous CMake. Il décrit comment ton projet doit être configuré, compilé et lié.

```
1 cmake_minimum_required(VERSION 3.25.2)
2 project(CMakeSFMLProject LANGUAGES CXX)
3
4 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
5
6 include(FetchContent)
7 FetchContent_Declare(SFML
8     GIT_REPOSITORY https://github.com/SFML/SFML.git
9     GIT_TAG 3.0.0)
10 FetchContent_MakeAvailable(SFML)
```

```
11
12 add_executable(main main.cpp)
13 target_link_libraries(main PRIVATE sfml-graphics)
14 target_compile_features(main PRIVATE cxx_std_17)
```

```
1 cmake_minimum_required(VERSION 3.25.2)
2 project(CMakeSFMLProject LANGUAGES CXX)
```

- Spécifie la version minimale de CMake requise.
- Définit le nom de ton projet (`CMakeSFMLProject`) et précise que c'est un projet en **C++**.

```
1 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
```

- Change l'emplacement des exécutables générés.
- Ici, les exécutables seront placés dans le dossier `/build/bin`.

```
1 include(FetchContent)
2 FetchContent_Declare(SFML
3   GIT_REPOSITORY https://github.com/SFML/SFML.git
4   GIT_TAG 3.0.0)
5 FetchContent_MakeAvailable(SFML)
```

- **FetchContent** est une méthode puissante qui télécharge et configure automatiquement des bibliothèques externes.
- Ici, il télécharge et configure la bibliothèque **SFML** (version 3.0.0) directement depuis GitHub.

```
1 add_executable(main main.cpp)
2 target_link_libraries(main PRIVATE sfml-graphics)
3 target_compile_features(main PRIVATE cxx_std_17)
```

- **add_executable** crée ton exécutable principal (`main`) en utilisant ton fichier `main.cpp`.
- **target_link_libraries** indique que ton exécutable doit être lié avec **SFML** (le module graphique).
- **target_compile_features** précise que ton code utilise les fonctionnalités du standard **C++17**.

J'ai créé un fichier `main.cpp` ou le contenu sera provisoire

```
1 #include <SFML/Graphics.hpp>
2
3 int main()
4 {
```

```
5 auto window = sf::RenderWindow(sf::VideoMode({1920u, 1080u}), "YaltaChess");
6 window.setFramerateLimit(144);
7
8 while (window.isOpen())
9 {
10     while (const std::optional<sf::Event> event = window.pollEvent())
11     {
12         if (event->isClosed())
13         {
14             window.close();
15         }
16     }
17
18     window.clear();
19     window.display();
20 }
21 }
```

Commandes pour compiler mon code principal

```
1 cmake -B build // par defaut mais dans mon cas, c'est la ligne du bas
2 cmake -B build_main -S main
```

La commande initiale (`cmake -B build`) **génère les fichiers de configuration** (Makefiles, cache, etc.) et télécharge les dépendances (comme SFML via `FetchContent`). C'est normal que ce soit plus long la première fois.

```
1 cmake --build build // par defaut mais dans mon cas, c'est la ligne du bas
2 cmake --build build_main
```

La commande suivante (`cmake --build build`) **compile tout le projet**, y compris les dépendances. C'est aussi plus long au premier passage. Cette commande **ne recompilera que les fichiers modifiés**, rendant la compilation beaucoup plus rapide.(builds "incrémentiels")

j'ai crée un dossier `tests` où je vais placer le code pour tester

J'ai créer un fichier `testSFML3.cpp`

```
1 #include <SFML/Graphics.hpp>
```

```

2
3 int main()
4 {
5     auto window = sf::RenderWindow(sf::VideoMode({1920u, 1080u}), "CMake Test
6 SFML");
7     window.setFramerateLimit(144);
8
9     while (window.isOpen())
10    {
11         while (const std::optional event = window.pollEvent())
12         {
13             if (event->is<sf::Event::Closed>())
14             {
15                 window.close();
16             }
17
18             window.clear();
19             window.display();
20         }
21     }

```

j'ai créé un fichier `CMakeLists.txt`

```

1 cmake_minimum_required(VERSION 3.25.2)
2 project(TestSFML LANGUAGES CXX)
3
4 include(FetchContent)
5 FetchContent_Declare(SFML
6     GIT_REPOSITORY https://github.com/SFML/SFML.git
7     GIT_TAG 3.0.0)
8 FetchContent_MakeAvailable(SFML)
9
10 add_executable(testSFML3 testSFML3.cpp)
11 target_link_libraries(testSFML3 PRIVATE sfml-graphics)
12 target_compile_features(testSFML3 PRIVATE cxx_std_17)

```

Commandes pour lancer le code de test SFML

```
1 | cmake -B build_test -S tests
```

Génération des fichiers de configuration

```
1 | cmake --build build_test
```

Compilation de tout le projet

```
1 | ./build_test/testSFML3
```

Lancement du test

EE

Étape 4: mise en place de la structure MVC

Voici ma structure optimisée avec SFML 3 pour bien démarrer mon projet.

Cette base inclut les bonnes pratiques et une architecture claire:

- **Séparation claire** : Chaque composant respecte bien le pattern MVC.
- **Lisibilité** : Les classes sont courtes et claires.
- **Extensibilité** : Cette structure permet d'ajouter facilement des entités, des scènes ou d'autres fonctionnalités.
- **Optimisation SFML 3** : Utilisation de `std::optional` pour améliorer la gestion d'événements.

L'architecture **MVC** (Modèle-Vue-Contrôleur) est une structure de code puissante qui permet de séparer la logique métier, l'affichage et la gestion des interactions utilisateur.

L'architecture MVC se divise en trois parties principales :

- **Modèle (Model)** → Gestion des données et logique métier.
- **Vue (View)** → Gestion de l'affichage graphique avec SFML.
- **Contrôleur (Controller)** → Gestion des événements et interactions utilisateur.

```
1 /YaltaChess
2 |
3 |--- /main
4 |   |--- CMakeLists.txt
5 |   |--- Model.h
6 |   |--- Model.cpp
7 |   |--- view.h
8 |   |--- view.cpp
9 |   |--- Controller.h
10 |   |--- Controller.cpp
11 |   |--- main.cpp
12 |--- /build_main    <-- (sera créé lors de la compilation)
```

1 fichier `/main/Model.h`

Gère les données, les règles du jeu, et la logique métier.

```
1 #ifndef MODEL_H
2 #define MODEL_H
3
4 #include <SFML/Graphics.hpp>
5
6 class Model {
7 public:
8     Model();
9     void update(); // Mise à jour des données du modèle (ex : position des pièces
d'échecs).
10 };
11
12 #endif // MODEL_H
```

1 fichier `/main/Model.cpp`

Gère les données, les règles du jeu, et la logique métier.

```

1 #include "Model.h"
2
3 Model::Model() {
4     // Initialisation des données (ex : position initiale des pièces)
5 }
6
7 void Model::update() {
8     // Exemple : Mettre à jour les positions des pièces, gérer les règles du jeu,
9     // etc.
10 }
```

Rôle du **Model** :

- Stocke les informations clés (ex : position des pièces d'échecs).
- Met à jour les données en fonction des règles métier (ex : déplacement légal des pièces).

1 fichier `/main/view.h`

Gère l'affichage et le rendu graphique à l'écran.

```

1 #ifndef VIEW_H
2 #define VIEW_H
3
4 #include <SFML/Graphics.hpp>
5
6 class View {
7 public:
8     View(sf::RenderWindow& window);
9     void render(); // Dessine les éléments graphiques
10 private:
11     sf::RenderWindow& m_window; // Référence vers la fenêtre SFML
12 };
13
14 #endif // VIEW_H
```

1 fichier `/main/view.cpp`

Gère l'affichage et le rendu graphique à l'écran.

```

1 #include "view.h"
2
3 View::View(sf::RenderWindow& window) : m_window(window) {}
4
5 void View::render() {
6     m_window.clear(sf::Color::Black); // Couleur de fond par défaut
7     // Dessine les éléments graphiques ici: les pièces, l'échiquier, etc
8     m_window.display();
9 }
```

Rôle du `view` :

- Affiche les éléments graphiques.
- Récupère les informations du `Model` pour afficher correctement les données à l'utilisateur.
- Ne contient **aucune** logique métier.

1 fichier `/main/controller.h`

Gère les entrées utilisateur (clics, clavier) et met à jour le modèle et la vue.

```

1 #ifndef CONTROLLER_H
2 #define CONTROLLER_H
3
4 #include <SFML/Graphics.hpp>
5 #include "Model.h"
6 #include "view.h"
7
8 class Controller {
9 public:
10     Controller(Model& model, View& view);
11     void processEvents(sf::RenderWindow& window); // Gère les événements SFML
12 private:
13     Model& m_model;
14     View& m_view;
15 };
16
17 #endif // CONTROLLER_H
```

1 fichier `/main/controller.cpp`

Gère les entrées utilisateur (clics, clavier) et met à jour le modèle et la vue.

```
1 #include "Controller.h"
2
3 Controller::Controller(Model& model, view& view) : m_model(model), m_view(view) {}
4
5 void Controller::processEvents(sf::RenderWindow& window) {
6     while (const std::optional event = window.pollEvent()) {
7         if (event->is<sf::Event::Closed>()) {
8             window.close();
9         }
10        // Ici, gestion des clics sur les pièces d'échecs, les déplacements, etc.
11    }
12 }
```

Rôle du **Controller** :

- Récupère les événements utilisateurs (clics, mouvements de souris, etc.).
- Met à jour le **Model** en fonction de ces interactions.
- Informe la **view** des changements pour que l'affichage soit correct.

1 fichier `/main/main.cpp`

Point d'entrée principal du programme qui connecte le modèle, la vue et le contrôleur.

```
1 #include <SFML/Graphics.hpp>
2 #include "Model.h"
3 #include "View.h"
4 #include "Controller.h"
5
6 int main() {
7     auto window = sf::RenderWindow(sf::VideoMode({1920u, 1080u}), "SFML MVC
Example");
8     window.setFramerateLimit(144);
9
10    Model model; // Création du modèle
11    View view(window); // Création de la vue (lié à la fenêtre)
12    Controller controller(model, view); // Création du contrôleur
13
14    while (window.isOpen()) {
15        controller.processEvents(window); // Gestion des événements (entrées
utilisateurs)
16        model.update(); // Mise à jour des données
17        view.render(); // Rendu graphique et mise à jour
affichage
18    }
19 }
```

```
20     return 0;  
21 }
```

Rôle du `main` :

- Initialise les trois composants (`Model`, `View`, `Controller`).
- Utilise une boucle infinie pour :
 - Traiter les événements avec le **Controller**.
 - Mettre à jour les données avec le **Model**.
 - Afficher les résultats avec la **View**.

Cycle de fonctionnement du code (Flux MVC)

Voici le cycle complet de fonctionnement :

1. Démarrage du programme

→ Le `main` crée une fenêtre SFML et initialise les trois composants.

2. Événements utilisateur (input)

→ Le `Controller` intercepte les interactions (ex : clic sur une pièce).

→ Il met à jour le `Model` en conséquence (ex : déplacement d'une pièce).

3. Mise à jour des données

→ Le `Model` applique les règles du jeu (ex : vérifie si le déplacement est légal).

4. Affichage (output)

→ La `View` récupère les informations du `Model` et met à jour l'écran.

5. Répétition

→ Ce cycle se répète tant que la fenêtre est ouverte.

Je dois aussi modifier le fichier `/main/CMakeLists.txt`

```
1 cmake_minimum_required(VERSION 3.25.2)  
2 project(SFML_MVC_YaltaChess LANGUAGES CXX)  
3  
4 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)  
5  
6 include(FetchContent)  
7 FetchContent_Declare(  
8     SFML
```

```

9      GIT_REPOSITORY https://github.com/SFML/SFML.git
10     GIT_TAG 3.0.0
11 )
12 FetchContent_MakeAvailable(SFML)
13
14 # Inclusion de tous les fichiers du dossier actuel (car CMakeLists.txt est dans
15 /main)
15 file(GLOB_RECURSE SRC_FILES ./*.cpp)
16
17 # Création de l'exécutable avec tous les fichiers sources
18 add_executable(SFML_MVC_YaltaChess ${SRC_FILES})
19
20 # Ajout des bibliothèques SFML
21 target_link_libraries(SFML_MVC_YaltaChess PRIVATE sfml-graphics sfml-window sfml-
22 system)
23
24 # Standard C++17
24 target_compile_features(SFML_MVC_YaltaChess PRIVATE cxx_std_17)
25
26 # Inclusion des répertoires pour les headers
27 target_include_directories(SFML_MVC_YaltaChess PRIVATE ${CMAKE_SOURCE_DIR}/main)
28

```

1. Définition de la version de CMake

```
1 | cmake_minimum_required(VERSION 3.25.2)
```

- Cette ligne indique la version minimale de CMake requise.
→ **VERSION 3.25.2** est nécessaire car SFML 3 utilise des fonctionnalités modernes de CMake.

2. Nom du projet

```
1 | project(SFML_MVC_YaltaChess LANGUAGES CXX)
```

- Définit le nom du projet (**SFML_MVC_YaltaChess**).
- **LANGUAGES CXX** indique que ton projet est en C++.

3. Définir le répertoire de sortie des exécutables

```
1 | set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
```

- Cette ligne précise que l'exécutable sera généré dans le dossier `/bin` à l'intérieur du répertoire de build (`build_main`).
 - Cela permet de garder une structure propre avec un exécutable bien séparé des fichiers de compilation.

4. Gestion de la bibliothèque SFML via `FetchContent`

```

1 | include(FetchContent)
2 | FetchContent_Declare(
3 |     SFML
4 |     GIT_REPOSITORY https://github.com/SFML/SFML.git
5 |     GIT_TAG 3.0.0
6 |
7 |     FetchContent_MakeAvailable(SFML)

```

- `include(FetchContent)` : Active le module `FetchContent` qui permet de télécharger et de configurer des bibliothèques directement depuis GitHub.
- `FetchContent_Declare()` : Indique à CMake d'aller chercher SFML 3 sur le dépôt GitHub officiel.
- `GIT_TAG 3.0.0` : Utilise la version 3.0.0 de SFML (la plus récente).
- `FetchContent_MakeAvailable(SFML)` : Télécharge et configure automatiquement SFML dans ton projet.

5. Inclusion des fichiers sources

```

1 | file(GLOB_RECURSE SRC_FILES *.cpp)

```

- Cette commande recherche **récursivement** tous les fichiers `.cpp` dans le répertoire `/main` (car ton `CmakeLists.txt` est dans `/main`).
 - Cela permet d'ajouter automatiquement tous tes fichiers sources sans les lister manuellement.
 - **Exemple :** Si tu ajoutes `Piece.cpp`, il sera automatiquement inclus.

6. Crédit de l'exécutable

```

1 | add_executable(SFML_MVC_YalDashess ${SRC_FILES})

```

- Cette ligne crée l'exécutable nommé `SFML_MVC_YalDashess` en utilisant tous les fichiers contenus dans la variable `${SRC_FILES}`.
 - C'est l'étape clé qui génère ton programme final.

7. Ajout des bibliothèques SFML

```
1 | target_link_libraries(SFML_MVC_YaltaChess PRIVATE sfml-graphics sfml-window sfml-system)
```

- **target_link_libraries** : Relie ton projet aux bibliothèques nécessaires de SFML :
- **sfml-graphics** : Pour la gestion des graphismes (sprites, formes, etc.).
- **sfml-window** : Pour la gestion des fenêtres et des événements.
- **sfml-system** : Pour les outils de gestion du temps et les utilitaires.

8. Activation du standard C++17

```
1 | target_compile_features(SFML_MVC_YaltaChess PRIVATE cxx_std_17)
```

- Active la norme **C++17** pour ton projet.
→ SFML 3 est conçu pour fonctionner avec cette version ou supérieure.

