SIRTET

H04

HSU, Shang Ling
HUANG, Yi Feng
LEE, Pin Hung

gamewindow.ui/.h/.cpp

```
namespace Ui {
   class gamewindow;
}
```

Ui::gamewindow is the class generated along with gamewindow.ui. It will be used to construct the new instance for the data member ui owned by our gamewindow.

```
class gamewindow : public QWidget
{
   Q_OBJECT
```

Inheritance hierarchy: QObject > QWiget > gamewindow

```
public:

   explicit gamewindow(QWidget *parent = nullptr, GameInstance*
_sirtet_game=nullptr);
```

The only and explicit constructor of gamewindow.
AiMoveButton connected here.
Helper function make_grid() called here.
[Parameter]

- parent: nullptr. This widget is not owned by any parent widget.
- _sirtet_game: As a gamewindow is “had” by some GameInstance, we pass the pointer to the GameInstance to the gamewindow.

```
   ~gamewindow() override;
```

The destructor delete the ui (of type Ui::gamewindow) and all the Squares.

```
   void update_ui();
```

Request the entire board, next, held, and line counter from the GameInstance and update the ui accordingly.

```
   void show_game_over();
```

Show the GameOverLabel and disable the AIMoveButton.

```
private:

    void closeEvent(QCloseEvent *event) override;
         Emit the close event for the mainwindow to handle.

    void keyPressEvent(QKeyEvent *event) override;
         Collect all key press event and call corresponding functions of
         the GameInstance.
         It ignores the event if the game has been over.
         Otherwise, certain keys are handled:
          • Left/Right/Down key for basic movements.
          • Up key for clockwise rotate.
          • Enter key for direct drop.
          • Shift key for held.

    void make_grid();
         Make instances of all the squares in the UI.

    Ui::gamewindow *ui;
    Square* square[25][10];
    GameInstance* sirtet_game;

private slots:
    void aiMoveButton_clicked_handler();
         Call the one AIMove from the GameInstance.

signals:
    void closed();
};
```

square.h/.cpp

```
typedef std::map<std::string, std::string> StyleMap;

class Square : public QLabel
{
   Q_OBJECT
        Inheritance hierarchy: QObject > QLabel > Square

public:
   Square(QWidget* parent=nullptr, int _row=0, int _col=0);
        [Paremeter]
         • parent: The gamewindow containing this Square.
         • _row: The y coordinate of the square. From bottom to top.
         • _col: The x coordinate of the square. From left to right.

   void set_color(int _color);
        Set the color of this square. Read in the corresponding file
        according to that color.
        Note: _color==-2 means the preview one.

   int get_color() const;
        The accesssor of data member color.

private:
   void render();
        Render this square according to the style sheet.

   void setStyle(std::string key, std::string value);
        Set a certain style.

   void applyStyle();
        Apply that style to the style sheet.

   int row, col;
   int color;
   StyleMap style;

   static const int OFFSET_X = 16;
   static const int OFFSET_Y = 16;
   static const int SQUARE_WIDTH = 32;
   static const int SQUARE_HEIGHT = 32;

};
```

mainwindow.ui/.h/.cpp

```
namespace Ui {
      class MainWindow;
}
        Ui::MainWindow is the class generated along with MainWindow.ui. It
        will be used to construct the new instance for the data member ui
        owned by our MainWindow.

class MainWindow : public QMainWindow
{
   Q_OBJECT
        Inheritance hierarchy: QObject > QMainWindow > MainWindow

public:
   explicit MainWindow(QWidget *parent = nullptr);
        Set up the UI and connect the StartButton.

   ~MainWindow();
        Delete the GameInstance and Ui::MainWindow owned.

private:
   Ui::MainWindow *ui;
   GameInstance* sirtet_game;

private slots:
   void startButton_clicked_handler();
        When StartButton is clicked, check whether there is already a game
        being played. If so, simply delete it and start a new
        GameInstance. The GameInstance will can the GameWindow later on so
        we do not have to handle it here.
        Note: The mainwindow is hidden upon the click of StartButton. It
        will be shown again when the gamewindow is closed.

   void game_window_closed_handler();
        As states, the mainwindow is hidden upon the click of StartButton.
        It will be shown again when the gamewindow is closed.
};
```

