i. Problem statement

This assignment requires students to create a program to simulate the board game Monopoly. In this game, two players (including one human player and one computer player) take turns to roll a six-sided dice and move around the game board, buying and investing properties. The rule is listed as follows:

- To play this game, every player must set up an account first with a positive starting balance of, say, 2000. The program is supposed to be able to track the balance changes as the game is going.
- This game is played on a game board as shown in Fig. 1. This game board consists of 38 squares. Each square has a price tag (this price may be generated randomly within a price range, say, from 10 to 300) for the ownership except for the left top corner square which is the starting square ("GO" square, each game starts from here). Each player's account balance will be increased by 200 for each time the player passes this square (the "GO" square).
- User and the computer take turns to roll a dice. The outcome of each rolling (a random number within the range of 1 to 6) decides how many squares user/computer can advance in a clockwise direction on the board. After the player have landed on a square: - if this square is unoccupied (except "GO" and "JAIL"): the player can decide whether or not you should buy it;
  - if this square is occupied by the player, then nothing needs to be done;
  - if this square is occupied by the opponent and the adjacent squares are unoccupied or occupied by the player: he/she will be fined by 10% of the square price;
  - if this square and one of its adjacent squares are both occupied by the opponent: the player will be fined by 20% of the square price;
  - The fine is topped at 20% of the square price even if more than 2 consecutive squares have been occupied by your opponent.
- The fine for each square can be further increased by 5% if the owner decides to invest on the square he/she has just bought off. The size of investment is half of the square price (which means the owner needs to pay 1.5×price to buy and invest the square).
- If the player lands on the "JAIL" square (the right bottom square), he/she needs to wait for a round before continue moving;
- The game ends when either one of the players declares bankruptcy (the balance <=0) or you have chosen to quit the game.

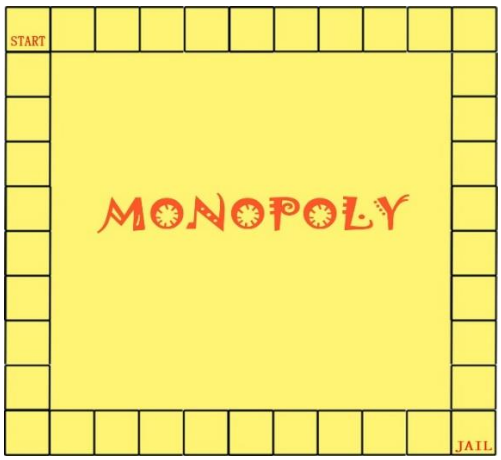


Fig.1. Game board

ii. Analysis

Based on the requirement, three classes are conceived and the class members of each class are specified in the following tables:

Table 2.1. Class members of Dice class

Dice class	
class members	responsibility
`int max, min`	The integers define the range of a random number. E.g. "Dice (6,1)" constraints the random number's range from 1 to 6.
`int number`	An integer used to store the random number generated by the method `roll_dice()`.
`void roll_dice()`	A method that would generate a random integer within the given range.
`int get_dice()`	A method that would receive the random integer generated by `roll_dice()` and pass it to `number`.

The dice class functions as a random number generator, which can simulate a six-sided dice. It could also be applied to define the computer player's strategies.

Table 2.2. Key class members of Player class

Player class	
Key class members	responsibility
`int balance`	An integer representing the player's account balance
`int X, Y, iniX, iniY, W, Len, Xmax, Ymax`	This set of integers would record the player's position and size on the game board. X, Y stand for the current position (X, Y); iniX and iniY stand for the initial position; W and Len stand for the width and length of the figure; Xmax and Ymax stand for the maximum coordinates on X and Y axis.
`void move(int)`	A method used to define the player's movement on the game board
`int get_place()`	Each square on the game board has an index. This method would return the index of the player's current position.
`void addBalance(int)` `void withdrawBalance(int)`	These methods would respectively increase and withdraw the player's balance by certain number.

The player class defines the attributes and behaviors of a player. In this program, two objects of player class should be initialized to respectively record the information of the human player (player-1) and the computer player (player-2) and the balances of both players are initially set to be 2000.

Table 2.3. Key class members of Asset class

Asset class	
Key class members	responsibility
`int place`	Define the index of each square
`int value`	Define the price of each square
`int own`	Define the ownership of the asset. Own is 0 means the asset is unoccupied; 1 means bought by player-1; 2 means bought by player-2

`int getValue()`
`void setOwn(int num)`
`int getOwn()`
`int getPlace()`
	Getters and setters

The asset class defines the attributes of each asset and each square on the game board stands for an asset. Therefore, 38 objects of asset class should be initialized in this program. According to the game's rule, each object has different "value" and "place", but "own" is initially set to be 0.

iii. Design

1. JAIL Pause

Figure 3.1 depicts the overall flowchart of the program, in which user would control Player_1 and start his/her round by rolling the dice. Normally, Player_1 and Player_2 would carry out their rounds alternately until one of them wins the game.

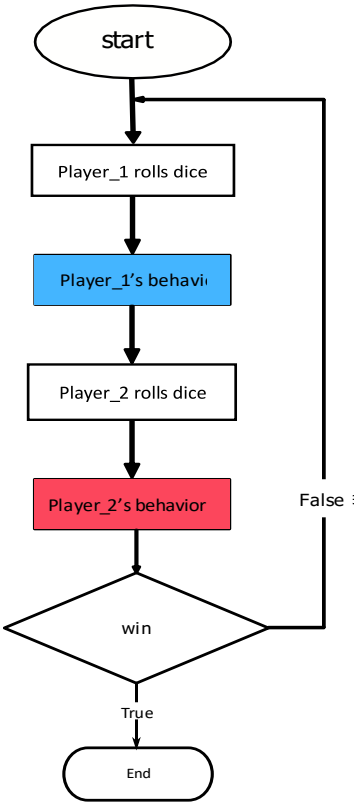


Figure 3.1. The overall flowchart of Monopoly

However, according to the rule, when one of the players enter the special square JAIL, his/her next round would be deprived and the opponent would get two consecutive rounds. To achieve this goal, the flowchart is extended: the variable next_round is declared to count the rounds of Player_1 staying in the JAIL and is initially assigned 0. If Player_1 firstly enter the JAIL, next_round would be added by 1. In the next step that checks whether next_round < 1, the player_1's round would be skipped and next_round would be subtracted by 2. Then, during the next round, the value of Player_1's next_round is less than 1 and finally set to be 0, therefore, the player could execute his/her round as normal. Another variable next_round2 performs the same operations for Player_2. The extended flowchart is given in Figure 3.2.

