Project : “Chess World Game Using Unreal Engine”

Team members: 24125073 - Nguyễn Trọng Hùng Phong

24125019 - Nguyễn Chí Tính

Date of Submission: 21 / 12 / 2024

Table of contents:

1. Project Overview:

Summary: The Chess World Game using Unreal Engine is a 3D chess game that enhances traditional gameplay with immersive graphics and interactive environments.

Aims: Make the chess game feature detailed 3D models, smooth animations, AI opponents, and multiplayer support, aiming to offer a more engaging and modern chess experience.

2. Game features:

- A chess game with 3D model UI.
- Display moves history.
- Add piece animations smoothly.
- Implement sound effects for piece moves, check, and checkmate.
- Access game settings for customization(board color, piece design, sound effects).
- Handle user inputs smoothly via mouse.
- Save the current game state and load a previously saved game.
- Two-player mode
- AI mode with three levels(easy, medium, hard).

3. Architecture:

Overview:

This project explores the architecture of a 3D Chess Game implemented in UE4. The system is developed almost in C++ and structured based on Object-Oriented Programming (OOP) principles, with a clear separation between the Logic Module and Graphics Module. In Logic Module, we built all chess-related algorithms based on 2D board structure, including piece movements, checkmate, promotion, and other game rules. For Graphic Module, this module's responsibility is to render a graphic board and handle events from users. We set up actors and components like cameras, lightning, box_collision, object, physical properties of actors in the UE4 map. We designed a basic blueprint and used C++ to develop them to support interactive user interfaces.

Input from Player or AI:

Player Input: The player interacts with the 3D GraphicBoard by dragging and dropping pieces, or left click to choose and right click to move pieces. These interactions are encoded into 2D coordinates (x, y) for the LogicBoard.

AI Input (AI Gen Step): The AI generates piece movements based on 2D LogicBoard using genStep() function. This move was implemented by format: initial position (x, y) and final position (fX, fY). This was encoded in the opposite direction to 64-tiles array (tilesCollection[64]) by a mapping with formula (x * 8 + y), where each tile corresponds to positions of pieces in 3D GraphicBoard.

Logic Module Processing:

Board Management: The Board class holds the LogicBoard, which is a 8x8 matrix, where each cell holds a reference to a piece (such as King, Knight, Rook, Pawn, etc.). The Board is updated when a move is made. With cells. With empty cells, we create a class Blank to avoid memory leaks and ensure valid reference to have better memory management.

Piece Movement & Validation: Each chess piece inherits Piece class(), for instance: Knight class has name and color properties, has its movement logic. When the player selects a piece, an event is handled. The generateMove() (based on Validator.h) function is used to generate all possible valid moves for that piece based on the current state of LogicBoard.

- The initial coordinates (x, y) were encoded from clicking, dragging events used to check the potential valid move pairs (x, y), (fX, fY). If a matching is found, this move is valid.

- When a player makes a move on the Graphic Board, now we can ensure that move is valid. So we encoded 3D position (x, y, z) to 2D coordinates matrix[8][8] of LogicBoard.

AI Move Generation: We use the minimax algorithm and alpha-beta pruning. The AI evaluates the available moves based on a strategy, which involves calculating the best move considering possible future states of the board (numbers of future states follow depth of the tree).

Update Logic Board:

When we have user movements or AI generative movements, those movements are validation moves. The Logic Board is updated to reflect the new position of the pieces.

Graphic Board Rendering Layer:

After the logic layers processes and the Logic Board is updated, the rendering layer must update the Graphic Board based on the Logic Board. This phase has to update the 3D position of Graphic Pieces and change suitable material of the 3D piece to match with 2D LogicBoard.

This phase also handles special cases, such as:

