11 Building our own game (2/4) - Chess Pieces

Functional and Implementation Guidelines

Functional Guidelines

1. Abstract away Chess Initialization Logic into a different file
   1. Free your “gameplay” objects for actual gameplay
2. Give concrete implementation to individual chess pieces with their real movement patterns
   1. Pawn Piece
      1. Don’t implement en passant or piece promotion chess features at this stage
   2. Rook Piece
   3. “Unfinished” Piece - a placeholder for all chess pieces left to be implemented
   4. Feel free to implement some (all) of the chess pieces
      1. Don’t try to implement the King “special” features at this stage
3. Implement a feature, which visualizes all possible moves for a selected chess piece
   1. Give different visual effects for all possible move scenarios
      1. normal movement to unoccupied game board position
      2. movement that would capture an enemy piece
      3. movement over friendly figures - (actually is a forbidden move) but visualizes the defence structure
   2. Hide the movement patterns of a chess piece once it is de-selected

Implementation Guidelines

1. Implement PieceHandlerPopulator
   1. Move the init logic behind the Piece creation our of the business logic from PieceHandler
2. Start using a polymorphic approach when the ChessPiece class is used
3. Implement a UnfinishedPiece class
   1. Has a small exclamation mark in the top left corner of the piece (to indicate that the piece is not fully implemented).
   2. Use RED text for the exclamation mark
   3. Structure - always place the base struct as a first member

| class UnfinishedPiece: public ChessPiece {  Text \_notReadyText;  }; |
| --- |

* 1. Expand the ChessDefines

| enum class TileType : uint8\_t {  MOVE, GUARD, TAKE  }; |
| --- |

* 1. Public API - the same as the base struct. “Polymorphism” should be simulated + 1 additional “getMoveTilesUnfinishedPiece” method. This method should return all possible TileTypes - MOVE, GUARD, TAKE

| std::vector<TileData> UnfinishedPiece::getMoveTiles(  const std::array<PlayerPieces, Defines::PLAYERS\_COUNT> &activePieces) const {  return std::vector<TileData> { };  } |
| --- |

1. Implement ChessPieceResolver - resolving at runtime which functionality should be called (polymorphism)
   1. Has the same API as the base ChessPiece struct
   2. Add a bool flag in the ChessPiece ‘isUnfinished’ - will be removed later on
   3. Implementation

| void setBoardPosChessPieceResolver(struct ChessPiece \*piece,  const struct BoardPos \*boardPos) {  if (piece->isUnfinished) {  setBoardPosUnfinishedPiece((struct UnfinishedPiece\*)piece, boardPos);  return;  }  switch (piece->pieceType) {  case KING:  case QUEEN:  case BISHOP:  case KNIGHT:  case ROOK:  case PAWN:  setBoardPosChessPiece(piece, boardPos);  break;  default:  LOGERR("Error, received unsupported pieceType: %d", piece->pieceType);  break;  }  } |
| --- |

1. Implement ChessStructs.h
   1. Create additional defines

| enum Direction {  UP\_LEFT,  UP,  UP\_RIGHT,  RIGHT,  DOWN\_RIGHT,  DOWN,  DOWN\_LEFT,  LEFT,  DIRECTION\_COUNT  }; |
| --- |

* 1. Structure

| struct TileData {  BoardPos boardPos;  TileType tileType = TileType::MOVE;  };  using MoveDirection = std::vector<BoardPos>; |
| --- |

* 1. Expand the BoardUtils

| static int32\_t getOpponentId(int32\_t activePlayerId);  static BoardPos getAdjacentPos(Defines::Direction dir,  const BoardPos &currPos);  static bool doCollideWithPiece(const BoardPos &selectedPos,  const ChessPiece::PlayerPieces &pieces,  int32\_t \*outCollisionRelativeId = nullptr);  static TileType getTileType(const BoardPos &boardPos,  const ChessPiece::PlayerPieces &playerPieces,  const ChessPiece::PlayerPieces &enemyPieces); |
| --- |

1. Start implementing ROOK struct (polymorphic approach)
   1. Reuse all base methods but the “getMoveTilesRook”

| std::vector<TileData> getMoveTiles(  const std::array<PlayerPieces, Defines::PLAYERS\_COUNT> &activePieces) const final; |
| --- |

* 1. Implement a helper method to get all possible rook moves **within the board** not accounting for any other pieces/figures

| std::vector<MoveDirection> Rook::getBoardMoves() const {  constexpr auto allowedDirs = 4;  constexpr std::array<Defines::Direction, allowedDirs> rookDirs { Defines::UP,  Defines::RIGHT, Defines::DOWN, Defines::LEFT };  constexpr auto maxRootMoves = 14;  std::vector<MoveDirection> boardMoves(allowedDirs);  BoardPos futurePos;  for (int32\_t dirIdx = 0; dirIdx < allowedDirs; ++dirIdx) {  boardMoves[dirIdx].reserve(maxRootMoves);  futurePos = \_boardPos;  futurePos = BoardUtils::getAdjacentPos(rookDirs[dirIdx], futurePos);  //...  }  return boardMoves;  } |
| --- |

* 1. Implement the getMoveTilesRook method

| std::vector<TileData> Rook::getMoveTiles(  const std::array<PlayerPieces, Defines::PLAYERS\_COUNT> &activePieces) const {  const std::vector<MoveDirection> boardMoves = getBoardMoves();  //...  for (const auto &moveDir : boardMoves) {  if (moveDir.empty()) {  continue;  }  //…  }  } |
| --- |

1. GameBoard class should remember the active move tiles
   1. Achieve this by using an interface

| virtual void onPieceGrabbed(const BoardPos &boardPos,  const std::vector<TileData> &moveTiles) = 0;  virtual void onPieceUngrabbed() = 0;  virtual bool isMoveAllowed(const BoardPos &pos) const = 0; |
| --- |

1. Implement Movement Selector class
   1. Mark/unmark possible movement tiles
   2. Structure

| class MoveSelector {  enum InternalDefines {  MAX\_ACTIVE\_TILES = 28  };  std::array<Image, MAX\_ACTIVE\_TILES> \_tileImgs;  size\_t \_activeTiles { 0 };  }; |
| --- |

* 1. Public API

| void draw();  void markTiles(const std::vector<TileData> &markedTiles);  void unmarkTiles(); |
| --- |

1. Implement erase of enemy player pieces when TileType::TAKE is processed
2. Implement Pawn class using the same analogy as the Rook
   1. Do not try to implement everything in one method - separate them

| std::vector<TileData> Pawn::getMoveTiles(  const std::array<PlayerPieces, Defines::PLAYERS\_COUNT> &activePieces) const {  if (Defines::WHITE\_PLAYER == \_playerId) {  return getWhiteMoveTiles(activePieces);  }  return getBlackMoveTiles(activePieces);  } |
| --- |

1. Don’t try to implement Piece Promotion mechanism. It will be covered in later topics