Underground 游戏设计分报告

姓名：覃捷

学号：3170101281

指导老师：袁昕

1. 分工任务及解决方案

1.1 分工任务

本人负责的分工任务如下：

（1） 迷宫随机生成算法的实现和迷宫的显示

（2） 人物与怪物的显示，移动和碰撞检测

（3） 迷宫中圆形视野显示和刷新

1.2 解决方案和设计思路

（1）迷宫生成和显示

迷宫的生成是本迷宫游戏的重要内容。一个好的迷宫应该是随机的，复杂的，并且迷宫中通路和围墙的比例应该比较均匀。查找了一些资料后，我决定使用随机 Prim 算法实现迷宫的随机生成。

Prim 算法是经典的计算最小生成树算法。Prim 算法从点的方面考虑构建一颗最小生成树，大致思想是：设图 G 顶点集合为 U，首先任意选择图 G 中的一点作为起始点 a，将该点加入集合 V，再从集合 U-V 中找到另一点 b 使得点 b 到 V 中任意一点的权值最小，此时将 b 点也加入集合 V；以此类推，现在的集合 V={a，b}，再从集合 U-V 中找到另一点 c 使得点 c 到 V 中任意一点的权值最小，此时将 c 点加入集合 V，直至所有顶点全部被加入 V，此时就构建出了一颗 MST。因为有 N 个顶点，所以该 MST 就有 N-1 条边，每一次向集合 V 中加入一个点，就意味着找到一条最小生成树的边。

而在本游戏中，Prim 算法可以用来生成迷宫。我们令初始时迷宫中的每个单元格都为墙。随机选择一个单元格作为墙的通路，然后把该单元格的所有邻墙加入列表。当列表中还有墙时，从列表里随机选一个墙。如果这面墙分隔的两个单元格只有一个单元格被访问过，那就从列表里移除这面墙，即把墙打通，让未访问的单元格成为迷宫的通路，否则把这个格子的墙加入列表。如果墙两面的单元格都已经被访问过，那就从列表里移除这面墙。当列表中没有墙时，算法终止，迷宫生成完毕。

生成迷宫的代码如下：

```
void Sequence(Coordinated *Arry, int n, int &sum)   //把元素从数组删除
{
    int i;
    for (i = n; i < sum; i++)
    {
        Arry[i] = Arry[i + 1];
    }
    sum--;
}
```

```
void Store(Coordinated *map_1, Coordinate *map_2, int map, int &k,
int n)     //把元素储存进数组
{
    if (((map + 2) % n != 0)&&map_2[map + 1].Judgemet_2 == 0)
    {
        map_1[k].adress = map + 1;
        map_2[map_1[k].adress].Judgemet_2 = 1;
        k++;
    }
    if (((map - 1) % n != 0) && map_2[map - 1].Judgemet_2 == 0)
    {
        map_1[k].adress = map - 1;
        map_2[map_1[k].adress].Judgemet_2 = 1;
        k++;
    }
    if (((map - n) / n != 0) && map_2[map - n].Judgemet_2 == 0)
    {
        map_1[k].adress = map - n;
        map_2[map_1[k].adress].Judgemet_2 = 1;
        k++;
    }
    if (((map + n) / n != n - 1) && map_2[map + n].Judgemet_2 == 0)
    {
        map_1[k].adress = map + n;
        map_2[map_1[k].adress].Judgemet_2 = 1;
        k++;
    }
}

void PrimeMap(Coordinate *map, int n)
{
    Coordinated *map2 = new Coordinated[(n*n-3*n-2)/2];
    srand((unsigned)time(0));

    int k = 2;
    map2[0].adress = n + 2;
    map2[1].adress = 2 * n + 1;
    map[n + 2].Judgemet_2 = 1;
    map[2 * n + 1].Judgemet_2 = 1;

    int b;
    while(k!=0)
    {
        b = rand() % k;
```

```
        if ((map2[b].adress / n) % 2 == 0)
            if (map[map2[b].adress + n].Judgment + map[map2[b].adress -
n].Judgment > 0)
            {
                map[map2[b].adress + n].Judgment = 0;
                map[map2[b].adress].Judgment = 0;
                map[map2[b].adress - n].Judgment = 0;
                Store(map2, map, map2[b].adress + n, k, n);
                Store(map2, map, map2[b].adress - n, k, n);
                Sequence(map2, b, k);
                continue;
            }
            else
            {
                Sequence(map2, b, k);
                continue;
            }
        if ((map2[b].adress / n) % 2 == 1)
            if (map[map2[b].adress + 1].Judgment + map[map2[b].adress -
1].Judgment > 0)
            {
                map[map2[b].adress + 1].Judgment = 0;
                map[map2[b].adress].Judgment = 0;
                map[map2[b].adress - 1].Judgment = 0;
                Store(map2, map, map2[b].adress + 1, k, n);
                Store(map2, map, map2[b].adress - 1, k, n);
                Sequence(map2, b, k);
                continue;
            }
            else
            {
                Sequence(map2, b, k);
                continue;
            }
    }
     delete []map2;
}
```