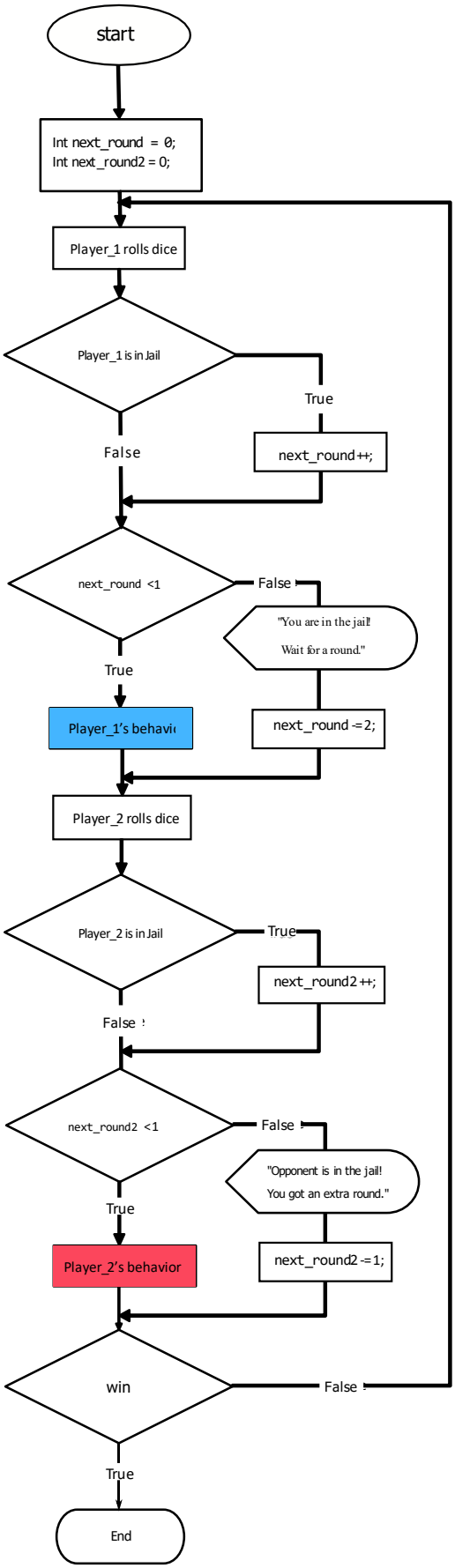


Figure 3.2. The extened flowchart to perform JAIL pause

2. Player's Behavior

Player_1's behavior is designed and indicated in Figure 3.3. The player's figure would firstly move forward n squares (n is the dice number rolled by the payer previously) on the gameboard. Then, the player can choose to buy or invest the current asset.

i. Purchasing

If the player decides to buy, the program would firstly rule out the 'GO' and 'JAIL' squares, and then check whether the asset has been occupied –if it is unoccupied, the player's balance would be reduced by the asset value and the asset's ownership would belong to the player; If occupied, the program would further check the asset's ownership and specify who occupied the asset.

ii. Investment

If the player decides to invest, the program would exclude the unoccupied square and the square occupied by opponent by similar conditional statements in Buy procedure. Only the square owned by the player can be invested and the player has to withdraw 50% of the current asset's price from his/her balance account for investment.

iv. Is_occupied Judgement

In order to identify the ownership of an asset, two methods are provided. The first option is to apply two int vectors "Vec_Owned" and "Vec_Occupied" to store the indexes of assets brought by the player and opponent. The two vectors are initially empty and if the player and opponent buy assets in their rounds, the index of assets they brought would be respectively pushed back into the two vectors. During the Buy and Invest process the program would determine whether the current asset the player stays is occupied or not by matching its index with the elements of the two vectors—If the index is repeated in the vector "Vec_Owned", it would meant that the current asset has been owned by the player; if in "Vec_Occupied", the asset has been occupied by the opponent; if neither in "Vec_Owned" nor in "Vec_Occupied", the asset is unoccupied.

Another option is to check the variable "own" of each asset object. Asset class contains an int variable "own" (mentioned in Analysis section). The own variable of each asset is initially set to zero, meaning that it is unoccupied. When Player_1 buys one asset, its "own" would be converted to 1; whereas when Player_2 buys the asset, its own would be converted to 2. Therefore, the program could identify an asset's ownership by checking its variable "own".

v. Penalty

The penalty operation can be performed by applying the aforementioned two methods to identify ownerships of assets. If the current asset is occupied by the opponent while both of the adjacent assets are not occupied by the opponent, the player would be penalized by 10% of the current asset's value; If the current asset and at least one of the adjacent assets are occupied by the opponent, the player would get 20% penalty.

vi. Player_2's behavior

Player_2 is controlled by the computer and its behavior should be similar to that of Player_1. A series of strategies are defined to perform Player_2's behavior:

The purchasing strategy is designed based on Player_2's balance as well as the asset's price. When player_2's balance is more than or equal to 300, it would preferentially buy the assets with prices larger than 150; while for other assets cheaper than 300, the player could randomly choose to buy or not, which can be achieved by instantiating Dice class; When its balance is less than 300, it would stop buying any asset. As for investing strategy, Player_2 would randomly invest any occupied assets when its balance account is more than 600; otherwise, it would stop

investing.

