大作业文档

司路阳 未央-软件31 2023012824 sily23@mails.tsinghua.edu.cn

模块之间逻辑关系

本项目主要由以下几个模块组成：

Scene模块：包含BattleScene和Scene类，负责管理游戏的整体场景，更新游戏状态，并处理用户的输入。还有CommandDialog类和GameOverDialog类，分别负责唤出作弊码窗口和游戏结束窗口。

Item模块：包含游戏中的所有物品（如武器、护甲、头部装备、腿部装备等），每个物品都有对应的类进行管理。

Character模块：负责管理游戏中的角色（如Link和Rival），包括角色的状态更新、动作执行、以及与物品的交互。

Condition模块：处理角色和物品的各种状态效果，如冻结、燃烧、触电等。

Game模块：主要包含游戏的启动、场景的初始化，以及结束对话框的处理。

这些模块通过类的继承、信号和槽机制，以及CMake构建系统进行组织和连接。项目通过类的继承实现了各个模块之间的高度复用，同时通过信号和槽机制实现了不同模块之间的通信。

程序运行流程

程序运行流程 启动游戏：MyGame类初始化后，创建BattleScene，并设置游戏场景视图。

场景初始化：BattleScene类负责初始化地图、物品、角色，以及各种游戏元素，并开始游戏主循环。

主循环更新：在每一帧中，BattleScene通过调用update函数更新场景中所有物体的位置、状态，以及检查是否有物体需要移除或处理特定效果。

用户输入：BattleScene监听键盘事件，处理用户输入，控制角色动作（如移动、攻击、使用物品等）。

状态管理：在游戏过程中，Condition模块处理角色和物品的各种状态效果（如冻结、燃烧等），并更新其表现。

游戏结束：当角色的血量降至0时，弹出GameOverDialog对话框，玩家可以选择重新开始游戏或退出游戏。

作弊码输入：用户可以通过按/键弹出CommandDialog窗口，输入作弊码直接生成指定的物品。

完成的要求

需求点1：基本移动和生命值（10%）

具体要求

- 使用按键1和按键2控制人物左右移动（2%）。

在 `Character` 类中，代码片段定义了如何根据按键的状态来更新角色的速度和翻转。当左键按下时，速度向左增加，角色水平翻转。当右键按下时，速度向右增加，角色水平翻转。如果没有按下任何水平移动的按键，速度向0靠近。

```
if (isLeftDown()) {
      velocity.setX(qMax(velocity.x() - moveSpeed,-0.3)); // 左键按下，速度向
左增加
      setTransform(QTransform().scale(1, 1)); // 设置角色的水平翻转
  }
  if (isRightDown()) {
      velocity.setX(qMin(velocity.x() + moveSpeed,0.3)); // 右键按下，速度向右
增加
      setTransform(QTransform().scale(-1, 1)); // 设置角色的水平翻转
  }
  else{
      //如果没有按下任何水平移动的按键，速度向0靠近
      if (velocity.x() > 0) {
          velocity.setX(qMax(velocity.x() - moveSpeed/2, 0.00000001));
      } else if (velocity.x() < 0) {
          velocity.setX(qMin(velocity.x() + moveSpeed/2, -0.00000001));
      }
  }
```

- 使用按键3控制人物跳跃，能够跳到更高的平台上（2%）。

  代码片段定义了如何根据按键的状态来更新角色的跳跃状态。当按下W键时，如果角色没有被冻结，则设置跳跃按键被按下，并更新角色的下降速度和加速度。

```
case Qt::Key_W:
    if (link != nullptr && link->beFrozen == false) {
        link->setJumpDown(true);
        link->downSpeed = -1;
        link->downAcceleration = gravity.getGravity();
    }
    break;
```

- 实现重力加速度，使人物和物体在悬空时会按照重力自然下坠（3%）。

  在 `Gravity` 类中，我们定义了重力加速度以及如何将其应用于一个 `Item`。重力通过 `setGravity` 方法设置，并通过 `getVelocity` 和 `setVelocity` 方法获取和设置物体的速度。`setPos` 方法用于更新物体的位置，考虑到了重力加速度的影响。

```
// gravity.cpp 片段
void Gravity::setVelocity(Item *item, double deltaTime) {
    velocity = item->downSpeed + gravity * deltaTime;
    item->downSpeed = velocity;
}
```

```
void Gravity::setPos(Item *item, double deltaTime) {
    if(item->downAcceleration != 0) {
        auto y = item->downSpeed * deltaTime + 0.5 * gravity * deltaTime *
deltaTime;
        item->setPos(item->pos() + QPointF(0, y));
    } else {
        item->setPos(item->pos() + QPointF(0, 0));
    }
}
```

- 支持两个玩家同屏对战，使用两套不同的键盘按键映射分别操作两个角色（3%）。

  建立两套映射即可

```
void BattleScene::keyPressEvent(QKeyEvent *event) {
  if(event->isAutoRepeat()){
      return;
  }
  switch (event->key()) {
  case Qt::Key_A:
      if (link != nullptr && link->beFrozen == false) {
          link->setLeftDown(true);
      }
      break;
  case Qt::Key_D:
      if (link != nullptr && link->beFrozen == false) {
          link->setRightDown(true);
      }
      break;
  case Qt::Key_S:
      if (link != nullptr && link->beFrozen == false) {
          link->setPickDown(true);
      }
      break;
  case Qt::Key_W:
      if (link != nullptr && link->beFrozen == false) {
          link->setJumpDown(true);
          link->downSpeed = -1;
          link->downAcceleration = gravity.getGravity();
      }
      break;
  case Qt::Key_J:
      if (link != nullptr && link->beFrozen == false) {
          if (link->melee != nullptr && link->melee->isVisible()) {
              link->setAttackDown(true);
              attackDone(link, rival);
              rival->updateHealthBar();
          }
          else if(link->bow != nullptr && link->bow->isVisible()){
              link->setThrowDown(true);
          }
      }
```

```
        break;
    case Qt::Key_Q:
        if (link != nullptr && link->beFrozen == false) {
            link->setThrowDown(true);
        }
        break;
    case Qt::Key_L:
        if (link != nullptr && link->beFrozen == false){
            link->setChangeDown(true);
        }
        break;
    case Qt::Key_K:
        if (link!=nullptr && link->beFrozen == false){
            link->setChangeArrowDown(true);
            link->changeArrow();
        }
        break;
    case Qt::Key_Left:
        if (rival != nullptr && rival->beFrozen == false) {
            rival->setLeftDown(true);
        }
        break;
    case Qt::Key_Right:
        if (rival != nullptr && rival->beFrozen == false) {
            rival->setRightDown(true);
        }
        break;
    case Qt::Key_Down:
        if (rival != nullptr && rival->beFrozen == false) {
            rival->setPickDown(true);
        }
        break;
    case Qt::Key_Up:
        if (rival != nullptr && rival->beFrozen == false) {
            rival->setJumpDown(true);
            rival->downSpeed = -1;
            rival->downAcceleration = gravity.getGravity();
            //character->downAcceleration = 0.03;
        }
        break;
    case Qt::Key_Shift:
        if (rival != nullptr && rival->beFrozen == false) {
            if (rival->melee != nullptr) {
            rival->setAttackDown(true);
            attackDone(rival, link);
            link->updateHealthBar();
            }
            else if(rival->bow != nullptr && rival->bow->isVisible()){
                rival->setThrowDown(true);
            }
        }
        break;
    case Qt::Key_Call:
        if (rival != nullptr && rival->beFrozen == false) {
```

```
            rival->setThrowDown(true);
        }
        break;
    case Qt::Key_Space:
        if (rival != nullptr && rival->beFrozen == false) {
            rival->setChangeDown(true);
        }
        break;
    case Qt::Key_B:
        if (rival != nullptr && rival->beFrozen == false) {
            rival->setChangeArrowDown(true);
            rival->changeArrow();
        }
        break;
    default:
            Scene::keyPressEvent(event);
        }
    } //按键按下事件
```