完成迷宫的生成后，我们在 Qt 的主窗口上使用 Qt 自带的绘制函数进行绘制。迷宫生成后保留一个二维数组用来保存迷宫中各个单元格的属性信息，并且从通路单元格中随机挑选两个单元格作为出口和入口。使用 Qt 自带的绘制方块的函数进行绘制，并加载迷宫墙和通路的对应图片进行显示。绘制迷宫的代码如下：

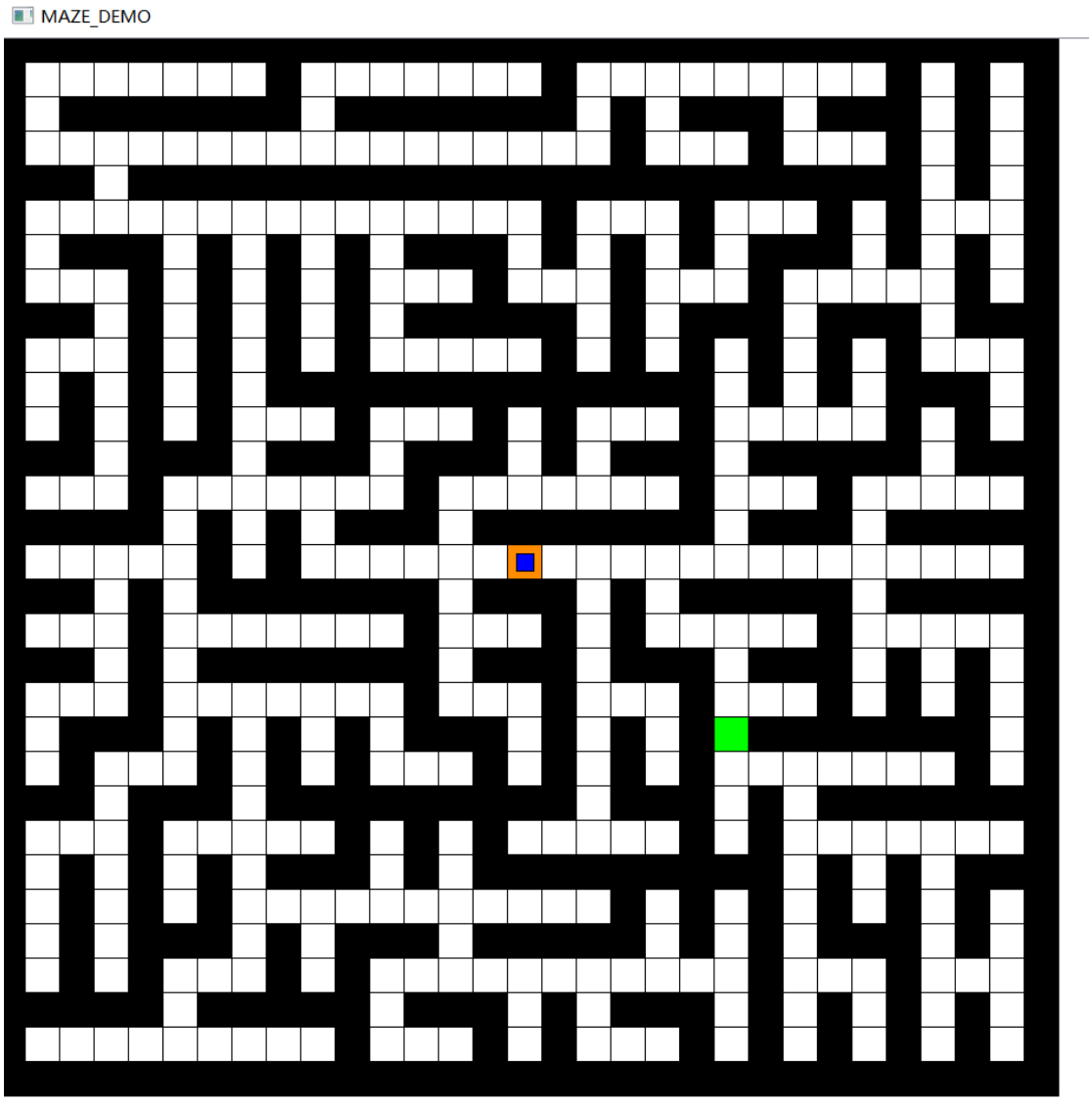
```
void game_control::paint(QPainter *painter,
          const QStyleOptionGraphicsItem *option,
          QWidget *widget)
```

```
{
    Q_UNUSED(option);  //标明该参数没有使用
    Q_UNUSED(widget);
    int central_x=(int)maze->get_man().get_x()/BLOCK_SIZE;
    int central_y=(int)maze->get_man().get_y()/BLOCK_SIZE;
    QColor black(0,0,0);
    QColor white(255,255,255);
    QColor lime(0,255,0);
    QPixmap pix;
    painter->setBrush(Qt::black);
    painter->drawRect(-WINDOW_SIZE/2,-
WINDOW_SIZE/2,WINDOW_SIZE*1.2,WINDOW_SIZE*1.2);
    for(int i=central_y-PAINT_BLOCK_NUMBER-
1;i<central_y+PAINT_BLOCK_NUMBER+1;i++){
        for(int j=central_x-PAINT_BLOCK_NUMBER-
1;j<central_x+PAINT_BLOCK_NUMBER+1;j++){

if(i>=0&&i<maze->get_col_size()&&j>=0&&j<maze->get_row_size()){
              switch (maze->get_block(i,j).get_type()) {
              case 0:pix.load(":/new/image/wall_1.png");break;
              case 1:pix.load(":/new/image/wall_dark.png");break;
              case 2: pix.load(":/new/image/exit_2.png");break;
              case 3: pix.load(":/new/image/enter_2.png");break;
              }

painter->drawPixmap(j*maze->get_block(i,j).get_block_size()-
maze->get_man().get_x(),i*maze->get_block(i,j).get_block_size()-
maze->get_man().get_y(),maze->get_block(i,j).get_block_size(),maze->g
et_block(i,j).get_block_size(),pix);
          }
       }
    }
}
```

