中国海洋大学

OCEAN UNIVERSITY OF CHINA

《程序设计基础实践课程报告》

课程设计报告

题　　目：	MUD 游戏 Potter 2077 的规划与制作
上课时间：	周一 1-4 周二 5-8
授课教师：	张树刚
姓　　名：	洪佳荣 薛全笑 高翊航
组　　别：	第七组
日　　期：	2023.9.14

MUD 游戏 Potter 2077 的规划与制作

1 概览

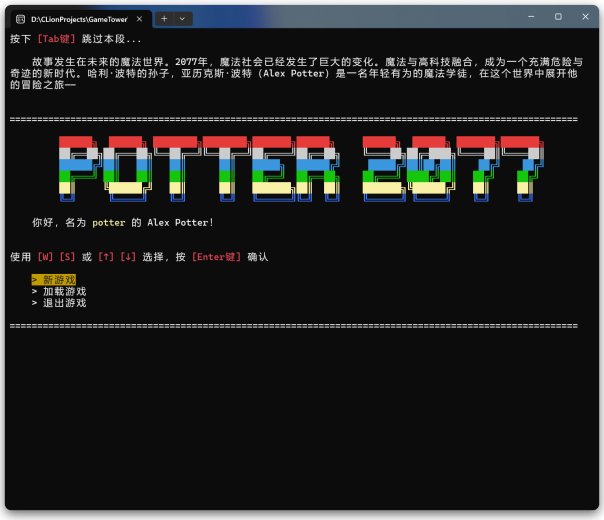
1.1 故事背景

故事发生在未来的魔法世界。2077年，魔法社会已经发生了巨大的变化。魔法与高科技融合，成为一个充满危险与奇迹的新时代。哈利·波特的孙子，亚历克斯·波特（Alex Potter）是一名年轻有为的魔法学徒，在这个世界中展开他的冒险之旅。

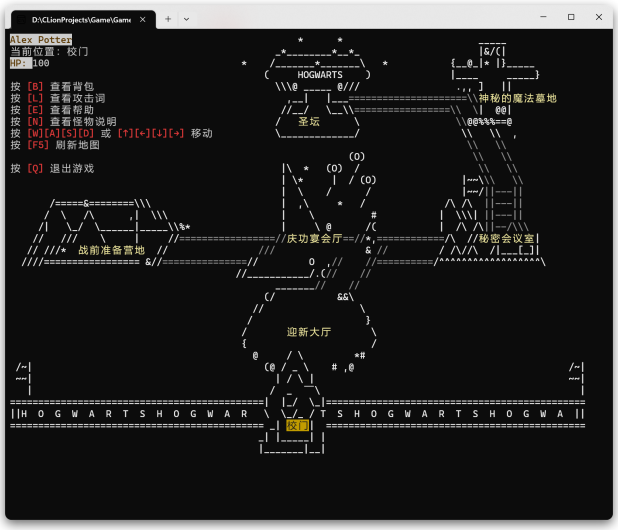
1.2 游戏玩法

玩家通过键盘操作前往地图中的不同区域，根据进度需要参与游戏设定的战斗场景。战斗过程中，玩家通过打 Boss 区域的词进行防御，打玩家区域的词进行攻击；当 Boss 区域的词触底的时候，玩家受到伤害。为辅助玩家进行游戏进程，玩家在战斗过程中可以使用道具辅助战斗。若战斗获胜，则游戏进程前进，玩家可通过解锁的剧情进一步了解整个故事；若战斗失败，玩家则会回到战斗前的状态。玩家的目标是通过对战，揭示一个故事的谜团。

