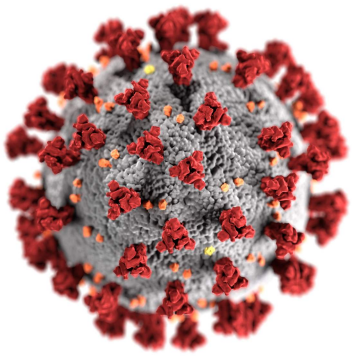
Hong Kong University of Science and Technology

2020-2021 Fall Semester

Project Documentation for COMP2012H

-Virus Fighter-



By LIN,Yuchen WEI,Yuanjing YANG,Lin

Project #H11

December 4, 2020

Contents

1 Overview **1**

2 GameEngine - The logic of the whole game **1**

3 Player **10**

3.1 Appendix: The logic of changing the player/enemy's photo 17

4 Enemy (Base class) **20**

4.1 Anti (Derived class) 25

5 Boss **27**

6 Bullet (Base class) **32**

6.1 Virus (Base class & Derived class) 33

6.1.1 VirusC (Derived class) 36

6.2 Weapon 37

7 Props (Bass Class) **39**

7.1 Freezer (Alias: disinfector) 39

7.2 BloodBag (Alias: pills) 40

7.3 Medal 41

7.4 Mask (Alias: shield) 41

8 GUI at the end of the game **41**

8.1 Win 42

8.2 Lose 43

8.3 Ranking 45

1 Overview

Virus Fighter is a Bullet Hell game under the game genre Shoot'em up(aka STG). In this type of game, the player has to combat large amounts of enemies by shooting at them while dodging their fire. We chose to implement this kind of game because it is fun and easy to play. Besides, we believe that it would be an excellent opportunity for us to make fair use of OOP since there are at least four different types of objects (i. e., player, enemy, bullet, boss) in the game.

The idea of our game mainly comes from the unusual and challenging year. We name our game Virus Fighter to memorize the tough but admirable and memorable process of people from different nations and backgrounds working together to fight against covid-19. Besides, we thought the virus element might add to the uniqueness of the game.

Virus Fighter is equipped with essential features of the Bullet Hell game. The player could move up/down/left/right by pressing "WSAD" and shot by pressing space. There are seven types of objects: player, enemy, boss, weapon, virus, boss virus, and props. To win the game, the player has to dodge the viruses shot from enemies while killing enemies by shooting at them to score. An enemy might leave a prop after it dies, the player could pick it up and gain some advantages. When the score reaches certain bars, the level goes up, and game status changes. The ultimate boss will come up at the last level. The final goal for the player is to kill the boss and score as much as he/she can. If the player is killed by the boss, he/she loses. However, even if the player has killed the boss, the game will not end until the time is up, or health decreases to zero. During the time between the boss died and the game ends, the player is encouraged to score as much as he/she can since the score will be counted for ranking.

2 GameEngine - The logic of the whole game

```
1 /* Header File for GameEngine except include part */
2 namespace Ui {
3 class GameEngine;
4 }
5
```

```
6  class GameEngine :  public QMainWindow
7  {
8      Q_OBJECT
9  public:
10     explicit GameEngine(QString name = NULL,QWidget *parent = nullptr);
11     ~GameEngine();
12     void game_begin();
13     void update_speed_and_potential();
14     void update_speed_and_potential_frozen();
15     void update_speed_and_potential_shield();
16     void write_in_ranking_and_finish_game();
17     Player* player;
18     Score* score;
19     Health * health;
20     QGraphicsScene *scene;
21     int level = 0;
22     QString name;
23     Ui::GameEngine *ui;
24     int enemy_num = 0;
25     bool is_end(){return end;}
26 private slots:
27     void Quit_clicked();
28     void create_boss();
29     void set_focus();
30     void game_logic_control();
31     void level_control();
32     void set_score();
33     void set_health();
34     void set_level();
35     void set_time();
36     void on_Esc_clicked();
```

```
private:
    QTimer* timer_for_enemy;
    QTimer* timer_for_boss;
    QTimer* timer_for_clock;
    QTimer* timer_for_focus;
    QTimer* timer_for_level;
    vector<vector<vector<int>>> level_array=/* neglected here */ ;
    vector<vector<vector<int>>> level_array_frozen=/* neglected here */  ;
    vector<vector<vector<int>>> level_array_shield=/* neglected here */  ;
    Boss* boss = nullptr;
    int time = 200;
    bool end = false;
    int score_record = 0;
    QMediaPlayer *main_background_music;
};
```

The GameEngine class is the main controller of the whole game, equipped with a GUI game surface, a set of functions and variables (and objects) which play a vital role in the entire game.
Unlike the game implemented in PA4 and other similar games, the shooting game requires not only the input event-based reaction of the program, but also the capability of continuing the game smoothly in parallel with the player's input. As a result, several `QTimer` are used to update different displays and parameters in the game to bring about better game experience. In addition, there are utility functions used to handle the update of parameters and GUI surface.

(1) Basics

a) `score` and `health`

They are class objects in charge of the score and the health of the player, respectively. As strange as it may seem, instead of using a set of integers to record the value of score and health, two classes are implemented so that the function of recording and alternating can be implanted into the object `score` and `health` as well.

b) timer_for_enemy,timer_for_clock,timer_for_focus,timer_for_level

```
timer_for_enemy = new QTimer();
QObject::connect(timer_for_enemy,SIGNAL(timeout()),player,SLOT(spawn()));

timer_for_focus = new QTimer();
QObject::connect(timer_for_focus,SIGNAL(timeout()),this,SLOT(set_focus()));
QObject::connect(timer_for_focus,SIGNAL(timeout()),this,SLOT(set_score()));
QObject::connect(timer_for_focus,SIGNAL(timeout()),this,SLOT(set_health()));
QObject::connect(timer_for_focus,SIGNAL(timeout()),this,SLOT(set_level()));
timer_for_focus->start(500);

timer_for_clock = new QTimer();
QObject::connect(timer_for_clock,SIGNAL(timeout()),this,SLOT(set_time()));
timer_for_clock->start(1000);

timer_for_level = new QTimer();
QObject::connect(timer_for_level,SIGNAL(timeout()),this,SLOT(level_control()));
timer_for_level->start(200);
```

It can been seen that timer_for_focus is in charge of the update of score, health, and level readings on the GUI surface as well as setting the focus (to prevent other programs interrupting the keyboard input).

c) set-functions

```
void GameEngine::set_score()
{
    QLCDNumber* lcd=ui->Score;
    lcd->display(score->getScore());
    lcd->show();
}
void GameEngine::set_health()
{
    QProgressBar* health_dis=ui->Health;
```

```
    health_dis->setValue(health->getHealth());
    health_dis->show();
}
void GameEngine::set_level()
{
    QLCDNumber* level_lcd=ui->Level;
    level_lcd->display(level);
    level_lcd->show();
}
void GameEngine::set_time()
{
    time-=1;
    QLCDNumber* time_lcd=ui->Time_left;
    time_lcd->display(time);
    time_lcd->show();
}
```

These are slot functions triggered by the `QTimer` from time to time. Corresponding `QLCDNumber` and `QProgressBar` have been designed in `gameengine.ui`.