需求点2：多种地形（10%）

- 地图中存在不同高度的平台（5%）。

  `Item` 类负责管理游戏中的物品，包括它们的位置和状态。它也包含了用于检测物品是否在地面上的逻辑。

```
bool Item::isOnGround(Item *item) {
    if (item != nullptr) {
        QPointF Position = item->pos();
        int blockX = static_cast<int>(Position.x()) / 80;
        int blockY = static_cast<int>(Position.y()) / 80;
        double absence = 30;
        int n_blockY = static_cast<int>(Position.y() + absence) / 80;

        if (blockY < 0 || blockY >= 9 || blockX < 0 || blockX >= 16) {
            return false;  // 防止越界访问
        }

        if (blocks[blockY][blockX] != 0) {
            return true;
        }

        if (blocks[blockY][blockX] == 0 && blocks[n_blockY][blockX] != 0) {
            return false;
        }
    }
    return false;
}
```

- 平台支持不同的材质，以不同的外观呈现（5%）。

地图由一个二维数组 `blocks` 表示，其中不同的数字代表不同的平台类型。

```
// Item.cpp 片段
const int BattleScene::blocks[9][16] = {
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 4, 4, 4, 4, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{3, 3, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 3, 3},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}}; //方块数组初始化
```

为数组值不同的地方新建不同类型的砖块即可完成不同材质不同外观不同高度的平台的创建

```
addItem(map); //添加地图
for(int i = 0; i < 9; i++) {
    for(int j = 0; j < 16; j++) {
        switch (blocks[i][j]) {
        case 1:
            blockGrid[i][j] = new Soilblock();
            break;
        case 2:
            blockGrid[i][j] = new Grassblock();
            break;
        case 3:
            blockGrid[i][j] = new Ironblock();
            blockGrid[i][j]->electrocuted = new Electrocuted();
            blockGrid[i][j]->electrocuted->setScale(1.5);
            blockGrid[i][j]->electrocuted->setParentItem(blockGrid[i][j]);
            blockGrid[i][j]->electrocuted->setPos(20 - blockGrid[i][j]-
>electrocuted->boundingRect().width() / 2,
                                                  20 - blockGrid[i][j]-
>electrocuted->boundingRect().height() / 2);
            blockGrid[i][j]->electrocuted->setVisible(0);
            //blockGrid[i][j]->electrocuted->setZValue(1);
            break;
        case 4:
            blockGrid[i][j] = new Stoneblock();
            blockGrid[i][j]->fire = new Fire();
            blockGrid[i][j]->fire->setScale(0.6);
            blockGrid[i][j]->fire->setParentItem(blockGrid[i][j]);
            blockGrid[i][j]->fire->setPos(80 - blockGrid[i][j]->fire-
>boundingRect().width() / 2,
                                          90 - blockGrid[i][j]->fire-
>boundingRect().height() / 2);
            blockGrid[i][j]->fire->setVisible(0);
            //blockGrid[i][j]->fire->setZValue(1);
            break;
```

```
            default:
                blockGrid[i][j] = nullptr;
                continue;
            }

            // 在将效果添加到场景之前设置方块的位置
            blockGrid[i][j]->setPos(j * blockWidth, i * blockWidth);

            //将方块添加到场景
            addItem(blockGrid[i][j]);

            //现在添加效果（只添加一次，作为开关情况的一部分）
            if (blockGrid[i][j]->electrocuted != nullptr) {
                addItem(blockGrid[i][j]->electrocuted);
            }

            if (blockGrid[i][j]->fire != nullptr) {
                addItem(blockGrid[i][j]->fire);
            }
        }
    }
```

需求点3：生命值系统（10%）

具体要求

- 每个玩家各有一定的生命值，支持显示生命值（5%）。

  在 `Character` 类中，代码片段定义了如何创建一个图形项来表示角色的生命值条。生命值条的矩形位置、大小和颜色都被设置好了。

```
// Character.cpp 片段
Character::Character(QGraphicsItem *parent) : Item(parent, ""),
healthBar(new QGraphicsRectItem(this)) {
    // ...（生命值条创建逻辑省略）
    healthBar->setRect(-50,-220,100,10); //设置生命值条的矩形
    healthBar->setBrush(QBrush(Qt::green)); //设置生命值条的颜色
}
```

- 受到伤害会减少生命值。当一方生命值首先降低至低于零时，游戏结束，判定另一方胜利（5%）。

  在 `Character` 类中，代码片段定义了如何更新角色的生命值和生命值条。当设置新的生命值时，会根据生命值的比例来计算生命值条的宽度，并更新生命值条的位置和颜色。如果生命值较低，生命值条的颜色会变为红色。

```
// Character.cpp 片段
void Character::setHealth(qreal newHealth) {
    this->health = newHealth;
    updateHealthBar();
```

```
}

void Character::updateHealthBar() {
    int width = 100; //生命值条的宽度
    qreal healthBarWidth = (width * health) / 100; //生命值是0-100，计算生命值
条的宽度
    healthBar->setRect(-50,-220,healthBarWidth,10); //设置生命值条的矩形
    if(health <= 20){
        healthBar->setBrush(QBrush(Qt::red)); //如果生命值较低，设置生命值条的
颜色为红色
    }
}
```

游戏胜利界面

为了实现一个游戏结束的窗口，当一方的血量小于等于 0 时弹出窗口并显示获胜方的消息，使用 `QDialog` 来创建一个模态对话框，并在对话框中包含两个按钮："重新开始"和"退出游戏"。

实现步骤

创建 GameOverDialog 类： 继承自 `QDialog`，用于显示游戏结束的窗口。

在 `BattleScene` 中检测角色血量： 当检测到角色血量小于等于 0 时，弹出游戏结束窗口。

处理重新开始和退出游戏按钮： 重新开始按钮重启程序，退出按钮关闭程序。

代码实现

1. 创建 `GameOverDialog` 类

```
#include <QDialog>
#include <QPushButton>
#include <QVBoxLayout>
#include <QLabel>

class GameOverDialog : public QDialog {
    Q_OBJECT

public:
    explicit GameOverDialog(const QString& winner, QWidget *parent = nullptr) :
QDialog(parent) {
        setWindowTitle("Game Over");

        QLabel *label = new QLabel(winner, this);
        label->setAlignment(Qt::AlignCenter);

        QPushButton *restartButton = new QPushButton("重新开始", this);
        QPushButton *quitButton = new QPushButton("退出游戏", this);

        QVBoxLayout *layout = new QVBoxLayout(this);
        layout->addWidget(label);
```

```
        layout->addWidget(restartButton);
        layout->addWidget(quitButton);

        connect(restartButton, &QPushButton::clicked, this,
&GameOverDialog::restartGame);
        connect(quitButton, &QPushButton::clicked, this,
&GameOverDialog::quitGame);
    }

signals:
    void restartGame();
    void quitGame();
};
```

2. 在 BattleScene 中检测角色血量

```
void BattleScene::checkGameOver() {
    if (link->health <= 0) {
        showGameOverDialog("P2 胜利!");
    } else if (rival->health <= 0) {
        showGameOverDialog("P1 胜利!");
    }
}

void BattleScene::showGameOverDialog(const QString& winner) {
    GameOverDialog *dialog = new GameOverDialog(winner, this);
    connect(dialog, &GameOverDialog::restartGame, this,
&BattleScene::restartGame);
    connect(dialog, &GameOverDialog::quitGame, this, &BattleScene::quitGame);
    dialog->exec();  // 显示模态对话框
}
```

3. 处理重新开始和退出游戏按钮

```
void BattleScene::restartGame() {
    // 重新启动程序
    QProcess::startDetached(QCoreApplication::applicationFilePath());
    QCoreApplication::quit();
}

void BattleScene::quitGame() {
    // 退出程序
    QCoreApplication::quit();
}
```

4. 在 BattleScene::update() 中调用 checkGameOver

```
void BattleScene::update() {
    // 更新游戏逻辑
    // ...

    // 检查游戏是否结束
    checkGameOver();

    // 继续更新
    Scene::update();
}
```