9. Inclusion des répertoires de headers

```
1 | target_include_directories(SFML_MVC_YaltaChess PRIVATE ${CMAKE_SOURCE_DIR}/main)
```

- Indique à CMake d'inclure le dossier `/main` pour retrouver les fichiers `.h` (headers).
→ Cela permet que les `#include "Model.h"` ou `#include "Controller.h"` fonctionnent sans chemin compliqué.

10. Commandes finales pour compiler et exécuter

Dans ton terminal, à la racine de ton projet :

► Générer les fichiers de build

```
1 | cmake -B build_main -S main
```

- Génère le répertoire `build_main` avec les fichiers de compilation.

► Compiler le projet

```
1 | cmake --build build_main
```

- Compile ton projet et génère l'exécutable dans `build_main/bin/SFML_MVC_Yaltachess`.

► Exécuter ton projet

```
1 | ./build_main/bin/SFML_MVC_Yaltachess
```

- Lance ton application SFML.

Étape 5: création de la hiérarchie orientée objet des pièces

Chaque pièce d'échecs Yalta sera un objet dérivant d'une classe abstraite commune (`Piece`), selon ce schéma :

```
1 | Piece (abstraite)
2 | |
3 |   └── Roi
4 |   └── Dame
5 |   └── Tour
6 |   └── Fou
7 |   └── Cavalier
8 |   └── Pion
```

Organisation des fichiers

```
1 | /main
2 |   ├── Model.h
3 |   ├── Model.cpp
4 |   ├── View.h
5 |   ├── View.cpp
6 |   ├── Controller.h
7 |   ├── Controller.cpp
8 |   └── main.cpp
9 |
10 |   └── /pieces
11 |     ├── Piece.h
12 |     ├── Piece.cpp
13 |     └── Roi.h
```

```
14 └── Roi.cpp
15 └── Dame.h
16 └── Dame.cpp
17 └── Tour.h
18 └── Tour.cpp
19 └── Fou.h
20 └── Fou.cpp
21 └── Cavalier.h
22 └── Cavalier.cpp
23 └── Pion.h
24 └── Pion.cpp
25
```

Fichier `pieces/Piece.h`

```
1 #ifndef PIECE_H
2 #define PIECE_H
3
4 #include <SFML/Graphics.hpp>
5
6 enum Couleur { BLANC, NOIR, ROUGE };
7
8 // Classe abstraite représentant une pièce du jeu Yalta
9 class Piece {
10 protected:
11     sf::Vector2i position; // Position sur l'échiquier
12     Couleur couleur;      // Couleur de la pièce (joueur)
13
14 public:
15     Piece(sf::Vector2i pos, Couleur coul);
16     virtual ~Piece() = default;
17
18     // Vérifie la validité du déplacement spécifique à chaque pièce
19     virtual bool mouvementValide(sf::Vector2i nouvellePos) const = 0;
20
21     // Dessine graphiquement la pièce (SFML)
22     virtual void dessiner(sf::RenderWindow& window) const = 0;
23
24     // Accesseurs
25     sf::Vector2i getPosition() const;
26     void setPosition(sf::Vector2i nouvellePos);
27     Couleur getCouleur() const;
28 };
29
30 #endif
```

Fichier `pieces/Piece.cpp`

```
1 #include "Piece.h"
2
3 Piece::Piece(sf::Vector2i pos, Couleur coul)
4     : position(pos), couleur(coul) {}
5
6 sf::Vector2i Piece::getPosition() const {
7     return position;
8 }
9
10 void Piece::setPosition(sf::Vector2i nouvellePos) {
11     position = nouvellePos;
12 }
13
14 Couleur Piece::getCouleur() const {
15     return couleur;
16 }
```

Un exemple de pièce du jeu qui va hériter de `Piece`: fichiers `Roi.h` et `Roi.cpp`

```
1 ifndef ROI_H
2 define ROI_H
3
4 include "Piece.h"
5
6 // Classe Roi héritant de Piece
7 class Roi : public Piece {
8 public:
9     Roi(sf::Vector2i pos, Couleur coul);
10    bool mouvementValide(sf::Vector2i nouvellePos) const override;
11    void dessiner(sf::RenderWindow& window) const override;
12 };
13
14 endif
```

```
1 include "Roi.h"
2 include <cmath>
3
4 Roi::Roi(sf::Vector2i pos, Couleur coul) : Piece(pos, coul) {}
5
6 bool Roi::mouvementValide(sf::Vector2i nouvellePos) const {
7     int dx = std::abs(nouvellePos.x - position.x);
```

```

8     int dy = std::abs(nouvellePos.y - position.y);
9     return (dx <= 1 && dy <= 1);
10 }
11
12 void Roi::dessiner(sf::RenderWindow& window) const {
13     // Ici, dessiner le Roi (sprite ou forme)
14 }
15

```

Étape 6: création de la hiérarchie orientée objet des cases

La structure avec les cases intégrées

```

1 /main
2 |--- Model.h
3 |--- Model.cpp
4 |--- View.h
5 |--- View.cpp
6 |--- Controller.h
7 |--- Controller.cpp
8 |--- main.cpp
9 |
10 └── /pieces
11     |   └── Piece.h
12     |   └── Piece.cpp
13     |   └── Roi.h
14     |   └── Roi.cpp
15     |   └── Dame.h
16     |   └── Dame.cpp
17     |   └── Tour.h
18     |   └── Tour.cpp
19     |   └── Fou.h
20     |   └── Fou.cpp
21     |   └── Cavalier.h
22     |   └── Cavalier.cpp
23     |   └── Pion.h
24     |   └── Pion.cpp
25 |
26 └── /cases (nouveau dossier)
27     └── Case.h
28     └── Case.cpp
29
30

```

Fichier `cases/Case.h`

```

1 #ifndef CASE_H
2 #define CASE_H
3
4 #include <SFML/Graphics.hpp>
5 #include "../pieces/Piece.h"
6
7 class Case {
8 private:
9     sf::Vector2i position;
10    Piece* piece; // Pointeur vers une pièce ou nullptr si vide
11
12 public:
13    Case(sf::Vector2i pos);
14
15    bool estOccupee() const;
16    Piece* getPiece() const;
17    void setPiece(Piece* p);
18    sf::Vector2i getPosition() const;
19
20    void dessiner(sf::RenderWindow& window);
21};
22
23#endif
24
```

Fichier cases/Case.cpp

```

1 #include "Case.h"
2
3 Case::Case(sf::Vector2i pos) : position(pos), piece(nullptr) {}
4
5 bool Case::estOccupee() const { return piece != nullptr; }
6
7 Piece* Case::getPiece() const { return piece; }
8
9 void Case::setPiece(Piece* p) { piece = p; }
10
11 sf::Vector2i Case::getPosition() const { return position; }
12
13 void Case::dessiner(sf::RenderWindow& window) {
14     sf::RectangleShape carre({100.f, 100.f});
15     carre.setPosition(position.x * 100.f, position.y * 100.f);
16     carre.setFillColor((position.x + position.y) % 2 ? sf::Color(209,139,71) :
17     sf::Color(255,206,158));
18     window.draw(carre);
19 }
```

Quelques modifications doivent être apportées:

Fichier `cases/case.h`

```
1 // Case.h (correction)
2 #ifndef CASE_H
3 #define CASE_H
4
5 #include <SFML/Graphics.hpp>
6 #include "../pieces/Piece.h"
7
8 class Case {
9 private:
10     sf::Vector2i position;
11     Piece* piece; // Pointeur vers une pièce ou nullptr
12
13 public:
14     Case(sf::Vector2i pos);
15
16     bool estOccupee() const;
17     Piece* getPiece() const;
18     void setPiece(Piece* p);
19     sf::Vector2i getPosition() const;
20
21     // Correction : ajout de const ici
22     void dessiner(sf::RenderWindow& window) const;
23 };
24
25 #endif
26
```

Fichier `cases/Case.cpp`

```
1 #include "Case.h"
2
3 Case::Case(sf::Vector2i pos) : position(pos), piece(nullptr) {}
4
5 bool Case::estOccupee() const { return piece != nullptr; }
6
7 Piece* Case::getPiece() const { return piece; }
8
9 void Case::setPiece(Piece* p) { piece = p; }
```

```

11 sf::Vector2i Case::getPosition() const { return position; }
12
13 // Correction ici :
14 void Case::dessiner(sf::RenderWindow& window) const {
15     sf::RectangleShape carre({100.f, 100.f});
16     carre.setPosition(sf::Vector2f(position.x * 100.f, position.y * 100.f));
17     carre.setFillColor((position.x + position.y) % 2 ? sf::Color(209,139,71) :
18         sf::Color(255,206,158));
19     window.draw(carre);
20 }
```

Fichier `CMakeLists.txt` pour prendre en compte les pièces et cases

```

1 cmake_minimum_required(VERSION 3.25.2)
2 project(SFML_MVC_Yaltachess LANGUAGES CXX)
3
4 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
5
6 include(FetchContent)
7 FetchContent_Declare(
8     SFML
9     GIT_REPOSITORY https://github.com/SFML/SFML.git
10    GIT_TAG 3.0.0
11 )
12 FetchContent_MakeAvailable(SFML)
13
14 # Inclusion de tous les fichiers .cpp du dossier courant ET des sous-dossiers
# (pieces et cases)
15 file(GLOB_RECURSE SRC_FILES
16     ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp
17     ${CMAKE_CURRENT_SOURCE_DIR}/pieces/*.cpp
18     ${CMAKE_CURRENT_SOURCE_DIR}/cases/*.cpp
19 )
20
21 # Création de l'exécutable avec tous les fichiers sources
22 add_executable(SFML_MVC_Yaltachess ${SRC_FILES})
23
24 # Ajout des bibliothèques SFML
25 target_link_libraries(SFML_MVC_Yaltachess PRIVATE sfml-graphics sfml-window sfml-
system)
26
27 # Standard C++17
28 target_compile_features(SFML_MVC_Yaltachess PRIVATE cxx_std_17)
29
30 # Inclusion des répertoires pour les headers (y compris sous-dossiers)
31 target_include_directories(SFML_MVC_Yaltachess PRIVATE
```

```

32     ${CMAKE_CURRENT_SOURCE_DIR}
33     ${CMAKE_CURRENT_SOURCE_DIR}/pieces
34     ${CMAKE_CURRENT_SOURCE_DIR}/cases
35 )
36

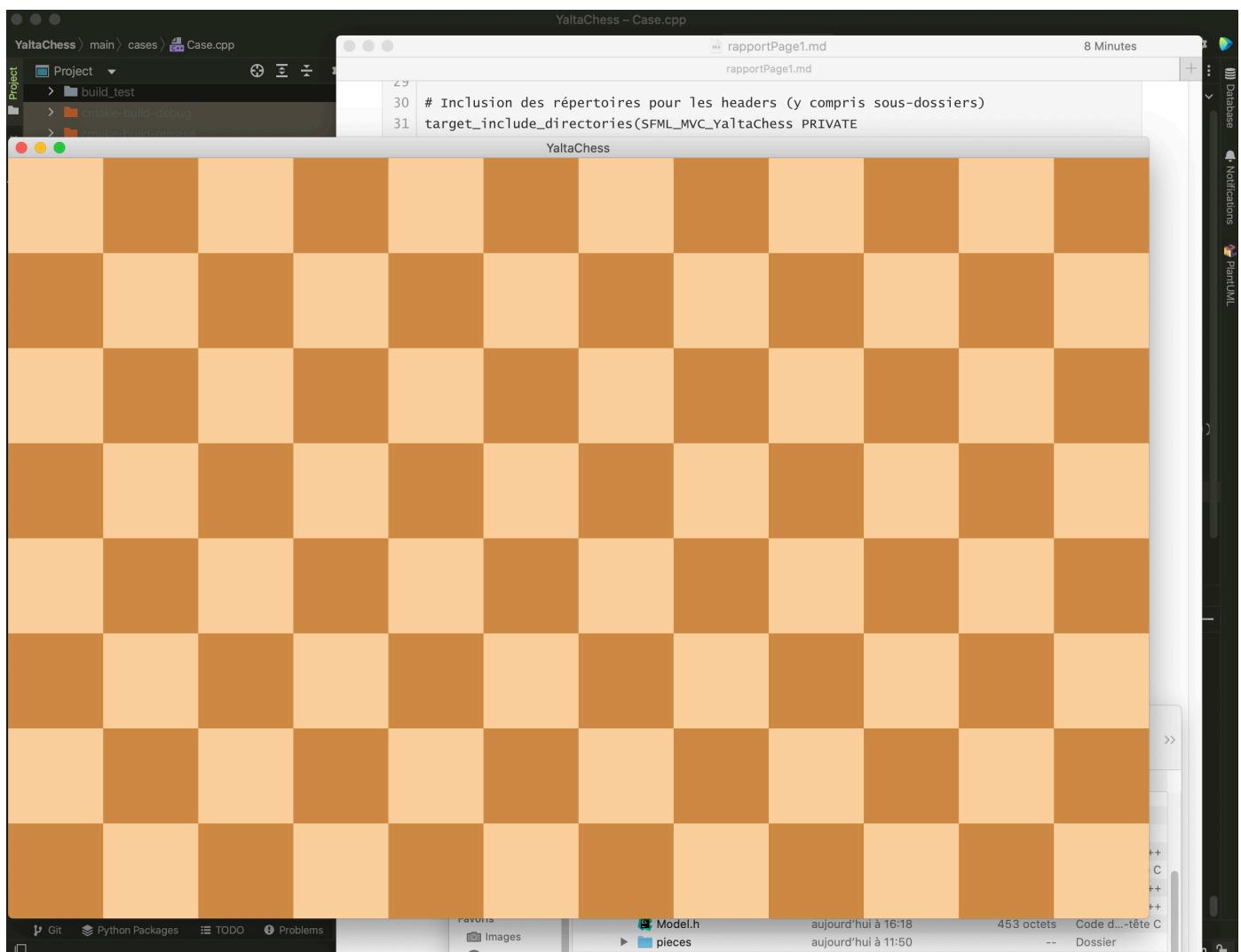
```

on compile de nouveau

```

1 cmake -B build_main -S main
2 cmake --build build_main
3
4 ./build_main/bin/SFML_MVC_Yaltachess

```



Notre échiquier actuel utilise des cases carrées disposées dans un tableau 2D simple. C'est une première étape. Or, l'échiquier Yalta est hexagonal avec 96 cases trapézoïdales!

Pour gérer ce type spécifique d'échiquier, il faut adapter ta structure graphique et logique.

Comment dessiner l'échiquier Yalta sous SFML clairement

- Chaque case sera dessinée comme un **polygone SFML** (`sf::ConvexShape`).

```
1 void Model::initialiserEchiquierYalta() {
2     std::vector<std::vector<sf::Vector2f>> coordcases = {
3         {{400,400},{450,380},{500,400},{450,420}},
4         {{500,400},{550,380},{600,400},{550,420}},
5         {{345.f,825.f},{395.f,805.f},{445.f,825.f},{395.f,845.f}},
6         //etc...
7     };
}
```

- Chaque case aura une coordonnée spécifique (personnalisée).
- On stockera dans notre classe `Case` chaque position sous forme de polygone (`sf::ConvexShape`) et on l'affichera ensuite directement.