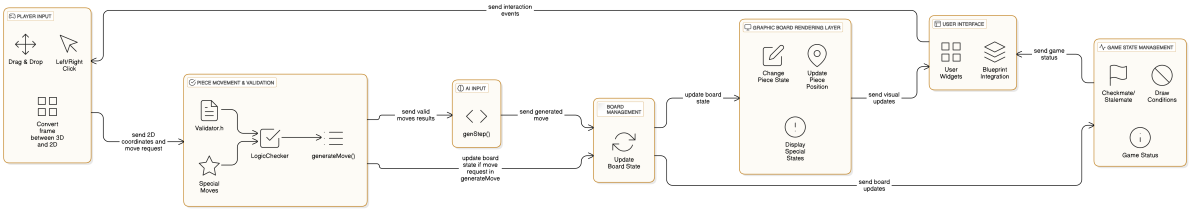
- Checkmate or Stalemate: If the game reaches a checkmate or stalemate condition, the game's end state is displayed.
- Castling: If the user opts for castling, the pieces' movements are reflected in the 3D display.
- En Passant: The pawn's en passant move is shown in the graphics when applicable.
- Promotion: When a pawn reaches the opponent's back rank, the user is prompted to choose a new piece, and the graphics are updated accordingly.
- Game State Management: tracking and determining the status of game such as checkmate, stalemate, draw conditions, game over.

User interface:

Management handles user interaction and presentation through C++ and Unreal Engine's user widgets:

- User Widgets**:** Created using C++ classes, these widgets allow users to interact with the game (e.g., selecting pieces, making moves, displaying game status).

- Blueprint Integration**:** A simple Blueprint setup is used to add a widget to the viewport.
- User Interaction: The UI responds to user actions, such as dragging and dropping pieces, clicking to select, and displaying real-time game status (e.g., check, checkmate, turn indicator).



4. Implementation Details:

Code Organization: The provided code is a part of a chess engine that handles various aspects of chess gameplay, such as piece movement, game state validation, and board representation. The code is structured into the following components:

1. **Core Components**:
   - **applyMove**: Applies a chess move to the board, handling special cases like castling, en passant, and pawn promotion.

```
void applyMove(Board &board, const pair <char, pair <
ii, ii>> Move) {
    auto &[tp, move] = Move;

    int x = move.first.first, y = move.first.second;
    int fX = move.second.first, fY = move.second.seco
nd;

    board.matrix[fX][fY] = board.matrix[x][y];
    board.matrix[x][y] = new Blank(); // apply move

    char cn = board.matrix[fX][fY]->name;
    if (cn == 'K') {
        int id = board.matrix[fX][fY]->color;
        board.kingx[id] = fX, board.kingy[id] = fY;
```

```
// Navigate King
    }
    if (cn == 'K' || cn == 'R' || cn == 'P') {
        board.matrix[fX][fY]->cMove += 1; // Update c
Move
    }

    if (tp == 'C') {
        if (fY < y) board.matrix[x][fY + 1] = board.m
atrix[x][1], board.matrix[x][1] = new Blank();
        else board.matrix[x][fY - 1] = board.matrix
[x][8], board.matrix[x][8] = new Blank(); // Move Roo
k in Castling
    }
    else if (tp == 'E') {
        board.matrix[x][fY] = new Blank(); // Capturn
Pawn in En Passant
    }
    else if (tp == 'n' || tp == 'b' || tp == 'r' || t
p == 'q') {
        board.matrix[fX][fY]->name = tp - 32; // Prom
oted Piece
    }

    if (!board.all_piece_moves.empty()) {
        auto &[tp2, move2] = board.all_piece_moves.ba
ck();
        fX = move2.second.first, fY = move2.second.se
cond;
        if (board.matrix[fX][fY]->name == 'P' && boar
d.matrix[fX][fY]->cMove == 1)
            board.matrix[fX][fY]->cMove ++; // Update
cMove for case En Passant not immediately
    }

    board.all_piece_moves.push_back(Move); // Update
history move
}
```