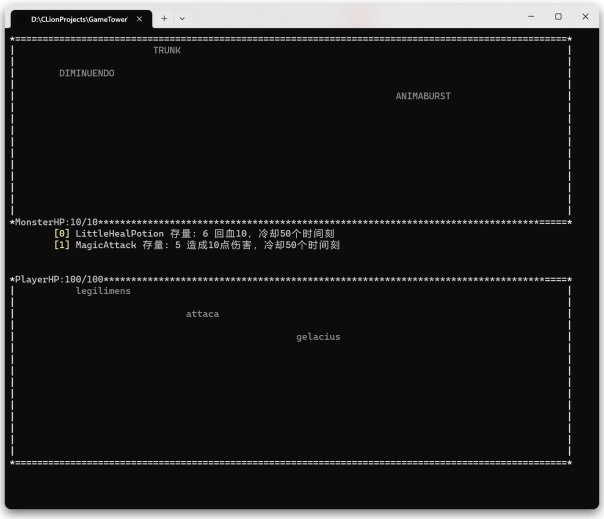
1.3 游戏截图



游戏开始界面



游戏地图



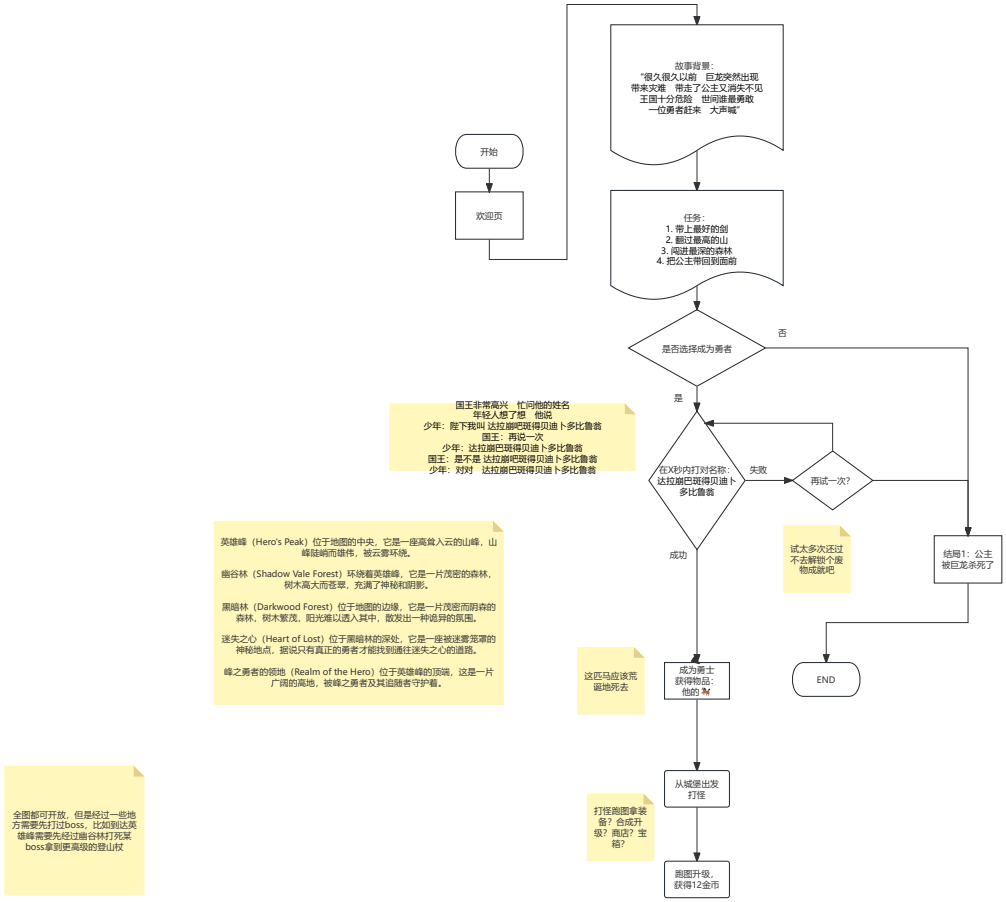
游戏战斗界面

游戏致谢页面

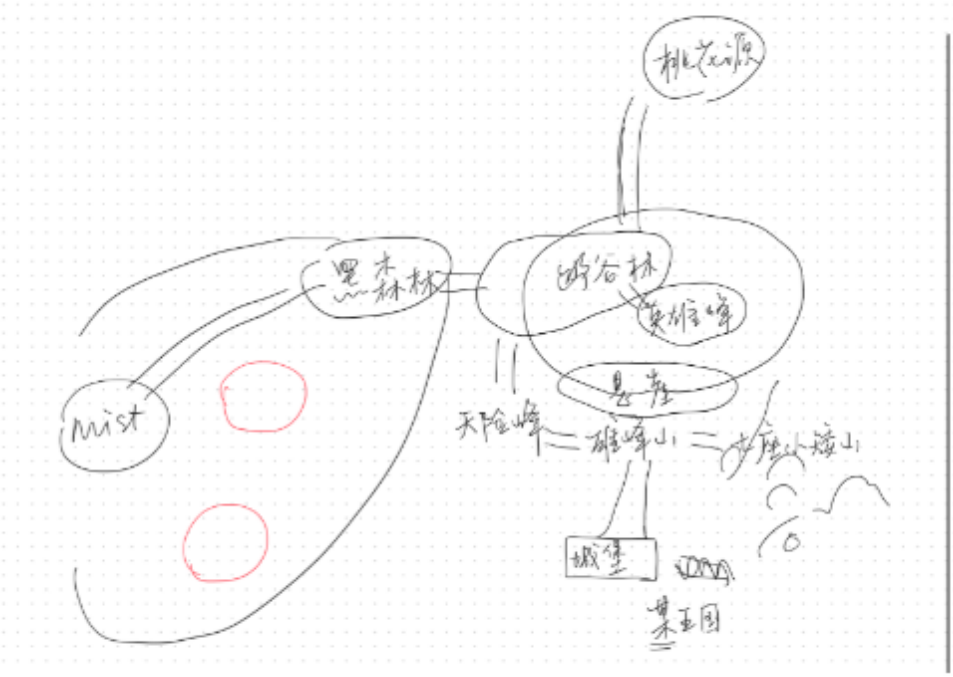
2 分析

2.1 选择游戏的类型和题材

开发初期，本小组选定的方向也是一个“骑士救公主”类型的游戏，很快便联想到了前些年大火的歌曲《达拉崩吧》的背景故事——公主被恶龙抓走，骑士勇夺十二金币和恶龙对决救下公主。但如此类型的 RPG 游戏容易出现同质化的现象。于是，经过分析和《达拉崩吧》复杂的人名带来的灵感，我们根据 MUD 游戏纯文字的特点，就文字本身提出了类“金山打字通”的游戏玩法，而游戏背景则设定在《达拉崩吧》构建的童话世界。



游戏初期规划流程图



游戏初期规划地图

经过进一步分析，在控制台中打汉字难度较大，还需要考虑到编码问题。于是，此时游戏背景被进一步修改为英文环境下的《达拉崩吧》。联想到《哈利波特》的咒语和《达拉崩吧》的汉字一样充满着无序，彼时《霍格沃兹之遗》也正流行，我们将背景修改为西方的魔法世界。游戏名 *Potter 2077* 缝合了《哈利波特》和《赛博朋克2077》两款大作的名称。

经过上述阶段的分析，我们的游戏确定为发生在未来的魔法世界中的打字游戏。

2.2 开发需求分析