初步显示效果图如下:

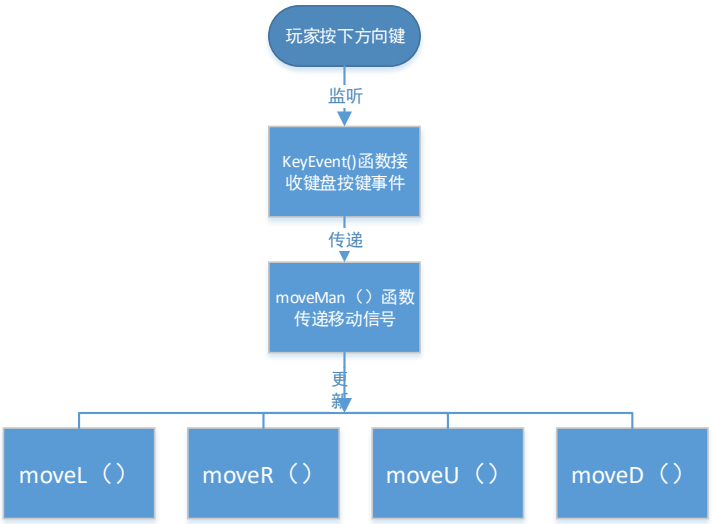


可以看到使用随机 Prim 算法生成的迷宫复杂，自然并且比较有挑战性，符合我们的需求。

（2）人物的移动和显示

为了实现人物的移动，我们需要接收玩家键盘的输入。Qt 中自带了 *KeyEvent（）* 函数可以监听玩家的键盘输入，我们只需要将键盘输入转换为人物坐标变换的信号即可。我们规定玩家通过键盘上的方向键控制人物的左右移动，每次监听到方向键对应按键信息就触发一个槽函数，更新人物的坐标信息，在下一次窗口刷新时显示新的位置。