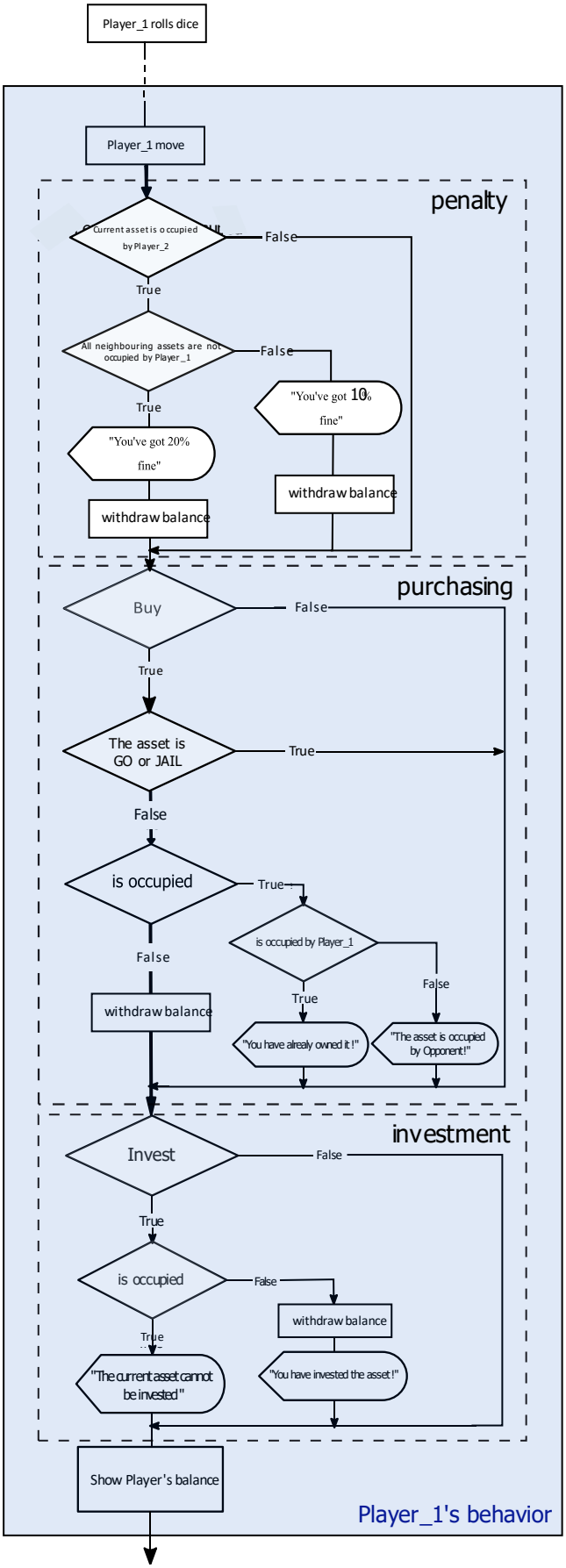

Player_1 rolls dice
Player_1 move
penalty
Current asset is occupied by Player_2
False
True
All neighbouring assets are not occupied by Player_1
False
True
"You've got 10% fine"
"You've got 20% fine"
withdraw balance
withdraw balance
purchasing
Buy
False
True
The asset is GO or JAIL
True
False
is occupied
True
False
is occupied by Player_1
True
False
withdraw balance
"You have alreay owned it !"
"The asset is occupied by Opponent!"
investment
Invest
False
True
is occupied
False
True
withdraw balance
"The current asset cannot be invested"
"You have invested the asset !"
Show Player's balance
Player_1's behavior

Figure 3.3. The flowchart of Player_1's behavior

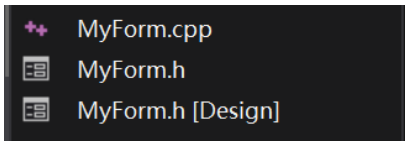
3. Graphical User Interface

Graphical User Interface (GUI) is invoked in this program to design a visible and flexible interaction with user through graphical icons (e.g. button) and visual indicators (e.g. label), instead of text-based user interfaces.

1.Setup

MS Visual Studio provides direct access to set up GUI in Windows form:

- Select Visual C++ and open a CLR empty project;
- Inside the project, choose Add New Item and select Visual C++. Add a Windows FORM under UI with its default name "MyForm". Then an empty window form is obtained (show in Figure 3.4 (a)).
- Two files are built followed by adding the Windows Form, which can be found in "Solution Explorer" on the right side: The source file "MyForm.cpp" is used to start the form and the header file "MyForm.h" is used to program the Form. Additionally, Visual Studio also provides a Form designer "MyForm.h [Design]", which displays a design view of the Form and provides graphical manipulation instead of codes.



- In "MyForm.h [Design]", several components (including Buttons, Labels and Picture Boxes) are added to the Form by dragging them from Common Controls list under the Toolbox on the left side. The previous empty Form would now become a Monopoly game page (shown in Figure 3.4 (b)). The four buttons Roll, Buy, Invest and Surrender would execute the player's operations. A gameboard as well as two figures of players are also included due to aesthetic concerns.

