Project Documentation

Table of Contents

Overview

The project represents a turn-based game that combines elements of resource management, tactical combat, and territorial control. It is inspired by famous strategy games such as Civilization, Total War: Warhammer, Risk, and Age of Empires. The game is designed for two players sharing the same screen. The key features for the software are as follows:

1. Core Gameplay Features
    a. Resource management: Players gather resources primarily by constructing specialized buildings on map tiles. The buildings generate passive income at the end of each turn. Resources need to be managed effectively as they are needed for further development of the territories and further construction of other buildings. Resources are also needed for deploying armies and for colonizing new tiles, which is essential for territorial expansion.
    b. Territorial expansion: The game begins with players controlling a single base tile and one army unit. Expansion can be achieved by moving armies to neutral tiles, which can then be colonized for a resource cost.

The game encourages strategic growth by an increasing cost structure for each additional tile captured. To conquer enemy territory, the tiles of the enemy need to be neutralized through combat.

c. Construction: Players can place buildings on tiles to enhance resource generation. Each building serves a specific function and requires resources for both construction and ongoing upkeep, necessitating planning to balance expansion with sustainability.

Software Structure

Army Class

The Army class represents player-controlled military units in the game. Armies are necessary for territorial expansion and combat, offering a variety of unit types with distinct characteristics. The class encapsulates the properties and behaviors of an army, including its type, location, owner, and combat attributes.

An army has the following private attributes:

- owner_: A shared pointer to the Player who owns the army.
- location_: The current location of the army on the map, represented as a pair of integers (x, y).
- unitCount_: The number of units in the army.
- type_: The army's type (e.g., Infantry, Cavalry, Artillery, Marine).
- attack_ and defense_: The army's attack and defense stats, determined by its type.
- canAdvance_: A map that defines the types of terrain the army can traverse. This varies by army type:
    - Infantry: Can traverse grass but not water or rock.
    - Marine: Can traverse grass and water but not rock.

- hasCompletedTurn_: A boolean indicating whether the army has completed its actions for the current turn.

The class has the following methods:

Class constructor:

Army(ArmyType type, std::pair<int, int> location, std::shared_ptr<Player> owner, int unitCount): Initializes the army with its type, location, owner, and unit count. The constructor also sets terrain movement capabilities and assigns combat attributes based on the army type:

- Infantry: Moderate attack and defense (2 each).
- Cavalry: Stronger attack and defense (3 each).
- Artillery: High attack and defense (5 each).
- Marine: Moderate attack and defense (2 each) with water traversal capabilities.

Methods responsible ith ownership management:

- std::shared_ptr<Player> getOwner() const: Returns the current owner of the army.
- std::shared_ptr<Player> setOwner(std::shared_ptr<Player> player): Transfers ownership of the army to a new player, returning the previous owner.

Methods responsible for location management:

- std::pair<int, int> getLocation() const: Returns the current location of the army.
- int getLocationX() const and int getLocationY() const: Return the x and y coordinates, respectively.
- std::pair<int, int> setLocation(std::pair<int, int> location): Updates the army's location and returns the previous location.

Methods responsible for the unit count:

- int getUnitCount() const: Retrieves the current unit count.
- void setUnitCount(int unitCount): Updates the unit count.
- void incrementUnitCount(): Increases the unit count by one.

Methods responsible for combat attributes:

- ArmyType getType() const: Returns the type of the army.

- int getAttack() const and int getDefense() const: Return the army's attack and defense stats, respectively.

Method responsible for turn management:

- bool canAdvance(TileType type): Determines whether the army can move to a specified tile type. Movement rules are:
    - Grass: All armies can advance.
    - Water: Only Marines can advance.
    - Rock: No armies can advance.

Method responsible for information retrieval:

- const std::string toString() const: Returns a formatted string containing detailed information about the army, including its type, owner, location, and unit count.

The class also has a couple of supporting functions:

- getArmyTypeName: Converts the ArmyType enumeration into a human-readable string (e.g., "Infantry," "Cavalry").
- getArmyDeploymentCost: Returns the resource cost for deploying an army of a specific type. Deployment costs vary:
    - Infantry: Requires food.
    - Cavalry: Requires additional food.
    - Artillery: Requires food, stone, and gold.
    - Marine: Requires food and wood.

