

Quoridor Technical Documentation

18.06.2023

Language and Library choice

To recreate the game Quoridor as a computer program we have decided to use python for the following reasons :

- **Cross-Platform Compatibility** : python is cross platform meaning it should run the same on every operating system an user might want to play on.
- **Security** : as python integrates a garbage collector it would avoid vulnerabilities related to memory management in lower level languages.
- **Rapid development and ease to maintain** : Python's high-level nature allows for faster development and ease to maintain.

As for the external library we used Pygame for the graphics :

- **High performance** : pygame is an API to the popular C library SDL2 which is really performant to create games.
- **Huge Community** : since SDL2 is a really popular library it enables great maintainability.

Python inbuilt library we used :

- **Asyncio** : asyncio enabled us to treat the user's input as async so the program wouldn't have to wait for events to do something else.
- **Secrets** : a "Random" Ai was asked for the game although computer can only generate pseudo-randomness (except if they have external devices to measure "random" events), so we decided to use the best source of randomness of the computer generally used to do cryptographic stuff.

Data Structures

Game.state.structs.GameStructs :

This is a class responsible to hold General Game Data (not specific data of an element like a wall placement) the complete list of held data is :

```
- Self.input_queue -> Queue of inputs from the user
- Self.mouse_position_queue -> Queue of mouse positions
- Self.hovered_wall -> Bool to avoid double hovered wall
- Self.placed_wall -> Bool to avoid double wall placement
- Self.is_running -> Bool to make run the game loop
- self.WIDTH -> Window width
- self.HEIGHT -> Window height
- self.BOARD_SIZES -> list of all possible board size
- self.NUMBERS_OF_BARRIERS -> List of options for barriers
- self.NUMBERS_OF_PLAYERS -> List of possible number of players
- self.NUMBERS_OF_BOTS -> List of possible number of bots
- self.MAX_NUMBER_OF_BARRIERS -> Dict of max barriers for each board size
- self.TILE_SIZES -> Dict of Tile size for each board size
- self.WALL_WIDTH -> Dict of Wall width for each board size
- self.WALL_HEIGHT -> int of wall Height
- Self.current_player -> int to track player turn
- self.WIN_MESSAGE -> The win message to display (str)
- self.PLAYER_COLORS -> List of pawns colors
- Self.remaining_walls -> List to track remaining walls per player
- Self.bot_instances -> Dict of Bot instance (player:bot)
```

Game.elements.board.Board :

This is the class responsible to hold every instance of elements of the board, it hold instances of Tiles (`self.tiles`) and Pawns (`self.pawns`)

Game.elements.tile.Tile :

This is the class responsible of Tiles on the board it hold those attributes :

```
- self.x -> x position of the tile on the screen (int)
- self.y -> y position of the tile on the screen (int)
- self.x_index -> x index of the tile in the board (int)
- self.y_index -> y index of the tile in the board (int)
- self.size -> tile size (int)
- Self.wall_up -> instance of the top wall
- Self.wall_right -> instance of the right wall
- Self.wall_down -> instance of the bottom wall
- Self.wall_left -> instance of the left wall
- Self.pawn -> contain pawn instance if the pawn is on the Tile
- self.COLOR -> tile color
- self.HOVER_COLOR -> color of the tile when hovered
```

Game.elements.wall.Wall :

This is the class responsible of the Walls on the board it hold those attributes :

```
- self.x -> x position of the wall on the screen (int)
- self.y -> y position of the wall on the screen (int)
- self.type -> either 'horizontal' or 'vertical'
- self.active -> bool to deactivate the wall on the borders
- Self.placed -> bool to know if the wall is placed
- self.PLACED_COLOR -> placed color of the wall
- self.UNPLACED_COLOR -> color of the wall when it's nor placed
- self.HOVER_COLOR -> color of the wall when it is hovered
```

Game.element.pawn.Pawn :