解释

`GameOverDialog` 类： 这是一个自定义的对话框，显示游戏结束的消息和两个按钮（重新开始和退出游戏）。

`checkGameOver()` 函数： 每次更新时检查双方的血量，决定是否弹出游戏结束窗口。

`restartGame()` 和 `quitGame()` 函数： 分别处理重新启动程序和退出程序的逻辑。

整体逻辑

每次更新游戏时，检查 link 和 rival 的血量，如果一方的血量小于等于 0，就会弹出游戏结束的窗口。 玩家可以选择重新开始游戏或退出游戏。

需求点4：物品掉落（10%）

具体要求

- 双方玩家开始时没有武器不能攻击，武器等其他物品会从空中随机出现，按照重力规律下落（4%）。

  在 `BattleScene` 类中，代码片段定义了如何随机生成物品并将其添加到场景中。物品包括武器、盔甲和消耗品，它们会根据重力规律下落。

  ```
  void BattleScene::spawnRandomItem()
  {
      int randomItemIndex = QRandomGenerator::global()->bounded(0, 31); // 生成0到11之间的随机数
      int randomXPosition = QRandomGenerator::global()->bounded(0, 1280); // 随机生成X位置

      Item *newItem = nullptr;

      switch (randomItemIndex) {
      case 0:
          newItem = new WoodShortSword();
          break;
      case 1:
          newItem = new WoodLongSword();
          break;
  ```

```
        case 2:
            newItem = new WoodStaff();
            break;
        case 3:
            newItem = new ThunderSword();
            break;
        case 4:
            newItem = new FireSword();
            break;
        case 5:
            newItem = new IceSword();
            break;
        case 6:
            newItem = new IronStaff();
            break;
        case 7:
            newItem = new IronLongSword();
            break;
        case 8:
            newItem = new IronShortSword();
            break;
        case 9:
            newItem = new IronBow();
            break;
        case 10:
            newItem = new IronStrongBow();
            break;
         case 11:
             newItem = new IronHandBow();
             break;
       case 12:
            newItem = new WoodBow();
            break;
         case 13:
            newItem = new WoodStrongBow();
            break;
        case 14:
            newItem = new WoodHandBow();
             break;
        case 15:
            newItem = new NormalArrow();
            break;
        case 16:
            newItem = new FireArrow();
            break;
        case 17:
            newItem = new IceArrow();
            break;
        case 18:
            newItem = new ThunderArrow();
            break;
         case 19:
            newItem = new FlamebreakerArmor();
            break;
```

```
        case 20:
            newItem = new IceArmor();
            break;
        case 21:
            newItem = new ThunderArmor();
            break;
        case 22:
            newItem = new OldShirt();
            break;
        case 23:
            newItem = new CapOfTheHero();
            break;
        case 24:
            newItem = new FlameCap();
            break;
        case 25:
           newItem = new IceCap();
           break;
        case 26:
           newItem = new ThunderCap();
            break;
        case 27:
            newItem = new FlameTrousers();
            break;
        case 28:
            newItem = new IceTrousers();
            break;
        case 29:
            newItem = new ThunderTrousers();
            break;
        case 30:
            newItem = new WellWornTrousers();
             break;
        default:
            //newItem = new Item(); // 默认物品，防止出错
            break;
        }

        if (newItem) {
            newItem->setPos(randomXPosition, 0); // 设置物品初始位置
            //newItem->setPos(newItem->pos() + QPointF(0, -newItem-
>boundingRect().height())); // 使物品显示在地面上
            newItem->setZValue(1);
            addItem(newItem); // 添加到场景
            dropItems.append(newItem);
            auto mountable = dynamic_cast<Mountable *>(newItem);
            mountable->unmount();
            applyGravity(newItem);

            // 在3秒后移除物品，如果未被拾取
            QTimer::singleShot(3000, [this, newItem]() {
                removeItemAfterDelay(newItem);
            });
        }
```

```
}

void BattleScene::removeItemAfterDelay(Item* item) {
    if (item) {  // 如果物品存在且未被拾取
        auto mountable = dynamic_cast<Mountable *>(item);
        if(mountable->isMounted()){
            return;
        }

        dropItems.removeOne(item);  // 从掉落物品列表中移除
        removeItem(item);  // 从场景中移除
        delete item;  // 删除物品
    }
}

void BattleScene::applyGravity(Item* item) {
    QTimer *gravityTimer = new QTimer(this);
    connect(gravityTimer, &QTimer::timeout, [this, item, gravityTimer]() {
        item->setPos(item->pos().x(), item->pos().y() + 10); // 使物品每帧向下
移动
        if (item->pos().y() >= 720 || item->isOnGround(item)) { // 检测是否落
地
            gravityTimer->stop();
            gravityTimer->deleteLater();
        }
    });
    gravityTimer->start(16); // 每16ms（约60帧每秒）更新位置
}
```

- 当掉落物品在玩家附近时，玩家使用按键4捡起物品。如果是武器则会装备，如果是消耗品则物品效果立刻生效（3%）。

  `Mountable` 类提供了一个接口，用于处理可装备物品的挂载和卸载。

```
// Mountable.cpp 片段
void Mountable::mountToParent() {
    mounted = true;
}

void Mountable::unmount() {
    mounted = false;
}
```

  在 `BattleScene` 类中，代码片段定义了如何处理拾取物品。玩家在拾取时，会寻找最近的未挂载的可挂载物品，并将其拾取。如果拾取的物品是护甲、帽子、裤子、近战武器、弓或箭，则根据物品类型进行不同的处理。

```
void BattleScene::processPicking() {
Scene::processPicking();
```

```
//qDebug() << "Processing picking.";
if (link->isPicking()) {
    auto mountable = findNearestUnmountedMountable(link->pos(), 100.); //查找
最近的未挂载的可挂载物品，距离阈值为100
    if (mountable != nullptr) {
        Item* tempItem;
        tempItem = dynamic_cast<Item*>(pickupMountable(link, mountable)); //
拾取可挂载物品
        if(tempItem != nullptr){
            dropItems.push_back(tempItem);
        }
    }
}
if (rival->isPicking()) {
    auto mountable = findNearestUnmountedMountable(rival->pos(), 100.); //查
找最近的未挂载的可挂载物品，距禒阈值为100
    if (mountable != nullptr) {
        Item* tempItem;
        tempItem = dynamic_cast<Item*>(pickupMountable(rival, mountable));
//拾取可挂载物品
        if(tempItem != nullptr){
            dropItems.push_back(tempItem);
        }
    }
}
} //处理拾取

Mountable *BattleScene::findNearestUnmountedMountable(const QPointF &pos,
qreal distance_threshold) {
Mountable *nearest = nullptr; //最近的可挂载物品
qreal minDistance = distance_threshold; //最小距离为距离阈值

for (QGraphicsItem *item: items()) {
    if (auto mountable = dynamic_cast<Mountable *>(item)) { //如果是可挂载物品
        if (!mountable->isMounted()) {
            qreal distance = QLineF(pos, item->pos()).length(); //计算距离
            if (distance < minDistance) {
                minDistance = distance; //更新最小距离
                nearest = mountable; //更新最近的可挂载物品
            }
        }
    }
}

return nearest;
} //查找最近的未挂载的可挂载物品

Mountable *BattleScene::pickupMountable(Character *character, Mountable
*mountable) {
if (auto armor = dynamic_cast<Armor *>(mountable)) {
    return character->pickupArmor(armor); //拾取护甲
}
else if (auto cap = dynamic_cast<HeadEquipment *>(mountable)) {
    return character->pickupHeadEquipment(cap); //拾取帽子
```

```
}
else if (auto trousers = dynamic_cast<LegEquipment *>(mountable)) {
    return character->pickupLegEquipment(trousers); //拾取裤子
}
else if (auto melee = dynamic_cast<MeleeWeapon *>(mountable)) {
    return character->pickupMelee(melee); //拾取近战武器
}
else if (auto bow = dynamic_cast<Bow *>(mountable)) {
    return character->pickupBow(bow); //拾取弓
}
else if (auto arrow = dynamic_cast<Arrow *>(mountable) ) {
    if(character->bow != nullptr)
    {
        return character->pickupArrow(arrow);
    }       //拾取箭
}
return nullptr;
} //拾取可挂载物品
```