The army type determines its role in combat and strategy. For example:

1. The infantry is versatile and cost-effective
2. Cavalry offers stronger offense and defense which is suitable for assaults
3. Artillery provides heavy firepower, ideal for sieges
4. Marines enable unique strategies by crossing water tiles.

Armies are owned by players, with ownership influencing resource management and strategic positioning. The toString method facilitates UI integration by providing a detailed textual representation of the army's state.

Building Class

The Building class encapsulates the logic and behavior associated with constructing and managing buildings in the game. Buildings are critical gameplay elements, providing resources and strategic advantages to players. The class structure includes two key components: Building and BuildingBlueprint.

BuildingBlueprint Class

The BuildingBlueprint class represents a template for creating buildings. It allows configuration of the type, resource cost, and resource gain for specific buildiing types. It has the following attributes:

- type_: (Private) The type of building, represented by an enumeration BuildingType.
- resourceCost_: (Private) The resources required to construct the building.
- resourceGain_: (Private) The resources generated by the building each turn.

The methods of the class are:

Class constructor:

BuildingBlueprint(): Initializes the blueprint with a default label ("Blueprint").

Methods responsible for type management:

- BuildingType getType() const: Returns the type of the building.
- void setType(BuildingType type): Sets the building type and updates its label, resource cost, and resource gain based on predefined values:
  - WOOD_CUTTER: Gains wood each turn.
  - FARM: Gains food each turn.
  - MINE: Gains stone each turn.

    - MARKET: Gains gold each turn.

Methods responsible for resource management:

- Resources getResourceCost() const: Returns the resource cost for constructing the building.
- Resources getResourceGain() const: Returns the resources generated by the building each turn.

Building Class

The Building class models an actual building instance in the game. Each building has a unique ID, coorsinator, owner, and resource-related properties.

A building has the following attributes:

- idCounter_: (Static) An atomic counter for generating unique IDs for each building.
- id_: The unique ID of the building.
- type_: The type of the building (e.g., WOOD_CUTTER, FARM).
- resourceCost_: The resources required to construct the building.
- resourceGain_: The resources provided by the building each turn.
- owner_: A shared pointer to the player who owns the building.
- x_, y_: The coordinates of the building on the map.

The methods of the Building class are:

Constructor for the class:

> Building(BuildingType type, Resources resourceCost, Resources resourceGain, std::shared_ptr<Player> owner, int x, int y): Initializes the building with the specified type, cost, gain, owner, and location. Assigns a unique ID using the static idCounter_.

Methods responsible with the ID and location of the building:

- int getId() const: Returns the unique ID of the building.
- int getX() const and int getY() const: Return the building's x and y coordinates, respectively.

- std::pair<int, int> getXY() const: Returns the building's coordinates as a pair.

Methods responsible with type and ownership of the bulding:

- BuildingType getType() const: Returns the building type.
- std::shared_ptr<Player> getOwner() const: Retrieves the current owner of the building.
- std::shared_ptr<Player> setOwner(std::shared_ptr<Player> player): Changes the building's owner and returns the previous owner.

Methods responsible with resource management:

- Resources getResourceCost() const: Returns the resource cost for constructing the building.
- Resources nextTurn() const: Calculates and returns the resources generated by the building during a turn based on its type.

Method for information retreival:

- std::vector<std::string> getInfo() const: Generates detailed information about the building, including its owner, type, and resource output.

The Building class integrates with the game's resource management system by consuming resources during construction (resourceCost_) and producing resources (resourceGain_) during each turn. Buildings are associated with a specific player through the owner_ attribute, enabling strategic decision-making and player-specific benefits.

The type of building determines its strategic value. For example:

- **Wood Cutter**: Provides wood, useful for construction.
- **Farm**: Supplies food, essential for army maintenance.
- **Mine**: Produces stone, valuable for advanced structures.
- **Market**: Generates gold, a versatile resource for various expenditures.