Problème rencontré pour la création de l'échiquier avec cette méthode

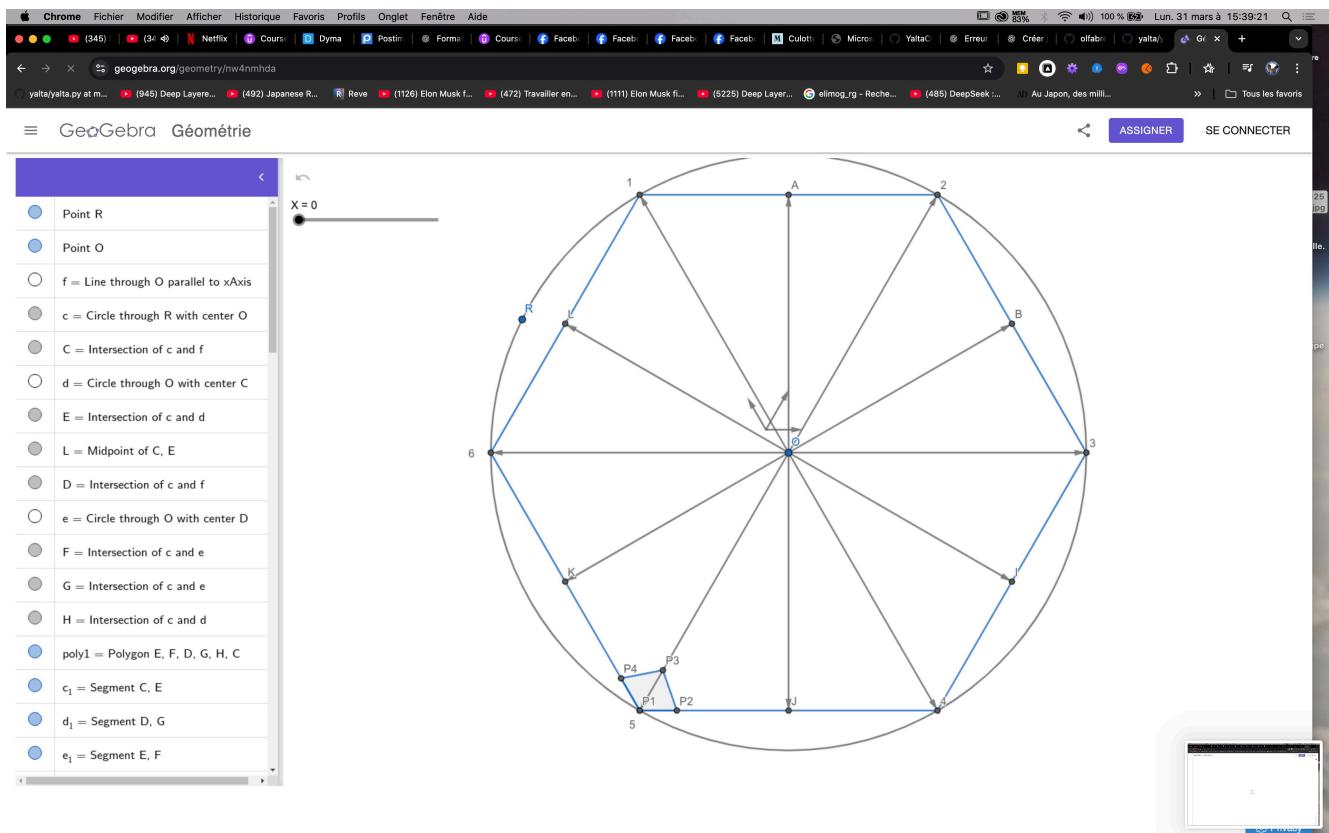
En effet, cela signifie que je serais obligé de prendre toutes les coordonnées des 96 cases à partir du logiciel Photoshop sur une vraie photo d'un échiquier Yalta en fond.

Le travail reste tout de même **colossal**.

Y'aurait-il une autre solution ?

oui d'après mes recherches sur internet, j'ai trouvé une autre solution. Beaucoup moins chronophage et longue à réaliser:

1. La structure globale



- On part d'un **hexagone** (une forme à 6 côtés égaux).
- Cet hexagone central est divisé en **6 zones** distinctes appelées "**sextants**" (comme 6 parts égales d'un gâteau hexagonal).
- Chaque sextant contient des **petites cases** (cellules) dans lesquelles les pièces d'échecs peuvent se déplacer.

2. La forme de chaque case

- Contrairement aux cases habituelles (carrées), ici chaque case est un **quadrilatère** (4 côtés, mais pas forcément égaux).
- Pourquoi des quadrilatères ?
Car c'est la forme géométrique qui permet de parfaitement remplir chaque sextant pour obtenir l'échiquier hexagonal global sans trous ni chevauchements.

3. Comment on dessine précisément chaque case ?

Voici simplement comment c'est fait dans ton programme :

Étape par étape :

Étape 1 : On choisit un **point central** de l'écran (`mid`) :

```
1 | mid = (largeur_ecran / 2, hauteur_ecran / 2)
```

Étape 2 : Autour de ce point central, on calcule **6 points extérieurs** pour créer l'hexagone global :

```
1 | points_hexagone = [
2 |     haut-gauche, haut-droite, droite,
3 |     bas-droite, bas-gauche, gauche
4 | ]
```

Étape 3 : Pour chaque sextant (zone), on prend deux points consécutifs de l'hexagone pour dessiner les cases à l'intérieur en effectuant des calculs géométriques précis (on appelle ça **interpolation linéaire**).

Étape 4 : On utilise ces points pour calculer les coins de chaque petite case (quadrilatère). Cela donne à chaque petite case sa position précise sur l'écran.

4. Alternance de couleurs des cases

Chaque case est alternativement colorée en blanc ou en noir selon une règle mathématique simple :

- Si la somme des coordonnées de la case (en x, y, et la zone) est paire, la case est blanche ; sinon, elle est noire :

```
1 | couleur = ((x + y + zone) % 2 == 0) ? blanc : noir;
```

Cela crée l'effet échiquier caractéristique.

5. Centrage final sur l'écran

Pour que l'échiquier soit parfaitement centré dans ta fenêtre graphique :

On décale toutes les cases d'une même distance (ici 50 pixels) vers la droite et vers le bas, afin que l'échiquier apparaisse exactement au milieu de l'écran :

```
1 | centre_final = (largeur_echiquier / 2 + décalage_x, hauteur_echiquier / 2 + décalage_y)
```

En résumé, dessiner l'échiquier Yalta, c'est :

- Créer un hexagone, puis le découper en 6 zones égales.
- Remplir chaque zone par des cases à 4 côtés, calculées mathématiquement.
- Alterner blanc/noir les cases.
- Décaler légèrement le tout pour le centrer sur l'écran.

Rappel pour compiler

```
1 cmake -B build_main -S main
2 cmake --build build_main
3
4 ./build_main/bin/SFML_MVC_Yaltachess
```

Étape 7: création d'une fenêtre intro

Je me suis servi d'une I.A. générative pour créer l'image png qui porte le nom de mon application.

On a crée une classe Intro pour typer un objet de type Intro. On a donc créer un objet Intro qui s'occupe d'afficher l'image d'introduction

```
1 Intro intro(window);
2 if (!intro.initialize()) { // Charge l'image intro.png
3     return -1; // Si l'image n'est pas trouvée, le programme s'arrête.
4 }
```

On a ajouté une boucle de façon que tant que l'intro n'est pas finie (elle dure 5 secondes), la fenêtre reste ouverte et montre l'image.

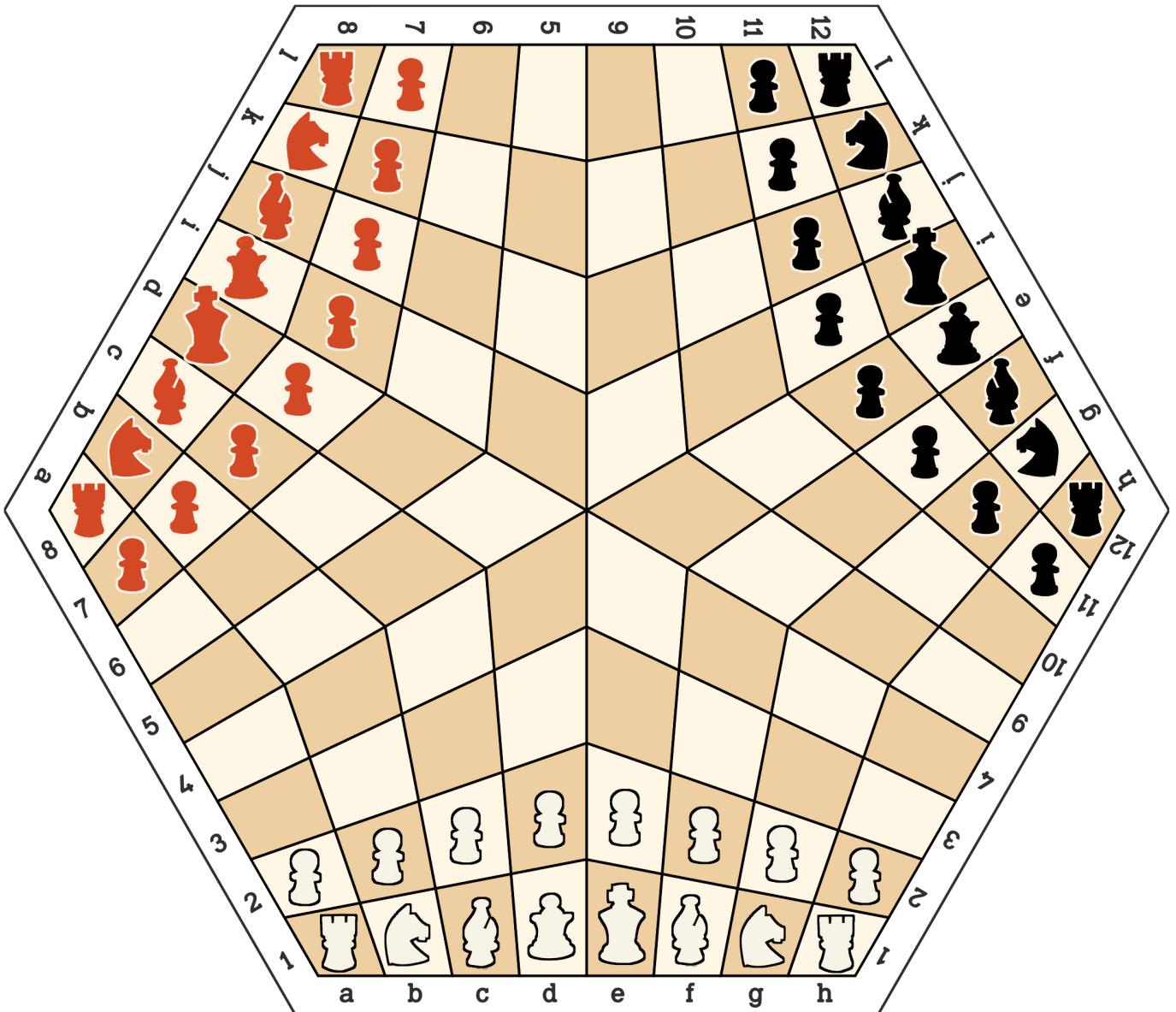
Dans le fichier `intro.cpp` on s'occupe de charger l'image (initialisation) et redimensionne l'image pour qu'elle s'adapte à la fenêtre du jeu (1100 x 1100)

Nous avons aussi intégré la police d'écriture **Press Start 2P** récupérée chez google font <https://fonts.google.com/share?selection.family=Press+Start+2P> pour faire un titre au dessus de l'image

Nous avons aussi intégré une musique style des jeux des années 80 (libre de droits) que nous avons téléchargée sur le site <https://pixabay.com/fr/music/search/genre/jeux%20vid%C3%A9o/?page=3>

Étape 8: création des labels

Nous devons créer les labels comportant lettres et chiffres. La numérotation est assez particulière sur un jeu d'échec Yalta. De plus les labels doivent être orientés selon le côté: perpendiculaire à l'arête du côté.



Donc le but est de placer les chiffres et lettres de la même manière que sur la photo ci-dessus avec un bandeau blanc autour du périmètre de l'échiquier. C'est dans `View.cpp` que tout est réalisé.

ici un extrait du code qui permet d'affecter à chaque côté, les labels

```

1 void YaltaChessView::initBorderLabels()
2 {
3     const float WIDTH      = 1000.f;
4     const float OFFSET     = 50.f;
5     const float OUTSET    = 25.f;      // distance label → bordure
6
7     // Étiquettes par côté (dans l'ordre horaire)
8     const std::vector<std::vector<std::string>> labels = {
9         /* 0 (num.) */ { "8", "7", "6", "5", "9", "10", "11", "12" },
10        /* 1 (lettres) */ { "l", "k", "j", "i", "e", "f", "g", "h" },
11        /* 2 (num.) */ { "12", "11", "10", "9", "4", "3", "2", "1" },
12        /* 3 (lettres) */ { "h", "g", "f", "e", "d", "c", "b", "a" },
13        /* 4 (num.) */ { "1", "2", "3", "4", "5", "6", "7", "8" },

```

```

14     /* 5 (lettres) */ { "a","b","c","d","i","j","k","l" }
15 };
16 ...

```

ici un extrait du code qui permet d'orienter à chaque côté, les labels (Police Robot de Google)

```

1 // override angles pour côtés lettres
2     float angleDeg = baseAngle;
3     if      (k == 1) angleDeg = 240.f;
4     else if (k == 3) angleDeg = 0.f;
5     else if (k == 5) angleDeg = 120.f;
6
7     txt.setRotation(sf::degrees(angleDeg));
8     txt.setPosition(pos);
9     borderLabels.push_back(txt);
10 ...

```

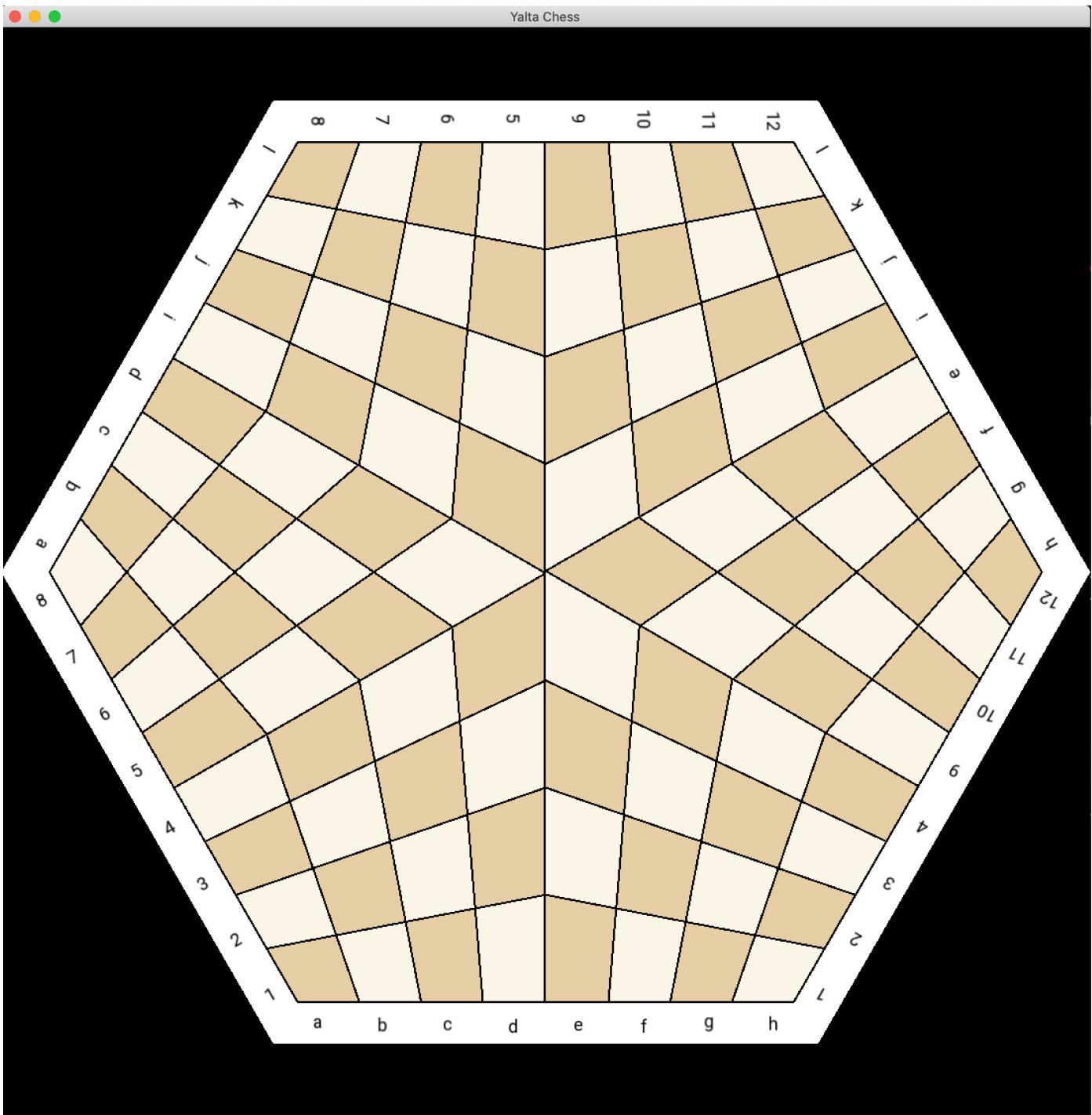
ici un extrait pour le contour de l'échiquier (bordure blanche entourée d'un trait noir)

```

1 // 2) Bordure blanche : hexagone plus grand de BORDER_WIDTH vers l'extérieur
2 sf::ConvexShape whiteBorder;
3 whiteBorder.setPointCount(6);
4 for (int i = 0; i < 6; ++i)
5 {
6     // direction unitaire
7     sf::vector2f dir = v[i];
8     float len = std::sqrt(dir.x*dir.x + dir.y*dir.y);
9     if (len != 0) dir /= len;
10
11    // point : sommet board + déplacement
12    sf::vector2f pt = mid + v[i] + dir * BORDER_WIDTH;
13    whiteBorder.setPoint(i, pt);
14 }
15 whiteBorder.setFillColor(sf::Color::white);
16 window.draw(whiteBorder);
17 ...