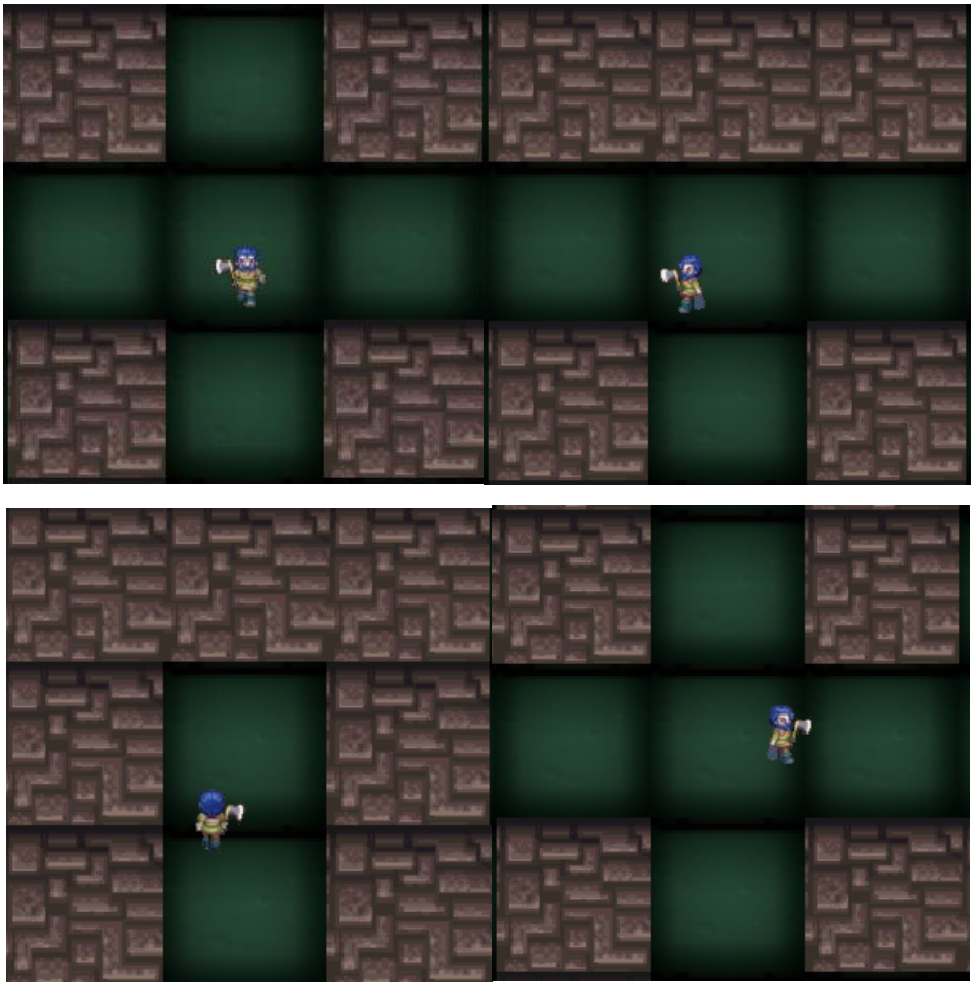
我在 *player* 类中定义了玩家的坐标信息 *x* 和 *y*，并且定义了四个用于更新坐标的函数 *moveL（）*，*moveR（）*，*moveU（）*，*moveD（）* 分别对应左右上下移动，分别更新四个方向上的坐标。这四个函数会被上层的 *game_control* 类的 *moveMan（）* 函数调用，而 *view* 层的 *KeyEvent（）则调用 moveMan（）函数*，以此实现键盘按键信息的传递和人物位置信息的更新。函数调用关系如下：



同理，每次玩家按下键盘也会出发怪物的位置更新，而怪物的位置更新是根据另一套随机算法实现的。

绘制人物时直接调用 Qt 自带的 *paint（）* 函数，直接根据人物的坐标加载人物图片即可完成绘制。由于人物移动时移动方向会改变，所以需要根据不同的移动方向加载不同方向的人物图片。

人物的显示效果图如下：



（3）人物，怪物与墙的碰撞检测

在迷宫中人物和怪物都只能在通路中移动，如果碰到围墙则不能向前继续移动，所以我们需要设计一套碰撞检测算法。最简单且有效的算法即是计算人物和怪物周围的特定大小范围内是否存在围墙单元块，如果存在则人物不能继续向该方向移动，否则可以。同时，为了降低计算成本，碰撞检测算法放在了 *moveMan（）* 函数中，即只会在检测到键盘方向键输入时调用，降低了计算成本。相似的，人物与怪物的碰撞检测也使用类似算法。

碰撞检测算法代码如下：

```
void maze_model::moveMan(int _direction){
    man.set_direction(_direction);
   if(man.get_direction()==0){//试着往左走

if(!(maze[(int)man.get_y()/BLOCK_SIZE][(int)man.get_x()/BLOCK_SIZE-
1].get_type()==0&&man.get_x()-
((int)man.get_x()/BLOCK_SIZE)*BLOCK_SIZE<=man.get_step())){
            man.moveL();
        }
    }
    else if(man.get_direction()==1){

if(!(maze[(int)man.get_y()/BLOCK_SIZE][(int)man.get_x()/BLOCK_SIZE+1]
.get_type()==0&&((int)man.get_x()/BLOCK_SIZE+1)*BLOCK_SIZE-
man.get_x()-man.get_size()<=man.get_step())){
            man.moveR();
        }
    }
    else if(man.get_direction()==2){
        if(!(maze[(int)man.get_y()/BLOCK_SIZE-
1][(int)man.get_x()/BLOCK_SIZE].get_type()==0&&man.get_y()-
((int)man.get_y()/BLOCK_SIZE)*BLOCK_SIZE<=man.get_step())){
            man.moveU();
        }
    }
    else{

if(!(maze[(int)man.get_y()/BLOCK_SIZE+1][(int)man.get_x()/BLOCK_SIZE]
.get_type()==0&&((int)man.get_y()/BLOCK_SIZE+1)*BLOCK_SIZE-
man.get_y()-man.get_size()<=man.get_step())){
            man.moveD();
        }
    }
}
```