- **undoMove**: Reverts the last move on the board, restoring the board state and handling special moves.

```
void undoMove(Board &board, const pair <char, pair <i
i, ii>> Move, Piece* past) {
    auto &[tp, move] = Move;

    int x = move.first.first, y = move.first.second;
    int fX = move.second.first, fY = move.second.seco
nd;

    board.matrix[x][y] = board.matrix[fX][fY];
    board.matrix[fX][fY] = past; // undo move

    char cn = board.matrix[x][y]->name;
    if (cn == 'K') {
        int id = board.matrix[x][y]->color;
        board.kingx[id] = x, board.kingy[id] = y; //
Navigate King
    }
    if (cn == 'K' || cn == 'R' || cn == 'P') {
        board.matrix[x][y]->cMove -= 1; // Update cMo
ve
    }

    if (tp == 'C') {
        if (fY < y) board.matrix[x][1] = board.matrix
[x][fY + 1], board.matrix[x][fY + 1] = new Blank();
        else board.matrix[x][8] = board.matrix[x][fY
- 1], board.matrix[x][fY - 1] = new Blank(); // Move
Rook back to its original position
    }
    else if (tp == 'E') {
        board.matrix[x][fY]->name = 'P';
        board.matrix[x][fY]->color = 1 ^ board.matrix
[x][y]->color;
        board.matrix[x][fY]->cMove = 1; // Replace ca
ptured Pawn
```

```
    }
    else if (tp == 'n' || tp == 'b' || tp == 'r' || t
p == 'q') {
        board.matrix[x][y]->name = 'P'; // Demote
    }

    board.all_piece_moves.pop_back(); // Update histo
ry move

    if (board.all_piece_moves.empty()) return ;

    auto &[tp2, move2] = board.all_piece_moves.back
();
    fX = move2.second.first, fY = move2.second.secon
d;
    if (board.matrix[fX][fY]->name == 'P' && board.ma
trix[fX][fY]->cMove == 2)
        board.matrix[fX][fY]->cMove --; // Update cMo
ve for case En Passant not immediately
}
```

- **generateMoves**: Generates all checked valid moves and for a player, including special moves like castling, en passant and pawn promotions.

```
vector <pair <char, pair<ii, ii>>> generateMoves(Boar
d &board, int turn) {
    vector <pair <char, pair<ii, ii>>> validMoves;

    castling(board, turn, validMoves); // push castli
ng

    for (int i = 1; i <= 8; i++) {
        for (int j = 1; j <= 8; j++) {
            Piece* piece = board.matrix[i][j];
            if (piece->color != turn) continue;

            vector<pair<int, int>> pieceMoves;
            get_valid_move(pieceMoves, board, i, j, p
```

```
iece->name);

            int tmp2 = turn ? 1 : 8;
            for (const auto &[x, y] : pieceMoves) {
                Piece* tmp = board.matrix[x][y];
                pair <char, pair <ii, ii>> move = {tm
p->name, {mp(i, j), mp(x, y)}};

                if (piece->name == 'P' && j != y && t
mp->color == -1)
                    move.fi = 'E'; // En Passant

                applyMove(board, move);

                if (!isKingInCheck(board, turn)) {
                    if (piece->name == 'P' && x == tm
p2) { // push all valid promotion
                        move.fi = 'q', validMoves.pus
h_back(move);
                        move.fi = 'r', validMoves.pus
h_back(move);
                        move.fi = 'b', validMoves.pus
h_back(move);
                        move.fi = 'n', validMoves.pus
h_back(move);
                    }
                    else validMoves.push_back(move);
// push normal move
                }

                undoMove(board, move, tmp);
            }
        }
    }

    return validMoves;
}
```

- **isKingInCheck**: Determines whether a player's king is in check.

```
bool isKingInCheck(Board &board, const int &turn) {
    int kingX = board.kingx[turn];
    int kingY = board.kingy[turn];

    for (int i = 1; i <= 8; i++)
        for (int j = 1; j <= 8; j++) {
            Piece* piece = board.matrix[i][j];
            if (piece->color != -1 && piece->color != t
                vector<pair<int, int>> validMoves;
                get_valid_move(validMoves, board, i, j,
                for (const auto &move : validMoves)
                    if (move.first == kingX && move.sec
                        return true;
            }
        }
    return false;
}
```

2. **Special Rules**:
   - **Castling:** Managed through the castling function, which ensures all conditions are met and verifies that the king does not move through or into a square under attack.
   - **En Passant**: Handled in validator, ensuring correct pawn capture when the opponent's pawn moves for the first time.
3. **Chess Bot**:
   - **pieceValue**: Assigns a numeric value to each piece for evaluation purposes.
   - **Position Scores**: Defined in PIECE_POS_POINTS to encourage optimal placement of pieces during gameplay.
   - **evaluateBoard:** assesses the value of the chessboard state based on pieces, their positions, and game outcomes (win, loss, draw)
   - **minimaxAlphaBeta:** This is the recursive function implementing the Minimax algorithm with Alpha-Beta Pruning, Move Sorting (prioritizes moves that are more likely to yield better outcomes, improving the efficiency of Alpha-Beta pruning). It evaluates the board state to determine the best move for the current player.

- **genStep:** This function generates the best move for the current player using the minimaxAlphaBeta algorithm.