Gamestate Class

The Gamestate class is the central controller for the game, managing the game\s overall logic, namely players, armies, buildings, and tiles. It ensures there are no issues between turn transitions, resource allocation and game state updates.

A gamestate has the following private attributes:

- turn_: The current turn number in the game.
- activePlayerID_: The ID of the player whose turn is active.
- window_: A shared pointer to the SFML render window, facilitating graphical output.
- players_: A vector of shared pointers to all players in the game.
- armies_: A vector of all armies present on the map.
- buildings_: A vector of all buildings constructed during the game.
- townhalls_: A vector of all town halls in the game.
- map_: A vector representing the game map, composed of Tile objects.
- num_rows_: The number of rows in the game map grid.

The Gamestate class has the following methods:

Methods responsible with the initialization process:

- GameState(std::shared_ptr<sf::RenderWindow> window, int turn): Initializes the game state with the specified render window and starting turn.
- void loadMap(const std::vector<Tile>& tiles, int num_rows): Loads the game map using a vector of Tileobjects and sets the number of rows.
- void loadMapFromString(const std::string str, int num_rows): Parses a string representation of the map to initialize the game grid. Characters such as 'G', 'W', and 'R' represent Grass, Water, and Rock tiles, respectively.

Methods responsible with turn management:

- void tick(): Executes the current player’s turn logic.
- void nextTurn(): Advances the game to the next turn, updating resources, resetting army states, and switching to the next player.

Methods responsible with player management:

- void addPlayer(std::shared_ptr<Player> player): Adds a player to the game.
- int getActivePlayerID(): Retrieves the ID of the current active player.
- void setActivePlayerID(int playerID): Sets the ID of the active player.
- std::shared_ptr<Player> getWinner() const: Determines if there is a winner (i.e., the last surviving player) and returns their shared pointer.
- void handleGameover(std::shared_ptr<Player> player): Handles player elimination by setting their state to inactive and removing their tiles, armies, and buildings from the game.

Methods responsible with army management:

- void addArmy(ArmyType type, int x, int y, std::shared_ptr<Player> owner, int unitCount): Adds a new army to the game at the specified location.
- void moveArmy(Army& army, int x, int y): Moves an army to a new location, handling merging or combat if another army is present.
- std::vector<Army>::iterator findArmyByLocation(int x, int y): Finds an army at the specified location.
- bool tileHasArmy(int x, int y): Checks if a tile has an army present.
- std::vector<std::pair<int, int>> getArmyMovementRange(Army& army): Determines the range of tiles an army can move to during its turn.

Methods responsible with building and tile management:

- void addBuilding(const Building& building): Adds a new building to the game.
- bool canPlaceBuilding(int x, int y, int playerID): Checks if a building can be placed at the specified location.
- void addTownhall(const TownHall& townhall): Adds a new town hall to the game.
- Tile& getTile(int xPos, int yPos): Retrieves the tile at the specified coordinates, throwing an exception if the coordinates are invalid.
- std::vector<Tile> getNeighbourTiles(int xPos, int yPos, int radius): Retrieves tiles within a specified radius of the given coordinates.

- std::vector<Tile> getClaimedTiles(const std::shared_ptr<Player> player): Returns a list of tiles owned by a player.

Methods responsible with visibility management:

- std::vector<Tile> getVisibleTiles(std::shared_ptr<Player> player): Retrieves all tiles visible to a player, based on the positions of their armies, buildings, and town halls.
- std::vector<Building> getVisibleBuildings(std::shared_ptr<Player> player): Returns a list of buildings visible to a player.
- std::vector<Army> getVisibleArmies(std::shared_ptr<Player> player): Returns a list of armies visible to a player.

Methods responsible with combat and colonization:

- void colonize(Army& army): Allows an army to colonize the tile it occupies, transferring ownership to its player.
- void placeSoldiers(std::shared_ptr<Player> player, ArmyType armyType): Handles soldier placement near a town hall, merging with existing armies if applicable.