piece .h/ .cpp

```
#ifndef PIECE_H
#define PIECE_H

#include <utility>
using namespace std;

class Piece{
public:

    Piece(const int (&board)[25][10], int type, int orientation = 0,
pair<int, int> coordinate = make_pair(20, 4), bool alive = true);
        Constructor, take in a reference to the game board (a [25][10] int
array) to refer to in other method function, type referring to the type of
type of tetris, a pair of int for coordination, and an bool indicating
whether the piece is still in player's control.

    virtual ~Piece()=default;
      Destructor, default destructor is used as there is no dynamic data
member.

    int get_type() const;
      Getter for the type of this tetris, return the data member "type".

    int get_orientation() const;
      Getter for the orientation of this tetris, return the data member
"orientation".

    bool get_alive() const;
      Getter for the alive state of this tetris, return the data member
"alive".
    pair<int, int> get_coordinate() const;
      Getter for the coordination of this tetris, return the data member
"coordinate".

    virtual void get_occupied_coordinates(pair<int,int> args[4]) const =
0;
      Getter for the blocks occupied by this tetris. This is a pure
virtual function as different tetris occupies different blocks.

    void move_left();
      Attempt to move the block to left by one unit on the board.
      First get the current occupied coordinates and reduce their x
coordinates by one. If the new occupying four blocks is already occupied
or out of boundary, move it back to the original coordination.
```

```
    void move_right();
```
Attempt to move the block to right by one unit on the board.

First get the current occupied coordinates and increase their x coordinates by one. If the new occupying four blocks is already occupied or out of boundary, move it back to the original coordination.

```
    void move_down();
```
Attempt to move the block down by one unit on the board.

First get a copy of the current coordinates and reduce the copies' y coordinates by one. If the new occupying four blocks is already occupied or out of boundary, its bottom touch the ground or other blocks. Then change the piece's "alive" to false, meaning it can no longer be controlled by player. If the new occupying four blocks is not occupied and not out of boundary, then reduce this block's y coordinate by one.

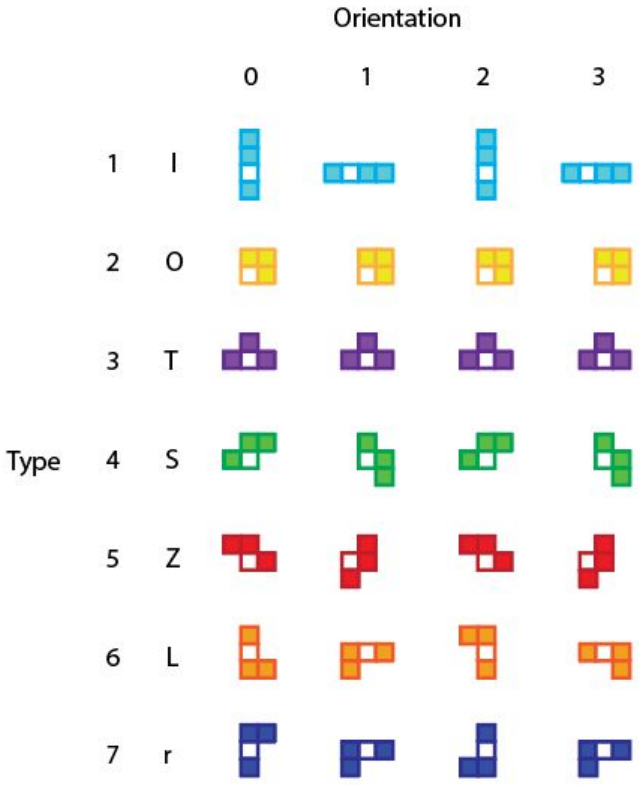
```
    void drop();
```
Move the block down until it touches the bottom of game map or its bottom touches a block.