```

voici le résultat



Étape 9: création et positions des pions

Il est conseillé d'utiliser des images au format png transparent. Je vais télécharger toutes les images de pions d'un jeu d'échec à l'adresse suivante: https://commons.wikimedia.org/wiki/Category:SVG_chess_piece_S

Nous avons donc ajouté les ressources d'images png pour toutes les pièces nommées sous la forme:

- Tour_White.png
- Tour_Black.png

nous avons défini une énumération des couleurs dans `Piece.h`

```
1 | enum Couleur { BLANC, NOIR, ROUGE };
```

Chaque pièce porte donc une valeur `BLANC`, `NOIR` ou `ROUGE`.

Dans le constructeur de `Model` (`Model::Model()`), on parcourt un tableau statique `SETUP[12][12]` : chaque entrée est un `pair<couleur, type>`. Si `couleur < 0` la case est vide, sinon on instancie la bonne sous-classe de `Piece` (Roi, Pion, etc.) avec la couleur et la position `(x, y)`, et on l'ajoute au vecteur `pieces`.

2 / Chargement des textures et coloration

`ResourceManager::loadAll()` charge les PNG nommés par exemple `"Pion_White.png"` ou `"Tour_Black.png"` dans une map `textures` de `string→Texture`.

Pour le joueur ROUGE, on utilise la texture blanche mais on recolore le sprite en rouge juste avant le dessin:

```
1 | if (p->getCouleur() == ROUGE)
2 |     spr.setColor(color(195, 83, 51));
```

Cela applique une teinte rouge sur l'ensemble des pixels non transparents.

3/ Placement et centrage des sprites

Chaque pièce connaît sa position grille `vector2i g` (indices de ligne/colonne). La fonction `gridToPixel(g)` convertit ces indices en coordonnées pixels au centre de la case hexagonale.

Pour centrer le sprite sur ce point, j'ai fait:

```
1 | auto ts = tex.getSize();                      // taille en pixels de la texture
2 | spr.setOrigin({ ts.x/2.f, ts.y/2.f });        // origine au centre
3 | spr.setPosition(gridToPixel(p->getPosition()));
```

la position du sprite correspond exactement au centre de la case

Pour placer correctement les pièces (informations trouvées sur internet): on représente l'échiquier sous forme d'une grille logique 12×12 (indices 0 à 11 en X et Y). Chaque case de la grille est décrite par un `pair<int, int>`

Pourquoi un tableau 12×12 pour un jeu à 3 joueurs ?

L'échiquier en étoile a 6 « bras » de longueur 4 : on choisit une grille 12×12 , plus simple à parcourir, et on marque `-1` partout où il n'y a pas de case valide. Seules les zones centrales et terminales de chaque bras portent des pièces de départ. Les `initialiserEchiquier()` s'occupe, indépendamment, de traduire ces indices `(x, y)` en formes hexagonales et positions pixels à l'écran.

Le premier entier (`couleur`) indique la couleur du joueur:

0 → BLANC

1 → ROUGE

2 → NOIR

<0 (dans mon programme -1) → pas de pièce dans cette case

Le second entier (`type`) indique le type de pièce:

0 → Roi

1 → Pion

2 → Cavalier

3 → Fou

4 → Tour

5 → Dame

en résumé

Le tableau `SETUP` encode, pour chaque coordonnée logique, la couleur et le type de la pièce de départ (ou -1 si vide).

Dans le constructeur de `Model`, on boucle sur ce tableau, on convertit dans mon programme, ces codes en instances concrètes (`new Roi`, etc.), et on remplit le `vector<Piece*> pieces` pour pouvoir ultérieurement afficher et manipuler toutes les pièces du jeu!

```

14     // y=5
15     { {1,2},{1,1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{1,1},{1,1} },
16     // y=6
17     { {1,3},{1,1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1} },
18     // y=7
19     { {1,0},{1,1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1} },
20     // y=8
21     { {-1,-1},{-1,-1},{-1,-1},{-1,-1},{2,4},{2,2},{2,3},{2,5},{2,4},{2,1} },
22     // y=9
23     { {-1,-1},{-1,-1},{-1,-1},{-1,-1},{2,1},{2,1},{2,1},{2,1},{2,2},{2,1} },
24     // y=10
25     { {-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{2,3} },
26     // y=11
27     { {-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{2,0} },
28 {2,1},{-1,-1},{-1,-1} }
29
30 Model::Model() {
31     initialiserEchiquier();
32     for (int y = 0; y < 12; ++y) {
33         for (int x = 0; x < 12; ++x) {
34             auto [coul, type] = SETUP[y][x];
35             if (coul < 0) continue;
36             Vector2i grid{x,y};
37             Piece* p = nullptr;
38             Couleur cc = (coul==0? BLANC : coul==1? ROUGE : NOIR);
39             switch(type) {
40                 case 0: p = new Roi(grid, cc); break;
41                 case 1: p = new Pion(grid, cc); break;
42                 case 2: p = new Cavalier(grid, cc); break;
43                 case 3: p = new Fou(grid, cc); break;
44                 case 4: p = new Tour(grid, cc); break;
45                 case 5: p = new Dame(grid, cc); break;
46             }
47             pieces.push_back(p);
48         }
49     }
50 }
```

Toujours dans `ResourceManager::loadAll()`, on génère en parallèle avec un procédé qui est un `maskTexture` (procédé trouvé sur internet)

- on récupère l'`Image` de la texture originale,
- on crée une image du même format,
- pour chaque pixel où `alpha>0`, on écrit `color(255,255,255,alpha)`, sinon `transparent`,
- on recharge cette image dans un `Texture maskTex` et on l'ajoute à `maskTextures`.

Cela donne une silhouette blanche de la pièce basée sur son canal alpha

Au moment du dessin (`YaltaChessView::draw()`), pour chaque pièce:

- on récupère `maskTex = getMask(key)` et on crée un `maskspr` centré comme le sprite original,
- on choisit une `outlinecolor` (noir pour les pièces blanches, crème pour les autres),
- on dessine 8 copies de `maskspr` décalées de `±thickness` en x et/ou y (tableau `offsets`) pour former le contour,
- on dessine le sprite normal par-dessus.

Le résultat est un liseré net, sans artefact, parfaitement calé sur la forme de la pièce.

Étape 11: gestion des noms des joueurs et attribution couleur

Le jeu d'échec Yalta est un jeu d'échec à 3 joueurs. Pour mon application, le premier joueur est un humain et les deux autres sont des joueurs fictifs dans un premier temps

Pour l'instant, c'est toujours l'humain (joueur) qui doit commencer à jouer.

L'application va choisir au hasard quelle est la couleur attribuée à chacun (son côté).

Je souhaite que les autres joueurs fictifs soient définies par des prénoms au hasard.

Sur chaque côté il y aura affichée le prénom des joueurs fictifs et "Vous" pour représenter le joueur.

Pour connaître quel est le tour de la personne qui doit jouer, un point vert sera affiché à côté des prénoms.

Par conséquent les autres auront un point rouge.

Le joueur aura la main seulement sur ces pièces du jeu.

1/ Pour la première étape je souhaite tirer au hasard, un côté à chacun avec un prénom au hasard et pour le joueur humain le prénom "Vous" affiché sur le côté attribué. Ensuite placer un point vert clignotant pour désigner que c'est son tour à joueur et rouge pour les autres.

- Structure des joueurs dans le modèle

- Définir un type `PlayerInfo` pour stocker l'état de chaque joueur dans `model.h`. je l'ai placé dans une structure car ce ne sont que des données à gérer

```

1 enum Couleur { BLANC, ROUGE, NOIR };
2 struct PlayerInfo {
3     std::string name;    // "Humain" ou prénom
4     Couleur color;      // côté du joueur
5     bool isHuman; // true si joueur réel
6 };

```

- Tirage aléatoire des prénoms et des couleurs

- Liste de prénoms à choisir

```

1 std::vector<std::string> botNames = {
2     "Alice", "Bob", "Charlie", "Diane", "Eve", "Frank"
3 };

```

- liste de couleurs

```
1 std::vector<Couleur> cols = { BLANC, ROUGE, NOIR };
```

- Mélange aléatoire dans `model.cpp`

```

1
2 void Model::initialiserJoueurs() {
3     static std::mt19937_64 rng{std::random_device{}()};
4     std::shuffle(cols.begin(), cols.end(), rng);
5     std::shuffle(botNames.begin(), botNames.end(), rng);
6
7     players.clear();
8     players.push_back({ "Humain", cols[0], true });
9     players.push_back({ botNames[0], cols[1], false });
10    players.push_back({ botNames[1], cols[2], false });
11    currentPlayerIdx = 0; // L'humain commence
12 }

```

- `std::mt19937_64` : moteur de génération pseudo-aléatoire.

`std::shuffle` (`<algorithm>`) : mélange uniforme du vecteur.

- Affichage des prénoms sur leurs côtés

Dans le fichier `view.cpp` on récupère les joueurs

```
1 auto const& players = model.getPlayers();
2 int currentPlayerIdx = model.getCurrentPlayerIdx();
```

On calcule la position autour du plateau

```
1 Vector2f mid(OFFSET + BOARD_SIZE/2.f,
2                 OFFSET + BOARD_SIZE/2.f);
3 float infoRadius = BOARD_SIZE/2.f + 30.f;
4 for (size_t i = 0; i < players.size(); ++i) {
5     auto const& p = players[i];
6     float angleDeg = 0.f;
7     // angles corrigés pour BLANC en bas, ROUGE en haut-gauche, NOIR en haut-droite
8     switch (p.color) {
9         case BLANC: angleDeg = 90.f; break;
10        case ROUGE: angleDeg = -150.f; break;
11        case NOIR: angleDeg = -30.f; break;
12    }
13    float a = angleDeg * 3.14159265f / 180.f;
14    vector2f pos = mid + vector2f(std::cos(a), std::sin(a)) * infoRadius;
```

bien choisir les côtés !

on dessine le texte

```
1 Text label(coordFont,
2             p.isHuman ? "Vous" : p.name,
3             20);
4 label.setFillColor(Color::Black);
5 label.setPosition(pos + Vector2f(-20.f, -10.f));
6 window.draw(label);
```

coordFont : police chargée au constructeur.

On aligne légèrement le texte pour qu'il n'empêtre pas sur le point.

- Indicateurs rouge/vert clignotant

- j'ai ajouté `sf::clock blinkClock;` dans `YaltaChessView` et je l'initialise dans le constructeur (`blinkClock.restart();`).
 - on calcule l'état "on/off" chaque frame

```
1 const float blinkPeriod = 0.5f;
2 bool blinkon = std::fmod(
3     blinkClock.getElapsedTime().asSeconds(),
4     blinkPeriod * 2.f
5 ) < blinkPeriod;
```

- o on dessine le cercle

```

1 circleShape dot(8.f);
2 dot.setOrigin({8.f, 8.f});
3 dot.setPosition(pos + vector2f(-40.f, 0.f));
4 if ((int)i == currentIndex) {
5     // actif : clignote vert
6     if (blinkOn) {
7         dot.setFillColor(Color::Green);
8         window.draw(dot);
9     }
10 } else {
11     // autres : rouge fixe
12     dot.setFillColor(Color::Red);
13     window.draw(dot);
14 }
```

circleShape : forme SFML pour un rond.

blinkClock : mesure le temps écoulé et permet de découper les cycles.

Étape 12: Gestion du premier événement souris

Je souhaite quand c'est mon tour de jouer (cercle vert clignotant) au passage de ma souris sur mes pièces, il y a un changement (la case prend la couleur orange). Ce que l'on appelle un **hover**.

1/ Boucle principale et gestion des événements

Mon `main.cpp` appelle à chaque frame le `controller.handleEvent(event)` pour traiter tous les événements dont les déplacements de la souris `MouseMoved` mais aussi ses actions `view.draw()`

2/ Capture du déplacement de la souris

Dans `Controller::handleEvent` je vérifie que l'évènement est bien un `MouseMoved`

Je récupère la position écran (`event.mouseMove.x`, `event.mouseMove.y`) et je la convertit en coordonnées monde avec

```
1 | vector2f world = view.getWindow().mapPixelToCoords(pixel);
```

J e m'assure bien que c'est bien au tour de l'humain à jouer ("vous")

```
1 | players[model.getCurrentPlayerIdx()].isHuman
```

3/ Recherche de la case sous le curseur

Dans le HandleEvent:

- Je parcourt toutes les cases (pointeur Case) de model.getCases()
- Pour chaque case, on appelle une méthode `contientPoint(world)` pour déterminer si le point est à l'intérieur du polygone convexe
- Je vérifie que la case est bien occupée `c->estOccupee()` (par une de mes pièces) que la pièce m'appartient bien (`c->getPiece()->getCouleur() == players[...].color`).
- Le premier c (case) qui passe ces tests est stocké dans `hoveredCase`, puis on appelle `view.setHoveredCase(hoveredCase)`.

4/ Mise à jour de la liaison pièces ⇒ cases

Dans `YaltachessView::draw()` avant de dessiner:

- Je remet à zéro avec la méthode `c->setPiece(nullptr)` pour chaque case
- Je parcourt toutes les pièces (`model.getPieces()`), on calcule leur centre en pixels (avec `gridToPixel`), puis on redonne à la case correspondante son pointeur de pièce (`c->setPiece(p)`).

