**高级语言程序设计大作业实验报告**

**姓名：黄浩天 学号：2413421**

**一、作业题目**

基于QT框架的2D双人对战游戏开发

**二、开发软件**

Qt Creator

Visual Studio

**三、课题要求**

1. 图形化界面：使用QT图形化平台进行程序开发，实现具有交互功能的图形界面。

2. 面向对象编程：运用面向对象的编程思想，将游戏中的各个元素（如玩家、平台、技能球等）封装成类，并通过类的继承、多态等特性来构建游戏逻辑。

3. 游戏功能实现：实现游戏的基本功能，包括玩家控制、碰撞检测、攻击机制、游戏模式切换、游戏结束判定等。

**四、主要流程**

1. 整体流程

游戏启动后，展示主菜单界面。玩家选择“开始游戏”，弹出游戏模式选择对话框，有“经典模式”和“进化模式”可选。选择模式后进入游戏界面。

在游戏界面，玩家1用wasd移动，j近战攻击，k远程攻击；玩家2用上下左右键移动，1近战攻击，2远程攻击。游戏实时检测玩家输入，处理移动、攻击操作，同时进行碰撞检测，包括玩家与平台、玩家之间、剑气与玩家以及技能球与玩家之间的碰撞，并根据结果处理，如调整玩家位置、计算伤害等。

进化模式下，每隔一定时间随机生成技能球，玩家碰到可获得增益效果。当有玩家生命值降为0时，游戏结束，弹出对话框，玩家可选择返回主菜单或退出游戏。

2. 算法或公式

 玩家移动算法：玩家按下方向键，改变横坐标或纵坐标实现移动。如玩家1按A键，调用`moveLeft`函数，通过`setPos(x() - moveSpeed \* weapons[currentWeaponIndex]->speedModifier, y())`向左移动，移动速度`moveSpeed`和武器速度修正系数`weapons[currentWeaponIndex]->speedModifier`决定实际移动距离。

 跳跃算法：玩家按跳跃键（玩家1为W，玩家2为Up）且着地时，设置垂直速度`verticalSpeed`为负的跳跃力`jumpForce`向上跳跃。后续更新中，累加重力加速度`gravity`模拟重力，限制最大下落速度`maxFallSpeed`。如在`update`函数中，通过`if (!isGrounded) { verticalSpeed += gravity; if (verticalSpeed > maxFallSpeed) { verticalSpeed = maxFallSpeed; } setPos(x(), y() + verticalSpeed); }`实现跳跃和下落模拟。

 碰撞检测算法:

    1.玩家与平台碰撞检测：获取玩家脚部碰撞区域和平台边界矩形，判断是否相交。若相交且玩家满足条件（如正在下落或静止、脚部在平台上方一定容错范围内、水平方向也在平台范围内），则判定玩家落在平台上，调整位置并设为着地状态。如在`checkCollisions`函数中，对玩家1与平台的碰撞检测代码：

if(platform->isColliding(player1)) {

    QRectF feetRect1 = player1->getFeetRect();

    QRectF platformRect = platform->boundingRect();

    platformRect.translate(platform->pos());

    if(feetRect1.bottom() <= platformRect.top() + 15) {

        if(player1->getVerticalSpeed() >= 0 || player1->isGroundedState()) {

            if(feetRect1.intersects(platformRect)) {

                if (player1->x() + player1->boundingRect().width() - 5 > platformRect.left() &&

                    player1->x() + 5 < platformRect.right()) {

                    if (player1->getVerticalSpeed() > 0) {

                        player1->setPos(player1->x(), platformRect.top() - player1->boundingRect().height() + 5);

                    }

                    player1->setGrounded(true);

                    player1OnPlatform = true;

                }

            }

        }

    }

}

```

    2.玩家之间攻击碰撞检测：玩家攻击且未命中目标时，获取攻击范围和对方玩家边界矩形，判断是否相交。若相交，对对方玩家造成伤害并标记已命中目标。以玩家1攻击玩家2为例：

if (player1->isAttacking &&!player2->isDeadState() &&!player1->hasHitTarget()) {

    QRectF attackRect1 = player1->boundingRect();

    attackRect1.translate(player1->pos());

    QRectF player2Rect = player2->boundingRect();

    player2Rect.translate(player2->pos());

    if (attackRect1.intersects(player2Rect)) {

        player2->takeDamage(player1->getCurrentWeaponDamage());

        player1->setHitTarget(true);

    }

}

```

    3.剑气碰撞检测：用`QList`存储所有剑气，遍历检查剑气之间以及剑气与玩家之间的碰撞。对于剑气之间的碰撞，根据伤害值判断处理；对于剑气与玩家的碰撞，若满足条件则对玩家造成伤害并删除剑气。如：

```cpp

for (int i = 0; i < allSwordQis.size(); ++i) {

    for (int j = i + 1; j < allSwordQis.size(); ++j) {

        SwordQi \*qi1 = allSwordQis[i];

        SwordQi \*qi2 = allSwordQis[j];

        if (qi1->belongsToPlayer1() != qi2->belongsToPlayer1() &&

            qi1->collidesWithItem(qi2) &&

            !swordQisToDelete.contains(qi1) &&

            !swordQisToDelete.contains(qi2)) {

            int damage1 = qi1->getDamage();

            int damage2 = qi2->getDamage();

            if (damage1 == damage2) {

                swordQisToDelete.append(qi1);

                swordQisToDelete.append(qi2);

            } else if (damage1 > damage2) {

                qi1->setDamage(damage1 - damage2);

                swordQisToDelete.append(qi2);

            } else {

                qi2->setDamage(damage2 - damage1);

                swordQisToDelete.append(qi1);

            }

        }

    }

}

for (SwordQi \*swordQi : allSwordQis) {

    if (swordQisToDelete.contains(swordQi)) continue;

    if (swordQi->collidesWithItem(player1) &&!swordQi->belongsToPlayer1() &&!player1->isDeadState()) {

        player1->takeDamage(swordQi->getDamage());

        swordQisToDelete.append(swordQi);

    }

    if (swordQi->collidesWithItem(player2) && swordQi->belongsToPlayer1() &&!player2->isDeadState()) {

        player2->takeDamage(swordQi->getDamage());

        swordQisToDelete.append(swordQi);

    }

}

for (SwordQi \*swordQi : swordQisToDelete) {

    if (swordQi && gameScene->items().contains(swordQi)) {

        gameScene->removeItem(swordQi);

        delete swordQi;

    }

}