Move the piece down until it is not alive.

```
    void rotate();
```
Attempt to move the block to left by one unit on the board.

First get the current occupied coordinates and reduce their x coordinates by one. If the new occupying four blocks is already occupied or out of boundary, try moving it left and right by one and two units, if the moves are still invalid, move it back to the original coordination.

```
protected:
    const int (&board) [25][10];
```
“board” is a constant reference that refer to the game board, which is a 25*10 int array. The class member functions access the status of the game board through this reference.
```
    int type;
```
“type” refers to the type of tetris of this piece. There are 7 types of piece, I, O, T, Z, S, L, and r.
```
    int orientation;
```
“orientation” refers to the current state of rotation of the piece.
```
    pair<int, int> coordinate;
```
“coordinate” is a pair of int, the first int refers to the x coordinate and the second refers to the y coordinate.
```
    bool alive;
```
“alive” indicates whether is this piece in the player’s control, which also means if the piece is falling.
```
};
```



The specification of the type and orientation is shown above, the white block represents the center for indicating coordinate.

```
#endif // PIECE_H
```

l .h / .cpp

```
#ifndef L_H
#define L_H
#include "piece.h"

class L: public Piece{
public:
    L(const int (&board)[25][10], int type = 6, int orientation = 0, pair<int,
int> coordinate = make_pair(23, 4), bool alive = true);
        Constructor, take in a reference to the game board (a [25][10] int
array) to refer to in other method function, type referring to the type of
type of tetris, a pair of int for coordination, and an bool indicating
whether the piece is still in player's control.
    virtual void get_occupied_coordinates(pair<int,int> args[4]) const
override;
        Takes in args, a array of pair, and copy the coordinates of the 4
occupying blocks of this piece to the array.
};

#endif // L_H
```

o .h / .cpp

```
#ifndef O_H
#define O_H
#include "piece.h"

class O: public Piece{
public:
    O(const int (&board)[25][10], int type = 2, int orientation = 0, pair<int,
int> coordinate = make_pair(23, 4), bool alive = true);
        Constructor, take in a reference to the game board (a [25][10] int
array) to refer to in other method function, type referring to the type of
type of tetris, a pair of int for coordination, and an bool indicating
whether the piece is still in player's control.
    virtual void get_occupied_coordinates(pair<int,int> args[4]) const
override;
        Takes in args, a array of pair, and copy the coordinates of the 4
occupying blocks of this piece to the array.
};

#endif // O_H
```

.t / .cpp

```
#ifndef T_H
#define T_H
#include "piece.h"

class T: public Piece{
public:
    T(const int (&board)[25][10], int type = 3, int orientation = 0, pair<int,
int> coordinate = make_pair(23, 4), bool alive = true);
        Constructor, take in a reference to the game board (a [25][10] int
array) to refer to in other method function, type referring to the type of
type of tetris, a pair of int for coordination, and an bool indicating
whether the piece is still in player's control.
    virtual void get_occupied_coordinates(pair<int,int> args[4]) const
override;
        Takes in args, a array of pair, and copy the coordinates of the 4
occupying blocks of this piece to the array.
};

#endif // T_H
```

s.h / .cpp

```
#ifndef S_H
#define S_H
#include "piece.h"

class S: public Piece{
public:
    S(const int (&board)[25][10], int type = 2, int orientation = 0, pair<int,
int> coordinate = make_pair(23, 4), bool alive = true);
        Constructor, take in a reference to the game board (a [25][10] int
array) to refer to in other method function, type referring to the type of
type of tetris, a pair of int for coordination, and an bool indicating
whether the piece is still in player's control.
    virtual void get_occupied_coordinates(pair<int,int> args[4]) const
override;
        Takes in args, a array of pair, and copy the coordinates of the 4
occupying blocks of this piece to the array.
};

#endif // S_H
```