（4）怪物随机游走算法和怪物显示

为了添加娱乐性，我们在迷宫中添加了一个怪物，这个怪物需要实现随机游走的功能。最简单的想法是直接使怪物随机向四个方向移动，但是这样的随机移动虽然的确随机，但是

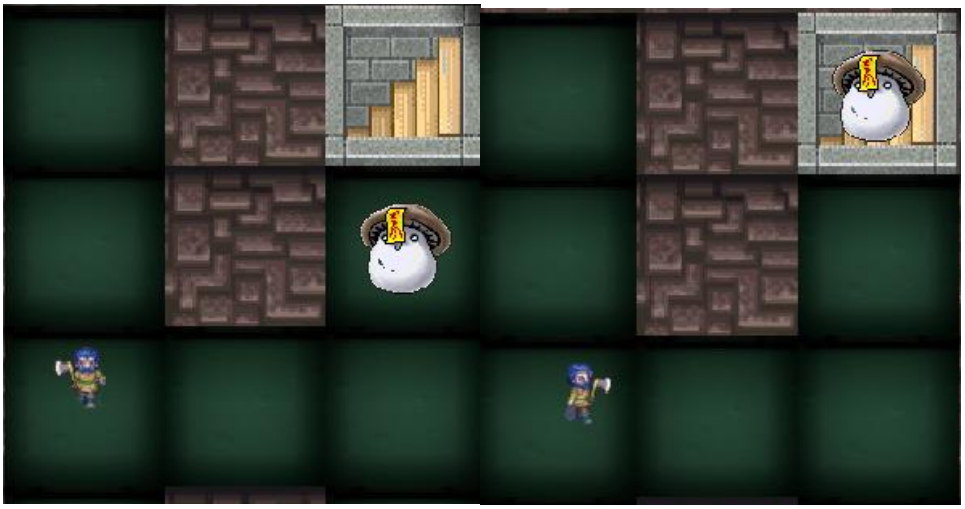
由于四个方向移动的等可能性，非常容易出现怪物在某个区域转圈的情况，而我们需要的是让怪物能够在整个迷宫中随机游走。为了实现这一功能，我实现了一种新的随机算法。这种随机算法的主要思想是当怪物当前位置在一条直线通路上时，则该怪物只会从通路的一头走到另一头。而当怪物走到一个分岔路口时，则随机判断需要向哪个方向走，并且一次性走到通路的末尾。每次走到一个分岔路口，就会重新选择要向哪个方向行走。这个算法实现了怪物的随机行走，并且能让怪物在整个迷宫随机游走。该算法的实现代码如下：

```
void maze_model::moveMonster(int _direction){
    static int count=0;
    static int dirc=0;
    int pos_x=(monster.get_x()/BLOCK_SIZE);
    int pos_y=monster.get_y()/BLOCK_SIZE;
    int dirc_count=0;
    if(pos_x-1>=0){
        if(maze[pos_y][pos_x-1].get_type()!=0) dirc_count++;
    }
    if(pos_x+1<col_size){
        if(maze[pos_y][pos_x+1].get_type()!=0) dirc_count++;
    }
    if(pos_y-1>=0){
        if(maze[pos_y-1][pos_x].get_type()!=0) dirc_count++;
    }
    if(pos_y+1<row_size){
        if(maze[pos_y+1][pos_x].get_type()!=0) dirc_count++;
    }
    if(dirc_count>2){
        count=0;
    }
    if(count==0){
        int l=0,r=0,u=0,d=0;

        for(int i=pos_y+1;i<col_size;i++){
            if(maze[i][pos_x].get_type()==0) break;
            d++;
        }
        for(int i=pos_y-1;i>=0;i--){
            if(maze[i][pos_x].get_type()==0)break;
            u++;
        }
        for(int i=pos_x+1;i<row_size;i++){
            if(maze[pos_y][i].get_type()==0)break;
            r++;
        }
        for(int i=pos_x-1;i>=0;i--){
            if(maze[pos_y][i].get_type()==0)break;
```

```
            l++;
        }
        switch (rand()%4) {
        case 0:count=l;dirc=0;break;
        case 1:count=r;dirc=1;break;
        case 2:count=u;dirc=2;break;
        case 3:count=d;dirc=3;break;
        }
        monster.set_direction(dirc);
    }
    if(count){
        switch (dirc) {
        case 0:monster.set_x(monster.get_x()-BLOCK_SIZE);count-
-;break;
        case 1:monster.set_x(monster.get_x()+BLOCK_SIZE);count-
-;break;
        case 2:monster.set_y(monster.get_y()-BLOCK_SIZE);count-
-;break;
        case 3:monster.set_y(monster.get_y()+BLOCK_SIZE);count-
-;break;
        }
    }
}
```

怪物的显示与人物的显示相似，即绘制怪物时直接调用 Qt 自带的 *paint ()* 函数，直接根据怪物的坐标加载人物图片即可完成绘制。由于怪物移动时移动方向会改变，所以需要根据不同的移动方向加载不同方向的人物图片。显示效果图如下：