4. **Game State Management**:
    - The GameState enum tracks the game's status (ONGOING, WHITE_WINS, BLACK_WINS, DRAW).
    - Move history is maintained in board.all_piece_moves, and captured pieces are stored for undo functionality.
    - **isGameOver**: Checks if the game has ended due to checkmate, stalemate, or draw conditions(insufficient pieces, 50 moves rule).
    - **Threefold Repetition:** Encodes the board state to detect if the same position has occurred three times.

Libraries Used:

- **#include <bits/stdc++.h>**: Provides access to standard STL containers and algorithms, streamlining coding tasks.
- **Custom Headers**:
    - **Player.h**: Likely defines player-specific functionality.
    - **Validator.h**: Includes logic for validating piece-specific moves (e.g., getPawnMoves, getKnightMoves).

Major Components:

1. **Board Class**:
    - Represents the chessboard, storing piece positions in board.matrix and tracking king positions (kingx, kingy) for each color.
    - Maintains move history and repetition counters for advanced draw detection.
2. **Piece Class**:
    - Represents a chess piece with attributes such as name, color, and cMove (move count for special rules).
3. **Move Representation**:
    - Each move is stored as a pair of coordinates and a type (char) for special moves like castling (C), en passant (E), or promotion (n, b, r, q). Otherwise, the type is normal

move (‘space’) or capture move (P, N, B, R, Q respectively pawn, knight, bishop, rook, queen).

4. **Validator Class:**
   - Implements base movement rules for pawns, knights, bishops, rooks, queens, and kings.
   - Ensures moves respect constraints like board boundaries, piece color, and special move rules.
5. **Player Class** encapsulates player-specific attributes and behaviors, such as:

   Identity and Role:
   - The type of player (e.g., human or AI).
   - The player's color (white or black).

   Game Actions:
   - Making a move (based on AI logic or user input).
   - Keeping track of pieces and moves.

Graphic Board Implementation

Chess Controller:

This module functions as the main entry point of the system, serving as the intermediary between the 2D Logic Board and the 3D Graphic Board. It manages core functionalities, including user event handling, player switching, undo/redo operations, and saving/loading the game state.

Input from Player or AI:

The functions `OnLeftMouseClick()`, `OnLeftMouseHandle()`, and `OnRightMouseClick()` are the primary methods for processing player inputs, including left-click, right-click, and piece dragging actions. Below is a detailed analysis of the `OnLeftMouseClick()` function, which is responsible for selecting a chess piece:

Function `OnLeftMouseClick()` Overview:

This function handles player interactions when the left mouse button is clicked to select and move a chess piece. Key operational steps are as follows:

Resetting Initial State:

Upon a mouse click, all relevant states, including cursor type, piece position, and references, are reset to ensure a clean interaction state.

```
CurrentMouseCursor = EMouseCursor::GrabHandClosed;
CurrentPlayer->mousePosition.X = NULL;
CurrentPlayer->mousePosition.Y = NULL;
CurrentPlayer->setCurrentPiece(nullptr);
CurrentMouseCursor = DefaultMouseCursor;
```

Clearing Previous Selection:

If the player had previously selected a chess piece, the "selection" mode is canceled to prevent conflicting states.

```
if (this->SelectedPiece != nullptr) {
    this->SelectedPiece->SetDeselected();
}
Deselect();
```

Identifying the Selected Chess Piece:

- After the mouse click, the function determines the piece's position based on the 3D game board.
- This position is then mapped to corresponding 2D logical coordinates.

```
FVector fromPos = SelectedPiece->tileOccupied->GetActorLoca
tion();
std::pair<int, int> fromCoord = ConvertToLogicCoordinates(f
romPos);
SelectPiece(fromCoord.first, fromCoord.second);
```

Verification of the Piece Belonging to the Current Player:

- Verify whether the selected piece belongs to the color corresponding to the current player.
- If correct, the `isPending` status is activated, allowing the player to move the piece.

```
if (SelectedPiece->isWhite() == CurrentPlayer->isWhite()) {
    isPending = true;
    PieceLocation = SelectedPiece->GetActorLocation();
    CurrentPlayer->isPending = true;
    CurrentPlayer->setCurrentPiece(SelectedPiece);
    CurrentMouseCursor = EMouseCursor::GrabHandClosed;
    SelectedPiece->setOverlap(true);
    SelectedPiece->StaticMesh->SetSimulatePhysics(false);
}
```

Similarly, the `OnLeftMouseRelease()` function handles the event of the user dragging a piece, while the `OnRightMouseClick()` function is used to move a piece without the need for drag-and-drop, using the right mouse button.