5/ Dessin avec highlight orange

Toujours dans la méthode draw():

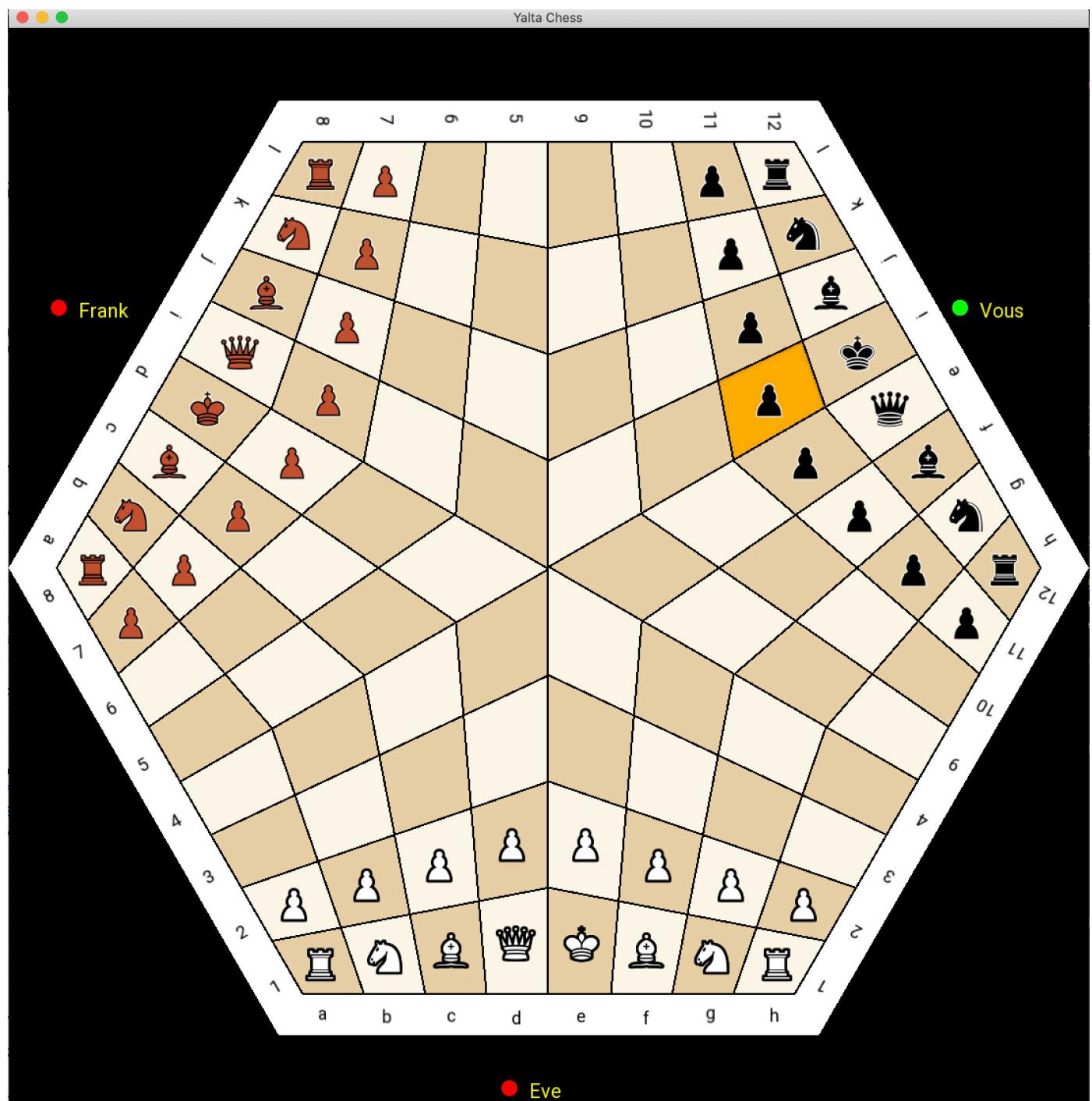
```
1 for (Case* c : model.getCases()) {  
2     if (c == hoveredCase) {  
3         // copie du ConvexShape  
4         ConvexShape highlight = c->getShape();  
5         // fond orange semi-transparent (ici R=255, G=165, B=0, alpha=240)  
6         highlight.setFillColor(Color(255,165,0,240));  
7         // contour inchangé  
8         highlight.setOutlineColor(Color::black);  
9         highlight.setOutlineThickness(2.f);  
10        window.draw(highlight);  
11    } else {  
12        window.draw(*c); // dessin normal  
13    }  
14 }
```

J'ai ajusté l'alpha (dernier paramètre) pour la transparence afin de ne pas avoir un orange trop pur et pas très jolie à l'affichage. L'orange est obtenu en combinant (255, 165, 0).

6/ Rafraîchissement de la fenêtre

`window.display()` affiche le résultat à l'écran.

Le passage de la souris, le hover, sur mes pièces fonctionne parfaitement. Nous pouvons continuer la programmation restante



Étape 13: Gestion du second événement souris

Rappel pour compiler

```
1 cmake -B build_main -S main
2 cmake --build build_main
3
4 ./build_main/bin/SFML_MVC_Yaltachess
```

Je souhaite à ce stade, pouvoir interagir sur mes pièces avec ma souris (sélection/désélection, déplacement des pièces) et qu'il m'indique aussi les positions possibles pour le déplacement de mes pièces.

Pour ce faire, je dois avancer en plusieurs étapes:

1/ Faire en sorte que chaque Case connaisse sa position logique (coordonnées grille)

- cela permet de traduire un clic pixel en coordonnées grille
- de demander au `Model` quelle pièce se trouve sur cette case
- générer et tester des déplacements en se basant sur des `Vector2i {x,y}`

J'ai ajouté un membre `gridPos` et un getter dans `Case`

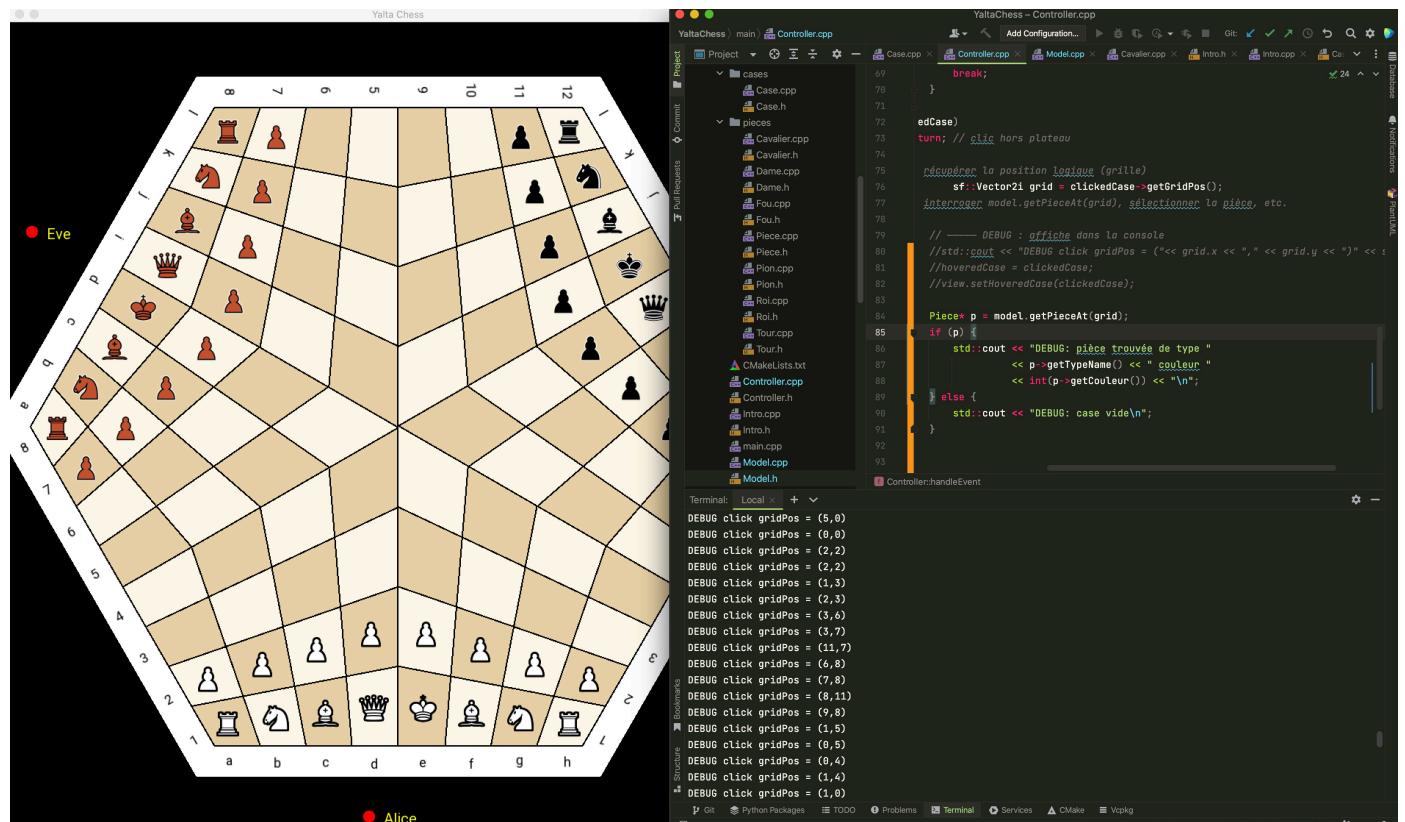
2/ intercepter l'évènement `MouseButtonPressed`

J'ai ajouté un bloc `MouseButtonPressed` dans le controller qui déclare `clickedCase` et récupère la case cliquée et appelle ensuite `getGridPos()`

J'ai placé un témoin pour visualiser si cela fonctionne correctement

```
1 // ----- DEBUG : affiche dans la console
2     std::cout << "DEBUG click gridPos = (" 
3             << grid.x << "," << grid.y << ")" << std::endl;
4 // ----- VISUEL : force le highlight de la case cliquée
5     hoveredCase = clickedCase;
6     view.setHoveredCase(clickedCase);
7 }
```

A chaque click sur une case, les coordonnées sont affichées sur le terminal

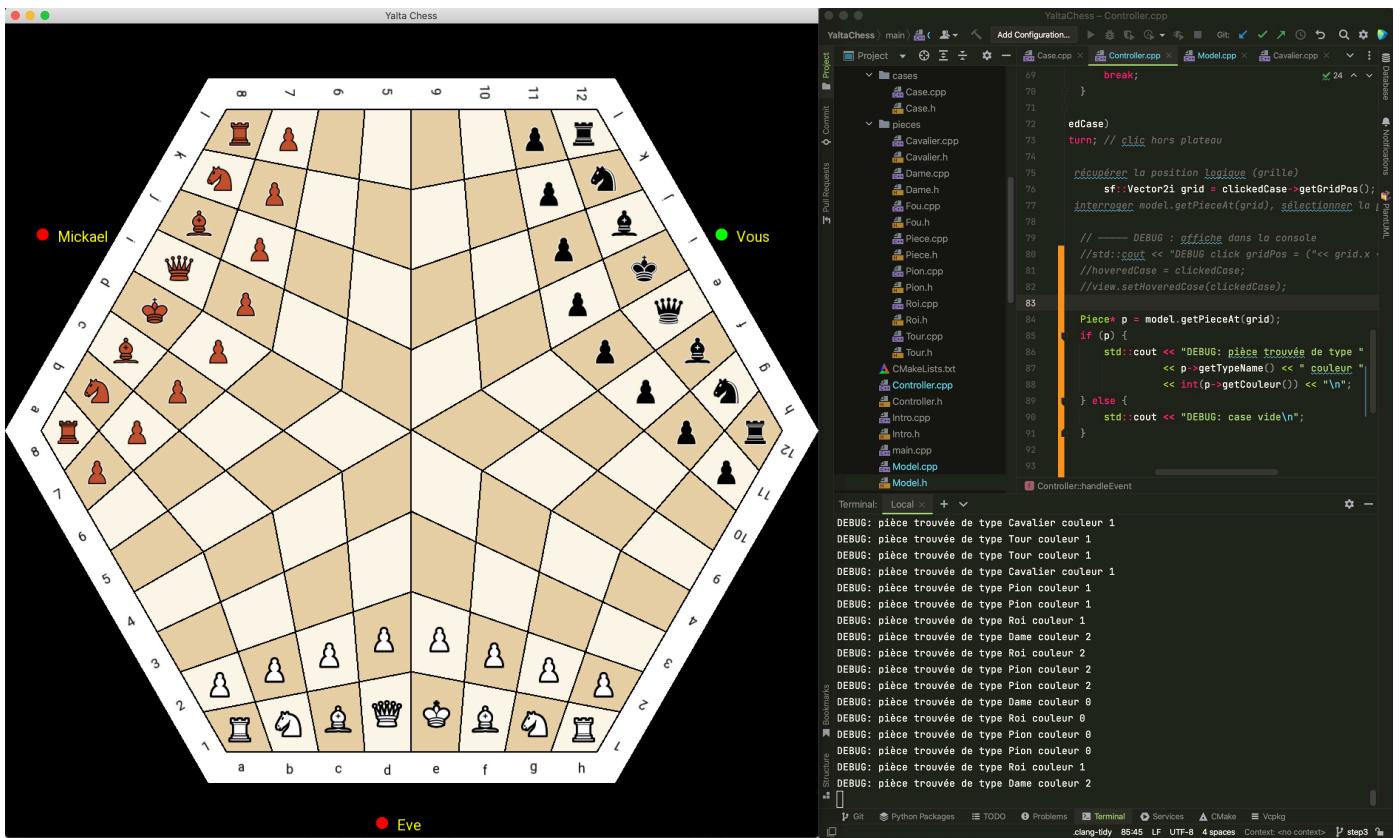


Ensuite j'ai implémenté une méthode pour connaitre le type et la couleur d'une pièce qui a été sélectionnée (click souris). On va l'appeler `Model::getPieceAt(sf::Vector2i)`

Mise en place d'un déboguage pour savoir si nous avons bien mis en place cette méthode

```

1 sf::Vector2i grid = clickedCase->getGridPos();
2 Piece* p = model.getPieceAt(grid);
3 if (p) {
4     std::cout << "DEBUG: pièce trouvée de type "
5             << p->getTypeName() << " couleur "
6             << int(p->getCouleur()) << "\n";
7 } else {
8     std::cout << "DEBUG: case vide\n";
9 }
```



La prochaine étape est de faire passer mon `Controller` de la simple détection des clics de ma souris à un gestionnaire en deus phases: sélection/désélection et déplacement et aussi, d'ajouter un moyen de visualiser les coups possibles (coups légaux)

On va se servir à la fois de notre Controller pour la sélection des pièces et coups possibles et de notre View pour dessiner le surlignage.

Je clique sur ma pièce, elle devient `selectedPiece` et la case correspondante devient verte, et ses cases destinations apparaissent vertes aussi (comme une assistance visuelle des coups possibles).

Je clique sur une des cases vertes, cela appelle `model.movePiece()`, ma pièce bouge puis tout est réinitialisé.

la gestion se fait avec l'évènement `MouseButtonPressed`

```
1 #include <SFML/Window/Event.hpp>
2 #include <SFML/Window/Mouse.hpp>
3
4 if (event.is<sf::Event::MouseButtonPressed>()) {
5     ...
```

Techniquement, dans `Controller`, au premier clic, on récupère la pièce, on calcule `getLegalMoves(model)` on convertit ces coordonnées `vector2i` en liste de `Case*` et on appelle `view.setHighlightedCases` pour afficher les cases en vert.

Au second click, si la case est dans `legalMoves`, on appelle `model.movePiece`, puis on réinitialise tout (`selectedPiece`, `legalMoves`, surlignage).

Attention: dans cette phase, la liste des coups possibles selon les pièces ne sont pas encore totalement parfaits. Il ya des erreurs que je corrigerais après dans une autre étape.