This is the class responsible of the Pawns on the board it hold those attributes :

```
- self.x -> x position of the Pawn in the grid (int)
- self.y -> y position of the Pawn in the grid (int)
- self.color -> color of the pawn
- self.pawn -> contain the Circle instance representing the pawn
```

Pawn Move algorithm

The pawn moving algorithm is triggered in the main.py when there's a right click that collides with a tile, it then checks `is_move_possible()` (located in `Game.elements.board`), moves the pawn to the Tile the user clicked if it's possible (with the use of `pawn.move()`) and switch player turn with `witch_player_turn()` (placed in `game.events.local_game` (it modifies the struct)).

The `is_move_possible()` method check the following (it returns a bool):

- Calculate the distance the move will make (any move > 1 is not valid)
- Verify that the direction of the movement doesn't collide with any wall (with the `tile.wall_xxxx` attributes)
- If there's a player in the Tile to move it check if there's a free tile behind the Tile with the player on (based on the direction of the move)

The `move()` method will just set the pawn position to the clicked tile or to the tile behind if there's a player on the way

Walls Placement algorithm

The wall placement is also triggered when a wall collides with the click coordinate.


We also implemented a way to visualize where the wall is by calling an `hover()` method everytime the pointer position collide with a wall.

The first step in the Wall placement algorithm is getting the neighbor wall since by default a wall is the size of a single tile getting its neighbor allows us to make the wall 2 times the size. (we use the method `get_neighbor(x, y, orientation)` (from `Game.elements.board`) to get it).

After that we check `is_wall_placeable()` (from `Game.elements.board`) (return a bool), then we set the flag `placed_wall` in the main struct to avoid double wall placement (otherwise it happens when you click where 2 wall collide), afterward we place the 2 walls (clicked wall and neighbor) and check if the wall is blocking a player from winning by calling `is_path_to_victory_for_all_players()` (return a bool and is once again placed in `game.elements.board`).

If that's the case we remove the 2 placed walls and wait for another event.

Otherwise we decrement the wall counter with `decrement_wall_counter()` (placed in `game.events.local_game` (it modifies the struct)) and switch player turns.



The `get_neighbor(x, y, orientation)` method is pretty straight forward, since it just get the tile at position `x` and `y`, and based on the orientation it give the next tile (if it go over the board size it just select the board in the other way)

The `is_wall_placeable()` method checks the following conditions :

- Check if the clicked wall or the neighbor's wall is already placed
- Check if the wall is active (not on the borders)
- Check if a walls exist in the perpendicular direction (to avoid having a wall going through another wall)

The `is_path_to_victory_for_all_players()` use the bfs (Breadth First Search) algorithm and it goes like this :

(It iterates for all player on the board)

Initialize `starting_tile` , the tile where the player's pawn is

Initialize `reachable_tiles` and add `starting_tile` to it the list

Initialize a 2d array `visited` filled with False (except at the coords where `starting_tile` is)

Initialize a `queue` to keep track of tiles to visit

Then while there's still something in the queue:

It gets the first tile in the Queue and get its possible neighbors (to move to) it then if the neighbor hasn't been visited (check in the visited 2d array) if it hasn't it adds it to the `queue` , append the neighbor to the `reachable_tiles` list and then set the False in the `visited` 2d array to True to not add it twice to the `reachable_tiles` .

After that, it checks if in `reachable_tiles` there's a tile that has coordinates that would match a victory for each players.

Multiplayer Protocol

To be able to communicate with the different computers that play the game we use a TCP server.

Each player will connect through a socket to this server.

The server holds a copy of the board (with the different classes) and numerates each player that is connected to it.

It then waits for either a move or wall_place packet from the player which it is the turn.

Once received it sends back the packet to all players connected so they can update their local version of the game.

The move packet holds the coordinates of the tile to move to and the wall_place packets hold the coordinates of the tile and the orientation of the wall.

For any other information please check the source code everything is documented and the project is well organized so you won't have trouble finding the information you want.