The GameState class ensures synchronized turn progression across players by updating resources and resetting army states. It provides efficient tile management through the map_ attribute and related methods, supporting core functionalities such as colonization, visibility, and player interactions. The class integrates combat mechanics, unit merging, and colonization processes to facilitate interactions between armies, buildings, and players. Additionally, it handles game-ending scenarios using the getWinner and handleGameover methods, ensuring a structured approach to determining victory or defeat.

Player Class

The Player class represents a participant in the game. It manages player-specific attributes, such as resources, turn order, and status. This class provides essential functionality for resource management, game participation, and player state tracking.

A player has the following attributes:

- id_: A unique identifier automatically assigned to each player upon creation.
- color_: The player's color, used for visual representation in the game.
- turnOrder_: The player's position in the turn sequence.
- resources_: The player's current resource pool, consisting of food, wood, gold, and stone.
- isAlive_: A boolean flag indicating whether the player is still active in the game.
- idCounter_: A static atomic counter used to generate unique player IDs.

The methods for the Player class are:

Constructor for the class:

> Player(sf::Color color, int turnOrder, Resources resources): Initializes the player with a specified color, turn order, and starting resources. Each player is automatically assigned a unique ID, and their initial state is set to "alive."

Methods responsible with resource management:

- const Resources& getResources() const: Retrieves the player's current resource pool.
- void modifyResources(const Resources& delta): Modifies the player's resources by adding the specified delta. Ensures no resource value drops below zero.

Methods responsible with player identification and actions:

- int getID(): Returns the player's unique ID.
- sf::Color getColor(): Returns the player's color for visual differentiation.
- int getTurnOrder() (if implemented): Would retrieve the player's turn order (not visible in provided code but likely in player.hpp).

Methods responsible with state management:

- bool getIsAlive() const: Checks if the player is still active in the game.

- void setIsAlive(bool isAlive): Updates the player's status, typically used to mark a player as "defeated."

Players use the method modifyResources to adjust their resources dynamically based on game events. The design ensures that resources cannot become negative, preventing invalid states. The isAlive_ flag tracks whether the player is still in the game. The flag determines when a player has been eliminated, such as losing all armies or their base.

PlayerAI Class

The PlayerAI class is responsible for implementing automated player behavior in the game. It uses predefined logic to perform actions such as building structures, managing armies, and interacting with the game environment. The class integrates with the GameState to control gameplay elements on behalf of an AI player.

An AI player has the following private attributes:

- self: A shared pointer to the PlayerAI instance, allowing the AI to reference itself within the game.
- gameState_: A pointer to the current GameState, enabling interaction with the game environment.
- currentActionStep_: Tracks the current step of the AI's action sequence.
- maxSteps_: Defines the maximum number of actions the AI can perform in a turn.
- toPlace: Specifies the next building type the AI intends to construct.
- enemyTownhallPosition_: Stores the position of the enemy's town hall, if discovered.

The PlayerAI class has the following methods:

- void addSharedPtr(std::shared_ptr<PlayerAI> refToSelf): Assigns a shared pointer reference to the AI instance for self-referencing during gameplay.

- void takeTurnActions(): Executes the AI's actions for the current turn, including building construction and army movements. Ends the turn if the maximum action steps are reached.

Methods responsible for army management:

- void doArmyActions(): Handles army-related decisions, including colonization, movement, and combat. The AI attempts to locate the enemy's town hall using a breadth-first search (BFS) algorithm and moves its armies accordingly.
- void spawnArmy(): Commands the AI to recruit new infantry units, using resources as necessary.
- void checkForEnemyTownhall(): Updates the AI's knowledge of the enemy town hall's position if it becomes visible.

Methods responsible for building management:

- void doBuildActions(): Executes building-related actions, including resource checks and placement of new structures. Ensures that the AI places buildings only on owned tiles without existing structures.
- bool canAffordBuilding(BuildingType building): Determines whether the AI has sufficient resources to construct a specified building.
- void setNextBuilding(): Cycles through the building types in a predefined sequence (e.g., Wood Cutter → Farm → Mine → Market).

The AI’s behavior is deterministic yet adaptable, enabling it to respond dynamically to the game environment while following predefined priorities.