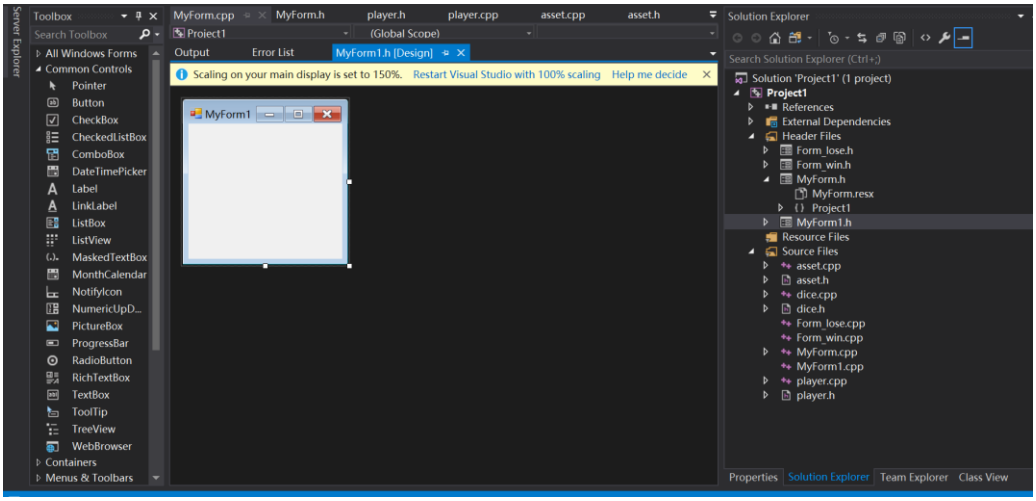


Figure 3.4(a). Demonstration of an empty Windows Form designer

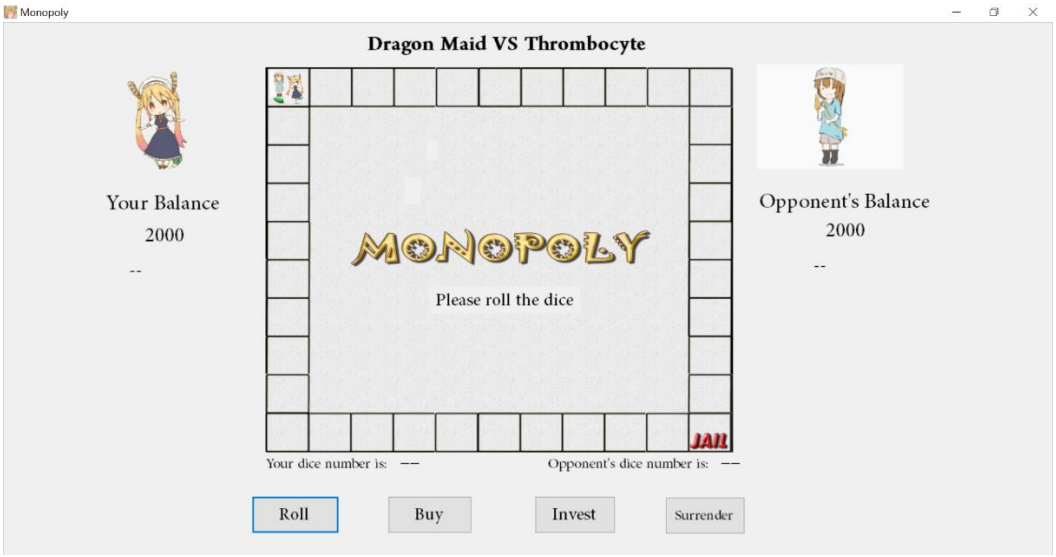


Figure 3.4 (b). Layout of Monopoly game page

2. Button Click Events

The program is expected to work when clicking the buttons, which are currently reactionless. Therefore, click event handlers would be added to the four buttons. Since Visual Studio provides a user-friendly Form designer "MyForm.h [Design]", an empty click event handler would be automatically generated in "MyForm.h" files by double-clicking one button icon in "MyForm.h [Design]". In the next step, the rest of codes in each click event handler should be filled: For "Roll_click()" handler, codes that executes the overall procedures of the game as well as JAIL Pause (displayed in Figure 3.2) should be added; For "Buy_click()" and "Invest_click()" handlers, codes performing Player_1's purchasing and investment procedures (displayed in Figure 3.3) would be respectively added; For "Surrender_click()" handlers, codes that exits the From would be added.

3. Victory and Failure Procedures

The final step is to design the victory and failure procedures in "Roll_click()" handler. At the end of each round, the program would settle the two players' balance. If Player_1's balance is less than or equal to 0 while Player_2's is positive or if "Surrender" button is clicked, a new form would be opened to sneer at the player's failure; Otherwise, if Player_1's balance is positive whereas Player_2's balance less than or equal to 0, another form would be opened to felicitate the player's victory. After the player closes the new form, the parent form is also exited automatically and then the program ends.

iv. Implementation

The program is submitted in the package Assessment 3. The source files can be found in Assessment 3\Project1 and the game can be run through Project1.exe file in Assessment 3\Debug.

v. Test

The running processes of the game Monopoly are displayed in this section.

1.The game page

The main interface of the game is shown in Figure 5.1. At the beginning of the game, the players' figure would land on the GO square. The balance accounts of both players are set to be 2000 and displayed on either side of the screen.

