Ludo GUI

Nikolaj Iversen, Nicolai Lynnerup

March 18, 2016

1   Getting the Game

The game can be found on

`https://gitlab.com/niive12/ludo-gui`

Or cloned with

`git clone git@gitlab.com:niive12/ludo-gui.git`

The code depends on Qt and qmake.

```
mkdir -p ludo-gui/build
cd ludo-gui/build
qmake ../Ludo
make
```

And then the binary should be called Ludo.

2   The Game Structure

The ludo game is structured after a turn in the game.

Each turn is a series of communications between the game class and the player class. In figure 1 is the turn illustrated. The game class updates the GUI. To make it easy to follow the game, two delays is inserted. The speed of the game is currently set in with the setGameDelay(msecs) function in the main loop. One is inserted just when the turn is given to a player and a longer one is given after the dice has been rolled. The die is printed in the GUI on a colored square, representing which turn it is to move. In figure 2 can the GUI be seen. The format of the positions is a std::vector¡int¿. In the beginning of the turn, both the positions and the dice roll is given to the player. The two inputs is combined in a struct called positions_and_dice which has the members pos and dice.

There is a lot of different rules to the game. There is no special house rules added to the game. The player has one chance to roll a 6 to get out. A player only sends competitor home if they share the same end position. This means if a piece is standing on every star, the first one will not be touched, but the second piece will be sent home. Two pieces of the same team counts as a globe and will send the piece that moves to it home.
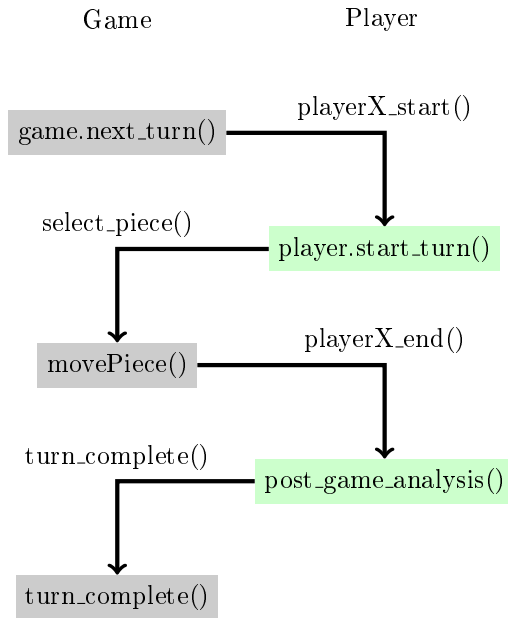
Game   Player

game.next_turn() ──────playerX_start()──────

                                    ↓
select_piece()   player.start_turn()
        ↓
movePiece() ──────────playerX_end()──────

                                    ↓
turn_complete()   post_game_analysis()
        ↓
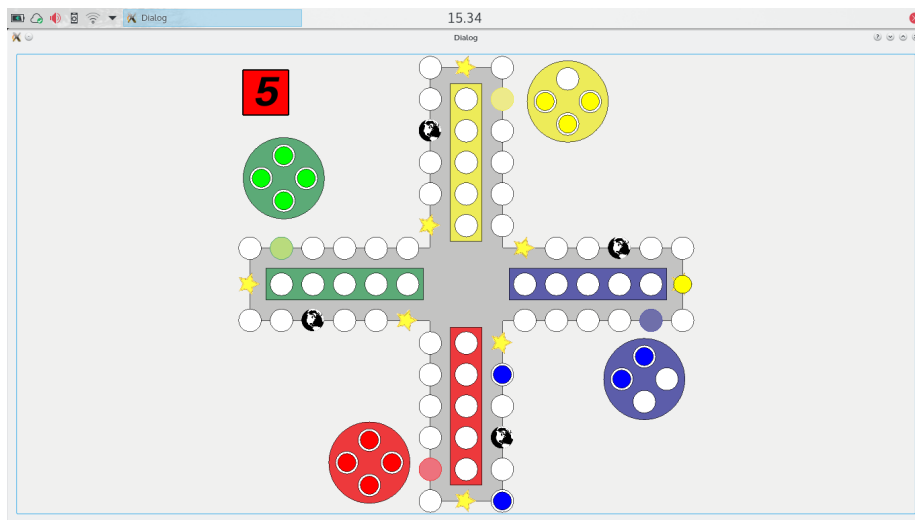turn_complete()

Figure 1: A turn in the game.

Figure 2: Screenshot of the game.

3 Implementing a Player

Two examples of a player is given in project. They should be implemented as a class which has the signals described in the previous section. For reference, it is also shown in Code Segment1. The private values and variables is up to the programmer, but they are included for completeness.

The positions are always sorted, so that index 0-3 is the players own pieces, 4-7 is the pieces of the next player clockwise on the board etc. The positions are also moved, such that they are relative to the starting point of the player. There is two "off board" positions a piece can be in. It can be in the home position which is called -1, and it can be in goal position, called 99. This allows for easy calculations of moving past field 57, so it moves backwards in the game. Field 0 is the first globe outside the start position, colored like the home field. Field 51 is the first step in the goal stretch.

An important note is that the player should check if it has won. The turn_complete sends a bool to the game, that, if true will declare them the winner and stop the game, and should be false if they have not won the game. The select_piece sends the index of the piece to the game.

```cpp
class ludo_player : public QObject {
    Q_OBJECT
private:
    std::vector<int> pos_start_of_turn;
    std::vector<int> pos_end_of_turn;
    int dice_roll;
    int make_decision();
public:
    ludo_player();
signals:
    void select_piece(int);
    void turn_complete(bool);
public slots:
    void start_turn(positions_and_dice relative);
    void post_game_analysis(std::vector<int> relative_pos);
};
```

Code Segment 1: Class skeleton as found in Ludo/ludo_player.h