在 `Character` 类中，代码片段定义了如何拾取不同类型的物品。拾取护甲、帽子、裤子、近战武器、弓和箭时，会根据物品类型进行不同的处理。例如，如果当前角色已经有近战武器，则不能拾取新的近战武器。如果当前角色有弓箭，则近战武器会隐藏。

```
Armor *Character::pickupArmor(Armor *newArmor) {
auto oldArmor = armor; //旧护甲
if (oldArmor != nullptr) { //如果旧护甲不为空
    oldArmor->unmount(); //卸载旧护甲
    oldArmor->setPos(newArmor->pos()); //设置旧护甲位置为新护甲位置
    oldArmor->setParentItem(parentItem());  //设置旧护甲的父节点为当前节点
}
newArmor->setParentItem(this); //设置新护甲的父节点为当前节点
newArmor->mountToParent(); //挂载新护甲到父节点
armor = newArmor; //设置新护甲
return oldArmor; //返回旧护甲
} //拾取护甲

HeadEquipment *Character::pickupHeadEquipment(HeadEquipment *
newHeadEquipment) {
auto oldHeadEquipment = headEquipment; //旧头部装备
if (oldHeadEquipment != nullptr) { //如果旧头部装备不为空
    oldHeadEquipment->unmount(); //卸载旧头部装备
    oldHeadEquipment->setPos(newHeadEquipment->pos()); //设置旧头部装备位置为
新头部装备位置
    oldHeadEquipment->setParentItem(parentItem());  //设置旧头部装备的父节点为
当前节点
}
newHeadEquipment->setParentItem(this); //设置新头部装备的父节点为当前节点
newHeadEquipment->mountToParent(); //挂载新头部装备到父节点
headEquipment = newHeadEquipment; //设置新头部装备
return oldHeadEquipment; //返回旧头部装备
}
```

```
LegEquipment *Character::pickupLegEquipment(LegEquipment * newLegEquipment)
{
auto oldLegEquipment = legEquipment; //旧腿部装备
if (oldLegEquipment != nullptr) { //如果旧腿部装备不为空
    oldLegEquipment->unmount(); //卸载旧腿部装备
    oldLegEquipment->setPos(newLegEquipment->pos()); //设置旧腿部装备位置为新腿
部装备位置
    oldLegEquipment->setParentItem(parentItem());  //设置旧腿部装备的父节点为当
前节点
}
newLegEquipment->setParentItem(this); //设置新腿部装备的父节点为当前节点
newLegEquipment->mountToParent(); //挂载新腿部装备到父节点
legEquipment = newLegEquipment; //设置新腿部装备
return oldLegEquipment; //返回旧腿部装备
}

Arrow* Character::pickupArrow(Arrow* newArrow) {
newArrow->setParentItem(this); // 设置新箭矢的父节点为当前角色
newArrow->mountToParent(); //挂载新箭矢到父节点
arrows.append(newArrow); // 将新箭矢添加到 arrows 向量中
newArrow->setVisible(0);
return nullptr;
}
MeleeWeapon* Character::pickupMelee(MeleeWeapon* newMelee) {
if (melee != nullptr) {  // 如果当前已经有近战武器，则不能拾取新的
    return nullptr;  // 返回nullptr表示没有拾取新武器
}

// 如果没有近战武器，执行正常的拾取逻辑
newMelee->setParentItem(this); // 设置新近战武器的父节点为当前节点
newMelee->mountToParent(); // 挂载新近战武器到父节点
melee = newMelee; // 设置新近战武器
if (bow != nullptr && bow->isVisible()) {
    melee->setVisible(0);  // 如果弓箭可见，则将近战武器隐藏
}
return nullptr;  // 返回nullptr表示没有替换旧武器
}


Bow *Character::pickupBow(Bow *newBow){
auto oldBow = bow;
if(oldBow != nullptr){
    oldBow->unmount();
    oldBow->setPos(newBow->pos());
    oldBow->setParentItem(parentItem());
    oldBow->setVisible(1);
}
newBow->setParentItem(this);
newBow->mountToParent();
bow = newBow;
if(melee != nullptr && melee->isVisible()){
    bow->setVisible(0);
}
```

```
return oldBow;
}
```

- 作弊码：实现一个文本框接受作弊码输入，使用作弊码可以立即生成任意指定的可掉落物体。这也是方便调试的有用工具（3%）。

1. Implementing CommandDialog Class

The CommandDialog class will be a simple dialog that includes a QLineEdit for input, a confirm button, and a cancel button.

```
CommandDialog.h
#ifndef COMMANDDIALOG_H
#define COMMANDDIALOG_H

#include <QDialog>
#include <QLineEdit>
#include <QPushButton>
#include <QVBoxLayout>
#include <QHBoxLayout>

class CommandDialog : public QDialog {
    Q_OBJECT

public:
    explicit CommandDialog(QWidget *parent = nullptr);

signals:
    void commandEntered(const QString &command);

private slots:
    void onConfirmButtonClicked();

private:
    QLineEdit *commandLineEdit;
    QPushButton *confirmButton;
    QPushButton *cancelButton;
};

#endif // COMMANDDIALOG_H
```

```
CommandDialog.cpp

#include "CommandDialog.h"

CommandDialog::CommandDialog(QWidget *parent) : QDialog(parent) {
    setWindowTitle("Enter Cheat Code");

    commandLineEdit = new QLineEdit(this);
    confirmButton = new QPushButton("Confirm", this);
```

```
    cancelButton = new QPushButton("Cancel", this);

    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    mainLayout->addWidget(commandLineEdit);

    QHBoxLayout *buttonLayout = new QHBoxLayout();
    buttonLayout->addWidget(confirmButton);
    buttonLayout->addWidget(cancelButton);
    mainLayout->addLayout(buttonLayout);

    setLayout(mainLayout);

    connect(confirmButton, &QPushButton::clicked, this,
&CommandDialog::onConfirmButtonClicked);
    connect(cancelButton, &QPushButton::clicked, this, &QDialog::reject);
}

void CommandDialog::onConfirmButtonClicked() {
    emit commandEntered(commandLineEdit->text());
    close();
}
```

2. Binding the Shortcut Key in MyGame

Modify MyGame.cpp to include a shortcut that triggers the CommandDialog when the / key is pressed.

```
MyGame.h
#ifndef MYGAME_H
#define MYGAME_H

#include <QMainWindow>
#include <QGraphicsView>
#include <QShortcut>
#include "Scenes/BattleScene.h"
#include "CommandDialog.h"

class MyGame : public QMainWindow {
    Q_OBJECT

public:
    explicit MyGame(QWidget *parent = nullptr);

private slots:
    void showCommandDialog();
    void processCommand(const QString &command);

private:
    QGraphicsView *view;
    BattleScene *battleScene;
    QShortcut *cheatShortcut;
    CommandDialog *commandDialog;
};
```

```
#endif // MYGAME_H
```

```
MyGame.cpp
#include "MyGame.h"
#include <QDebug>

MyGame::MyGame(QWidget *parent) : QMainWindow(parent) {
    battleScene = new BattleScene(this);
    view = new QGraphicsView(this);
    view->setScene(battleScene);

    // Set the view's window size to 1280x720
    view->setFixedSize((int) view->scene()->width(), (int) view->scene()-
>height());
    view->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    view->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

    setCentralWidget(view);
    setFixedSize(view->sizeHint());

    battleScene->startLoop();

    // Initialize CommandDialog
    commandDialog = new CommandDialog(this);

    // Bind the "/" key to show the command dialog
    cheatShortcut = new QShortcut(QKeySequence("/"), this);
    connect(cheatShortcut, &QShortcut::activated, this,
&MyGame::showCommandDialog);

    // Connect the dialog's commandEntered signal to the processCommand slot
    connect(commandDialog, &CommandDialog::commandEntered, this,
&MyGame::processCommand);
}

void MyGame::showCommandDialog() {
    commandDialog->show();
}

void MyGame::processCommand(const QString &command) {
    qDebug() << "Cheat code entered:" << command;
    battleScene->spawnItem(command);  // Assuming spawnItem is a method in
BattleScene
}
```

3. Implementing spawnItem Method in BattleScene

Finally, create the spawnItem method in BattleScene that will generate items based on the cheat code.