Implementing the Graphics Board: Performing a Move

The `MakeMove()` function is used to handle the movement of a piece on the board. Below is the main analysis:

Input State Validation:

- The function checks initial conditions, including whether a piece has been selected and if the destination square exists. If these conditions are not met, the function terminates.

Implementing the Graphics Board: Performing a Move

The `MakeMove()` function is used to handle the movement of a piece on the board. Below is the main analysis:

Input State Validation:

- The function checks initial conditions, including whether a piece has been selected and if the destination square exists. If these conditions are not met, the function terminates.

```
if (!SelectedPiece || !SelectedGrid) return;
```

Retrieve Logical Coordinates:

- Convert the position from the graphical space (3D) to logical coordinates (2D) in order to perform the logical processing steps on the chessboard.

```
FVector fromPos = SelectedPiece->tileOccupied->GetActorLoca
tion();
FVector toPos = SelectedGrid->GetActorLocation();
std::pair<int, int> fromCoord = ConvertToLogicCoordinates(f
romPos);
std::pair<int, int> toCoord = ConvertToLogicCoordinates(toP
os);
```

Move Validation:

- The `IsValidMove` function checks the validity of the move, including basic rules, castling status, or en passant captures.

```
int stepok = IsValidMove(logicBoard, fromPos, toPos, Next_T
urn == 1, Next_Turn);
```

Castling Handling:

- If the move is a castling, the position of the rook will be updated accordingly.

```
if (isCastling) {
    int rookY = (toY > fromY) ? 8 : 1; // Nhập thành phải h
oặc trái
    AChessPiece_Base* rookPiece = ChessBoard->TilesCollecti
on[rookIndex]->currentOccupyingPiece;
    if (rookPiece) {
        int newRookY = (toY > fromY) ? (toY - 1) : (toY +
1);
        Atile* newRookTile = ChessBoard->TilesCollection[ne
wRookIndex];
        rookPiece->tileOccupied->unoccupy();
        newRookTile->occupy(rookPiece);
        rookPiece->neverMove = false;
    }
}
```

En Passant Handling:

- If the move is an en passant capture, the captured pawn will be removed from the board.

```
if (isEnPassant) {
    int capturedPawnIndex = ((fromX - 1) * 8) + (toY - 1);
    AChessPiece_Base* capturedPawn = ChessBoard->TilesColle
ction[capturedPawnIndex]->currentOccupyingPiece;
    if (capturedPawn) {
        ChessBoard->TilesCollection[capturedPawnIndex]->uno
ccupy();
        capturedPawn->Discard();
    }
}
```

Update Score and Record the Move:

- When a piece is captured, the player's score is updated. The move is also recorded in the history.

```
if (SelectedGrid->isOccupied()) {
    CapturedPiece = SelectedGrid->currentOccupyingPiece;
    if (CurrentPlayer == PlayerOne) {
        PlayerOne->addScore(CapturedPiece->getPiecePoint
());
    } else {
        PlayerTwo->addScore(CapturedPiece->getPiecePoint
());
    }
    SelectedGrid->currentOccupyingPiece->Discard();
}
SaveCurrentMove(SelectedPiece, SelectedPiece->tileOccupied,
SelectedGrid, CapturedPiece);
```

Update the Chessboard:

- The piece is moved from its current square to the destination square. The logical state and the graphical interface are synchronized.

```
SelectedPiece->tileOccupied->unoccupy();
SelectedGrid->occupy(SelectedPiece);
```

Switch Turn:

- After the move is made, the turn is passed to the next player or AI (if the PvAI mode is enabled).

```
cntTurn = Next_Turn;
SwitchPlayer();
```

Implementing the Graphics Board: AI's Move Execution

The `AiMakeMove()` function handles the logic for a move controlled by the AI. It updates the game state, chessboard, and interface based on the move chosen by the AI.

Key functions include:

- Validating the move.
- Handling special moves (castling, en passant).
- Updating the player's score.
- Determining the game result.

