Specification Breakdown

The Royal Game of Ur is a multiplayer race board game. You can find the original specification in the *PRG - Semestral Project - 01 - Specification.pdf* file.

Functional requirements:

1. Game implementation
2. Main menu
3. Artificial player

Architecture/Design

Parts

The project is split into following parts:

- *Program* - main entry point, which holds global variables *gameForm, Form1* and *game*
- *Form1* - implements Main menu
- *GameForm* - implements game visualisation and main game loop (with timer)
- *Game* - holds the state of the game with *player1, player2* and *board*, implements main game logic (moving tokens etc.)
- *Player* - abstract class which is implemented either by *RealPlayer* or *AIPlayer*
- *Board* - has information about *tiles* and *dice*
- *Tile* - can be also *RosetteTile* or *EmptyTile* (*StartTile* or *EndTile*)
- *Dice*
- *Token* - has info about it's tile index and image

Game logic

The game logic is mainly implemented in the *GameForm* and *Game* classes.
Game has an attribute *turn* that specifies which player is playing this round. The game itself is split into five phases which are implemented in *Game* class:

1. (phase 0): Give the player the possibility to throw dice (call *player.ThrowDice()* method)
2. (phase 1): Wait for the player to throw dice (mostly useful in case of *RealPlayer* because the game needs to wait for a button press)
3. (phase 2): Show dice throwing animation and continue
4. (phase 3): Let the player choose token (if the player can play, which is checked by *Game*'s *CanPlay* method) and move it (if possible)
5. (phase 4): Redraw board, return to phase 0

The current phase is saved in the *phase* attribute of the *Game* class.
The main game loop is implemented with Winforms timer. Each tick, the game checks whether there is a winner, and if not, it acts according to the current phase (calls the appropriate *Game* method and redraws the board, if needed).

Dice

Each dice has its own 50% probability that it will display a dot. In the *Throw* method it generates a random double. If it's lesser than 0.5, it will not display a dot, otherwise it will.

Therefore, the probabilities of throwing different numbers are:
0: 1/16
1: 4/16
2: 6/16
3: 4/16
4: 1/16
This is later used in the Expectiminimax algorithm for the AI player.

Real Player

When *Real Player*'s *ThrowDice* method is called, it displays a button. Phase of the game does not change until the player clicks the button (*button2*).

The *ChooseToken* method changes *RealPlayer*'s *waitingForClick* attribute to *true*. This attribute is checked by PictureBoxes of the tokens and if it's true it saves the clicked token into the *chosenToken* attribute. When *ChooseToken* sees that *chosenToken* is not empty, it resets it back to null, changes *waitingForClick* to false and returns *chosenToken*.

AI Player

The AIPlayer has three possible difficulties, each using different algorithm to choose a token to move:

1. Easy - chooses the first possible token that can be moved
2. Medium - prefers tokens that land on a Rosette, get token to the end or capture opponent's token
3. Hard - uses Expectiminimax to find the optimal move

Expectiminimax is a version of minimax which takes into account random events. It counts the weighted average of the possible results according to their probabilities.

To try all the possible moves, Expectiminimax uses the *ReversibleMove* and *Reverse* methods of the *Game* class. *ReversibleMove* calls the *Move* method but before, it saves the indices of all the tokens. *Reverse* method restores the previous state of the game.

Heuristics for rating the board is implemented in the *RatePosition* method of the *Game* class. It adds points for each token in the end position and on a Rosette and it subtracts points for each token still at the start. This should encourage capturing opponent's tokens but in reality it often makes the AI want to get as many tokens as possible on the board. This heuristic works, however it might be useful to try to find a better one in the future.

Moving tokens

When a token is chosen, *Game* checks if it can be moved with the method *CanMove*. If yes, the new tile is counted by the *GetNewTile* method and then the tile is moved by the *Move* method.
Whether a tile is occupied is checked via the *occupiedBy* attribute of the *Tile* class.

Technical documentation

Note that this list does not contain all the methods used in the program because some of them seemed to be self-explanatory (e.g. *GameForm.DrawBoard*) or were explained in the previous section (e.g. *RealPlayer.ThrowDice*).

AIPlayer

ChooseToken

- Calls *Easy()*, *Medium()* or *Hard()* method according to the *difficulty* attribute.
- Returns *Token* to play.