Add the following slot in BattleScene.h:

```
public slots:
    void spawnItem(const QString &itemType);
BattleScene.cpp
```

Implement the spawnItem method in BattleScene.cpp:

```
#include "BattleScene.h"

void BattleScene::spawnItem(const QString &itemType) {
    Item *newItem = nullptr;

    if (itemType == "sword") {
        newItem = new MeleeWeapon();
    } else if (itemType == "arrow") {
        newItem = new Arrow();
    } else if (itemType == "shield") {
        newItem = new Armor();
    }

    if (newItem) {
        newItem->setPos(100, 100);  // Example position
        addItem(newItem);
        dropItems.append(newItem);
    } else {
        qDebug() << "Unknown cheat code:" << itemType;
    }
}
```

需求点5：近战武器（10%）

具体要求

- 能够渲染玩家装备的武器，有简单的攻击动画（2%）。 下以 `IronShortSword` 类的攻击为例

  在 `IronShortSword` 类中，代码片段定义了如何根据按键的状态来更新武器的位置。当按下攻击按键时，武器的位置会发生变化，从而实现攻击动画。

  ```
  void IronShortSword::attack() {
  //攻击函数
  if(pixmapItem != nullptr){
  setPos(-250,-170);
  }
  }

  void IronShortSword::attackStoped() {
  if (pixmapItem != nullptr) {
  ```

```
    setPos(-190, -170);  //设置近战武器的位置 适合铁短剑的位置
    pixmapItem->setPos(0, 0); //设置图片位置
}
}
```

- 玩家装备近战武器时，使用按键5可以进行攻击（2%）。
- 近战武器有多重类型（2%）：

  单手剑：朝角色面向方向进行攻击。

  矛：朝角色面向方向进行攻击，攻击距离更长。 （由于素材难找，我的项目用棍棒代替）

```
bool BattleScene::attackTrue(Character *attacker, Character *victim){
qDebug() << "Attacking!";
if(attacker->melee == nullptr){
    return false;
} else {
    if(attacker->melee->isTwoHanded == false){
        QPointF pos1 = attacker->pos();
        QPointF pos2 = victim->pos();
        qreal attackRange = attacker->melee->attackRange;

        // 计算攻击者与被攻击者的距离
        qreal distance = QLineF(pos1, pos2).length();

        if(distance < attackRange){
            // 计算攻击者朝向的向量
            QPointF directionVector = attacker->getDirection();
            QPointF vectorToVictim = pos2 - pos1;

            // 归一化向量
            vectorToVictim /= QLineF(pos1, pos2).length();

            // 计算两个向量的夹角的余弦值
            qreal dotProduct = QPointF::dotProduct(directionVector,
vectorToVictim);

            // 获取两个向量夹角的余弦值对应的弧度
            qreal angleRadians = std::acos(dotProduct);

            // 定义攻击弧的角度
            const qreal attackArcAngleRadians = M_PI / 3; // 60度对应的弧度

            // 判断夹角是否在定义的攻击弧内
            if(angleRadians <= attackArcAngleRadians){
                qDebug() << "Attacked!";
                return true; // 被攻击者在攻击者的前方弧内
            }
        }
    }
```

双手剑：按下攻击按键后间隔一段时间才会进行攻击，同时朝角色面向和背向的方向进行攻击。

```
else if(attacker->melee->isTwoHanded == true){
        // 双手剑的攻击延迟
        static QTimer *attackTimer = nullptr;
        if (attackTimer == nullptr) {
            attackTimer = new QTimer(this);
            attackTimer->setSingleShot(true);
        }

        if (!attackTimer->isActive()) {
            attackTimer->start(500);  // 假设延迟时间为500毫秒
            return false;  // 延迟期间不进行攻击判断
        }

        // 延迟结束后，计算是否命中
        QPointF pos1 = attacker->pos();
        QPointF pos2 = victim->pos();
        qreal attackRange = attacker->melee->attackRange;

        // 计算攻击者与被攻击者的距离
        qreal distance = QLineF(pos1, pos2).length();

        if (distance < attackRange) {
            // 计算攻击者朝向的向量（面向方向）
            QPointF directionVector = attacker->getDirection();
            QPointF vectorToVictim = pos2 - pos1;

            // 归一化向量
            vectorToVictim /= QLineF(pos1, pos2).length();

            // 计算两个向量的夹角的余弦值
            qreal dotProduct = QPointF::dotProduct(directionVector,
vectorToVictim);

            // 获取两个向量夹角的余弦值对应的弧度
            qreal angleRadians = std::acos(dotProduct);

            // 定义攻击弧的角度
            const qreal attackArcAngleRadians = M_PI / 3; // 60度对应的弧度

            // 判断夹角是否在定义的攻击弧内（面向方向）
            if (angleRadians <= attackArcAngleRadians) {
                qDebug() << "Attacked from the front!";
                return true;  // 被攻击者在攻击者的前方弧内
            }

            // 计算背向攻击：反转攻击方向
            QPointF reverseDirectionVector = -directionVector;

            // 计算背向攻击夹角
            dotProduct = QPointF::dotProduct(reverseDirectionVector,
vectorToVictim);
```

```
            angleRadians = std::acos(dotProduct);

            // 判断夹角是否在定义的攻击弧内（背向方向）
            if (angleRadians <= attackArcAngleRadians) {
                qDebug() << "Attacked from the back!";
                return true;  // 被攻击者在攻击者的背向弧内
            }
        }
    }
}
return false;
}
```

在 `BattleScene` 类中，代码片段定义了如何判断攻击是否命中以及如何计算伤害。攻击是否命中取决于攻击者和被攻击者之间的距离以及攻击者的朝向。如果攻击命中，会根据武器的元素类型和被攻击者的抗性来计算伤害。如果被攻击者处于冻结状态，则伤害会加倍。

```
void BattleScene::attackDone(Character *attacker, Character *victim) {
if (attacker != nullptr && victim != nullptr) {
    if (attacker->melee != nullptr) {
        int element = attacker->melee->element;  // 获取 element 属性

        // 检查并获取派生类的 element 值
        if (auto fireWeapon = dynamic_cast<FireSword*>(attacker->melee)) {
            element = fireWeapon->element;
        } else if (auto iceWeapon = dynamic_cast<IceSword*>(attacker-
>melee)) {
            element = iceWeapon->element;
        } else if (auto thunderWeapon = dynamic_cast<ThunderSword*>
(attacker->melee)) {
            element = thunderWeapon->element;
        }

        int damage = attacker->melee->damage;
        if (victim->beFrozen) {
            damage *= 2;  // 冻结状态下双倍伤害
            victim->beFrozen = false;
            victim->frozenEffect->setVisible(false);
            victim->setMovable(true);  // 恢复移动能力
            qDebug() << "Victim unfrozen.";
        }

        if (element == 0) {
            if (attackTrue(attacker, victim)) {
                victim->health -= damage;
            }
        } else if (element == 1) {
            if (attackTrue(attacker, victim)) {
                victim->health -= damage;
                if(victim->antiElement != 1 && victim->antiElement != 4 &&
victim->antiElement != 5 && victim->antiElement != 7){
```

```
                    victim->onFire = true;
                    victim->fireEffect->setVisible(true);
                    victim->startFireEffect();
                }

            }
        } else if (element == 2) {
            if (attackTrue(attacker, victim)) {
                victim->health -= damage;
                if(victim->antiElement != 2 && victim->antiElement != 4 &&
victim->antiElement != 6 && victim->antiElement != 7){
                    victim->beFrozen = true;
                    victim->frozenEffect->setVisible(true);
                    victim->startFrozenEffect();
                }
            }
        } else if (element == 3) {
            if (attackTrue(attacker, victim)) {
                victim->health -= damage;
                if(victim->antiElement != 3 && victim->antiElement != 5&&
victim->antiElement != 6 && victim->antiElement != 7){
                    victim->beThundered = true;
                    victim->electrocutedEffect->setVisible(true);
                    victim->startThunderEffect(this);
                }
            }
        }
    }
}
}
```