.z / .cpp

```
#ifndef Z_H
#define Z_H
#include "piece.h"

class Z: public Piece{
public:
    Z(const int (&board)[25][10], int type = 3, int orientation = 0, pair<int,
int> coordinate = make_pair(23, 4), bool alive = true);
        Constructor, take in a reference to the game board (a [25][10] int
array) to refer to in other method function, type referring to the type of
type of tetris, a pair of int for coordination, and an bool indicating
whether the piece is still in player's control.
    virtual void get_occupied_coordinates(pair<int,int> args[4]) const
override;
        Takes in args, a array of pair, and copy the coordinates of the 4
occupying blocks of this piece to the array.
};

#endif // Z_H
```

l .h / .cpp

```
#ifndef L_H
#define L_H
#include "piece.h"

class L: public Piece{
public:
    L(const int (&board)[25][10], int type = 2, int orientation = 0, pair<int,
int> coordinate = make_pair(23, 4), bool alive = true);
        Constructor, take in a reference to the game board (a [25][10] int
array) to refer to in other method function, type referring to the type of
type of tetris, a pair of int for coordination, and an bool indicating
whether the piece is still in player's control.
    virtual void get_occupied_coordinates(pair<int,int> args[4]) const
override;
        Takes in args, a array of pair, and copy the coordinates of the 4
occupying blocks of this piece to the array.
};

#endif // L_H
```

r. h / .cpp

```
#ifndef R_H
#define R_H
#include "piece.h"

class R: public Piece{
public:
    R(const int (&board)[25][10], int type = 3, int orientation = 0, pair<int,
int> coordinate = make_pair(23, 4), bool alive = true);
        Constructor, take in a reference to the game board (a [25][10] int
array) to refer to in other method function, type referring to the type of
type of tetris, a pair of int for coordination, and an bool indicating
whether the piece is still in player's control.
    virtual void get_occupied_coordinates(pair<int,int> args[4]) const
override;
        Takes in args, a array of pair, and copy the coordinates of the 4
occupying blocks of this piece to the array.
};

#endif // R_H
```