Les coups possibles sont définis dans chaque classe des pièces

un exemple ici pour le Cavalier.cpp

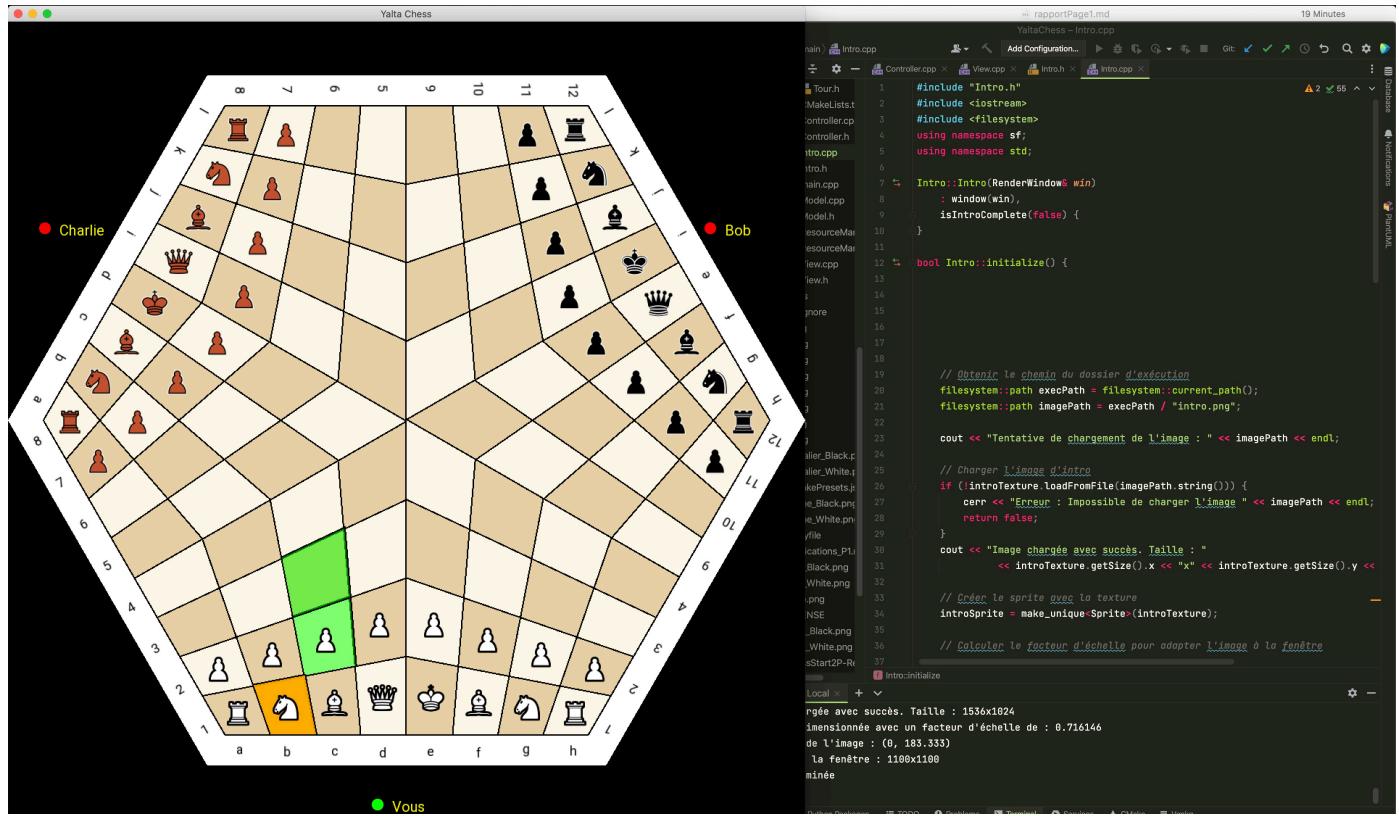
```
1 #include "Cavalier.h"
2 #include "Model.h"    // pour getPieceAt / isOccupied
3 using namespace sf;
4 using namespace std;
5
6 std::vector<Vector2i>
7 Cavalier::getLegalMoves(const Model& model) const {
8     static const array<Vector2i,8> jumps = {{
9         { 1, 2}, { 2, 1}, { 2,-1}, { 1,-2},
10        {-1,-2}, {-2,-1}, {-2, 1}, {-1, 2}
11    }};
12    vector<Vector2i> res;
13    for (auto d : jumps) {
14        Vector2i dest = position + d;
15        // bornes 0≤dest.x,y≤12
16        if (dest.x<0 || dest.x>=12 || dest.y<0 || dest.y>=12) continue;
17        if (!mouvementValide(dest)) continue;
18        Piece* cible = model.getPieceAt(dest);
19        if (!cible || cible->getCouleur() != coul)
20            res.push_back(dest);
21    }
```

```

22     return res;
23 }

```

Au final, voici une impression écran du résultat



Étape 14: Gestion des coups légaux sur un échiquier Yalta

Je pense que c'est la partie la plus dure avec le graphisme de l'échiquier. Je vais essayer de trouver des informations sur internet car bien que l'on trouve facilement pour un échiquier simple, je jeu d'échiquier Yalta est beaucoup plus complexe avec son style d'échiquier hexagonal à 3 joueurs.

Après des recherches sur internet, j'ai choisi les outils mathématiques avec lesquels je compte utiliser pour me faciliter la gestion des déplacements dans un grille complexe.

On va travailler avec des vecteurs de déplacement, c'est à dire avec des coordonnées cubiques ($x+y+z=0$). C'est à dire qu'à partir des positions de départ $\text{grid}(\text{colonne}, \text{ligne})$: ce qu'on appelle offset odd-r ou `Vector2i` comme integer, nous allons les convertir en coordonnées cubiques avec la méthode `hex::grilleverscube`

Pourquoi des coordonnées cubiques ? vs position (x,y) grille

- un seul conteneur de déplacements: toutes les directions possibles peuvent être représentées par 3 axes (dx, dy, dz) -> plus simple à itérer et à sommer pour les glissades
- le fait que $x + y + z = 0$ me garantit de rester sur le plan du jeu et évite des erreurs.

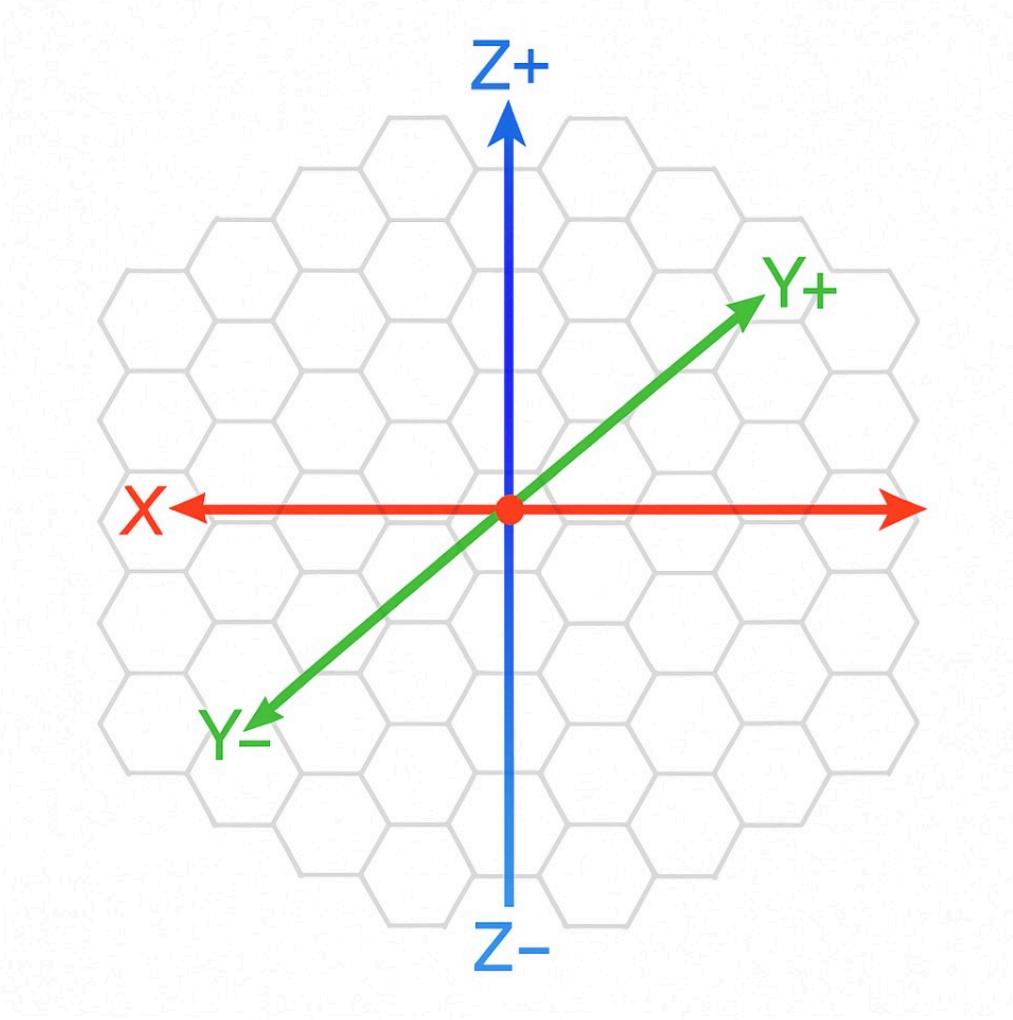
- La possibilité de gérer les glissades avec les pièces, sauts sans poser de problème avec les bordures compliquées d'un jeu d'échec Yalta
- Lisibilité et maintenance

```

1 // glisser en direction d (dx,dy,dz)
2 Cube nxt = { cur.x + d.x, cur.y + d.y, cur.z + d.z };

```

- Si on veut ajouter un nouveau type de pièce ou une règle spéciale, on ajoute simplement un vecteur ou une condition, sans retoucher toute la logique de voisinage.



Les coordonnées cubiques sont avant tout un outil mathématique qui simplifie tous les calculs de déplacement sur une grille hexagonale

Axe X (Rouge)

X^+ → flèche vers la droite (Est)

X^- → flèche vers la gauche (Ouest)

X^+ → **Est** : vecteur cubique (+1, -1, 0)

$X^- \rightarrow \text{Ouest}$: vecteur cubique (-1, +1, 0)

Axe Y (Vert)

$Y^+ \rightarrow$ flèche en diagonale vers le Nord-Est

$Y^- \rightarrow$ flèche en diagonale vers le Sud-Ouest

$Y^+ \rightarrow \text{Nord-Est}$: vecteur cubique (+1, 0, -1)

$Y^- \rightarrow \text{Sud-Ouest}$: vecteur cubique (-1, 0, +1)

Axe Z (Bleu)

$Z^+ \rightarrow$ flèche vers le haut (Nord)

$Z^- \rightarrow$ flèche vers le bas (Sud)

$Z^+ \rightarrow \text{Nord}$: vecteur cubique (0, -1, +1)

$Z^- \rightarrow \text{Sud}$: vecteur cubique (0, +1, -1)

Implémentation des coordonnées cubiques

L'implémentation des coordonnées cubiques (x,y,z) pour représenter chaque case du plateau va permettre en principe de surmonter:

La représentation des directions de déplacement devient complexe

Le calcul des distances entre cases n'est pas trivial

La détermination des cases voisines est moins intuitive

Les algorithmes de déplacement des pièces deviennent plus compliqués

Dans mon contexte de mon jeu:

- Chaque case est représentée par trois coordonnées (x,y,z) qui respectent la contrainte $x+y+z=0$
- Les six directions principales (Est, Nord-Est, Nord-Ouest, Ouest, Sud-Ouest, Sud-Est) sont représentées par des vecteurs simples
- Les calculs de distance et de voisinage deviennent plus intuitifs et cohérents

La structure `Cube` définie dans `HexagonalCubique.h` encapsule ces trois coordonnées :

```
1 | struct Cube { int x, y, z; };
```

Conversion entre systèmes de coordonnées

L'une des fonctionnalités clés est la conversion bidirectionnelle entre les coordonnées de grille 2D (utilisées pour l'affichage) et les coordonnées cubiques (utilisées pour la logique du jeu). Ces conversions sont implémentées dans le namespace `Hex` :

```
1 // Conversion d'une case en coordonnées "offset odd-r" (grille 2D) vers les
2 // coordonnées cubiques
3 inline Cube grilleversCube(const Vector2i &g) {
4     int q = g.x - (g.y - (g.y & 1)) / 2;
5     int r = g.y;
6     int x = q;
7     int z = r;
8     int y = -x - z;
9     return {x, y, z};
10}
11 // Conversion inverse : d'une position cubique vers la grille 2D "offset odd-r"
12 inline Vector2i cubeversGrille(const Cube &c) {
13     int q = c.x;
14     int r = c.z;
15     int col = q + (r - (r & 1)) / 2;
16     return {col, r};
17}
```

Ces fonctions permettent de maintenir deux représentations cohérentes du plateau :

- Une représentation visuelle en 2D pour l'interface utilisateur
- Une représentation logique en 3D (coordonnées cubiques) pour les calculs

Directions et voisage

Le système de coordonnées cubiques simplifie considérablement la définition des directions de déplacement. Les six directions principales sont définies comme des constantes :

```
1 static constexpr array<Cube, 6> DIRECTIONS = {
2     Cube{+1, -1, 0},    // 0 : Est
3     Cube{+1, 0, -1},   // 1 : Nord-Est
4     Cube{ 0, +1, -1},  // 2 : Nord-Ouest
5     Cube{-1, +1, 0},   // 3 : Ouest
6     Cube{-1, 0, +1},   // 4 : Sud-Ouest
7     Cube{ 0, -1, +1}   // 5 : Sud-Est
8 };
```

Cette représentation permet de calculer facilement les cases voisines en ajoutant simplement le vecteur de direction à la position actuelle :

```
1 | Cube voisin = position + DIRECTIONS[i];
```

Calcul des mouvements des pièces

L'utilisation des coordonnées cubiques simplifie considérablement l'implémentation des règles de déplacement des pièces. Par exemple, pour la Tour :

```
1 vector<Cube> mouvementsTour(const Cube position, const Model& model, Couleur
couleur) {
2     vector<Cube> resultat;
3
4     // La Tour se déplace en ligne droite dans les 6 directions hexagonales
5     for (const Cube& direction : DIRECTIONS) {
6         Cube positionActuelle = position;
7         while (true) {
8             positionActuelle = positionActuelle + direction;
9
10            // vérifier si on est toujours sur l'échiquier
11            if (!model.getCaseAtCube(positionActuelle)) break;
12
13            // vérifier si la case est occupée
14            if (auto piece = model.getPieceAtCube(positionActuelle)) {
15                // si la pièce est de couleur différente, on peut la capturer
16                if (piece->getCouleur() != couleur) {
17                    resultat.push_back(positionActuelle);
18                }
19                break; // on ne peut pas aller plus loin dans cette direction
20            }
21
22            // La case est vide, on peut y aller
23            resultat.push_back(positionActuelle);
24        }
25    }
26
27    return resultat;
28 }
```

Cette approche permet d'implémenter de manière élégante et cohérente les mouvements de toutes les pièces, y compris celles dont les déplacements sont plus complexes comme le Cavalier ou le Fou.

Conversion Cube vers Label

La notation algébrique "a1" par exemple est utilisée pour identifier les cases et enregistrer les coups. Pour maintenir cette convention familière aux joueurs d'échecs, il était nécessaire de développer un système de conversion entre les coordonnées cubiques et une notation algébrique adaptée au plateau hexagonal à trois joueurs pour au moins, afficher les coups joués sur la fenêtre principale.

La conversion entre coordonnées cubiques et notation algébrique est implémentée dans les fichiers `CubeToLabel.h` et `CubeToLabel.cpp`. Deux approches ont été explorées:

Conversion algorithmique : Une tentative initiale de conversion basée sur des calculs (visible dans la fonction `toAlgebrique` de `HexagonalCubique.h`)

Table de correspondance directe : L'approche finalement retenue, qui utilise une map statique pour associer directement chaque coordonnée cubique à son label :

```
1 std::string cubeToLabel(const Cube& c) {
2     // Table de correspondance directe entre coordonnées cube et labels
3     static const std::map<Cube, std::string, CubeCompare> cubeToLabelMap = {
4         {{4, -4, 0}, "A1"}, {{4, -5, 1}, "A2"}, {{3, -5, 2}, "A3"}, {{3, -6, 3},
5         "A4"}, 
6             // ... autres correspondances ...
7             {{9, -13, 4}, "L9"}, {{8, -12, 4}, "L10"}, {{7, -11, 4}, "L11"}, {{6, -10,
8             4}, "L12"}
9     };
10
11     auto it = cubeToLabelMap.find(c);
12     if (it != cubeToLabelMap.end()) {
13         return it->second;
14     }
15
16     // si les coordonnées ne sont pas trouvées, retourner une chaîne vide
17     return "";
18 }
```

Une fonction de test `testCubeToLabel` a également été implémentée pour vérifier la cohérence de la conversion sur l'ensemble des cases du plateau.