```

    4.技能球碰撞检测：遍历技能球列表，检查与玩家的碰撞。若碰撞，调用玩家`applySkill`函数应用技能效果，然后删除技能球。代码如下：

```cpp

for (auto it = skillBalls.begin(); it != skillBalls.end();) {

    SkillBall \*ball = \*it;

    if (ball->collidesWithItem(player1)) {

        player1->applySkill(ball->getType());

        gameScene->removeItem(ball);

        delete ball;

        it = skillBalls.erase(it);

        continue;

    }

    if (ball->collidesWithItem(player2)) {

        player2->applySkill(ball->getType());

        gameScene->removeItem(ball);

        delete ball;

        it = skillBalls.erase(it);

        continue;

    }

    ++it;

}

```

  5.游戏结束判定算法：游戏循环中，实时检查玩家生命值。当玩家1或玩家2生命值降为0时，判定游戏结束，停止游戏定时器，显示胜利者信息，弹出游戏结束对话框。如在`checkGameOver`函数中：

```cpp

if(player1->isDeadState() || player2->isDeadState()) {

    gameOver = true;

    gameTimer->stop();

    QGraphicsTextItem \*winnerText = new QGraphicsTextItem();

    winnerText->setDefaultTextColor(Qt::yellow);

    winnerText->setFont(QFont("Arial", 32, QFont::Bold));

    if(player1->isDeadState()) {

        winnerText->setPlainText("Player 2 Wins!");

    } else {

        winnerText->setPlainText("Player 1 Wins!");

    }

    winnerText->setPos(250, 250);

    gameScene->addItem(winnerText);

    onGameOver();

}

```

3. 单元测试

测试策略：游戏逻辑复杂，采用黑盒测试和白盒测试结合的方法。黑盒测试关注游戏整体功能正确性，如玩家操作响应、游戏模式切换、游戏结束判定等；白盒测试深入类内部方法，检查关键逻辑和算法实现，如碰撞检测算法、技能效果应用等。

 测试用例设计：

玩家移动测试：测试玩家1和玩家2的左右移动、跳跃操作，检查是否能正常移动，有无超出边界或异常情况。

攻击测试：测试玩家近战和远程攻击，检查攻击触发、范围和伤害计算，以及剑气发射和碰撞检测是否正常。

碰撞检测测试：测试玩家与平台、玩家之间、剑气与玩家以及技能球与玩家的碰撞，检查检测准确性和碰撞后处理逻辑，如位置调整、伤害计算、技能效果触发等。

游戏模式切换测试：在主菜单选择不同游戏模式，检查是否能正确切换，不同模式下功能（如技能球生成、剑气充能等）是否正常。

游戏结束测试：模拟玩家生命值降为0，检查游戏结束判定、胜利者信息显示、游戏结束对话框弹出，以及对话框按钮功能是否正常。

测试结果与分析：测试中发现部分碰撞检测误判，经调试是碰撞检测边界条件处理不当，特殊情况下玩家脚部与平台碰撞判定偏差。调整碰撞检测代码的容错范围和边界条件判断后解决问题。多次测试后，游戏各项功能正常运行，满足设计要求。

**五、收获**

1. QT框架的应用

本次游戏开发深入学习了QT框架。掌握创建图形化界面，包括使用窗口、按钮、标签等控件；学会利用信号与槽机制实现对象间事件通信，如连接按钮点击事件与处理函数；了解QT图形场景和视图系统，能创建游戏场景、添加和管理图形元素，如游戏中的玩家、平台、技能球等。为今后开发复杂图形化应用程序奠定基础。

2. 面向对象编程的深化理解

游戏开发充分运用面向对象编程思想。将游戏元素封装成类，如`Player`类、`Platform`类、`SkillBall`类等，实现数据和操作封装。通过类的继承创建`Player1`和`Player2`类继承自`Player`类，复用通用功能并个性化扩展。利用多态性，通过基类指针或引用调用子类实现方法，处理不同玩家操作和碰撞检测，提高代码灵活性和可扩展性。深入理解了面向对象编程的封装、继承、多态特性，能更好运用这些特性构建复杂软件系统。

3. 游戏开发中的算法与逻辑设计

游戏开发涉及多种算法和逻辑设计，如碰撞检测算法、玩家移动和攻击逻辑、游戏结束判定等。实现过程中需考虑边界条件和异常情况，确保游戏稳定性和公平性。如碰撞检测算法，精确计算图形对象边界矩形和相交情况，处理碰撞后位置调整和状态变化，实现准确碰撞检测。学会分析和设计复杂算法，优化算法提高程序性能和效率。处理游戏逻辑时，确保各功能模块协同工作，培养了系统思维和问题解决能力。