gameinstance .h / .cpp

```
#ifndef GAMEINSTANCE_H
#define GAMEINSTANCE_H

#include <QObject>
#include "piece.h"
#include <queue>
#include <cstdlib>
#include <algorithm>
#include <QTimer>

using namespace std;

class gamewindow;

class GameInstance : public QObject{
    Q_OBJECT
public:
    GameInstance();
      Default constructor for GameInstance, initialize all the variables.
The constructor of GameWindow will be called here. The timer will also be
initialized here, and the timeout signal and handler is connected.

    ~GameInstance();
      Default destructor for GameInstance, the game_window will be deleted
here. (We don't need to delete the active_piece, since it will already be
deleted before cheking game over.)

    void startGraphicUI();
      Show the game_window and set the timer to 800, which means the timer
will emit timeout signal every 800 milliseconds.

    void get_board(int arr[25][10]) const;
      Getter function for the board, game_window will call this function
to show the graphic UI. We adjust the board here to make the current piece
(which is labelled -1 in the board) show its correspond color, and also
calculate the "preview" blocks.

    int get_current_type() const;
      Getter function for the type of active_piece.

    gamewindow* get_game_window() const;
      Getter fucntion for the game_window.
```

```
    int get_lines() const;
```
Getter function for the current cleared lines.

```
    bool get_game_status() const;
```
Getter function for the game_status (0 means game over, 1 means not yet game over.)

```
    int get_next();
```
Getter function for the next_on piece. If the queue for the next_on piece is empty, call “add_queue()” first.

```
    int get_held() const;
```
Getter function for the current holding piece. If there is no current holding piece, return 0.

```
    void click_left();
```
Handler function when “left” is clicked. Try to move the active piece left, then refresh the board.

```
    void click_right();
```
Handler function when “right” is clicked. Try to move the active piece right, then refresh the board.

```
    void click_down();
```
Handler function when “down” is clicked. Try to move the active piece down, then refresh the board.

```
    void click_up();
```
Handler function when “up” is clicked. Try to rotate the active piece, then refresh the board.

```
    void click_space();
```
Handler function when “enter” is clicked. Try to drop the active piece, then refresh the board.

```
    void click_shift();
```
Handler function when “shift” is clicked. Try to hold the active piece, then refresh the board. Note that “shift” key can’t be clicked 2 times in a row without dropping (or move down to the bottom) any piece between them.

```
    void AI_move();
```
Handler function when “AI_move” is clicked. The AI will calculate all the possible moves of the active piece and the held piece (if no, the next piece) and choose the one with the highest resulting board score. If it is the active piece that yielded the highest score, do that move. (Spin to the best orientation, move to the best place, and drop.) Otherwise, do click_shift(). Note that it is impossible for this function to make 2 “click_shift()” decisions in a row without dropping (or move down to the bottom) any piece between them.

```
private:
    QTimer* timer;
```
Timer for emitting timeout signal every 800 milliseconds.

```
    gamewindow* game_window;
```
Pointer to the current GameWindow.

```
    Piece* active_piece;
```
Pointer to the current active piece. A piece will be constructed when player has control to it, and will be destructed when player lose control to it (either dropped, moved to the bottom of the board, or held).

```
    int lines;
```
The current number of cleared lines.

```
    int board[25][10];
```
The current board. “0” means an empty square, integers between 1 and 7 means colored blocks (they are the color of “I”, “O”, “T”, “S”, “Z”, “L”, “r” pieces respectively). “-1” means the current active piece.

```
    int next_on[7];
```
This array serves as a queue of next on pieces.

```
    int cursor;
```
The cursor means the number of “used pieces” of the next_on queue.

```
    int hold_piece;
```
The type of the current holding piece.

```
    bool clicked_hold;
```
True if shifted is just clicked. Will be reset to false when a piece is dropped or moved down to the bottom of the board.

```
    bool game_status;
```
The current game status, false means the game is over and true otherwise.

```
    int clear_lines();
```
See if there is any complete lines in the board. If so, clear them all and return the number of cleared lines in this fucntion.

```
    void add_queue();
```
Random shuffle integers from 1 to 7 and add to the next_on queue. Reset cursor to 0.

```
    void move_next_to_board();
```
Move the next piece to board and increase cursor by 1.

```
    void move_piece_to_board(int piece_type);
```
Move a piece with given type to board, do nothing to the cursor.

```
    bool check_game_over() const;
```
Check if the game is over by checking if there are any blocks above the 20th row.

```
    void refresh_board(bool update_ui);
```
Get the current occupied squares and update the board accordingly. If the active piece should be killed, kill it, check if the game is over, clear lines, then move the next piece to board. After all, update the ui if update_ui is true.

```
    int AI_calculate_board_score(bool arr[25][10]) const;
```
Calculate the “score” of a board. Each entry is either 0 (empty) or 1 (there is a block). First, we calculate the “height” of each column, which is defined as the highest block in that column. Then, we calculate 4 characteristic values “aggregate_height”, “complete lines”, “holes”, and “bumpiness”. “aggregate_height” is the sum of all heights of the 10 columns. “complete lines” is the number of completed rows with all 1. “holes” is the number of blocks which is empty, but there exist a block which is in the same column but higher than it. “bumpiness” is the sum of the absolute value of height differences of adjacent columns. Obviously, a board with smaller aggregate_height, larger complete_lines, fewer holes, and smaller bumpiness is considered a “better” board which should recieve a higher score. The final score is calculated as “(-510066) * aggregate_height + (760666) * complete_lines + (-356630) * holes + (-184483) * bumpiness”. The weights are trianed by a genetic algorithm, which is done locally.

```
private slots:
    void timer_handler();
      Called every 800 milliseconds, automatically “click_down()”.
};

#endif // GAMEINSTANCE_H
```