(2) Functions and variables related to the game's logic

a) `level_control()`

```
void GameEngine::level_control()
{
    //This function is used to check and upgrade the level of the
    ↪ player, then invoke different functions
    //to create events (), it can be seen that
    //the level of the player is determined by its score.

    //level_new is the variable determined in the current vall of
    ↪ level_control() function
```

```
8     int level_new = 0;
9     int score_new = score->getScore();
10    //if(health->getHealth() == 0 && end == false && score_new <= 170)
11
12    //To finish the game boss need to be killed and health runs out or
      ↪  time runs out
13    if(!Boss::bossIsKilled()&&(health->getHealth() <= 0||time<=0) &&
      ↪  end == false)
14    {
15        this->close();
16        //qDebug() << "Game finished";
17        Lose* new_lose = new
          ↪  Lose(score->getScore(),enemy_num,level,this);
18        new_lose->show();
19        // play background music
20        main_background_music->stop();
21        QMediaPlayer * music = new QMediaPlayer();
22
          ↪  music->setMedia(QUrl("qrc:/music/game_resource/music/sprint.mp3"));
23        music->play();
24
25        end = true;
26        return;
27    }
28    //else
      ↪  if((score_new>=380||Boss::bossIsKilled())&&time>=0&&health->getHealth()>=0&&e
29    else
      ↪  if((score_new>=380||Boss::bossIsKilled())&&(time<=0||health->getHealth()<=0)&
30    {
31        this->close();
32        score_record = score->getScore();
```

```
33          Win* new_win = new Win(score->getScore(),enemy_num,this);
34          new_win->show();
35          // play background music
36          main_background_music->stop();
37          QMediaPlayer * music = new QMediaPlayer();
38          music->setMedia(QUrl("qrc:/music/game_resource/music/legends
            ↪  never die.mp3"));
39          music->play();
40
41          end = true;
42          //write_in_ranking_and_finish_game(); //execute final progam
            ↪  after win has beem closed
43          return;
44      }
45      else if(score_new < 40) level_new = 1;
46      else if(score_new < 80) level_new = 2;
47      else if(score_new < 120) level_new = 3;
48      else if(score_new < 170) level_new = 4;
49      else if(score_new < 380&&!Boss::bossIsKilled()) level_new = 5;
50
51      // The following statements will be executed only when there's
        ↪  discrepancy between old level and new level
52      // Event-based reaction to save resource
53      if(level_new != level && level_new != 0)
54      {
55          level = level_new;
56          player->level_up(level_new);
57          update_speed_and_potential();
58          game_logic_control();
59          if(level_new == 1)
60          {
```

```
61              this->ui->textBrowser->setTextColor(QColor(255,0,0));
62              this->ui->textBrowser->QTextEdit::append("Game Start!");
63          }
64          else
65          {
66              this->ui->textBrowser->setTextColor(QColor(255,0,0));
67              this->ui->textBrowser->QTextEdit::append("Level up to " +
                ↪  QString::number(level_new) + " !");
68          }
69              //qDebug() << "Level now: "<<level;
70      }
71
72  }
```

This function is in charge of the update of the level with respect to the score of the player and evoke other utility functions (if needed). `level_new` will be calculated and compared with current level, then necessary level-up actions will be executed only when current level is different from previous level. The highest level of this game is 5. A player is considered to "win" if he(she) can score above 380 points or beat the boss before the time runs out.

b) `game_logic_control()`

```
1  void GameEngine::game_logic_control()
2  {
3      switch (level) {
4          case 1:{
5          timer_for_enemy->start(5000);
6          break;
7      }
8          case 2:{
9          timer_for_enemy->start(4000);
```

```
        break;
    }
        case 3:{
        VirusC::allowInfect();
        timer_for_enemy->start(3500);
        break;
    }
        case 4:{
        timer_for_enemy->start(3000);
        break;
    }
        case 5:{
        create_boss();
        timer_for_enemy->start(3000);
        break;
    }

    }
}
```

This function is called right after the level up of the player so that the frequency of enemy production can change accordingly.

c) `update_speed_and_potential()` & `_frozen()` & `_shield()`

```
void GameEngine::update_speed_and_potential()
{
    int layer = level-1;
    for(int i=0;i<3;++i)
        player->enemy_parameters[i]=level_array[layer][i];
}
```

When level goes up, the speed of the enemy, attack potential of the virus emitted

by the enemy and frequency of emission need to be altered as well. Hence three 3D arrays `level_array`,`level_array_frozen` and `level_array_shield` are used to record the information and assign the value to the constructor of player and enemy. Three arrays are used instead of one because some props in the game are designed to decrease the attack potential of the enemy or the virus-emission frequency in a short period of time. Once a certain prop is picked, other two functions will be executed to update parameters.

(3) `write_in_ranking_and_finish_game()` If the player wins at the end, this function will be executed to record the player's name and score into the file `Ranking.txt`. Data will be shown in the later ranking.