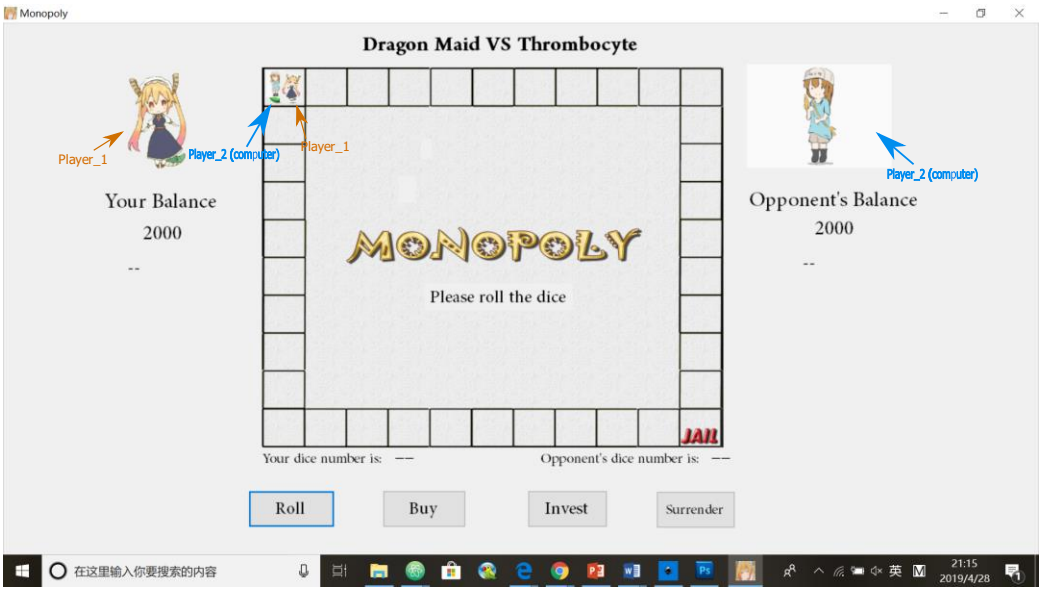


Figure 5.1. Layout of the game page

2.Rolling Dice

When clicking Roll button, Player_1 and Player_2 would respectively roll a six-sided dice, the dice numbers of both players are displayed at the bottom of the game board. The players would move forward by a certain number of squares based on the rolling outcomes. The price of the current asset would be randomly generated and displayed on the screen.

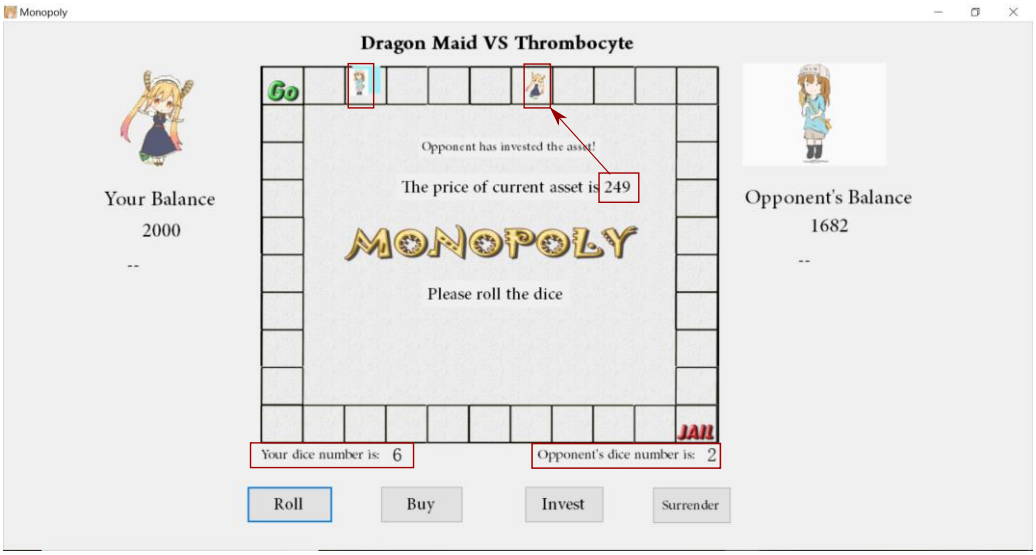


Figure 5.2. The click event of rolling dice

3.Buying assets

When the player clicks Buy button and if the current asset is unoccupied, a message stating that "You have bought the asset" would be displayed and the player's balance account would be reduced by the asset's price. In addition, a color block would cover the asset bought by the player to indicate its ownership—an orange block indicates the asset is owned by Player_1 and a blue block indicates the asset is occupied by the opponent, which is depicted in Figure 5.3 (a). If the payer is trying to buy an occupied asset, an error message box will be displayed, which is shown in Figure 5.3 (b) and (c).