Selectable Class

The Selectable class is an abstract base class designed to provide common functionality for objects that can be selected within the game, such as tiles, buildings, armies, and town halls. It offers a consistent interface for accessing shared properties like name, label, and additional details.

The class has a single attribute, **`name_`**. It is a string representing the name of the selectable object, primarily used for display purposes.

The class has the following methods:

- **`Selectable(std::string name)`**: Initializes the selectable object with a given name.
- **`std::string getName() const`**: Retrieves the name of the selectable object.
- **`virtual std::string getLabel() const = 0`**: A pure virtual function that must be implemented by derived classes to provide a label for the object.
- **`virtual std::vector<std::string> getInfo() const = 0`**: A pure virtual function that must be implemented by derived classes to return additional details about the object.

Tile Class

The Tile class represents an individual unit of the game map, providing the foundational structure for gameplay mechanics such as territory control, building placement, and army movement. Each tile has a type, ownership, and occupancy status.

A tile has the following attributes:

- x_, y_: The tile's coordinates on the game map grid.
- type_: The type of the tile (e.g., Grass, Water, Rock), represented by the TileType enumeration.
- owner_: A shared pointer to the Player who controls the tile.
- occupied_: A boolean flag indicating whether the tile is currently occupied (e.g., by an army or building).

The methods for the Player class are:

Class Constructor

> Tile(int x, int y, TileType type, std::shared_ptr<Player> owner): Initializes the tile with its coordinates, type, and owner. If no owner is specified, the tile defaults to unowned.

Methods repsonsible with position management:

- int getX() const and int getY() const: Retrieve the x and y coordinates of the tile.
- std::pair<int, int> getXY() const: Returns the tile's coordinates as a pair.

Methods for tile type management:

- TileType getType() const: Retrieves the tile's type.
- void setType(TileType type): Updates the tile's type.

Methods for managing the ownership of the tile:

- std::shared_ptr<Player> getOwner() const: Returns the current owner of the tile.
- std::shared_ptr<Player> setOwner(std::shared_ptr<Player> player): Assigns a new owner to the tile and returns the previous owner.
- void setOccupied(bool occupy): Sets the occupancy status of the tile.
- bool isOccupied() const: Checks whether the tile is occupied.

Method for information retrieval:

- std::vector<std::string> getInfo() const: Generates a list of strings containing details about the tile, such as its owner.

The Tile class represents the fundamental unt of the game map. Each tile is uniquely identified by its coordinates and may hold additional elements, such as armies or buildings.

Tile types influence gameplay, such as restricting army movement or affecting resource availability. For example:

- Grass: Default terrain allowing most interactions.
- Water: Typically impassable except for specific units.
- Rock: Provides restrictions, for example, armies cannot pass.

Townhall Class

The townhall is the player's headquarters, providing key functionalities such as recruiting soldiers and acting as a strategic hub on the map.

A townhall has the following attributes:

- id_: A unique identifier for the town hall instance.
- owner_: A shared pointer to the Player who owns the town hall.
- x_, y_: The coordinates of the town hall on the map.
- soldierCost_: The resource cost associated with recruiting a soldier from the town hall.
- label: The name of the town hall, primarily used for UI display.

The Townhall class has the following methods:

Class constructor

> TownHall(int id, int soldierCost, std::shared_ptr<Player> owner, int x, int y): Initializes the town hall with its ID, soldier recruitment cost, owner, and map coordinates.

Methods responsible with identification of the townhall:

- int getId() const: Retrieves the unique ID of the town hall.
- std::string getName() const: Returns the town hall's name (label).
- std::string getLabel() const: Retrieves the label associated with the town hall.
- std::shared_ptr<Player> getOwner() const: Retrieves the current owner of the town hall.

Methods responsible for location management

- int getX() const and int getY() const: Return the x and y coordinates of the town hall, respectively.
- std::pair<int, int> getXY() const: Returns the coordinates as a pair.
- std::pair<int, int> getPosition() const: Another method to retrieve the town hall's coordinates.