3 Player

Player class controls the action of the player such as pressing and releasing the key to move and shoot, getting more powerful as level up and display player related graphs on the GUI. The variables requested_ shooting_number, requested_speed and attack_potential are properties of the player and will increase as level up.

```
class Player:public QObject, public QGraphicsPixmapItem{
    Q_OBJECT
public:
    Player(QGraphicsItem * parent=0);
    ~Player();
    void keyPressEvent(QKeyEvent * event);
    void keyReleaseEvent(QKeyEvent * event);
    QPointF get_attack_dest(){
        return QPointF(x(),y());
    }
    void level_up(int level);
    vector<vector<int>>
    ↪  enemy_parameters={{1000,3,10,1},{1000,3,10,1},{1000,3,10,1}};
```

```

public slots:
    void spawn();
private:
    QMediaPlayer * bulletsound;
    int key[5]={0,0,0,0,0}; //Up(W),Down(S),Left(V),Right(D),Space
    void create_bullet_and_sound(int x, int y);
    int xcount=0; //used to change player movement photo
    int ycount=0; //used to change player movement photo
    int ix=0; //used to change player movement photo
    int iy=0; //used to change player movement photo
    int result=0; //used to change player movement photo
    QPixmap player_photo[12]; //used to change player movement photo
    void move();
    Weapon** weapon;
    int requested_shooting_number = 1;
    int requested_speed = 10;
    int attack_potential = 5;
    QTimer* shooting_inertia;
};
```

(1) Functions receiving input from user

```
void Player::keyPressEvent(QKeyEvent *event){
    switch ( event->key() )
        {
        case Qt::Key_W:
            key[0] = 1;
            break;
        case Qt::Key_S:
            key[1] = 1;
```

```
9               break;
10          case Qt::Key_A:
11              key[2] = 1;
12              break;
13          case Qt::Key_D:
14              key[3] = 1;
15              break;
16          case Qt::Key_Space:
17              key[4] = 1;
18              break;
19          }
20          emit move();
21  }
22  void Player::keyReleaseEvent(QKeyEvent *event)
23  {
24      switch ( event->key() )//event->isAutoRepeat()
25          {
26          case Qt::Key_W:
27              key[0] = 0;
28              break;
29          case Qt::Key_S:
30              key[1] = 0;
31              break;
32          case Qt::Key_A:
33              key[2] = 0;
34              break;
35          case Qt::Key_D:
36              key[3] = 0;
37              break;
38          case Qt::Key_Space:
39              key[4] = 0;
```

```
            break;
        }
}
```

Use an array `key[]` to present the status of the keys A,S,W,D and space. When user press and release the keyboard, record the behavior press as 1 and release as 0.

(2) Function change the properties of player according to level

```
void Player::level_up(int level)
{
    requested_shooting_number = level*0.7+1;
    requested_speed = 10 + level*0.6;
    attack_potential = 5 + level;
}
```

(3) Functions control the action of the player

```
void Player::move()
{
    if(key[0]){
        if(pos().y()>=-3)
            setPos(x(),y()-5);
        else
            setPos(x(),-8);
    }
    if(key[1]){
        if(pos().y()<=740)
            setPos(x(),y()+5);
        else
            setPos(x(),750);
    }
```

```
15      if(key[2]){
16          if(pos().x()>=-8)
17              setPos(x()-8,y());
18          else
19              setPos(-8,y());
20      }
21      if(key[3]){
22          if(pos().x()<415)
23              setPos(x()+5,y());
24          else
25              setPos(418,y());
26      }
27      if(key[4]){
28          if(game->is_end()||y()<=-5)
29              return;
30          if(shooting_inertia->remainingTime() > 0)
31              return;
32          shooting_inertia->start(200);
33          shooting_inertia->setSingleShot(true);
34          create_bullet_and_sound(x(),y());
35      }
36
37      if(key[2]==1){
38          if(xcount>0)
39              xcount=0;
40              --xcount;
41      }
42
43      if(key[3]==1){
44          if(xcount<0)
45              xcount=0;
```

```
46              ++xcount;
47      }
48
49      if(key[0]==1){
50          if(ycount>0)
51              ycount=0;
52              --ycount;
53      }
54
55      if(key[1]==1){
56          if(ycount<0)
57              ycount=0;
58              ++ycount;
59      }
60      ix=abs(xcount)%30/10;
61      iy=abs(ycount)%30/10;
62
63      if(xcount>0){
64          ix+=3;
65          result=ix;
66      }else if(xcount<0){
67          ix+=9;
68          result=ix;
69      }
70      if(ycount>0){
71          iy+=6;
72          result=iy;
73      }else if(ycount<0){
74          iy+=0;
75          result=iy;
76      }
```

```
77
78
      ↪  if((key[0]&&key[2])||(key[0]&&key[3])||(key[1]&&key[2])||(key[1]&&key[3]))
79          result=ix;
80      if(key[2]!=1 && key[3]!=1){
81          xcount=0;
82      }
83      if(key[0]!=1 && key[1]!=1){
84          ycount=0;
85      }
86      setPixmap(player_photo[result]);
87  }
```

When user press ASWD, the player will move left, downward, upward and right, unless it is at the boundary. When space is pressed, if the game is not end and the last bullet has been shot for more than 200ms, create bullet(s). The variable result is used to determine the moving direction of the player and show the corresponding graph (will be elaborated in the appendix).

```
1  void Player::create_bullet_and_sound(int x, int y)
2  {
3      double angle = PI;
4          weapon = new Weapon* [requested_shooting_number];
5          if(requested_shooting_number == 1)
6          {
7              weapon[0]=new Weapon(requested_speed,attack_potential,angle);
8              weapon[0]->setPos(x,y);
9              scene()->addItem(weapon[0]);
10         }
11         else {
12             for(int i = 0 ; i < requested_shooting_number ; i++)
```

```
13              {
14                weapon[i]=new Weapon(requested_speed,attack_potential,
15                angle-PI/6+i*PI/(3*(requested_shooting_number-1)));
16                weapon[i]->setPos(x,y);
17                scene()->addItem(weapon[i]);
18              }
19          }
20  }
```

Create weapon(s) according to variable `requested_shooting_number`, `requested_speed` and `attack_potential`. Parameter `x`, `y` are the position to create the bullet.

3.1 Appendix: The logic of changing the player/enemy's photo

Figure. Players heading towards different direction

To make the player and enemy move more vividly, player's gesture and movement can be adjusted in accordance with the input. And concerning functions are mainly implemented in the `move()` function.