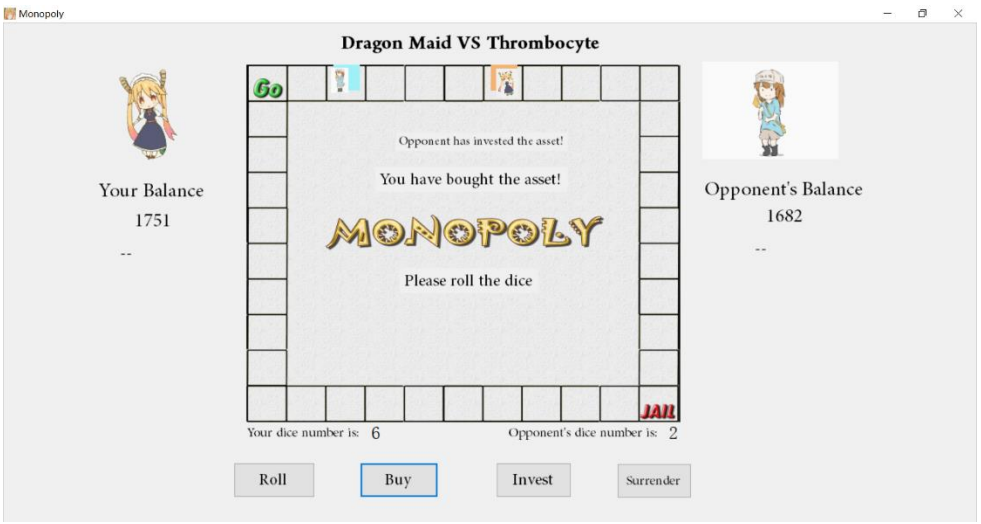


Figure 5.3 (a). The click event of buying an unoccupied dice

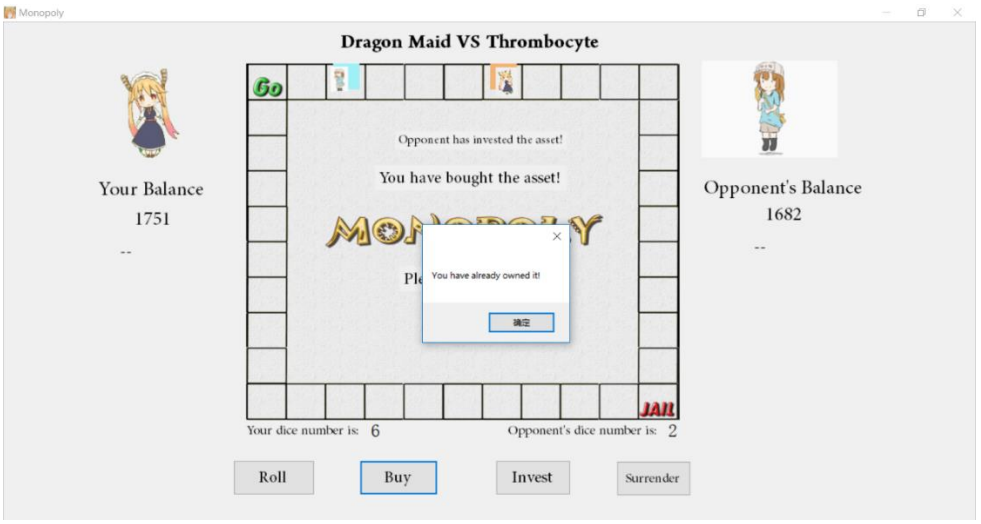


Figure 5.3 (b). When buying an asset owned by the player, the message “You have already owned it!” would be displayed.

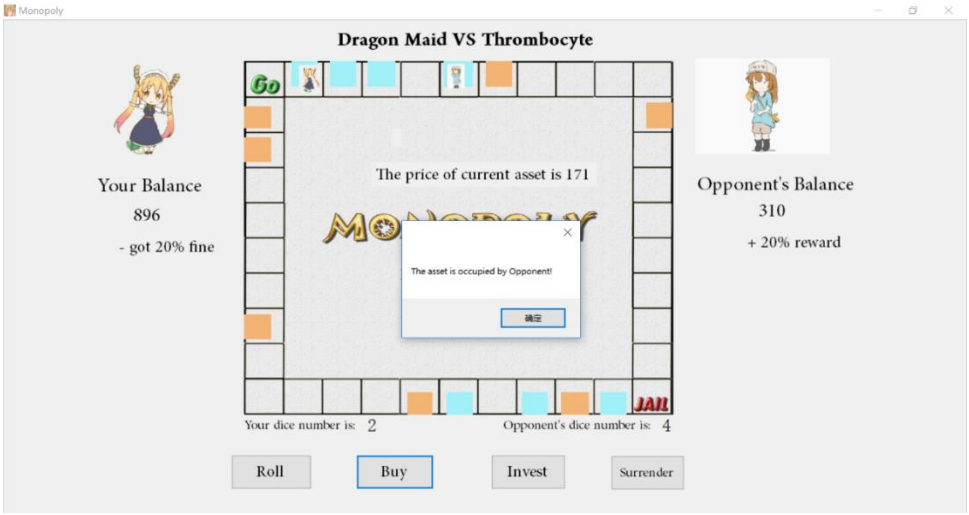


Figure 5.3 (c). When buying an asset occupied by the opponent, the message “The asset is occupied by Opponent!” would be displayed.

4.Investing assets

When the player invests an already-bought asset, the asset's value would increase by 25% of its original price and the account balance of the player would be reduced by 50% of the asset's original price. These processes are given in Figure 5.4 (a), (b) and (c)

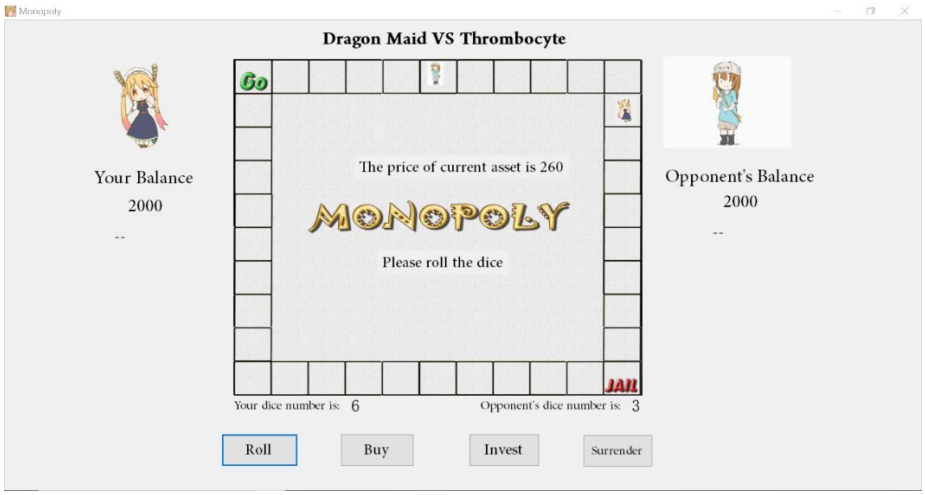


Figure 5.4 (a). The state before buying and investing. Note that the price of the current asset is 260.

Figure 5.4 (b). After buying and investing, the player's account is reduced by 150% of the current asset's price.