Conclusion: je pensais vraiment que toutes les obstacles allaient être plus facile à passer avec l'implémentation des coordonnées cube et des labels. Malheureusement pour moi, un bug s'est glissé dans mon code et impossible d'utiliser les avantages de ces coordonnées et de finaliser la totalité des fonctionnalités du jeu à ce stade.

Malgré les incohérences au niveau de la gestion des cases et pièces avec les coordonnées cube, j'ai implémenté des règles et contraintes pour les pièces du jeu.

Architecture des règles et contraintes

Au cœur de l'implémentation des règles se trouve la classe abstraite `Piece`, qui définit l'interface commune à toutes les pièces de mon jeu;

```
1 class Piece {
2 protected:
3     Cube positionCube; // Position cubique sur l'échiquier
4     Couleur couleur; // Couleur de la pièce (joueur)
5
6 public:
7     Piece(Cube pos, Couleur coul);
8     virtual ~Piece() = default;
9 }
```

```

10 // vérifie la validité du déplacement spécifique à chaque pièce
11 virtual bool mouvement valide(Cube nouvellePos) const = 0;
12
13 // retourne toutes les destinations valides selon le modèle
14 virtual vector<Cube> getLegalMoves(const Model&) const = 0;
15
16 // Dessine graphiquement la pièce (SFML)
17 virtual void dessiner(Renderwindow& window) const = 0;
18
19 void setPosition(Cube nouvellePos);
20 Couleur getCouleur() const;
21
22 virtual string getTypeName() const = 0;
23
24 // Accesseurs
25 const Cube& getPositionCube() const { return positionCube; }
26 void setPositionCube(const Cube& c) { positionCube = c; }
27
28 // Compatibilité IHM
29 Vector2i getPosition() const { return Hex::cubeversGrille(positionCube); }
30 };
31

```

Cette classe abstraite définit deux méthodes clés pour la gestion des règles:

- `mouvement valide(Cube nouvellePos)` : vérifie si un déplacement vers une position donnée est valide pour la pièce
- `getLegalMoves(const Model&)` : retourne toutes les destinations valides pour la pièce selon l'état actuel du jeu

Chaque type de pièce (Roi, Dame, Tour, Fou, Cavalier, Pion) hérite de la classe `Piece` et implémente ses propres règles de déplacement. Prenons l'exemple du Roi

```

1 class Roi : public Piece {
2 private:
3     Model* modelPtr;
4
5 public:
6     Roi(Cube pos, Couleur coul, Model* modelPtr);
7     bool mouvement valide(Cube nouvellePos) const override;
8     void dessiner(sf::RenderWindow& window) const override;
9     vector<Cube> getLegalMoves(const Model& model) const override;
10    string getTypeName() const override;
11

```

L'implémentation de `mouvement valide` et `getLegalMoves` pour le Roi utilise les fonctions utilitaires définies dans le namespace `Hex`:

```

1  bool Roi::mouvementValide(Cube nouvellePos) const {
2      auto mouvements = Hex::mouvementsRoi(positionCube, *modelPtr, couleur);
3      return std::find(mouvements.begin(), mouvements.end(), nouvellePos) != mouvements.end();
4  }
5
6  vector<Cube> Roi::getLegalMoves(const Model& model) const {
7      auto mouvements = Hex::mouvementsRoi(positionCube, model, couleur);
8
9      std::cout << "Mouvements légaux Roi = ";
10     for(const Cube& m : mouvements) std::cout << cubeToLabel(m) << ' ';
11     std::cout << '\n';
12
13     return mouvements;
14 }
```

et les règles de déplacement spécifiques à chaque type de pièce sont implémentées dans le namespace `Hex` du fichier `HexagonalCubique.cpp`. Toujours pour le Roi

```

1  vector<Cube> mouvementsRoi(const Cube position, const Model& model, Couleur couleur) {
2      vector<Cube> resultat;
3
4      // Le Roi se déplace d'une case dans toutes les directions
5      for (const Cube& direction : DIRECTIONS) {
6          Cube destination = position + direction;
7
8          // Vérifier si on est toujours sur l'échiquier
9          if (!model.getCaseAtCube(destination)) continue;
10
11         // Vérifier si la case est occupée
12         if (auto piece = model.getPieceAtCube(destination)) {
13             // Si la pièce est de couleur différente, on peut la capturer
14             if (piece->getCouleur() != couleur) {
15                 resultat.push_back(destination);
16             }
17             continue;
18         }
19
20         // La case est vide, on peut y aller
21         resultat.push_back(destination);
22     }
23
24     return resultat;
25 }
```

Gestion des événements utilisateur du contrôleur à la vue

Le contrôleur est responsable de la gestion des événements utilisateur et de la traduction de ces événements en actions sur le modèle:

Survol de la souris: le contrôleur détecte la case survolée et met à jour la vue en conséquence (il faut ne pas perdre en tête que SFML 3 rafraîchit le contenu périodiquement avec ce que l'on appelle la frame)

Clic sur une pièce : le contrôleur sélectionne la pièce et demande au modèle les mouvements légaux

Clic sur une destination : le contrôleur vérifie la validité du mouvement et demande au modèle d'effectuer le déplacement

```
1 void Controller::handleEvent(const sf::Event& event) {
2     // Gestion du survol de la souris
3     if (event.is<sf::Event::MouseMoved>()) {
4         // ...
5     }
6
7     // Gestion des clics
8     auto const* mouseBtn = event.getIf<sf::Event::MouseButtonPressed>();
9     if (!mouseBtn || mouseBtn->button != sf::Mouse::Button::Left) return;
10
11    // ...
12
13    // Si on a déjà une pièce sélectionnée
14    if (selectedCase) {
15        // Si on clique sur la même case, on désélectionne
16        if (clickedCase == selectedCase) {
17            selectedCase = nullptr;
18            view.setSelectedCase(nullptr);
19            view.setHighlightedCases({});
20            return;
21        }
22
23        // Vérifie si la case de destination est occupée par une pièce du joueur
24        courant
25        if (clickedCase->estOccupee() && clickedCase->getPiece()->getCouleur() ==
26        couleurCourante) {
27            return; // On ne peut pas déplacer une pièce sur une case occupée par
28        une pièce de même couleur
29        }
30
31        // Sinon on déplace la pièce vers la nouvelle case
32        Piece* pieceToMove = selectedCase->getPiece();
33        if (pieceToMove) {
34            if (!pieceToMove->mouvementValide(clickedCase->getCubePos())) {
35                view.setEventMessage("Déplacement illegal : coup refuse.");
36                selectedCase = nullptr;
37                view.setSelectedCase(nullptr);
38                view.setHighlightedCases({});
39                return; // on annule tout
```

```

37 }
38
39 // on déplace la pièce
40 model.movePieceCube(pieceToMove, clickedCase->getCubePos());
41
42 // on met à jour le message d'événement
43 string message = string("Le joueur ") +
44             (pieceToMove->getCouleur() == BLANC ? "Blanc" :
45             pieceToMove->getCouleur() == NOIR ? "Noir" : "Rouge")
46 +
47             " bouge son " + pieceToMove->getTypeName() +
48             " de " + cubeToLabel(selectedCase->getCubePos()) +
49             " vers " + cubeToLabel(clickedCase->getCubePos());
50
51 view.setEventMessage(message);
52
53 // On réinitialise la sélection
54 selectedCase = nullptr;
55 view.setSelectedCase(nullptr);
56 view.setHighlightedCases({});
57 }
58
59 // Si on n'a pas de pièce sélectionnée, on vérifie si on clique sur une pièce
60 // du joueur courant
61 if (clickedCase->estOccupee()) {
62     Piece* clickedPiece = clickedCase->getPiece();
63     if (clickedPiece && clickedPiece->getCouleur() == couleurCourante) {
64         selectedCase = clickedCase;
65         view.setSelectedCase(selectedCase);
66
67         // --- nouveau : calcule et souligne tous les déplacements légaux ---
68         vector<Cube> dests = clickedPiece->getLegalMoves(model);
69         vector<Case*> h1;
70         h1.push_back(selectedCase); // case source en orange
71
72         for (Cube c : dests)
73             if (auto ca = model.getCaseAtCube(c)) h1.push_back(ca);
74         view.setHighlightedCases(h1);
75     }
76 }
77
78 if (model.isPartieTerminee()) {
79     view.setEventMessage(model.getMessageFinPartie());
80 }

```

La vue `YaltaChessview` est responsable de l'affichage graphique du jeu. Elle reçoit des instructions du contrôleur pour mettre à jour l'affichage en fonction des actions de l'utilisateur.

Surbrillance des cases: la vue met en évidence la case sélectionnée et les destinations possibles

Affichage des messages: la vue affiche des messages d'événements (déplacement, capture, échec, etc.)

Rendu graphique: la vue dessine les pièces et les cases selon leur état actuel

```
1 void Yaltachessview::setHighlightedCases(const std::vector<Case*>& cases) {
2     highlightedCases = cases;
3 }
4
5 void Yaltachessview::setEventMessage(const string& message) {
6     eventText.setString(message);
7 }
```

Validation des mouvements et gestion des captures

La validation des mouvements se fait en deux étapes;

Vérification par la pièce: la méthode `mouvement valide` de la pièce vérifie si le mouvement respecte les règles spécifiques à ce type de pièce

Vérification par le contrôleur: le contrôleur vérifie des contraintes supplémentaires (tour du joueur, case occupée par une pièce amie, etc.)

La capture d'une pièce adverse est gérée par le modèle dans la méthode `movePieceCube`

```
1 void Model::movePieceCube(Piece* p, const Cube& dest) {
2     // Sauvegarde de la position d'origine
3     Cube from = p->getPositionCube();
4
5     // Vérifie si une pièce adverse occupe la destination
6     if (Piece* captured = getPieceAtCube(dest)) {
7         // Notifie les observateurs de la capture
8         notifyPieceCaptured(captured);
9
10        // Supprime la pièce capturée
11        removePiece(captured);
12    }
13
14    // Déplace la pièce
15    p->setPositionCube(dest);
16
17    // Notifie les observateurs du déplacement
18    notifyPieceMoved(p, from, dest);
19
20    // Passe au joueur suivant
21    nextPlayer();
22
23    // Vérifie si la partie est terminée
24    verifierFinPartie();
25 }
```

Gestion de l'échec et mat

Le modèle vérifie après chaque coup si un joueur est en échec ou en échec et mat

```

1 void Model::verifierFinPartie() {
2     // Vérifie si le joueur actuel est en échec
3     bool echec = estEnEchec(joueurActuel);
4
5     // Vérifie si le joueur actuel a des mouvements légaux
6     bool aMouvements = aMouvementsLegaux(joueurActuel);
7
8     if (echec && !aMouvements) {
9         // Échec et mat
10        partieTerminee = true;
11        messageFinPartie = "Échec et mat ! Le joueur " +
12                           (joueurActuel == BLANC ? "Blanc" :
13                            joueurActuel == NOIR ? "Noir" : "Rouge") +
14                           " a perdu.";
15
16        // Notifie les observateurs
17        notifyGameStateChanged(true, messageFinPartie);
18    }
19    else if (!aMouvements) {
20        // Pat
21        partieTerminee = true;
22        messageFinPartie = "Pat ! Le joueur " +
23                           (joueurActuel == BLANC ? "Blanc" :
24                            joueurActuel == NOIR ? "Noir" : "Rouge") +
25                           " ne peut plus bouger.";
26
27        // Notifie les observateurs
28        notifyGameStateChanged(true, messageFinPartie);
29    }
30 }
31 }
```

Ensuite j'ai implémenté des Design Paterns. J'ai eu l'aide d'une de Chatgpt pour l'implémentation pour le Design Pattern Observer

le Design Pattern Factory

Le pattern Factory est implémenté via la classe `PieceFactory` dans le fichier `Model.h`. Ce pattern permet de centraliser la création des différentes pièces du jeu d'échecs, en encapsulant la logique d'instanciation et en fournissant une interface unifiée pour créer des objets de différentes classes dérivées de `Piece`.

```
1 // Factory pour les pièces
```

```

2 class PieceFactory {
3 public:
4     static unique_ptr<Piece> createPiece(const string& type, const Cube& pos,
5 Couleur coul, Model* model) {
6         if (type == "Roi") return make_unique<Roi>(pos, coul, model);
7         if (type == "Pion") return make_unique<Pion>(pos, coul, model);
8         if (type == "Cavalier") return make_unique<Cavalier>(pos, coul, model);
9         if (type == "Fou") return make_unique<Fou>(pos, coul, model);
10        if (type == "Tour") return make_unique<Tour>(pos, coul, model);
11        if (type == "Dame") return make_unique<Dame>(pos, coul, model);
12
13        // Si le type n'est pas reconnu, on affiche une erreur
14        cerr << "Erreur : Type de pièce inconnu : " << type << endl;
15        return nullptr;
16    }
17
18    // Méthode pour obtenir la liste des types de pièces disponibles
19    static vector<string> getAvailablePieceTypes() {
20        return {"Roi", "Pion", "Cavalier", "Fou", "Tour", "Dame"};
21    }
22
23    // Méthode pour vérifier si un type de pièce est valide
24    static bool isValidPieceType(const string& type) {
25        auto types = getAvailablePieceTypes();
26        return find(types.begin(), types.end(), type) != types.end();
27    }
28};

```

Son fonctionnement:

Méthode de création centralisée: la méthode statique `createPiece` prend en paramètre le type de pièce (sous forme de chaîne de caractères), sa position, sa couleur et une référence au modèle.

Instanciation dynamique: en fonction du type spécifié, la factory crée l'instance appropriée de la classe dérivée de `Piece` (Roi, Dame, Tour, etc.) en utilisant `make_unique` pour une gestion automatique de la mémoire.

Validation des types: la méthode `isValidPieceType` permet de vérifier si un type de pièce est valide avant de tenter de le créer.

Enumération des types disponibles: la méthode `getAvailablePieceTypes` fournit la liste de tous les types de pièces que la factory peut créer.

La factory est principalement utilisée lors de l'initialisation du plateau de jeu dans la méthode `initialiserPieces()` du `Model`

```

1 // Extrait de Model.cpp (initialiserPieces)
2 void Model::initialiserPieces() {
3     // ...