As mentioned, twelve photos of the player has been included in the array `player_photo[12]`. The "link" variables between the keyboard input and the photo replacement are `xcount` and `ycount`. If the player moves to the right, then `xcount` should be positive and its value increases 1 once right press is detected, and vice versa. If none of the keys are pressed, both values will set to 0.

```
    if(key[2]==1){
        if(xcount>0)
            xcount=0;
            --xcount;
    }
    if(key[3]==1){
        if(xcount<0)
            xcount=0;
            ++xcount;
    }
    if(key[0]==1){
        if(ycount>0)
            ycount=0;
            --ycount;
    }
    if(key[1]==1){
        if(ycount<0)
            ycount=0;
            ++ycount;
    }
    // Case when no key in x/y direction is pressed
    if(key[2]!=1 && key[3]!=1){
        xcount=0;
    }
    if(key[0]!=1 && key[1]!=1){
        ycount=0;
    }
```

In addition, the player would be rather rigid if it stands still during moving, so `ix` and `iy` are used to create periodically "replacement" (by using modulus arithmetic) of the photo to show that the player is "walking".

```
    ix=abs(xcount)%30/10;
    iy=abs(ycount)%30/10;

    if(xcount>0){
        ix+=3;
        result=ix;
    }else if(xcount<0){
        ix+=9;
        result=ix;
    }
    if(ycount>0){
        iy+=6;
        result=iy;
    }else if(ycount<0){
        iy+=0;
        result=iy;
    }
```

It's also worthwhile to note that 0,3,6,9 are the numbers to "lift" the `ix` /`iy` to the given row of the photo (please note that the array index is numbered from first row to last row, from leftmost column to rightmost column in a given row). The sign of `xcount` /`ycount` is used to determine the selection of the photo.

```
    if((key[0]&&key[2])||(key[0]&&key[3])||(key[1]&&key[2])||(key[1]&&key[3]))
        result=ix;
```

This line is to make sure the hierarchy between directions such that the player will go "right" if it moves right and up simultaneously.

4 Enemy (Base class)

```
/* Header File for Enemy except include part */
class Enemy: public QObject,public QGraphicsPixmapItem{
    Q_OBJECT
public:
    Enemy(int period , int requested_potential , int requested_speed, int
    ↪  requested_shooting_number,QGraphicsItem * parent=nullptr);
    ~Enemy();
private slots:
    void move();
    virtual void shoot() = 0;
    void handle_move();
    void move_right();
    void move_left();

public slots:
    void get_attack(int attack);
protected:
    Health* health;
    const int SIZEW=100;
    const int SIZEH=100;
    int xcount=0; //used to change enemy's movement photo
    int ycount=0; //used to change enemy's movement photo
    int ix=0; //used to change enemy's movement photo
    int iy=0; //used to change enemy's movement photo
    int result=0; //used to change enemy's movement photo
    int get_prop; //Random number generated to see if a prop can be gotten
    QTimer* timer;
    int period = 1000;
    int requested_potential = 3;
```

```
29      int requested_speed = 10;
30      int requested_shooting_number = 1;
31      virtual void update_result();
32  };
```

There are three types of enemies–Anti, Devil and WildAnimal. They bare very similar properties so most of their data members and member functions are in their parent class, i.e. Enemy.

`period`, `requested_potential`, `requested_speed` and `requested_shooting_number` represent their shooting period, health, moving speed and number of shots of each time of shooting respectively, and they vary from different kinds of enemies and will change as game status changes(e.g. level goes up or player picks up props).

`move()`, `move_right()` and `move_left()` are the actual functions that make the enemy move down, right and left,

```
1  void Enemy::move(){
2      // move enemy down
3      setPos(x(),y()+2);
4
5      if(ycount<0)
6          ycount=0;
7      ++ycount;
8
9      // destroy enemy when it goes out of the screen
10     if (pos().y() > 800){
11         //decrease the health
12         game->health->decrease(2);
13
14         scene()->removeItem(this);
15         delete this;
16         return;
17     }
```

```
    update_result();
}
```

while handle_move() is the function that decides which direction should the enemy move and calls the three above functions under certain circumstances.

```
void Enemy::handle_move(){
    xcount=0;
    ycount=0;
    ix=0;
    iy=0;
    result=0;
    QTimer * timer_for_move_inertia = new QTimer(this);
    timer_for_move_inertia->setSingleShot(1);
    timer_for_move_inertia->start(999);
    QTimer * timer_for_move = new QTimer(this);
    timer_for_move->start(50);


    ↪  connect(timer_for_move_inertia,SIGNAL(timeout()),timer_for_move,SLOT(stop()));

    int random_number = rand() % 100;
    if(random_number <= 60){
    connect(timer_for_move,SIGNAL(timeout()),this,SLOT(move()));}
    else if(random_number > 60 && random_number <= 80)
    {connect(timer_for_move,SIGNAL(timeout()),this,SLOT(move_left()));}
    else if(random_number > 80)
    {connect(timer_for_move,SIGNAL(timeout()),this,SLOT(move_right()));}
}
```

void update_result() is the helper function that handle the direction of image(or which image

to display) after moving, so the enemy's image will always looks like facing the direction that it is moving to. int result represents the index of the image arrays which are defined in child classes. This function is overriden by the ones in its child classes since different types of enemies has different sets of images.

```
void Enemy::update_result()
{
    ix=abs(xcount)%30/10;
    iy=abs(ycount)%30/10;

    if(xcount>0){
        ix+=3;
        result=ix;
    }else if(xcount<0){
        ix+=9;
        result=ix;
    }

    if(ycount>0){
        iy+=6;
        result=iy;
    }else if(ycount<0){
        iy+=0;
        result=iy;
    }
}
```

`void get_attack(int attack)` is the function that receives damage when the enemy is hit by player's weapon, and it will be called in Weapon's function `void attack()`. It also generates props after it dies at its dying position. The props are generated randomly with slightly different opportunities. The total opportunity that a prop will be generate after the enemy died is 50%. There are four types of props which will be introduced later.

```
void Enemy::get_attack(int attack)
{
    health->decrease(attack);
    if(health->getHealth() <= 0)
    {
        game->score->increase();
        if(get_prop > 50)
        {
            if(get_prop>=88)
            {
                BloodBag* bloodbag = new BloodBag(10);
                scene()->addItem(bloodbag);
                bloodbag->setPos(pos().x(),pos().y());
            }
            else if(get_prop>=76)
            {
                Freezer* freezer = new Freezer();
                scene()->addItem(freezer);
                freezer->setPos(pos().x(),pos().y());
            }
            else if(get_prop>=64)
            {
                Mask* mask = new Mask();
                scene()->addItem(mask);
                mask->setPos(pos().x(),pos().y());
            }
            else if(get_prop>=50)
            {
                Medal* medal = new Medal(10);
                scene()->addItem(medal);
                medal->setPos(pos().x(),pos().y());
```

```
            }
        }
        scene()->removeItem(this);
        game->enemy_num++;
        delete this;
        return;
    }
}
```

4.1 Anti (Derived class)

```
/* Header File for Anti except include part */
class Anti : public Enemy
{
public:
    Anti(int period = 1000, int requested_potential = 3 , int requested_speed =
    ↪  10, int requested_shooting_number = 1);

private slots:
    void shoot() override;
    void update_result() override;

private:
    QPixmap anti_photo[12]; //used to change enemy's movement photo
    VirusA** virus;
};
```

Anti stands for antisocialist and it generates A type viruses.