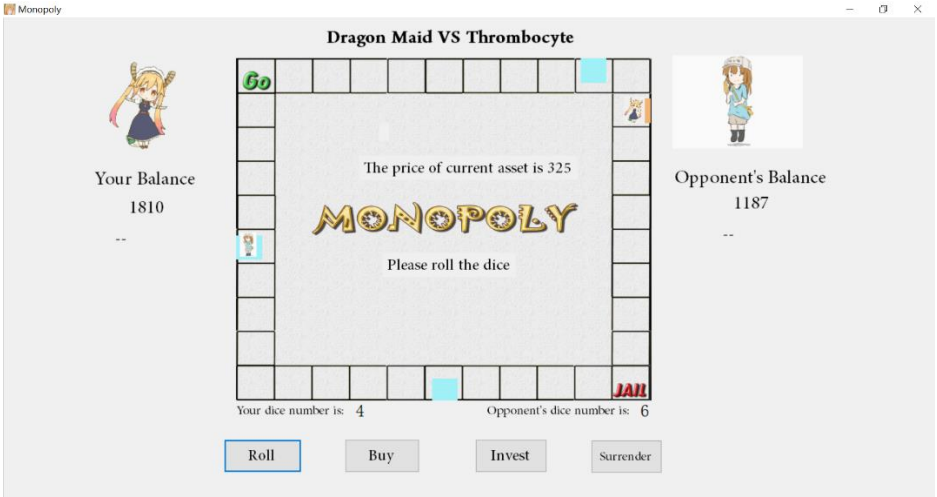


Figure 5.4 (c). After investing, the price of the asset is 325, 125% of the original price.

If the player mistakenly invests an invalid asset (including unoccupied asset, asset occupied by the opponent, GO and JAIL) a mistake message box will be displayed (shown in Figure 5.5).

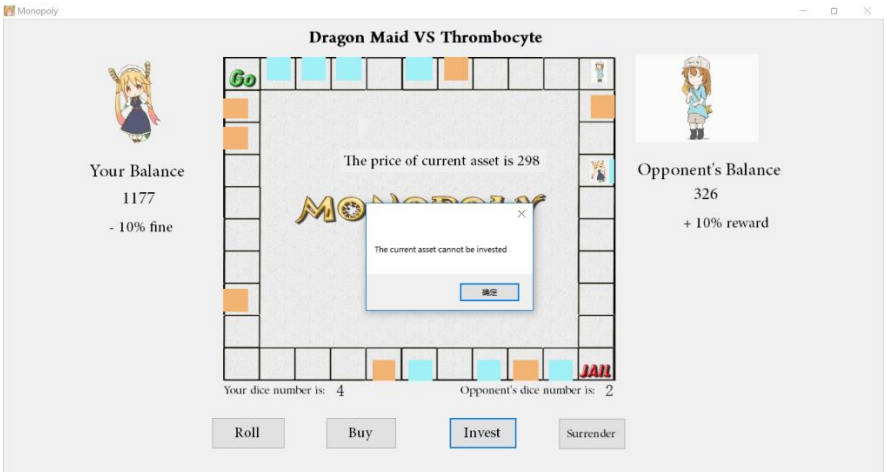


Figure 5.5. An error message is thrown to handle the invalid investment.

5. Penalty

The layouts of 10% and 20% penalties are depicted in Figure 5.5 (a) and (b).

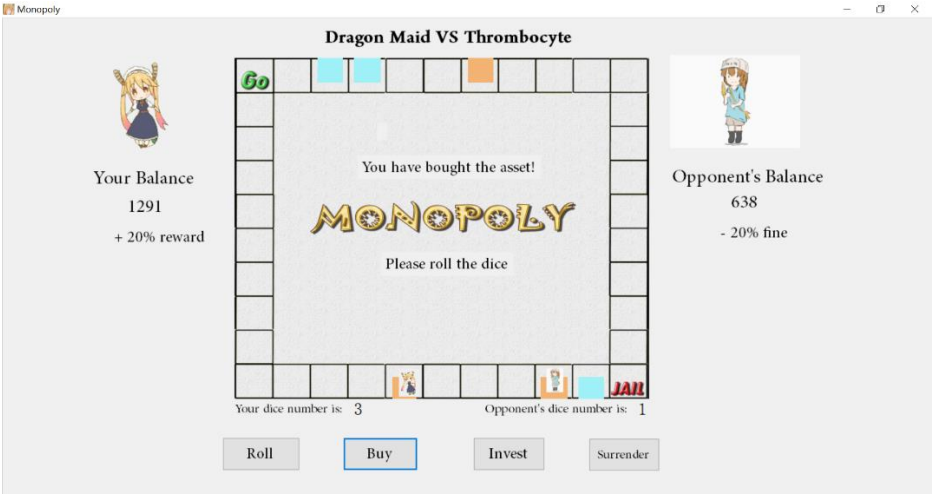


Figure 5.5(a). When the opponent got a 20% fine, the player would receive 20% reward

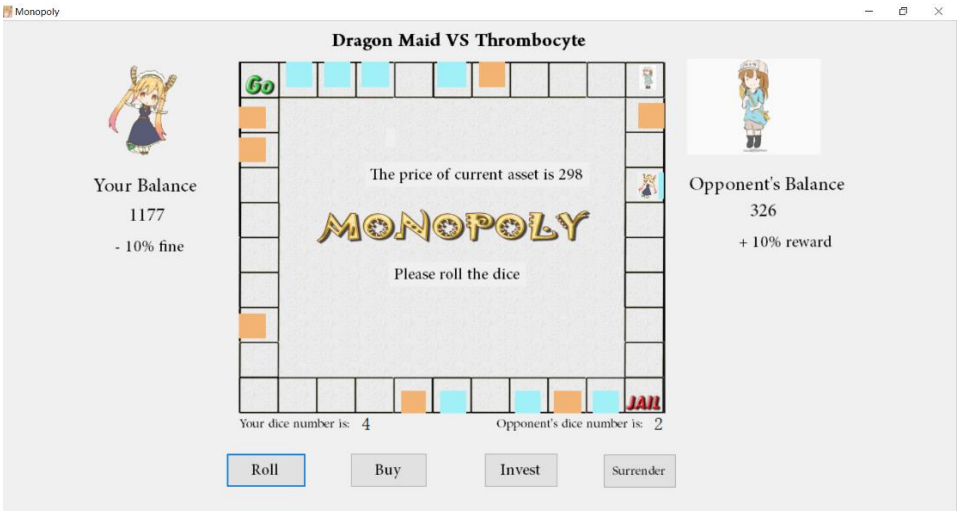


Figure 5.5(a). When the player got a 10% fine, the opponent would receive 10% reward

7. JAIL Pause

Figure 5.6 (a) and (b) respectively display the situations when Player_1 and Player_2 land in JAIL.