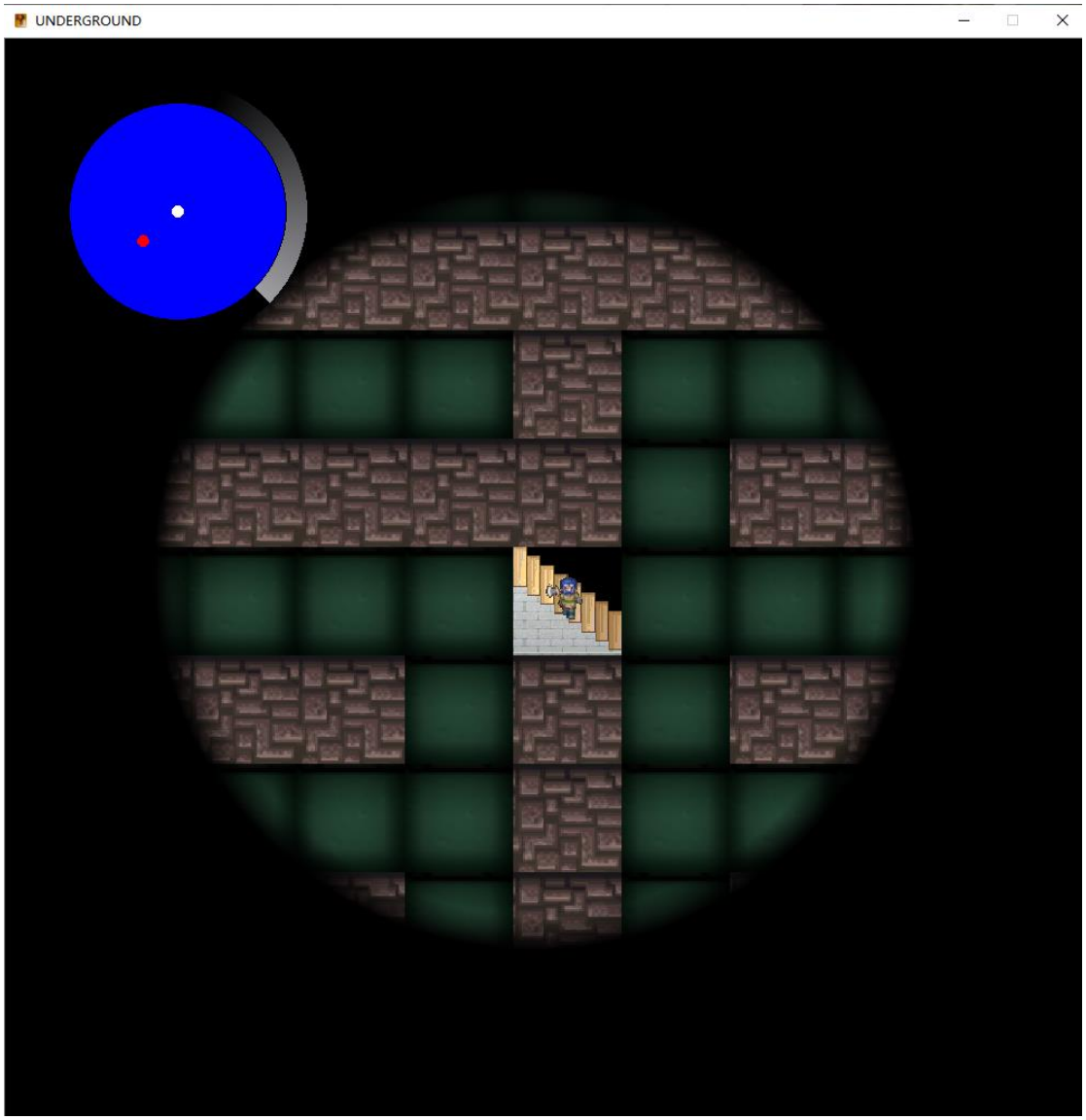
（5）迷宫圆形视野显示

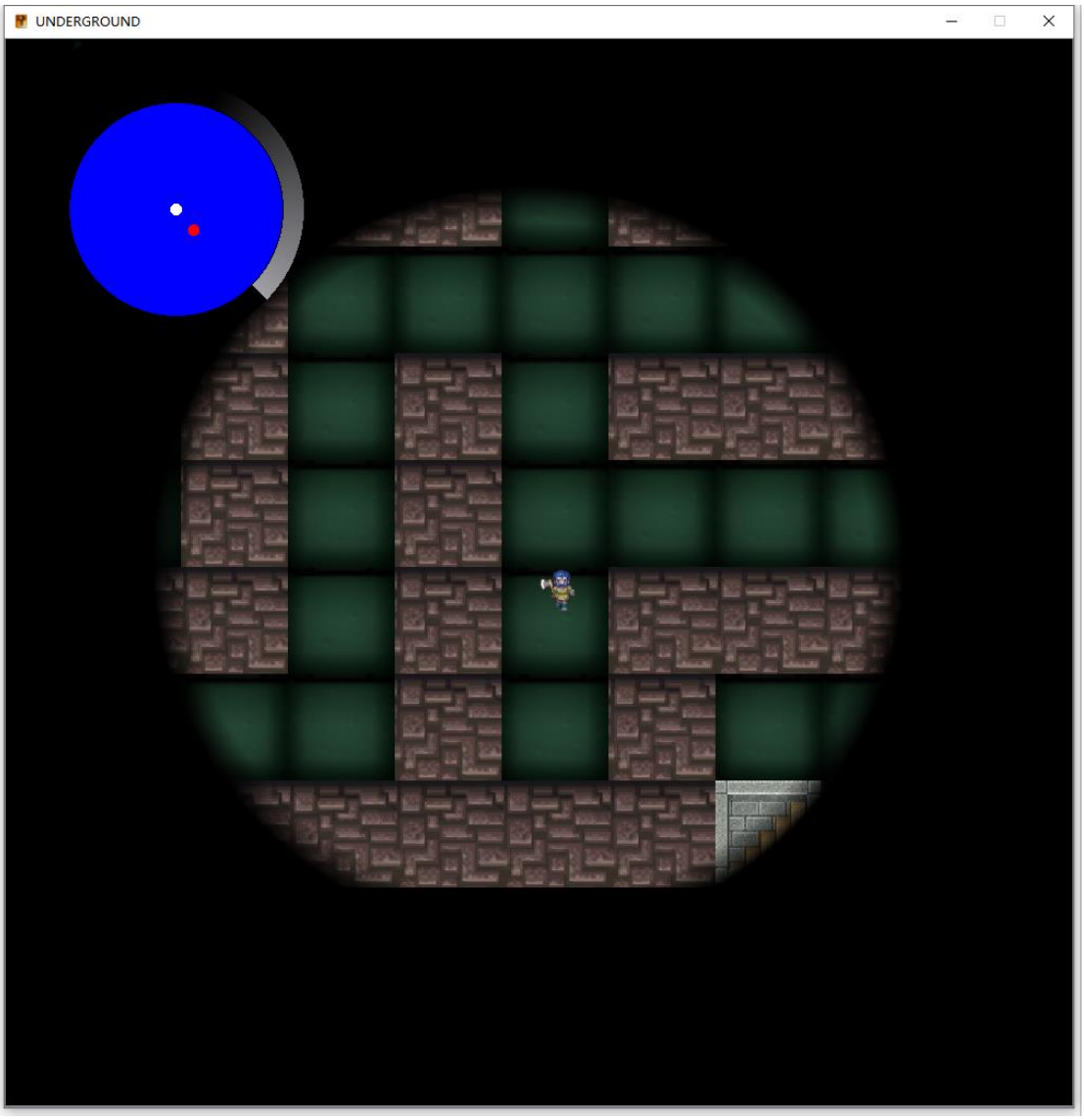
为了提高游戏的难度和真实性，我们需要控制人物可见迷宫的范围为一个圆形区域。同时为了不使这个圆形视野区域过于突兀，我们需要对该圆形区域的边界进行模糊边界处理。我使用 Qt 自带的颜色渐变函数，用来模拟圆形区域边界的模糊。而为了遮挡迷宫在玩家视野范围外的区域，我需要用一个圆环来实现覆盖，并且露出圆环内部的迷宫视野。所以需要使用 2 个同心圆，以玩家为中心，绘制一个黑色的圆环。迷宫圆形视野的显示代码如下：