`void shoot()` is the function that handles shooting and is connected to the timer. It first calculates the angle between the Anti itself and the player, then use this angle to generate A type virus/viruses.

The number of virus generated depends on the requested_sooting_number which will increase from 1 to 2 or 3 as level goes up and might decrease if the enemy is freezed, i.e. when player pick up a freezer.

```
1  void Anti::shoot()
2  {
3      if(game->is_end())
4          return;
5
6      QLineF ln(QPointF(x(),y()),game->player->get_attack_dest());
7      qreal angle=-::acos(ln.dx()/ln.length())+PI/2;
8      if(ln.dy()<0)
9          angle=PI-angle;
10
11     virus = new VirusA* [requested_shooting_number];
12     if(requested_shooting_number == 1)
13     {
14         virus[0]=new VirusA(requested_speed,requested_potential,angle);
15         virus[0]->setPos(x(),y());
16         scene()->addItem(virus[0]);
17     }
18     else {
19         for(int i = 0 ; i < requested_shooting_number ; i++)
20         {
21           virus[i]=new
             ↪  VirusA(requested_speed,requested_potential,angle-PI/6+i*PI/(3*(requested_shootin
22           virus[i]->setPos(x(),y());
23           scene()->addItem(virus[i]);
24         }
25     }
26 }
```

The function `void update_result()` which overrides the one in parent class just simply calls the one in parent class and then set the image as the result.

```
void Anti::update_result()
{
    Enemy::update_result();
    setPixmap(anti_photo[result]);
}
```

The other two types of enemies–Devil and WildAnimal are similar to Anti so will not be further explained here. The few differences are that Devil generates B type viruses while WildAnimal generates C type viruses. And these three kinds of enemies have different parameters(e.g. speed, requested_potential) and images.

5 Boss

The boss is the enemy that the player "must defeat" in order to win the game.

```
class Boss:public QObject, public QGraphicsPixmapItem
{
    Q_OBJECT
    static bool boss_is_killed;
 public:
    static bool bossIsKilled(){return boss_is_killed;}
    Boss();
    ~Boss();
    void get_attack(int attack);
private:
    int time = 0;
    int count_cycle;
    void Appear();
```

```
    void Move1();
    void Move2();
    QTimer* shoot_timer;
    double x,y,prev_x,prev_y;
    QPixmap boss_image;
    Health* health;
    int move_pattern =0;
    BossVirus* shot;
    int angle,raise;
    int raise2;
    int width;
    int height;
    double movex,movey;
    double virus_angle;
private slots:
    void Move();
    void Shoot_Barrage();
    void Shoot_Circle();
};
```

Compared with ordinary enemies, the boss has more patterns of moving and shooting. Its bullet has higher attack potential and different image. Move pattern is the indicator of the move state of the boss, and the movement of the boss is controlled by the following member function:

```
void Boss::Move()
{
    switch(move_pattern){
        case 0:
            Appear();
            break;
        case 1:
```

```
            Move1();
            break;
        case 2:
            Move2();
            break;
    }
    setPos(x,y);

}
```

The following function handles the way that the boss appears, `sin` function is used to make the movement more smoothly.

```
void Boss::Appear()
{
    double temp;
    angle+=2;
    temp=sin(angle*PI/180);
    y = prev_y+temp*movey;
    if(angle==90){
        move_pattern=1;
    }
}
```

After the boss moves into the place, Move1() will be activated:

```
void Boss::Move1()
{
    angle+=raise;
    y = prev_y+sin(90*PI/180)*movey-sin(1.5*(angle-90)*PI/180)*70;
    if(angle % 360==90){
```

```
        raise=-2;
        angle = 90;
        count_cycle=count_cycle+1;
    }else if(angle % 360==-90 || angle % 360==270){
        raise=2;
    }
    if(count_cycle == 1)
    {move_pattern=2;}
}
```

The boss will "swing" back and forth (as the function of `y` suggests) in the screen as the angle increases, Move2() will be executed once the angle reaches 450 degrees (`angle` is reset to 90 in order to prevent the integer being too large because only the remainder matters).

```
void Boss::Move2()
{
    angle+=2;
    x = prev_x+50*cos(3*angle*PI/180);
}
```

In this stage, the boss will move in a "curved" way.

When the boss is moving in above ways, it's also shooting simultaneously. The shooting pattern of the boss can be divided into two genres.

```
void Boss::Shoot_Barrage()
{
    if(game->is_end())
        return;
    BossVirus* shoot = new
    ↪ BossVirus(x+width/2,y+height/2,atan((x-game->player->x())/(y-game->player->y()))+
    (rand() % 5)*PI/180);
```

```
    game->scene->addItem(shoot);
    time+=1;
    if(time % 500 == 10*5)
    {
        shoot_timer->stop();
        disconnect(shoot_timer,SIGNAL(timeout()),this,SLOT(Shoot_Barrage()));
        connect(shoot_timer,SIGNAL(timeout()),this,SLOT(Shoot_Circle()));
        shoot_timer->start(100);
    }

}
```

Shoot_Barrage() produces a barrage of bullets heading towards the player with random discrepancy ranging from ±5 degrees. It's worthy to note that the shooting mode will change to Shoot_Circle() after a given period of time:

```
if(game->is_end())
        return;
    virus_angle+=30;
    for(int i=0;i<5;i++)
    {
        BossVirus* shoot = new
        ↪  BossVirus(x+width/2,y+height/2,virus_angle+i*2*PI/5);
        game->scene->addItem(shoot);
    }

    time+=1;
    if(time % 500== 10*40)
    {
        shoot_timer->stop();
        disconnect(shoot_timer,SIGNAL(timeout()),this,SLOT(Shoot_Circle()));
```

```
15          connect(shoot_timer,SIGNAL(timeout()),this,SLOT(Shoot_Barrage()));
16          shoot_timer->start(100);
17      }
```

For `Shoot_Circle()`, bullets will be divided into five branches and spread out. It's worthy to note that the shooting mode will change to `Shoot_Barrage()` after a given period of time. So a circulation of shooting between `Shoot_Barrage()` and `Shoot_Circle()` is formed.