Methods responsible for recruitment:

- bool canRecruitSoldier(Resources resources, ArmyType armyType) const: Determines if the town hall can recruit a soldier of the specified ArmyType based on the available resources and the resource cost of the soldier type. The method ensures that food, wood, gold, and stone requirements are met.
- int getSoldierCost() const: Returns the base resource cost for recruiting a soldier.

Methods for information retrievel:

- std::vector<std::string> getInfo() const: Returns a list of strings containing details about the town hall, such as its name and soldier recruitment cost.

UI Class

The UI class is responsible for managing the graphical user interface in the game. It handles player interaction, displays game elements, and provides a means for players to interact with the game state. This class integrates with SFML to render the UI components and supports actions such as resource management, unit movement, building placement, and turn progression.

An ui has the following attributes:

- player_: A shared pointer to the Player object associated with the UI.
- gameState_: A shared pointer to the GameState, allowing the UI to interact with the overall game logic.
- font: An SFML font object used for rendering text in the UI.
- window_: A shared pointer to the SFML render window, where all UI elements are drawn.
- selected_: A pointer to the currently selected object (e.g., a building, army, or tile).
- highlightedTiles_: A vector of tiles currently highlighted for user interaction.

- activeMenu_: Tracks the currently active menu (e.g., BUILD, ARMY, TOWNHALL).
- endTurnArea, mapArea, menuArea: Define the areas on the screen corresponding to specific UI interactions.

The ui class has the following methods:

Class constructor:

UI(std::shared_ptr<Player> player, std::shared_ptr<GameState> gameState, sf::Font font, std::shared_ptr<sf::RenderWindow> window): Initializes the UI with the player's information, game state, font, and render window. Sets the default menu to the BUILD menu.

Display Methods:

- **`void displayUI()`**: Renders all UI components, including the map, resource indicators, menus, and information boxes.
- **`void displayResources()`**: Displays the player's current resources (wood, food, gold, stone) on the screen.
- **`void displaySelected()`**: Displays information about the currently selected object, such as its name or label.
- `void displayEndRound()`: Displays the "End Turn" button if it's the player's turn, or "WAITING" otherwise.
- `void displayInfoBox()`: Shows detailed information about the currently selected object, such as stats or capabilities.
- `void displayMap()`: Renders the visible portion of the game map, including tiles, buildings, armies, and town halls.
- `void displayHighlightedtiles()`: Highlights specific tiles on the map for user interaction, such as movement suggestions or valid placement areas.
- `void displayMenu()`: Displays the currently active menu and its options, which vary depending on the context (e.g., building types in the BUILD menu).

- `void displayArmyMovementRange()`: Highlights the movement range of the currently selected army.

Methods responsible with handling interactions:

- `void processMouseButtonPressed(sf::Event event)`: Handles mouse button presses, determining which UI area was clicked and triggering the appropriate action.
- `void processMenuSelected(sf::Event event)`: Processes clicks within the menu area, updating the selected action or object.
- `void processTileSelected(int xPos, int yPos)`: Handles selection logic for tiles, including moving armies, placing buildings, and selecting town halls.
- `Selectable* getSelected() const`: Retrieves the currently selected object.
- `void setSelected(Selectable* selectable)`: Sets the currently selected object.
- `void resetSelected()`: Resets the current selection, deallocating memory if necessary (e.g., for a `BuildingBlueprint`).

Methods that provide drawing support:

- `void drawTile(const Tile& tile, int xPos, int yPos, bool isHighlighted)`: Draws a tile on the map, with optional highlighting.
- `void drawBuilding(const Building& building, int xPos, int yPos)`: Draws a building on the map.
- `void drawTownhall(const TownHall& townhall, int xPos, int yPos)`: Renders the town hall with a unique hexagonal shape and color.
- `void drawArmy(const Army& army, int xPos, int yPos)`: Displays the army's name and unit count on its corresponding tile.
- `void drawArmyMovementSuggestionMarker(const Army& army, int xPos, int yPos)`: Highlights possible movement destinations for the selected army.

The UI class is meant to provide a bridge between the player’s actions and the GameState, providing functionalities such as building placement, unit movement, and turn management.