- 近战武器有两种材质：金属质和木质（2%）。 在每个武器中都有成员变量

```
int material
```

0表示无材质（适用于元素武器），1表示铁质，2表示木质

- 投掷近战武器：使用按键6可以将手中的近战武器扔出，武器会按照玩家朝向飞出产生一 个水平的初速度，并且受到重力的影响发生下坠。武器接触人物或者平台后产生一次攻击 效果并消失（2%）。

在 `BattleScene` 类中，代码片段定义了如何处理投掷近战武器。如果近战武器被投掷，会根据其速度和位置来判断是否命中敌人。如果命中，会根据武器的元素类型和敌人的抗性来计算伤害，并应用相应的效果。

```
void BattleScene::processMeleeThrow(MeleeWeapon* melee) {
if (melee != nullptr && melee->isVisible()) {
    QRectF throwAttackRect = QRectF(melee->pos().x(), melee->pos().y(),
-100, 200);
    if (melee->speed.x() < 0) {
```

```
            throwAttackRect = QRectF(melee->pos().x(), melee->pos().y(), 100,
200);
            qDebug() << "Throwing weapon left.";
        }

        if (melee->beThrown) {
            if (throwAttackRect.contains(rival->pos())) {
                applyMeleeEffect(melee, rival);
                delete melee;
                melee = nullptr;
            }
            if (throwAttackRect.contains(link->pos())) {
                applyMeleeEffect(melee, link);
                delete melee;
                melee = nullptr;
            }
        }
        if(melee){
            if (melee->isOnGround(melee) && melee->beThrown) {
                igniteBlockIfLanded(melee);
                delete melee;
                melee = nullptr;
            }
        }
    }
    }

    void BattleScene::applyMeleeEffect(MeleeWeapon* melee, Character* victim) {
    if (melee) {
        int damage = melee->damage;

        // 如果角色被冻结，造成双倍伤害并解除冻结状态
        if (victim->beFrozen) {
            damage *= 2;
            victim->beFrozen = false;
            victim->frozenEffect->setVisible(false);
            victim->setMovable(true);
        }

        // 检查并应用伤害
        victim->setHealth(victim->health - damage);

        // 根据 element 属性应用效果
        int element = melee->element;
        if (auto fireWeapon = dynamic_cast<FireSword*>(melee)) {
            element = fireWeapon->element;
        } else if (auto iceWeapon = dynamic_cast<IceSword*>(melee)) {
            element = iceWeapon->element;
        } else if (auto thunderWeapon = dynamic_cast<ThunderSword*>(melee)) {
            element = thunderWeapon->element;
        }

        switch (element) {
        case 1:  // 火属性
```

```
        if (victim->antiElement != 1 && victim->antiElement != 4 && victim-
>antiElement != 5 && victim->antiElement != 7) {  // 如果没有火焰抗性
            victim->onFire = true;
            victim->fireEffect->setVisible(true);
            victim->startFireEffect();
        }
        break;
    case 2:  // 冰属性
        if (victim->antiElement != 2 && victim->antiElement != 4 && victim-
>antiElement != 6 && victim->antiElement != 7) {  // 如果没有冰霜抗性
            victim->beFrozen = true;
            victim->frozenEffect->setVisible(true);
            victim->startFrozenEffect();
        }
        break;
    case 3:  // 雷属性
        if (victim->antiElement != 3 && victim->antiElement != 5&& victim-
>antiElement != 6 && victim->antiElement != 7) {  // 如果没有雷电抗性
            victim->beThundered = true;
            victim->electrocutedEffect->setVisible(true);
            victim->startThunderEffect(this);
        }
        break;
    default:
        break;
    }
}
}
```

```
void Character::throwWeapon() {
if(melee == nullptr){
    return;
}
else if (melee != nullptr) {
    //qDebug() << "Throwing weapon, melee is not null";
    melee->unmount();
    melee->setParentItem(parentItem());
    melee->setPos(QPointF(pos().x()+100, pos().y()-50));
    melee->startThrown();
    melee->beThrown = true;
    //qDebug() << "Setting melee pos to:" << pos();
    QPointF direction = getDirection();
    QPointF speed = direction * 0.2;
    if(direction.x()>0){
        melee->setTransform(QTransform().scale(-1, 1), true); // 设置武器的水
平翻转
    }
    melee->speed = speed;
    //qDebug() << "Setting melee speed to:" << speed;
    melee->downAcceleration = gravity.getGravity();
    //qDebug() << "Setting melee downAcceleration to:" <<
```

```
gravity.getGravity();
    melee = nullptr;
}
}
```

需求点6：弓和箭（10%）

具体要求

- 支持至少三种不同的弓，每个弓有不同射程，一次能射出的箭的数量不同（2%）。
- 弓有两种材质：金属质和木质（2%）。

  `Throwable` 类提供了接口来处理可投掷物品，包括弓和箭。这个类允许物品开始和结束投掷状态。

```
// Throwable.cpp 片段
void Throwable::endThrown() {
    thrown = false;
}

void Throwable::startThrown() {
    thrown = true;
}
```

  `Bow` 类有六个子类，是木质、铁质、普通弓、强弓、手弩的不同组合。手弩有不为0的iniSpeed，能给射出的箭更大的初速度。而如果是强弓，将会执行两次射箭。至此即可覆盖所有要求。

```
if (isThrowDown()) {
    if(melee != nullptr && melee->isVisible()){
        throwWeapon();
    }
    else if(bow != nullptr && bow->isVisible()){
        archery();
        if(bow->capacity == 2)
        {
            changeArrow();
            archery();
        }
    }
}
```

- 同时拥有近战武器和弓时，玩家可以使用按键7在近战武器和弓之间切换（2%）。

  在 `Character` 类中，代码片段定义了如何切换角色的武器。如果当前角色装备了近战武器和弓箭，玩家可以使用特定的按键来切换这两种武器。如果当前显示的是近战武器，切换到弓箭；如果当前显示的是弓箭，切换到近战武器。

```
void Character::changeWeapon() {
// 提前返回，避免后续判断
if (melee == nullptr || bow == nullptr) {
    return;
}

if (melee->isVisible()) {
    // 如果当前显示的是近战武器，切换到弓箭
    melee->setVisible(false);
    bow->setVisible(true);
    setChangeDown(false);
} else if (bow->isVisible()) {
    // 如果当前显示的是弓箭，切换到近战武器
    bow->setVisible(false);
    melee->setVisible(true);
    if(!arrows.isEmpty()){
        arrows[arrowNum]->setVisible(false);
    }
    setChangeDown(false);
}
}
```

- 天空中会掉落箭，使用按键4捡起箭后会增加可用弹容量。使用按键5在不同类型的箭之间切换，使用按键6射出箭。箭的飞行轨迹类似于射箭但是射程更短（初速度更大）（2%）。
- 能够渲染飞行中箭和投掷武器（2%）。