```

```

4   for (int y = 0; y < 12; ++y) {
5     for (int x = 0; x < 12; ++x) {
6       auto [coul, type] = SETUP[y][x];
7       if (coul < 0 || type < 0) continue;
8
9       // Conversion des indices en types de pièces
10      string pieceType;
11      switch (type) {
12        case 0: pieceType = "Roi"; break;
13        case 1: pieceType = "Pion"; break;
14        case 2: pieceType = "Cavalier"; break;
15        case 3: pieceType = "Fou"; break;
16        case 4: pieceType = "Tour"; break;
17        case 5: pieceType = "Dame"; break;
18        default: continue;
19      }
20
21      // Conversion des indices en couleurs
22      Couleur couleur;
23      switch (coul) {
24        case 0: couleur = BLANC; break;
25        case 1: couleur = ROUGE; break;
26        case 2: couleur = NOIR; break;
27        default: continue;
28      }
29
30      // Création de la pièce via la factory
31      Cube cubePos = Hex::grilleVersCube({x, y});
32      auto piece = PieceFactory::createPiece(pieceType, cubePos, couleur,
33                                              this);
34      if (piece) {
35        pieces.push_back(std::move(piece));
36      }
37    }
38    // ...
39  }
40

```

Le Design Pattern Observer

Le pattern Observer est implémenté via les interfaces `GameObserver` et `GameObservable` dans le fichier `Model.h`, avec une implémentation concrète d'observateur dans la classe `GameLogger` (fichier `GameLogger.h`). Ce pattern permet de notifier automatiquement plusieurs objets (observateurs) des changements d'état dans un objet observé (le modèle du jeu), sans créer de couplage fort entre ces objets.

```

1 // Interface Observer
2 class GameObserver {
3 public:
4     virtual ~GameObserver() = default;
5     virtual void onPieceMoved(Piece* piece, const Cube& from, const Cube& to) = 0;
6     virtual void onPieceCaptured(Piece* captured) = 0;
7     virtual void onGameStateChanged(bool isGameOver, const string& message) = 0;
8 };
9
10 // Interface Observable
11 class GameObservable {
12 public:
13     virtual ~GameObservable() = default;
14     virtual void addObserver(GameObserver* observer) = 0;
15     virtual void removeObserver(GameObserver* observer) = 0;
16 protected:
17     virtual void notifyPieceMoved(Piece* piece, const Cube& from, const Cube& to) =
18 0;
19     virtual void notifyPieceCaptured(Piece* captured) = 0;
20     virtual void notifyGameStateChanged(bool isGameOver, const string& message) =
21 0;
21

```

La classe `Model` implémente l'interface `GameObservable` et gère une liste d'observateurs

```

1 class Model : public GameObservable
2 {
3 private:
4     vector<GameObserver*> observers;
5     // ...
6
7 public:
8     // ...
9
10    // Implémentation des méthodes de GameObservable
11    void addObserver(GameObserver* observer) override {
12        observers.push_back(observer);
13    }
14
15    void removeObserver(GameObserver* observer) override {
16        auto it = find(observers.begin(), observers.end(), observer);
17        if (it != observers.end()) {
18            observers.erase(it);
19        }
20    }
21
22 protected:

```

```

23     void notifyPieceMoved(Piece* piece, const Cube& from, const Cube& to) override
24     {
25         for (auto observer : observers) {
26             observer->onPieceMoved(piece, from, to);
27         }
28     }
29
30     void notifyPieceCaptured(Piece* captured) override {
31         for (auto observer : observers) {
32             observer->onPieceCaptured(captured);
33         }
34     }
35
36     void notifyGameStateChanged(bool isGameOver, const string& message) override {
37         for (auto observer : observers) {
38             observer->onGameStateChanged(isGameOver, message);
39         }
40     };

```

La classe `GameLogger` implémente l'interface `GameObserver` pour journaliser les événements du jeu

```

1  class GameLogger : public GameObserver {
2  private:
3      ofstream logfile;
4      // ...
5
6  public:
7      GameLogger() {
8          // Initialisation du fichier de log
9          // ...
10     }
11
12     void onPieceMoved(Piece* piece, const Cube& from, const Cube& to) override {
13         if (!logfile.is_open()) {
14             cerr << "Erreur : Fichier de log non ouvert dans onPieceMoved" << endl;
15             return;
16         }
17
18         string couleur;
19         switch(piece->getCouleur()) {
20             case BLANC: couleur = "Blanc"; break;
21             case NOIR: couleur = "Noir"; break;
22             case ROUGE: couleur = "Rouge"; break;
23         }
24
25         logfile << getCurrentTimestamp() << " - Déplacement : "
26             << couleur << " " << piece->getTypeName()
27             << " de (" << from.x << "," << from.y << "," << from.z << ")"

```

```

28         << " vers (" << to.x << "," << to.y << "," << to.z << ")\\n";
29     logFile.flush();
30 }
31
32 void onPieceCaptured(Piece* captured) override {
33     // Journalisation des captures
34     // ...
35 }
36
37 void onGameStateChanged(bool isGameOver, const string& message) override {
38     // Journalisation des changements d'état du jeu
39     // ...
40 }
41 };
42

```

Son fonctionnement:

Enregistrement des observateurs: les objets intéressés par les événements du jeu s'enregistrent auprès du modèle via la méthode `addobserver`.

Notification des événements: lorsqu'un événement important se produit (déplacement de pièce, capture, changement d'état du jeu), le modèle notifie tous les observateurs enregistrés en appelant les méthodes appropriées.

Traitement des notifications: chaque observateur traite les notifications selon les besoins spécifiques (journalisation, mise à jour de l'interface, etc.).

Le pattern Observer est utilisé pour notifier les événements importants du jeu, comme dans la méthode `movePieceCube` du `Model`

```

1 void Model::movePieceCube(Piece* p, const Cube& dest) {
2     // Sauvegarde de la position d'origine
3     Cube from = p->getPositionCube();
4
5     // vérifie si une pièce adverse occupe la destination
6     if (Piece* captured = getPieceAtCube(dest)) {
7         // Notifie les observateurs de la capture
8         notifyPieceCaptured(captured);
9
10        // Supprime la pièce capturée
11        removePiece(captured);
12    }
13
14    // Déplace la pièce
15    p->setPositionCube(dest);
16
17    // Notifie les observateurs du déplacement
18    notifyPieceMoved(p, from, dest);
19

```

```

20 // Passe au joueur suivant
21 nextPlayer();
22
23 // Vérifie si la partie est terminée
24 verifierFinPartie();
25 }
26

```

j'ai aussi implémenté un système d'affichage:

L'affichage des messages de coups joués s'inscrit dans l'architecture MVC (Modèle-Vue-Contrôleur) du jeu et implique principalement deux composants

La vue (YaltaChessView): responsable de l'affichage graphique des messages

Le contrôleur (Controller): responsable de la génération des messages en fonction des actions du joueur.

La classe `YaltaChessView` déclare un attribut `eventText` pour stocker le message à afficher et une méthode `setEventMessage` pour mettre à jour ce message

```

1 // Dans View.h
2 class YaltaChessView
3 {
4     private:
5         // ...
6         Text eventText; // Texte pour afficher les événements du jeu
7         // ...
8
9     public:
10        // ...
11        void setEventMessage(const string& message); // Méthode pour mettre à jour le
12        // ...
13    };

```

Le texte d'événement est initialisé dans le constructeur de `YaltaChessView`

```

1 // Dans View.cpp
2 YaltaChessView::YaltaChessView(RenderWindow &win, const Model &mod)
3     : window(win), model(mod), eventText(tempFont)
4 {
5     // ...
6
7     // Configure le texte d'événement
8     eventText.setFont(coordFont);
9     eventText.setString("Bienvenue dans Yalta Chess !");
10    eventText.setCharacterSize(24);
11    eventText.setFillColor(Color::White);

```

```

12     eventText.setPosition(Vector2f(20.f, 20.f)); // Position en haut à gauche
13
14 // ...
15 }
```

La méthode `setEventMessage` permet de mettre à jour le contenu du message

```

1 // Dans View.cpp
2 void YaltaChessView::setEventMessage(const string& message) {
3     eventText.setString(message);
4 }
```

La méthode `draw` de la vue s'occupe de rendre le texte à l'écran

```

1 // Dans View.cpp
2 void YaltaChessview::draw()
3 {
4     // ...
5
6     // Dessin du texte d'événement (en mode coordonnées fenêtre)
7     window.setView(window.getDefaultView());
8     window.draw(eventText);
9     window.setView(boardview);
10
11    // ...
12 }
```

Le contrôleur est responsable de la génération des messages en fonction des actions du joueur. Plusieurs types de messages sont générés

```

1 // Messages d'erreur pour les coups invalides
2 if (!pieceToMove->mouvement valide(clickedCase->getCubePos())) {
3     view.setEventMessage("Deplacement illegal : coup refuse.");
4     selectedCase = nullptr;
5     view.setSelectedCase(nullptr);
6     view.setHighlightedCases({});
7     return; // on annule tout
8 }
9
```

```

1 // Messages de confirmation pour les coups valides
2 // Dans Controller.cpp
3 // on déplace la pièce
4 model.movePieceCube(pieceToMove, clickedCase->getCubePos());
5
```

```

6 // On met à jour le message d'événement
7 string message = string("Le joueur ") +
8     (pieceToMove->getCouleur() == BLANC ? "Blanc" :
9      pieceToMove->getCouleur() == NOIR ? "Noir" : "Rouge") +
10     " bouge son " + pieceToMove->getTypeName() +
11     " de " + cubeToLabel(selectedCase->getCubePos()) +
12     " vers " + cubeToLabel(clickedCase->getCubePos());
13 view.setEventMessage(message);
14

```

```

1 // Messages de fin de partie
2 // Dans Controller.cpp
3 if (model.isPartieTerminee()) {
4     view.setEventMessage(model.getMessageFinPartie());
5 }

```

les messages des coups joués sont du type:

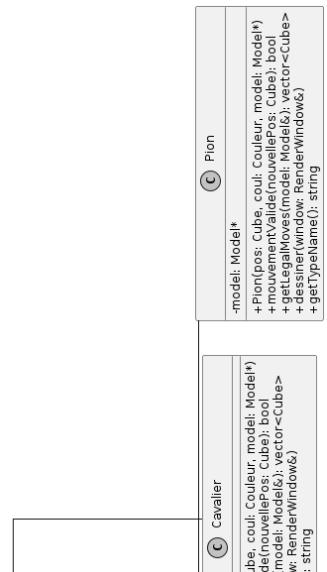
L'identification du joueur "Le joueur Blanc/Noir/Rouge" +

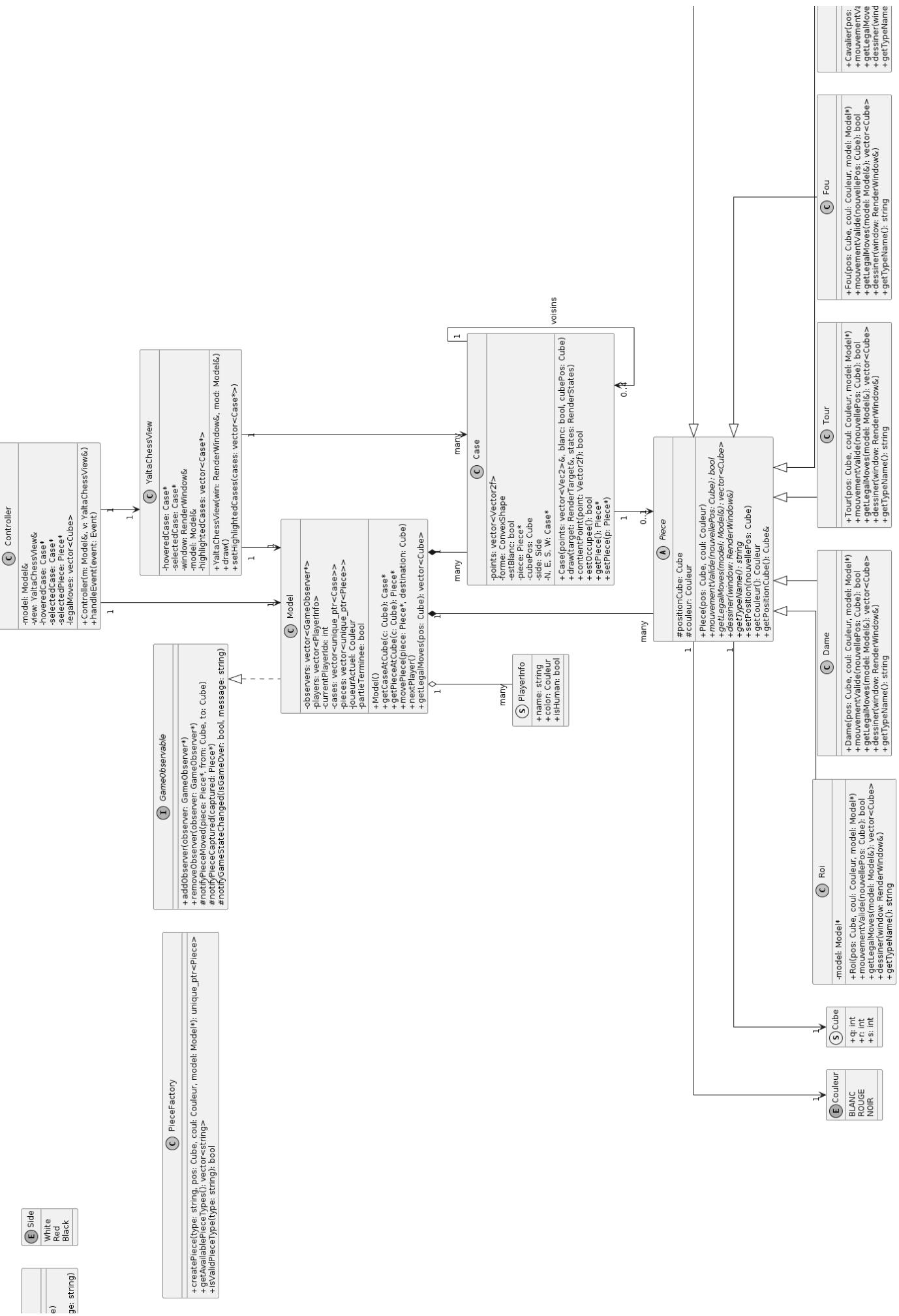
Le type de pièce déplacée "bouge son Roi/Dame/Tour/etc." +

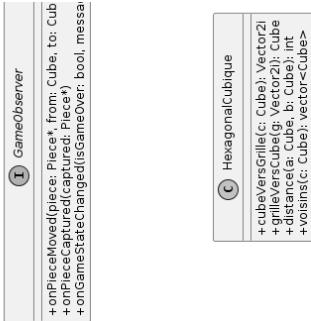
La position d'origine "de A1" +

La position de destination "vers B2"

Diagramme de classes plus détaillé







Conclusion

Réflexion personnelle sur le projet de jeu d'échecs à trois joueurs (Yalta)

Malgré les problèmes d'incohérences rencontrés au niveau des coordonnées cubiques et la non-finalisation de certaines fonctionnalités, ce projet de fin de semestre s'est révélé extrêmement enrichissant sur le plan pédagogique et technique.

Défis techniques et apprentissages

La mise en œuvre d'un système de coordonnées cubiques pour représenter un plateau hexagonal a constitué l'un des défis majeurs de ce projet. Bien que certaines incohérences persistent dans l'implémentation actuelle, ce travail m'a permis d'approfondir ma compréhension des systèmes de coordonnées non conventionnels et de leur application dans un contexte de jeu. La conversion entre différents systèmes de représentation (coordonnées cubiques, notation algébrique, coordonnées d'affichage) a nécessité une réflexion approfondie et m'a confronté à des problèmes mathématiques stimulants.

Application concrète du pattern MVC en C++

L'architecture Model-View-Controller (MVC) adoptée pour ce projet a considérablement amélioré ma compréhension de ce pattern de conception fondamental. Contrairement à son utilisation dans des frameworks web où le MVC est souvent imposé par l'infrastructure, l'implémentation manuelle en C++ m'a obligé à réfléchir attentivement à la séparation des responsabilités.

Exploration approfondie de SFML

L'utilisation de la bibliothèque SFML (Simple and Fast Multimedia Library) a été une expérience particulièrement formatrice. Au-delà des fonctionnalités de base pour l'affichage de formes et la gestion des événements, j'ai pu explorer des aspects plus avancés.

Implémentation de design patterns

L'intégration de design patterns comme Factory et Observer a renforcé ma compréhension de ces concepts théoriques en les appliquant à des problèmes concrets. La Factory pour la création des pièces et l'Observer pour la notification des événements du jeu ont démontré l'utilité de ces patterns pour créer un meilleur code.

En définitive, ce projet illustre parfaitement comment les difficultés techniques peuvent se transformer en opportunités d'apprentissage significatives lorsqu'elles sont abordées avec persévérance et curiosité. L'utilisation de l'IA m'a permis d'aller plus en profondeur dans les possibilités et réaliser des compléments d'informations au-delà d'un outil de débogage.