Mouse interactions are mapped to game actions, enabling intuitive selection of tiles and triggering of menus. Resources, selected objects, and game states are dynamically updated and reflected on the UI, ensuring real-time feedback to the player. Menus adapt based on the active context, offering relevant options for actions like constructing buildings, managing armies, or interacting with town halls.

UIManager Class

The UIManager class is responsible for managing the high-level state of the user interface in the game. It defines and controls the different UI states, such as the main menu, gameplay interface, and map editor by ensuring a smooth transition between these modes. Additionally, it centralizes constants for UI styling and interaction areas.

The private of the class are:

- **`state_`:** Tracks the current state of the UI, represented by the UIState enumeration. Possible states include:
  - **`MAINMENU`**: The main menu of the game.
  - **`GAMEPLAY`:** The active gameplay interface.
  - **`MAPEDITOR`:** The map editor mode.
- Predefined screen areas that map to specific interactions, represented as **`sf::IntRect`** objects:
  - **`endTurnArea`:** Defines the "End Turn" button area.
  - **`mapArea`:** Specifies the area where the game map is rendered.
  - **`menuArea`:** Identifies the area occupied by the action menu.
  - **`startArea`:** Defines the clickable area for starting the game from the main menu.
  - **`mapeditorArea`:** Specifies the clickable area for accessing the map editor.
- Color constants used throughout the UI:
  - **`UI_BG`:** Background color for UI elements.

  - **`DARK_GRAY, DARK_GREEN, LIGHT_GRAY:`** Colors used for buttons, outlines, and highlights.

UIManager has the following methods:

- **`UIManager()`**: Initializes the UIManager with the default state set to **`MAINMENU`**.
- **`void setState(UIState state)`** : Sets the current UI state.
- **`UIState getState() const`**: Retrieves the current UI state.

The UIManager class manages the active interface elements by controlling
the state_ attribute, ensuring that only the appropriate components are displayed for the current context, such as the main menu or gameplay interface.

The UIManager centralizes color definitions and interaction areas to maintain consistent styling and functionality across UI components.

Utils Module

The Utils module provides utility functions to create and manage graphical elements, such as rectangles and text, in the game. These functions facilitate the consistent rendering of UI components by encapsulating repetitive tasks and ensuring uniform styling.

The Utils module has the following functions:

1. Rectangle creation

- **`sf::RectangleShape createRect(float x, float y, float width, float height, sf::Color fillColor):`** Creates a rectangle with the specified position, size, and fill color.

- **`sf::RectangleShape createRect(int x, int y, int width, int height, sf::Color fillColor)`**: Overload of **`createRect`** accepting integer parameters for position and size.

- **`sf::RectangleShape createRect(float x, float y, float width, float height, sf::Color fillColor, float outlineSize, sf::Color outlineColor)`**: Creates a rectangle with an additional outline, defined by its size and color.

- **`sf::RectangleShape createRect(int x, int y, int width, int height, sf::Color fillColor, int outlineSize, sf::Color outlineColor)`**: Overload of **`createRect`** that uses integer parameters for position, size, and outline.

2. Text creation:

- **`sf::Text createTextForRect(sf::RectangleShape rect, sf::Font& font, std::string text, int charSize, sf::Color fillColor, ORIENTATION orientationX, ORIENTATION orientationY)`**: Creates a text object aligned relative to a rectangle. Supports horizontal (**`orientationX`**) and vertical (**`orientationY`**) alignment, including options for **`START`**, **`CENTER`**, and **`END`**.

- **`std::vector<sf::Text> createTextsForRect (sf::RectangleShape rect, sf::Font& font, std::vector<std::string> texts, int charSize, sf::Color fillColor, int offset, int lineSpacing)`**: Creates multiple text objects aligned relative to a rectangle, with specified line spacing and vertical offset. Texts that do not fit within the rectangle's bounds are skipped with a warning message.

The utility functions in the Utils module ensure a consistent and cohesive design for UI elements such as buttons, menus, and labels by abstracting the creation of rectangles and text. They support precise alignment for dynamic layouts and adaptive positioning, enhancing flexibility in UI design. The module also includes functions like **`createTextsForRect`** which help with error handling.