6 Bullet (Base class)

```
1  /* Header File for Bullet except include part */
2  class Bullet: public QObject,public QGraphicsPixmapItem {
3      Q_OBJECT
4  public:
5      enum class Type{WEAPON,VIRUS};
6  private:
7      Type type;
8      const int BULLET_BOUNDS[4]={-8,450,0,750};
9      QTimer* timer;
10 protected:
11     void connect_n_start();
12 public:
13     Bullet(){};
14     Bullet(Type type,QGraphicsItem * parent=0);
15     ~Bullet(){delete timer;};
16 private slots:
17     virtual void move()=0;
18     virtual bool check_collide()=0;
19     virtual void attack()=0;
```

```
20 public slots:
21     void check_out();
22 };
```

Since Bullet is the parent class for Weapon and Virus, it has an `enum class Type{WEAPON,VIRUS}`. The `BULLET_BOUNDS[4]` array is a const int array that contains the upper/lower/left/right bounds of bullets, which are used in `check_out()`.

`move()`, `check_collide()` and `attack()` are pure virtual functions since they are a bit different in Weapon class and Virus class. This makes Bullet an ABC.

The functions `connect_n_start()` and `check_out()` are the same for all the child classes of Bullet. The former connects `move()` as well as `attack()` to the timeout() signal separately and starts the timer, while the latter checks whether the bullet goes out of bound and remove it if it does.

6.1 Virus (Base class & Derived class)

```
1  /* Header File for Virus except include part */
2  class Virus: public Bullet{
3  public:
4      enum class VType{A,B,C,BOSS};
5      Virus(){};
6      Virus(VType vtype,int speed,int attack_potential,qreal angle);
7      ~Virus(){};
8      int get_attack_potential();
9      qreal get_angle(){return angle;}
10 protected:
11     const int WEITH{25};
12     const int HEIGHT{30};
13     int speed{0};
14     int attack_potential{0};
15     qreal angle{0};
```

```
16      const int xdir{0};
17      const int ydir{0};
18      VType vtype;
19  protected slots:
20      virtual void move() override;
21      virtual bool check_collide() override;
22      virtual void attack() override;
23  };
```

Since Virus is the parent class for four types of virus(A,B,C,BOSS), it has an `enum class VType{A,B,C,BOSS}`. `WEITH` and `HEIGHT` are used to scale the size of the image of the virus, we make it const so that we could change it more easily.

`speed` and `attack_potential` are inner properties of virus and are initialized to 0 by default to play safe. speed is the speed that the virus moves and attack_potential is the amount of damage player will get once been hit by the virus. Different types of virus has different speed and attack_potential. We want the game to be challengeable, so instead of shooting forward, we shoot the virus in the direction of the player. Thus we use `angle` to keep the angle between the enemy and player at the moment we shoot and use it to calculate `xdir` and `ydir`, which are the x and y components of the velocity of the virus respectively.

The function `move()` handles the movement of virus and remove it when it goes out of bound.(The move function for bossvirus is a little bit different from this.)

```
1  void Virus::move(){
2      setPos(x()+xdir,y()+ydir);
3      Virus::check_out();
4  }
```

The function `attack()` first check whether the virus collides with the player by calling `check_collide()` mentioned above, if it does, it gets its attack_potential by calling `get_attack_potential()` since attack_potential are private in VirusA/B/C/BOSS, then do the actual damage by decreasing health directly.

```
void Virus::attack(){
    if(check_collide()){
        int damage=get_attack_potential();
        game->health->decrease(damage);
        delete this;
    }
}
```

The function `check_collide()` checks whether the virus collides with the player and return a bool value.

```
bool Virus::check_collide()
{
    // get a list of all the items currently colliding with this bullet
    QList<QGraphicsItem *> colliding_items = collidingItems();

    // if it collide with the player, return true
    for (int i = 0, n = colliding_items.size(); i<n; ++i){
        auto &obj=*colliding_items[i];
        if (typeid(obj) == typeid(Player)){
            return true;
        }
    }

    return false;
}
```

6.1.1 VirusC (Derived class)

```
/* Header File for VirusC except include part */
class VirusC:public Virus
{
    Q_OBJECT
private:
    static bool allow_infect;
public:
    VirusC(int requested_speed,int requested_potential,qreal angle);
    ~VirusC(){};
    static void allowInfect(){
        allow_infect=true;
    };
private slots:
    void infect();
};
```

Since Viruses are similar, we only introduce VirusC here.

VirusC is the strongest type of virus, i.e., it has the greatest speed as well as attack_potential. It also has a special feature called `infect()` which does not exist in VirusA/B.

`bool allow_infect` and `void allowInfect()` should work for all C type virus so they are set to be static.

When the player reach level 3, the function `void allowInfect()` willed be called and set `bool allow_infect` to be true.

`void infect()` is connected to a timer in VirusC's constructor. What it does is to first check whether infect id allowed and then check whether it collides with a VirusA/B. If it does, the virus collides with it will be infected and become type C. This is done by killing the colliding virus and generate another type C virus at the same place with the same angle. There are 50% chances that a C type virus will infect another virus.

```
void VirusC::infect(){
    if(allow_infect){
        int prob=rand()%100;
        if(prob<50)
            return;
        // get a list of all the items currently colliding with this bullet
        QList<QGraphicsItem *> colliding_items = collidingItems();
        for (int i = 0, n = colliding_items.size(); i<n; ++i){
            auto &obj=*colliding_items[i];
            if (typeid(obj) == typeid(VirusA)||typeid(obj) == typeid(VirusB)){
                Virus* v=dynamic_cast<Virus*>(colliding_items[i]);
                VirusC*virus=new VirusC(5,3,v->get_angle());
                virus->setPos(colliding_items[i]->x(),colliding_items[i]->y());
                scene()->addItem(virus);
                scene()->removeItem(colliding_items[i]);
                delete colliding_items[i];
            }
        }
    }
}
```

6.2 Weapon

Weapon is the derived class of Bullet. Its variables weight and height are used to scale the size of its image so it is made constant to avoid unneeded change. Speed is the speed that the weapon moves and attack potential is the amount of damage enemy will received. `Xdir` and `ydir` are variables that help to control the direction of weapons.

```
class Weapon: public Bullet
{
private:
```

```
    const int WEITH{25};
    const int HEIGHT{30};
    int speed{10};
    int attack_potential{0};
    bool used = false;
    qreal angle{0};
    int xdir{0};
    int ydir{0};
public:
    //Weapon();
    Weapon(int speed,int attack_potential,qreal angle);
    ~Weapon(){};
private slots:
    virtual void move() override;
    virtual bool check_collide() override;
    virtual void attack() override;
};
```

(1) Function controls the movement of weapon

```
void Weapon::move(){
    setPos(x()+xdir,y()+ydir);
    if (pos().x()<-8||pos().x()>410)
    {
       xdir = -xdir;
    }
    if (pos().y() < 0||pos().y()>750)
    {
       scene()->removeItem(this);
       delete this;
    }
```

```
12 }
```

The function `move()` controls the movement of weapons. When it touches the left or right bound, it will bounce back and when touch upper or lower bound, it will be removed.

```
1 bool Weapon::check_collide(){return false;}
```

The check collide procedure is done in `attack()` since the implementation is different for different collide items, so it always returns false here.

