12 Building our own game (3/4) - Game Features and Animations

Functional and Implementation Guidelines

Functional Guidelines

1. Implement Game Logic - player turn mechanism
   1. Handle events only associated with the current active player
   2. Bonus: Try to implement an indicator who is the current active player
      1. It could be a text box, some animation or … get creative
2. Implement the feature piece promotion
   1. When a pawn reaches the end of the enemy lines - that pawn could be promoted to a Queen, Rook, Bishop or Knight piece
3. Implement a buttons panel with button associated with the piece promotion feature
   1. Let the player choose, which piece the pawn should be promoted to
4. Try to implement a feature, which flips the whole game board, when a player turn finished
   1. For example: the white pieces are faced closer to the bottom of the game board
   2. When the white finishes its move - flip the board so the black pieces are near the bottom of the game board
5. Implement a feature, which “flips” the input event when the board is “flipped” (associated with the feature described 4. )
   1. The input event should revert itself back to original behavior once its the white turn again

Implementation Guidelines

1. GameLogic struct (player turn mechanism).
   1. Structure

| class GameLogic {  int32\_t activePlayerId;  }; |
| --- |

* 1. Struct will be expanded in the future
  2. Public API

| int32\_t getActivePlayerId() const;  void finishTurn(); |
| --- |

1. Implement GameInterface

| virtual void finishTurn() = 0; |
| --- |

* 1. The game then passes it to the PieceHandler

1. Modify the PieceHandler to only handle events on the curr active player
2. BONUS: for homework implement text in the upper right corner to display the currently active player (WHITE or BLACK)
3. Expand the GameProxy with activePawnPromotion

| virtual void activatePawnPromotion() = 0; |
| --- |

1. Modify the Pawn to be able to activate the piece promotion

| void setBoardPos(const BoardPos& boardPos) final; |
| --- |

1. Implementat PiecePromotionPanel class
   1. Structure

| class PiecePromotionPanel {  enum InternalDefines {  BUTTONS\_COUNT = 4  };  std::array<PiecePromotionButton, BUTTONS\_COUNT> \_promotionBtns;  GameInterface \* \_gameInterface = nullptr;  bool \_isActive = false;  }; |
| --- |

* 1. Public API

| int32\_t init(const PiecePromotionPanelConfig &cfg);  void handleEvent(const InputEvent &e);  void draw();  void activate(int32\_t playerId);  bool isActive() const; |
| --- |

* 1. Implement a PiecePromotionPanelConfig (populate it in the Game.c)

| struct PiecePromotionPanelConfig {  GameInterface \*gameInterface = nullptr;  int32\_t whitePiecesRsrcId = 0;  int32\_t blackPiecesRsrcId = 0;  int32\_t buttonBgrRsrcId = 0;  int32\_t gameBoardWidth = 0;  int32\_t gameBoatdHeight = 0;  int32\_t buttonBgrWidth = 0;  int32\_t buttonBgrHeight = 0;  int32\_t buttonWidth = 0;  int32\_t buttonHeight = 0;  }; |
| --- |

1. Implement PiecePromotionButton
   1. Structure

| class PiecePromotionButton {  Image \_bgrImg;  PiecePromotionButtonConfig \_cfg;  }; |
| --- |

* 1. Public API

| int32\_t init(const PiecePromotionButtonConfig &cfg);  void draw() final;  void handleEvent(const InputEvent &e) final;  void activate(int32\_t activePlayerId); |
| --- |

* 1. Implement PiecePromotionbuttonConfig (populate in PiecePromotionPanel)

| struct PiecePromotionButtonConfig {  std::function<void(PieceType)> onBtnClicked; //capture the method by lambda  Point bgrPos;  int32\_t buttonBgrRsrcId;  int32\_t buttonWhitePieceRsrcId;  int32\_t buttonBlackPieceRsrcId;  int32\_t bgrWidth = 0;  int32\_t bgrHeight = 0;  int32\_t width = 0;  int32\_t height = 0;  PieceType pieceType;  }; |
| --- |

1. Finish the population of PiecePromotionPanel and PiecePromotionButton
2. Implement destroyButton() method inside the Button struct
3. The activatePiecePromotionButton method - destroys the btn and recreates it using the bgrImg as helper for relative position

| const int32\_t X\_Y\_DELTA = (\_bgrImg.getWidth() - \_cfg.width) / 2;  const Point btnPos = Point(\_bgrImg.getX() + X\_Y\_DELTA,  \_bgrImg.getY() + X\_Y\_DELTA); |
| --- |

1. Create an empty promotePiece method in the PieceHandler and call it from the Game.cpp
2. Bonus: For homework finish the implementation of the empty promotePiece() method
3. Implement BoardAnimator
   1. At the end of a player turn - rotate the board with 180 degrees and all pieces
   2. Structure

| class GameBoardAnimator {  GameInterface \*\_gameInterface = nullptr;  Image \*\_boardImg = nullptr;  int32\_t \_currRotation { 0 };  WidgetFlip \_targetFlipType = WidgetFlip::NONE;  }; |
| --- |

* 1. Public API

| int32\_t init(GameInterface \*gameInterface, Image \*boardImg);  void startAnim(int32\_t playerId); |
| --- |

* 1. Currently, the implementation of the animator will be empty and will directly invoke “on animEnd”, which will invoke setWidgetFlip(HORIZONTAL\_AND\_VERTICAL) on all pieces **AND** rotate the board Img with 180 degrees.
  2. Expand the game proxy

| virtual void setWidgetFlipType(WidgetFlip flipType) = 0; |
| --- |

1. The pieces are flipped, but they are not in their correct position
   1. BONUS: For homework think of a way how this can be done
2. Implement InputInverter class
   1. Place it in the Game class
   2. Structure

| class InputInverter {  int32\_t \_boardWidth { 0 };  int32\_t \_boardHeight { 0 };  WidgetFlip \_flipType = WidgetFlip::NONE;  }; |
| --- |

* 1. Public API

| int32\_t init(int32\_t boardWidth, int32\_t boardHeight);  void setBoardFlipType(WidgetFlip flipType);  void invertEvent(InputEvent & inputEvent); |
| --- |

* 1. Invert the event at the beginning of the Game handleEvent logic