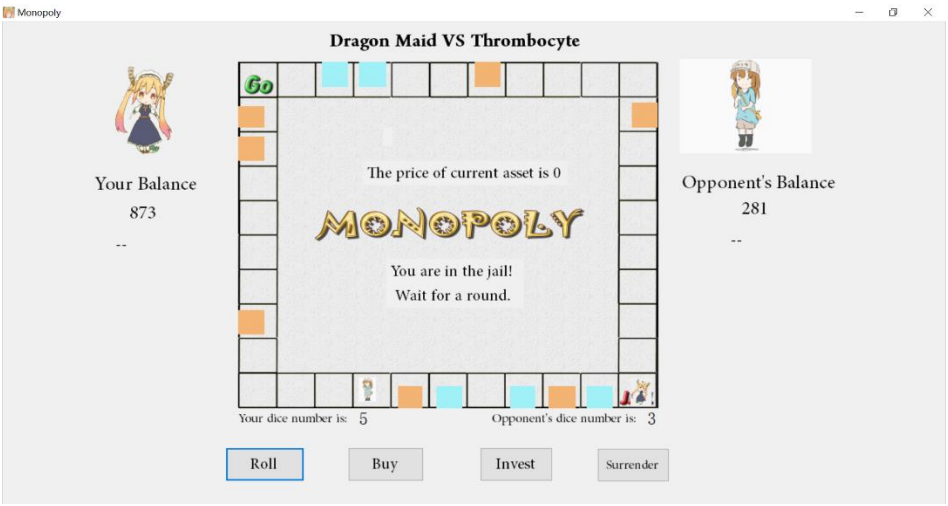


Figure 5.6 (a). Situation of Player_1 in JAIL

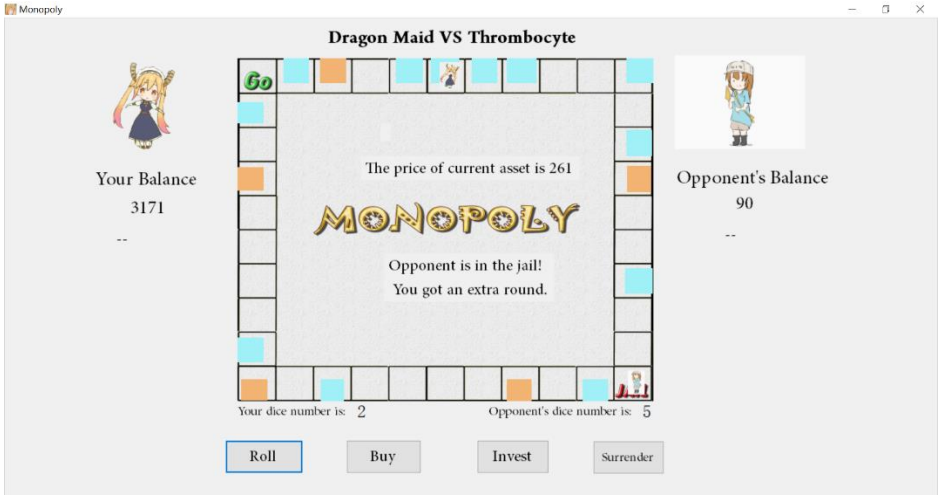


Figure 5.6 (B). Situation of Player_2 in JAIL

6. End of the Game

The player could end the game by defeating the opponent or losing the game or clicking Surrender button. If defeating the opponent, a “You win!” form will be displayed (shown in Figure 5.7 (a)), otherwise, a “You lose!” form will be displayed (shown in Figure 5.7 (b)). Finally, if the player closes the new form (either “You win” or “You lose”) the parent form will be automatically closed.

Figure 5.7 (a) Layout of "You win!" form

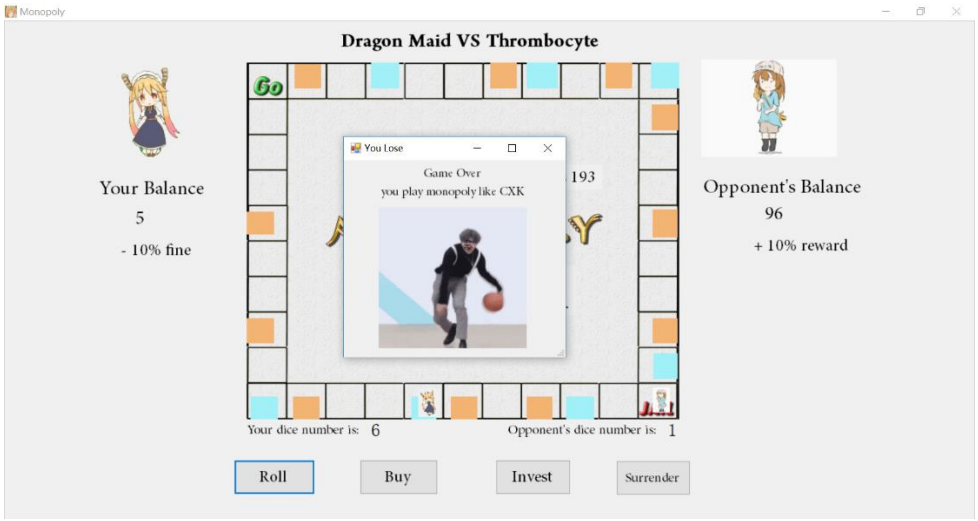


Figure 5.7 (b). Layout of "You lose!" form