(4) `attack()` (the code is a bit redundant so it's neglected here) The function `attack()`first check whether the weapon has been used since a weapon will be removed when it collide with an enemy. Then it will get a list of all the items currently colliding with the weapon, if there is an enemy, set the variable used to be true, remove the weapon, increase the score and damage will caused to the enemy according to the attack potential of weapon.

7 Props (Bass Class)

There are three types of props in our game: Freezer(disinfector), BloodBag(pills) and Medal. They are inherited from the base class `Props` and randomly generated after the death of the enemy. If the player hasn't picked up the props after 7.5 seconds, props will automatically disappear. The specific function of each prop is elaborated in the following.

Figure. Props (from left to right: Freezer, BloodBag, Medal, Mask)

7.1 Freezer (Alias: disinfector)

```
1 void Freezer::freeze()
2 {
```

```
    freeze_timer->start(10*1000);
    game->update_speed_and_potential_frozen();
}
```

As explained in GameEngine's section, update_speed_and_potential_frozen() is used to when the Freezer is obtained. freeze_timer is used to control the effective time of the prop to be 10 seconds. When the time is running out, finish() function in the base class will be evoked.

```
void props::finish()
{
    game->update_speed_and_potential();
    delete this;
}
```

7.2 BloodBag (Alias: pills)

The BloodBag is used to heal the player by a given amount add and display the message on the message box.

```
void BloodBag::add_health()
{
   game->ui->textBrowser->setTextColor(QColor(200,15,122));
   game->ui->textBrowser->QTextEdit::append(QString("Pills obtained with health:
   ↪  ") + QString::number(add));
   game->health->increase(add);
}
```

7.3 Medal

The Medal is used to add the score of the player by a given amount add and display the message on the message box.

```
void Medal::add_score()
{
    game->ui->textBrowser->setTextColor(QColor(122,122,122));
    game->ui->textBrowser->QTextEdit::append(QString("Medal obtained with score
    ↪  bonus:") + QString::number(add));
    for(int i = 0; i < add; i++)
    game->score->increase();
}
```

7.4 Mask (Alias: shield)

As explained in GameEngine's section, update_speed_and_potential_shield() is used to when the Mask is obtained.

```
   void Mask::shield()
{
    shield_timer->start(10*1000);
    game->update_speed_and_potential_shield();
}
```

8 GUI at the end of the game

A player is considered to "win" if he(she) can score above 380 points or beat the boss before the time runs out. When the time runs out or the player's health decreases to zero, the game will automatically end. If the player wins, a `Win` window will be shown with the score and enemies

eliminated. Then the program will ask the player to choose a `.txt` file to write in his data and update the ranking. If the player loses, there will be a `Lose` window showing the score, highest level and enemies eliminated.

8.1 Win

```
/* Header File for Win except include part */
namespace Ui {
class Win;
}

class Win : public QDialog
{
    Q_OBJECT
public:
    explicit Win(int score,int num_enemies,QWidget *parent = nullptr);
    void display_score();
    void display_enemy_num();
    ~Win();
    int score;
    int num_enemies;
private slots:
    void on_OK_clicked();
private:
    Ui::Win *ui;
};
```

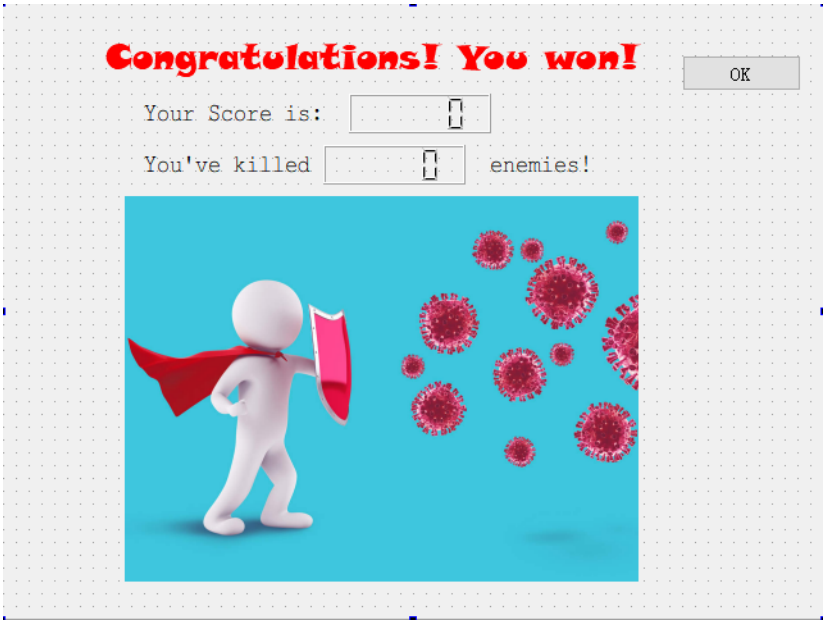


Figure. The GUI surface after winning the game

After the game has ended and given the fact that the player has won the game, a new GUI surface will be created. One can see that `score` and `num_enemies` are passed to the constructor of the `Win`. `display_score()` and `display_enemy_num()` are the corresponding handle functions to display the scores on the GUI surface.

The function `on_OK_clicked()` is used to manage the appearance of the window `Win`. The logic is that after OK is clicked, `Win` window will be appeared.