Move Validation and Execution

- **Handling En Passant:**

  Updates the "en passant" state of the pawns to ensure they are not captured in the next turn.

```
for (int i = 1; i <= 8; i++)
    for (int j = 1; j <= 8; j++)
        if (board.matrix[i][j]->name == 'P' && board.matrix
[i][j]->cMove == 1)
            board.matrix[i][j]->cMove = 2;
```

- **Execute the Move:**

  Executes the move based on the parameters `tp` (move type), `(x, y)` (starting position), and `(fX, fY)` (destination position). Handles pawn promotion and updates the king/rook status for castling.

```
applyMove(board, {tp, {mp(x, y), mp(fX, fY)}});
```

Handling Special Moves

- **Castling:**

  Move the rook to the new position when the AI performs castling.

```
if (tp == 'C') {
    int rookY = (fY > y) ? 8 : 1; // Xác định vị trí xe
    int newRookY = (fY > y) ? (fY - 1) : (fY + 1);
    Atile* newRookTile = ChessBoard->TilesCollection[newRoo
kIndex];
    rookPiece->tileOccupied->unoccupy();
    newRookTile->occupy(rookPiece);
    rookPiece->neverMove = false;
}
```

En Passant:

Remove the captured pawn in an en passant move and update the player's score.

```
if (tp == 'E') {
    int capturedPawnIndex = ((x - 1) * 8) + (fY - 1);
    AChessPiece_Base* capturedPawn = ChessBoard->TilesColle
ction[capturedPawnIndex]->currentOccupyingPiece;
    if (capturedPawn) {
        if (CurrentPlayer == PlayerOne) {
            HUD->widgetPoint1->addScore(capturedPawn->getPi
ecePoint());
            PlayerOne->addScore(capturedPawn->getPiecePoint
());
        } else {
            HUD->widgetPoint2->addScore(capturedPawn->getPi
ecePoint());
            PlayerTwo->addScore(capturedPawn->getPiecePoint
());
        }
        ChessBoard->TilesCollection[capturedPawnIndex]->uno
ccupy();
        capturedPawn->Discard();
    }
}
```

Update Game Status

- **Check Game Result:**

  Determine the game status (ongoing, game over, or draw) after the move.

```
GameState state = isGameOver(board, turn);
if (state == ONGOING) state = ThreeFoldRepetition(board, t
p);
if (state != ONGOING) {displayGameState(state);}
```

Switch Turn:

Ensure the turn is passed to the next player when the move is valid.

```
cntTurn ^= 1; // update turn if valid move
```

Save and Update Interface

- **Save Move History:**

  The move is recorded to support AI/Player interaction.

- **Update 2D Board and Interface:**

  Display the move and the result on the user interface.

Undo and Redo Moves

UndoLastMove

The `UndoLastMove()` function allows you to revert to the previous move by restoring the board state from history.

- **Condition Check:**
  - If the move history (`LogicMoveHistory`) contains one or fewer states, no undo is performed.

```
if (LogicMoveHistory.Num() <= 1)
  return;
```

- **Create a Copy of the Current State:**
  Save the current state into
  `LogicRedoStack` to support redo functionality later. The copy ensures each piece is recreated with the correct type and color.
- **Remove Current State and Restore the Previous State:**
  - Clear the current board.
  - Copy each piece from the previous state in history to the logical board (`logicBoard`).
  - Restore additional information such as the king's position, the turn order, and special states (like castling and en passant).
- **Update the Interface and Switch Player:**
  Update the interface to reflect the new board state and switch the turn back to the previous player.

```
UpdateGraphicsFromLogicBoard(-1, -1);
SwitchPlayer();
```

RedoLastMove

The `RedoLastMove()` function replays the undone move by restoring the state from `LogicRedoStack`.

- **Condition Check:**
  - If there are no states in `LogicRedoStack`, no redo is performed.

```
if (!CanRedo()) return;
```

- **Restore the State from `RedoStack`:**
  - Save the current state into `LogicMoveHistory` to support undo functionality.
  - Clear the current board.
  - Copy each piece from the next state (`nextState`) in `RedoStack` to the logical board (`logicBoard`).
  - Restore additional information like the king's position, the turn order, and special states.
- **Update the Interface and Switch Player:**
  Similar to

`UndoLastMove`, update the interface and switch the player's turn.

Additionally, there are other functions such as `SaveGameState`, `LoadGameState`, etc.