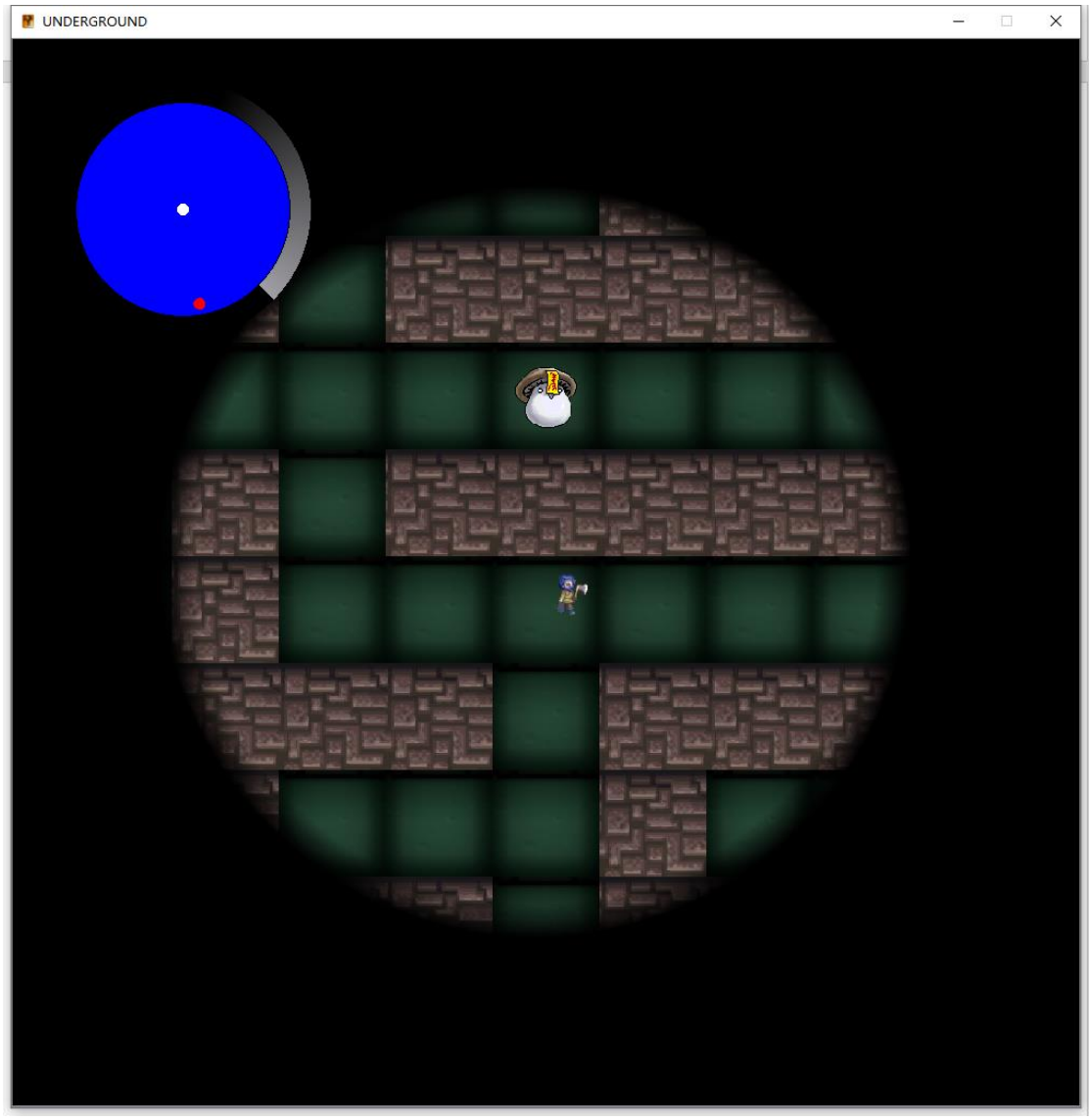
```
painter->drawPixmap(0,0,maze->get_man().get_size(),maze->get_man().ge
t_size(),pix);
    QColor white_op(255,255,255,0);
    QColor black_op(0,0,0,255);
    QRadialGradient
radialGradient(0,0,(PAINT_BLOCK_NUMBER+2)*BLOCK_SIZE*1.6,0,0);
    radialGradient.setColorAt(0.4,white_op);
    radialGradient.setColorAt(0.44,black_op);
    radialGradient.setSpread(QGradient::PadSpread);
    int radius = (PAINT_BLOCK_NUMBER+1)*BLOCK_SIZE*1.6;
        painter->save();
        painter->setPen(white_op);
        painter->setBrush(QBrush(radialGradient));
        //painter->setOpacity(0.5);
        QPainterPath bigCircle;
        bigCircle.addEllipse(-radius, -radius, radius * 2, radius *
2);
        radius = PAINT_BLOCK_NUMBER*BLOCK_SIZE*0.6;
        QPainterPath smallCircle;
        smallCircle.addEllipse(-radius, -radius, radius * 2, radius *
2);
        QPainterPath path = bigCircle - smallCircle;
        painter->drawPath(path);
```

```
        painter->restore();
```

迷宫圆形视野的显示如下:




UNDERGROUND



2. 心得体会

(1) 在该课程中，我对 C++多人协作编程有了更深入的了解，通过该迷宫游戏的制作，我对 C++最新的特性有了更深入的了解，提高了我的编程水平和对 C++的掌握程度。

(2) 为了实现该迷宫游戏，我们采用 Qt 作为游戏设计的底层平台。通过这十天来的学习，我对 Qt 游戏编程有了更深入的了解，也了解了 Qt 的很多特性，感受到了 Qt 的魅力。

(3) 老师在课上给我们介绍了 MVC 和 MVVM 等开发模式，我听了之后醍醐灌顶，终于了解了最新的多人协作开发模式，对 MVVM 模式有了更深入了解，也与同组的同学亲身实践了 MVVM 模式。虽然不能说的确完全的使用了 MVVM 模式，但是也对 MVVM 模式有了更深入的了解，受益匪浅。

(4) 本次课程要求我们使用各种工具来控制版本分发，进行多人协作等。我们使用了 Appveyor，GitHub 等平台进行多人协作和版本控制。使用这些工具，让我对多人协作编程有了更深入了解，也让我学会了如何高效的开发程序。

3. 改进意见

（1） 总的来说，该课程有很多亮点，但是也有一些不足。其中一点就是课程安排过于松散。只有刚开始的 3 天老师讲课，而后面的时间都由学生自由掌握。这样虽然给足了学生自由发挥的空间，但是也容易出现学生过于放飞自我而导致无法按时完成人物的情况。

（2） 老师讲课的内容过于笼统且宽旷，不够系统化。由于只有三天时间，所以讲的内容虽然比较多，但是不够深入，所以比较难让学生深入的了解。