8.2 Lose

```
1 /* Header File for Lose except include part */
2 namespace Ui {
3 class Lose;
4 }
5
6 class Lose : public QDialog
7 {
8     Q_OBJECT
```

```
9  public:
10     explicit Lose(int score,int num_enemies,int level,QWidget *parent =
       ↪  nullptr);
11     ~Lose();
12 private slots:
13     void on_OK_clicked();
14 private:
15     Ui::Lose *ui;
16     int score;
17     int num_enemies;
18     int level;
19     void display_score();
20     void display_enemy_num();
21     void display_level();
22 };
```

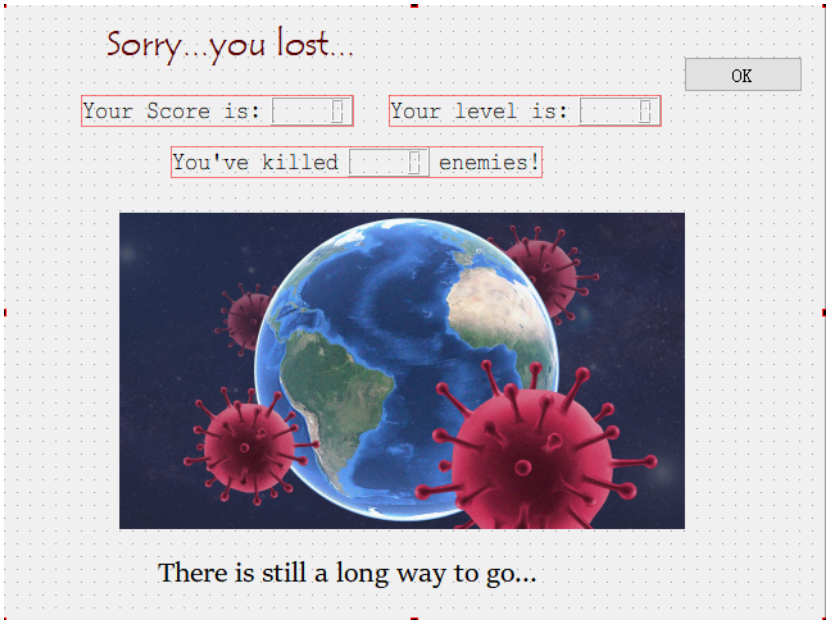


Figure. The GUI surface after losing the game

As one can see, the GUI surface and logic are similar to those of `Win`. Only `level` and corresponding display function `display_level()` is added.

8.3 Ranking

```
/* Header File for Ranking except include part */
namespace Ui {
class Ranking;
}
class Ranking : public QDialog
{
    Q_OBJECT
public:
    explicit Ranking(QString = NULL,QWidget *parent = nullptr);
    ~Ranking();
    void Read_the_file();
    void Set_the_table();
private slots:
    void on_Find_clicked();
    void on_name_requested_textChanged(const QString &arg1);
private:
    Ui::Ranking *ui;
    int player_num;
    QString filename;
    QString name_tentative;
    Hashtable hash;
    QList<string> names;
    QList<int> scores;
};
```

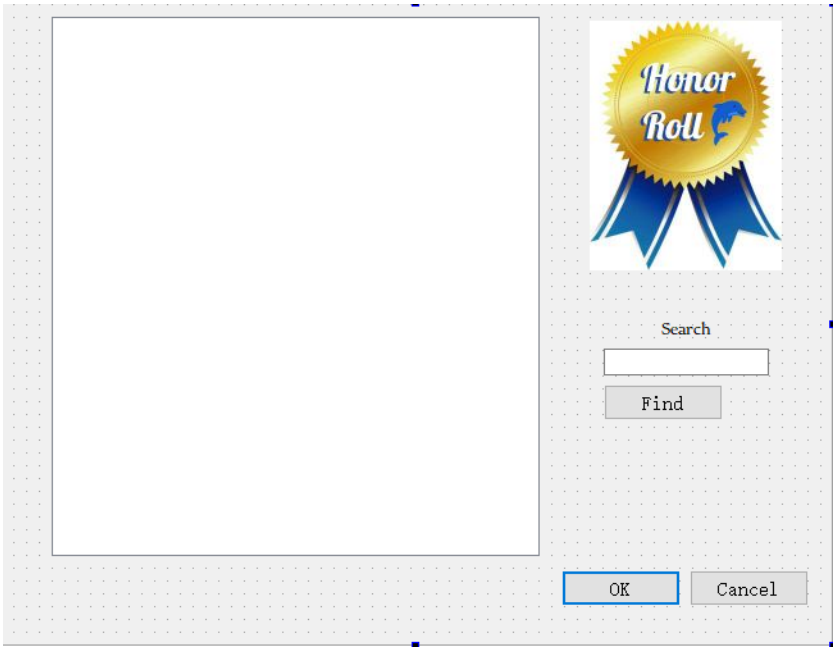


Figure. The GUI surface of ranking

As the class name suggests, this class deals with the retrieval and recording of the ranking and score information of the player(s). Because the ranking will be presented in the form of a `QDialog` in a new window, the class `Ranking` is inherited from `QDialog` and has a `ui` data member in charge of the GUI.

The whole process can be divided into a few stages:

(1) The program reads data from `Ranking.txt` in the file and record the names and scores of different players in two parallel `QLists`: `names` and `scores` (`QLists` are used instead of traditional arrays because its `append` function is of great convenience when dealing with a set of data with unknown size). The variable `player_num` is used to record the total number of players Then a nested `for` loop is executed to rearrange the score and player in the sequence of decreasing:

```
for (int i = 0; i < player_num; i++) {
        for (int j = i+1; j < player_num; j++) {
            if (scores[i] < scores[j]) {
                int temp;
                temp = scores[i];
```

```
                scores[i] = scores[j];
                scores[j] = temp;
                string temp2;
                temp2 = names[i];
                names[i] = names[j];
                names[j] = temp2;
            }
        }
    }
```

(2) To make the searching more efficient, a hashtable `hash` is used to store the names and corresponding index after rearrangement.

```
for(int i = 0; i < player_num; ++i)
        hash.add(names[i], i);
```

(3) Display the names and corresponding scores on the GUI

```
for (int ridx = 0 ; ridx < player_num ; ridx++ )
    {
          QTableWidgetItem* item1 = new QTableWidgetItem();
          item1->setText(QString::fromStdString(names[ridx]));
          ui->ranking->setItem(ridx,0,item1);

          QTableWidgetItem* item2 = new QTableWidgetItem();
          item2->setText(QString::number(scores[ridx]));
          ui->ranking->setItem(ridx,1,item2);
    }
```

(4) If "Find" button is clicked (with names typed in), the program will record the user input in the `QLineEdit` to the variable `name_tentative`. Then `on_Find_clicked()` will be called and

```
ui->ranking->selectRow(hash.FindScore(name_tentative.toStdString()));
```

will be executed to find and display the row of the wanted player.

Our appreciation goes to everyone who tried our game and gave valuable player feedback to help us improve it.

bgm used:

infinite–Lensko

Legends Never Die–Against the Current

Sprint–Yonetro/Seum Dero