ChessHUD: User Interface Management in a Chess Game

ChessHUD serves as the primary User Interface (UI) management class for the chess game, inheriting from the AHUD class in Unreal Engine. It functions as the central controller for coordinating the display and interaction of various in-game widgets.

Core Functions:

- Initialization and management of UI widgets
- Display and hide game screens
- Handle special interactions such as pawn promotion
- Manage game flow through different UI screens

Widget Initialization and Setup

The `BeginPlay` method is responsible for initializing and configuring the visibility of game widgets:

```
void AChessHUD::BeginPlay() {
    Super::BeginPlay();

    // Initialize settings widget
    if (settingWidget) {
        settingWidget->AddToViewport();
        settingWidget->SetVisibility(ESlateVisibility::Visi
ble);
    }

    // Initialize game mode widget
    if (chessModeWidget) {
        chessModeWidget->AddToViewport();
        chessModeWidget->SetVisibility(ESlateVisibility::Hi
dden);
    }

    // Initialize additional widgets
```

```
    if (levelWidget) {
        levelWidget->AddToViewport();
        levelWidget->SetVisibility(ESlateVisibility::Hidde
n);
    }

    // ... Similar initialization for other widgets
}
```

Screen Display Management

Winning Screen Display:

```
void AChessHUD::showWinningScreen() {
    winningScreen->AddToViewport();
}

void AChessHUD::ShowPlayerWinWidget(const FString& WinnerNa
me) {
    if (player1WinWidget) {
        player1WinWidget->SetNguoiThang(WinnerName);
        player1WinWidget->SetVisibility(ESlateVisibility::V
isible);
    }
}
```

Pawn Promotion Handling

```
void AChessHUD::ShowPromotionSelection(bool isWhite) {
    if (promotionWidget) {
        promotionWidget->SetVisibility(ESlateVisibility::Vi
sible);
    }
}

void AChessHUD::HandlePromotion(char pieceType) {
    if (AChessController* Controller =
        Cast<AChessController>(GetOwningPlayerController
```

```
())) {
        Controller->HandlePromotionSelection(pieceType);
    }
}
```

Settings and Menu Management

```
void AChessHUD::ShowSettingSelection(bool Force_on) {
    if (Force_on) {
        bIsSettingVisible = true;
        settingWidget->SetVisibility(ESlateVisibility::Visi
ble);
        return;
    }

    bIsSettingVisible = !bIsSettingVisible;
    settingWidget->SetVisibility(bIsSettingVisible ?
        ESlateVisibility::Visible : ESlateVisibility::Hidde
n);
}
```

Managed Widgets

ChessHUD manages several essential widgets as follows:

Core Widgets:

- **settingWidget:** Settings management
- **chessModeWidget:** Game mode selection
- **levelWidget:** Difficulty level selection
- **menuWidget:** Main menu interface

Specialized Functional Widgets:

- **promotionWidget:** Pawn promotion management
- **player1WinWidget:** Player victory notification
- **soundWidget:** Sound settings management

Save/Load Management Widgets:

- **saveGameWidget:** Game save functionality
- **loadGameWidget:** Game load functionality

5. Testing Strategy:

Unit Testing

- Test individual components:
  - Validate piece movements.
  - Check for legal/illegal moves.
  - Ensure special rules (e.g., en passant) work correctly.

Integration Testing

- Test interactions between components:
  - Logic and UI integration (e.g., updating the board after a move).
  - Communication between players and the AI engine.
  - Status updates (e.g., "checkmate" notification) after moves.

System Testing

- Test the entire game:
  - Starting, playing, and completing a match.
  - Switching between game modes (AI, local, online multiplayer).
  - Handling edge cases (e.g., forced draw due to insufficient material).

Regression Testing

- Ensure existing functionality remains intact when new features are added.

Performance Testing

- Measure response times in AI gameplay.

- Assess the system’s performance under high loads.

6. Future Improvements:

- Improved AI Performance (minimax tree with greater depth or dynamic depth, integrated AI such as Stockfish, …).
- User Experience Enhancements:
  - Move Suggestions: Add a "hint" system to suggest strong moves for users, especially useful for beginners.
  - Game Modes: Introduce additional modes such as timed games (e.g., blitz, bullet, classical) or puzzles and challenges (e.g., checkmate in X moves).
- Online Multiplayer and Leaderboard**.**
- Cross-Platform Compatibility: Mobile Version, Web-Based Chess, Cross-Platform Sync.