Easy

- Chooses the first possible token to play, using the *CanMove* method of the *Game* class.
- Returns *Token* to play.

Medium

- Searches for a token that lands on a Rosette, EndTile or captures opponent's token.
- Returns *Token* to play.

Hard

- Calls the *Expectiminimax* method with the specified depth and *game.DiceCount()*.
- Returns *Token* to play.

Expectiminimax

- Parameters:
    - *int depth:* how many moves into the future should the algorithm look
    - *Player player:* whose move does the algorithm examine
    - *int diceCount:* how many tiles will the token move
- Uses the Expectiminimax algorithm to find the optimal move.
- Calls itself recursively with decrementing depth until 0 depth is reached. Then uses *game.RatePosition()* method to rate the current position.
- Counts weighted average of all possible throws according to their probabilities.
- Returns the best *Token* to play and it’s expected value (*double*).

Dice

Throw

- Generates a random double, if it’s >= 0.5, sets dot to true, else it sets dot to false.

Game

Phase0

- Calls *ThrowDice* method of the current player.

Phase2

- If the player threw 0, skip one phase, else go to the next phase.

Phase3

- Parameters:
  - *int diceCount*: how many tiles should player move
- If the player can't move, go to the next phase and change who is playing (*turn* attribute).
- Else, let the player choose a token and if it can be moved, move it.
- If the token didn't land on a Rosette stone, change who is playing (*turn* attribute).

Phase4

- Reset *phase* attribute to 0.

CanPlay

- Parameters:
  - *Player player:* whose move to check
  - *int diceCount:* how many tiles should the token move
- Checks every *Token* in *player.tokens* with *CanMove* method. If any of them can move, returns true. Else returns false.
- Returns a *bool* that signifies whether the player has any possible moves.

CanMove

- Parameters:
  - *Token token:* token to move
  - *int count:* how many tiles should the token move
- The token cannot move if:
  - new tile would be out of range
  - new tile is Rosette and it's occupied by the other player
  - new tile is occupied by another token ot the same player
- Returns a *bool* signifying whether the token can move.

GetNewTile

- Parameters:
  - *Token token*: token to move
  - *int diceCount:* how many tiles should the token move
- Returns *int*, the index of the new tile.

Move

- Parameters:
  - *Token token:* token to move
  - *int diceCount:* how many tiles should the token move
- Moves the token to the tile with the index provided by *GetNewTile.*
- If a token lands on an EndTile, makes the token invisible.
- If it lands on a tile occupied by the other player, returns the other player's token back to start.
- Returns *true,* if the token landed on a Rosette, *false* otherwise.

RatePosition

- Heuristic for Expectiminimax.
- White tries to maximize, black to minimize.
- Each tile at the end gives +/-3 points.
- Each tile at a Rosette gives +/-1 point.
- Each tile at the start gives -/+2 points.
- Returns *int* rating of the position.

Reverse

- Parameters:
  - *int[] previousP1*: previous indices of white tokens
  - *int[] previousP2*: previous indices of black tokens
- Returns the game to the previous state.

ReversibleMove

- Parameters:
  - *Token token:* Token to move
  - *int diceCount:* How many tiles should the token move
- Saves the indices of every token into arrays (one for each player).
- Moves the given token.
- Returns *int[], int[]:* arrays of indices of previous token positions.

DiceCount

- Counts how many dice have a dot on them.
- Returns *int* count.

WhoIsPlaying

- Returns *Player*, whose turn it is (*player1* if *turn* is true, *player2* otherwise).

CreateBoard

- Initiates new board.

CheckWinner

- Checks if any player has *winning_count* tiles at the end.
- Returns winning *Player* or *null*.

GameForm

CalculateTilePositions

- Calculates position of each tile (depending on the width of the window).

TokenClicked

- Checks if the player is waitingForClick, if yes, saves clicked token into the *chosenToken* attribute of the player.

Tile

CalculatePosiiton

- Parameters:
  - *int width:* width of the console
  - *int i:* index of the tile
- Calculates position of the tile.
- Tiles are numbered from the right in the first and the third row (each row has 8 tiles, they are numbered from zero) and from the left in the second row.
- fillRect is one pixel smaller than the outer rect, this is for the picture to fit and not redraw the borders.
- Sets the *rect* and *fillRect* parameters of the *Tile*.