  在 `Character` 类中，代码片段定义了如何切换角色的箭矢和如何使用弓箭进行攻击。玩家可以使用特定的按键来切换当前选择的箭矢。当玩家准备攻击时，会从当前选择的箭矢中选择一个可见的箭矢，将其从角色手中射出，并设置为已射箭。

```
void Character::archery() {
if(bow == nullptr){
    return;
}
bool allNull = true;
Arrow* selected = nullptr;
for (Arrow* pointer : arrows) {
    if (pointer != nullptr) {
        allNull = false;
        break;
    }
}
if(allNull){
    return;
}
for(Arrow* arrow : arrows){
    if(arrow->isVisible() && arrow!=nullptr){
        selected = arrow;
        break;
```

```
    }
}
if(selected != nullptr){
    removeArrow(selected);
    selected->setRotation(45);
    //selected->setScale(0.3);
    selected->unmount();
    selected->setParentItem(parentItem());
    QPointF direction = getDirection();
    selected->setPos(QPointF(pos().x()+100, pos().y()-100));
    selected->startThrown();
    selected->beThrown = true;
    if(direction.x()>0){
        selected->setTransform(QTransform().scale(1, 1), true); // 设置武器的
水平翻转
    }
    if(direction.x()<0){
        selected->setTransform(QTransform().scale(-1, 1), true); // 设置武器
的水平翻转
    }
    selected->speed = direction * 0.5 + bow->iniSpeed;
    selected->downAcceleration = gravity.getGravity();

    isArrowFired = true;  // 标记为已射箭
}
}

void Character::changeArrow() {
//static int arrowNum = 0;  // 静态变量，用于追踪当前选择的箭矢

if (arrows.isEmpty()) {  // 如果箭矢列表为空，直接返回
    return;
}
// 隐藏当前箭矢
if (arrows[arrowNum] != nullptr) {
    arrows[arrowNum]->setVisible(false);
}

// 递增箭矢索引
arrowNum++;
if (arrowNum >= arrows.size()) {
    arrowNum = 0;  // 如果超过箭矢数量，重置为第一个箭矢
}

// 显示下一个箭矢
if (arrows[arrowNum] != nullptr) {
    arrows[arrowNum]->setParentItem(this);
    arrows[arrowNum]->mountToParent();
    arrows[arrowNum]->setVisible(true);
}

// 重置射箭状态，防止切换箭矢后自动射出
isArrowFired = false;
setChangeArrowDown(false);
```

```
}

void BattleScene::processArrowThrow(Arrow* arrow) {
if (arrow != nullptr && arrow->isVisible()) {
    QRectF throwAttackRect = QRectF(arrow->pos().x() + 50, arrow->pos().y(),
-100, 200);
    if (arrow->speed.x() < 0) {
        throwAttackRect = QRectF(arrow->pos().x() - 100, arrow->pos().y(),
-100, 200);
        qDebug() << "Throwing arrow left.";
    }

    if (arrow->beThrown) {
        if (throwAttackRect.contains(rival->pos())) {
            applyArrowEffect(arrow, rival);
            delete arrow;
            arrow = nullptr;
        }
        if (throwAttackRect.contains(link->pos())) {
            applyArrowEffect(arrow, link);
            delete arrow;
            arrow = nullptr;
        }
    }
    if(arrow){
        if (arrow->isOnGround(arrow) && arrow->beThrown) {
            igniteBlockIfLanded(arrow);
            delete arrow;
            arrow = nullptr;
        }
    }
}
}

void BattleScene::applyArrowEffect(Arrow* arrow, Character* victim) {
if (arrow) {
    int damage = arrow->damage;

    // 如果角色被冻结，造成双倍伤害并解除冻结状态
    if (victim->beFrozen) {
        damage *= 2;
        victim->beFrozen = false;
        victim->frozenEffect->setVisible(false);
        victim->setMovable(true);
    }

    // 检查并应用伤害
    victim->setHealth(victim->health - damage);

    // 根据 element 属性应用效果
    int element = arrow->element;
    if (auto fireArrow = dynamic_cast<FireArrow*>(arrow)) {
        element = fireArrow->element;
    } else if (auto iceArrow = dynamic_cast<IceArrow*>(arrow)) {
```

```
        element = iceArrow->element;
    } else if (auto thunderArrow = dynamic_cast<ThunderArrow*>(arrow)) {
        element = thunderArrow->element;
    }

    switch (element) {
    case 1:  // 火属性
        if (victim->antiElement != 1 && victim->antiElement != 4 && victim-
>antiElement != 5 && victim->antiElement != 7) {  // 如果没有火焰抗性
            victim->onFire = true;
            victim->fireEffect->setVisible(true);
            victim->startFireEffect();
        }
        break;
    case 2:  // 冰属性
        if (victim->antiElement != 2 && victim->antiElement != 4 && victim-
>antiElement != 6 && victim->antiElement != 7) {  // 如果没有冰霜抗性
            victim->beFrozen = true;
            victim->frozenEffect->setVisible(true);
            victim->startFrozenEffect();
        }
        break;
    case 3:  // 雷属性
        if (victim->antiElement != 3 && victim->antiElement != 5&& victim-
>antiElement != 6 && victim->antiElement != 7) {  // 如果没有雷电抗性
            victim->beThundered = true;
            victim->electrocutedEffect->setVisible(true);
            victim->startThunderEffect(this);
        }
        break;
    default:
        break;
    }
}
}

Arrow* Character::pickupArrow(Arrow* newArrow) {
// 假设你想保留原有的箭矢，这里直接添加新的箭矢到 arrows 中
newArrow->setParentItem(this); // 设置新箭矢的父节点为当前角色
newArrow->mountToParent(); //挂载新箭矢到父节点
arrows.append(newArrow); // 将新箭矢添加到 arrows 向量中
newArrow->setVisible(0);
return nullptr;
}

QVector<Arrow*> Character::removeAllArrows() {
    QVector<Arrow*> removedArrows = arrows; // 保存当前的箭矢
    arrows.clear(); // 清空 arrows 向量
    return removedArrows; // 返回被移除的箭矢
}

void Character::removeArrow(Arrow* selected) {
int removedIndex = arrows.indexOf(selected);
if (removedIndex != -1) {
```

```
    arrows.removeAt(removedIndex);
    if (arrowNum >= arrows.size()) {
        arrowNum = 0;  // 如果当前索引超出范围，重置为 0
    }
}
}
```

需求点7：火属性（5%）

具体要求

- 增加火属性的箭和火属性的近战武器。被这些武器攻击后进入着火状态（1%）。
- 玩家在着火物体周围会收到持续伤害，较长一段时间后恢复正常（1%）。
- 木质平台、木质武器（无论是否捡起）着火后会不断燃烧直至燃尽消失（1%）。
- 着火效果会有一定延时地在相互接触（距离小于一个阈值）的木质的平台、物品之间传播（1%）。
- 显示物品、地形和人物的燃烧效果（1%）。

  人物和可燃的砖块都有初始的挂载于其上的火焰效果，初始设置为不可见。如果被火焰属性攻击就会可见，之后开始执行对应效果。

```
void Character::initEffects() {
fireTimer = new QTimer(this);
freezeTimer = new QTimer(this);
thunderTimer = new QTimer(this);

QGraphicsScene::connect(fireTimer, &QTimer::timeout, [this]() {
    this->updateHealth(1);  // 火焰每秒减少1点血量
});

QGraphicsScene::connect(freezeTimer, &QTimer::timeout, [this]() {
    this->beFrozen = false;
    this->frozenEffect->setVisible(false);  // 冻结结束，隐藏特效
    freezeTimer->stop();
});

QGraphicsScene::connect(thunderTimer, &QTimer::timeout, [this]() {
    this->updateHealth(1);  // 雷击每秒减少1点血量
});
}

void Character::startFireEffect() {
fireTimer->start(1000);
if (melee != nullptr && melee->material == 2) {  // 如果是木质武器
    delete melee;
    melee = nullptr;
}
QTimer::singleShot(5000, [this]() {
```

```
    this->onFire = false;
    this->fireEffect->setVisible(false);
    this->fireTimer->stop();
});
}
```

传火

```
void BattleScene::spreadFire(int i, int j) {
if (i < 0 || i >= 9 || j < 0 || j >= 16 || blocks[i][j] != 4) {
    return;  // 确保只处理石头砖块
}
qDebug() << "Spreading fire to block (" << i << ", " << j << ")";
auto stoneBlock = dynamic_cast<Stoneblock*>(blockGrid[i][j]);
if (stoneBlock == nullptr || stoneBlock->fire == nullptr || stoneBlock-
>fire->isVisible()) {
    return;  // 如果砖块已经燃烧或不是石头，退出
}

// 设置火焰可见
stoneBlock->fire->setVisible(true);

// 延时扩散火焰到相邻的石头砖块
QTimer::singleShot(1000, [this, i, j]() {
    //spreadFire(i + 1, j);
    //spreadFire(i - 1, j);
    spreadFire(i, j + 1);
    spreadFire(i, j - 1);
});

// 延时删除燃尽的砖块
QTimer::singleShot(5000, [this, i, j]() {
    if (blockGrid[i][j] != nullptr) {  // 确保砖块仍然存在
        delete blockGrid[i][j];
        blockGrid[i][j] = nullptr;
        Item::blocks[i][j] = 0;
    }
});
}


void BattleScene::igniteBlockIfLanded(Item* item) {
int blockX = static_cast<int>(item->pos().x()) / 80;
int blockY = static_cast<int>(item->pos().y()) / 80;
if (blockGrid[blockY][blockX] != nullptr && blocks[blockY][blockX] == 4) {
    //blockGrid[blockY][blockX]->fire->setVisible(true);
    spreadFire(blockY, blockX);
}
}
```

需求点8 冰属性（5%）

具体要求

- 增加冰属性的箭和冰属性的近战武器。被这些武器攻击后进入冰冻状态（1%）。
- 玩家在冰冻状态下不能进行任何操作，较长一段时间后恢复正常（1%）。
- 在冰冻状态下受到攻击会收到双倍伤害，并立即解除冰冻状态（2%）。
- 显示人物的冰冻效果（1%）。

人物有初始的挂载于其上的冰冻效果，初始设置为不可见。如果被冰属性攻击就会可见，之后开始执行对应效果。

在每次输入都检测人物beFrozen的值，如果为真则这次输入不会改变任何事件。

碎冰双倍伤害可见上面的攻击、射箭、投掷中的damage值的确定。

```
void Character::startFrozenEffect() {
    freezeTimer->start(3000);  // 3秒后解除冰冻
    this->setMovable(false);   // 冰冻期间不能移动
    QTimer::singleShot(3000, [this]() {
        this->setMovable(true);  // 3秒后恢复移动能力
    });
}
```

需求点9：雷属性（5%）

具体要求

- 增加电属性的箭和电属性的近战武器。被这些武器攻击后进入触电状态（1%）。
- 玩家在触电状态下会收到持续伤害，一小段时间后恢复正常（1%）。
- 通电效果会快速在相互接触（距离小于一个阈值）的金属质的平台、物品（无论是否拿在 手上）之间传播（1%）。
- 人物触电后，如果他正手持一个金属质的武器，武器会掉到地上，需要重新捡起（1%）。
- 显示物品、地形的通电效果和人物的触电效果（1%）。

  人物和可导电的砖块都有初始的挂载于其上的通电效果，初始设置为不可见。如果被电属性攻击就会可见，之后开始执行对应效果。

```
void Character::startThunderEffect(BattleScene* scene) {
thunderTimer->start(1000);  // 每秒触发一次雷击伤害
qDebug() << "Thunder effect started";

if (melee != nullptr && melee->material == 1) {  // 如果角色持有金属武器
    qDebug() << "Metal weapon detected, unmounting melee weapon";
    melee->unmount();
    melee->setParentItem(parentItem());
```

```
    melee->setPos(QPointF(pos().x() + 100, pos().y() - 50));
}

// 传播电击效果到铁砖块
scene->spreadThunderEffect(this);

QTimer::singleShot(5000, [this]() {  // 5秒后停止雷击效果
    this->beThundered = false;
    this->electrocutedEffect->setVisible(false);
    this->thunderTimer->stop();
});
}

void BattleScene::spreadThunderEffect(Character* character) {
QPointF pos = character->pos();
int x = pos.x() / 80;
int y = pos.y() / 80;

// 检查角色是否站在铁砖块上
if (blocks[y][x] == 3 && blockGrid[y][x]->electrocuted != nullptr) {
    // 设置当前铁砖块的雷电效果为可见
    blockGrid[y][x]->electrocuted->setVisible(true);

    // 向左和向右传播雷电效果
    propagateThunderEffect(x, y, -1); // 向左传播
    propagateThunderEffect(x, y, 1);  // 向右传播
}
}

void BattleScene::propagateThunderEffect(int startX, int y, int direction) {
int x = startX + direction;
while (x >= 0 && x < 16) {
    // 检查是否为铁砖块
    if (blocks[y][x] == 3 && blockGrid[y][x]->electrocuted != nullptr) {
        blockGrid[y][x]->electrocuted->setVisible(true);
    } else {
        break;  // 不是铁砖块，停止传播
    }
    x += direction;
}
}
```

需求点10：多种盔甲（5%）

具体要求

- 玩家可以捡起随机掉落的盔甲替换当前装备的盔甲。盔甲分为头部、上身和下身三部分。设计至少三种盔甲，分别提供对火、冰、电的免疫效果（5%）。

  `FlamebreakerArmor` 类是 `Armor` 类的派生类，表示火焰护甲。它在构造函数中设置了抗性为火焰抗性。

```
// FlamebreakerArmor.cpp 片段
FlamebreakerArmor::FlamebreakerArmor(QGraphicsItem *parent) : Armor(parent,
":/Items/Armors/FlamebreakerArmor/BotW_Flamebreaker_Armor_Icon.png") {
    antiElement = 1; //火焰抗性
}
```

`IceArmor` 类是 `Armor` 类的派生类，表示冰霜护甲。它在构造函数中设置了抗性为冰霜抗性。

```
// IceArmor.cpp 片段
IceArmor::IceArmor(QGraphicsItem *parent) : Armor(parent,
":/Items/Armors/IceArmor/IceArmor.png") {
    antiElement = 2; //冰抗性
}
```

`OldShirt` 类是 `Armor` 类的派生类，表示旧衬衫。它在构造函数中设置了抗性为无抗性。

```
// OldShirt.cpp 片段
OldShirt::OldShirt(QGraphicsItem *parent) : Armor(parent,
":/Items/Armors/OldShirt/BotW_Old_Shirt_Icon.png") {
    antiElement = 0; //无抗性
}
```

`ThunderArmor` 类是 `Armor` 类的派生类，表示雷电护甲。它在构造函数中设置了抗性为雷电抗性。

```
// ThunderArmor.cpp 片段
ThunderArmor::ThunderArmor(QGraphicsItem *parent) : Armor(parent,
":/Items/Armors/ThunderArmor/Thunder_Armor.png") {
    antiElement = 3; //雷抗性
}
```

头部和腿部装备可以以此类推。antiElement的值，从0到3将分别对应无抗性、火抗性、冰抗、雷抗。

最终映射到人物抗性

```
void Character::updateAntiElement(){
int combinedResistance = 0;

// 直接读取装备的抗性值并组合它们
if (headEquipment) {
    combinedResistance |= headEquipment->antiElement;
}
if (legEquipment) {
    combinedResistance |= legEquipment->antiElement;
}
```

```
if (armor) {
    combinedResistance |= armor->antiElement;
}

// 根据组合后的抗性值更新角色的抗性
switch (combinedResistance) {
case 0: antiElement = 0; break; // 无抗性
case 1: antiElement = 1; break; // 抗火
case 2: antiElement = 2; break; // 抗冰
case 3: antiElement = 3; break; // 抗雷
case 4: antiElement = 4; break; // 有火有冰
case 5: antiElement = 5; break; // 有火有雷
case 6: antiElement = 6; break; // 有冰有雷
case 7: antiElement = 7; break; // 有火有冰有雷
}
}
```

参考文献、引用代码出处

Qt框架：本项目使用了Qt6的Core、Gui、Widgets模块，进行UI构建、事件处理、以及图形界面的呈现。

CMake构建系统：通过CMakeLists.txt定义了项目的构建过程，包含所有源文件、资源文件的管理，以及依赖库的链接。

说明

由于图片均在本地，难以获取其URL，因此没能在本文档中添加图片。您可在assets文件夹自行查找相应的图片。

感谢老师助教们的指导，感谢李泽诚同学提供的图片素材。