由于我们最先产生新想法是战斗形式，我们先对战斗界面和详细的战斗方式有了一定的构想。参考了金山打字通的打字游戏，在玩家打出字时应该有一定的反馈，比如对打出的字染色；然后在词语被打对时，应该有攻击效果。战斗的终点必然是玩家或者怪物任意一方血量降为 0。上面构想了玩家的攻击方式，而对于怪物，为了有更直观的“压迫感”，我们也参考了一些弹幕游戏和经典的“俄罗斯方块”，让怪物的词下落触底时对玩家造成伤害。当怪物被击败时，玩家自然进入下一阶段；而当怪物击败玩家时，玩家被恢复成原状态，进而准备下一次挑战。

进而，战斗界面的构想也产生了。初期我们希望像一些类宝可梦系游戏一样，左右分屏，分别代表怪物区域和玩家区域。了解了命令行按行输出的特点，左右分别输出在开发上可能有一定的困难，于是我们修改成了上下分屏。在中间加入物品区和血条区。

词语落下的时间是固定的，因此，在单位时间内能造成的伤害便是衡量游戏难易程度的一个指标。要提高单位时间伤害，之于玩家自身有着打字熟练度的差异，因此第一关的教学关卡不能设计得太难；往后要让玩家在游玩游戏的短时间内提升自身打字水平显然不太现实，因此需要根据不同阶段控制难度。控制难度的方法可以有控制词语下落速度、控制玩家攻击点数，经过综合考虑，我们决定使用后者，同时把攻击数绑定在词语本身的属性上，通过不同阶段给予不同难度、伤害的词汇来使玩家获得“升级”。

所以，对于一个战斗场景类，它应该有一个怪物属性，毕竟怪物只在战斗场景出现。然后是场景打字方法的实现。而战斗场景是否要和地点绑定，经过思考，得出的结论是战斗场景应该作为一个独立的场景，而地图地点只作为移动和剧情进行的载体。但为了和地点对应，它仍应该有一个坐标属性或者名称属性。

BOSS攻击区

打此处的词使该词消失以防御
若此处有词触底则对玩家造成伤害

BOSS HP 100/100

物品1
物品2

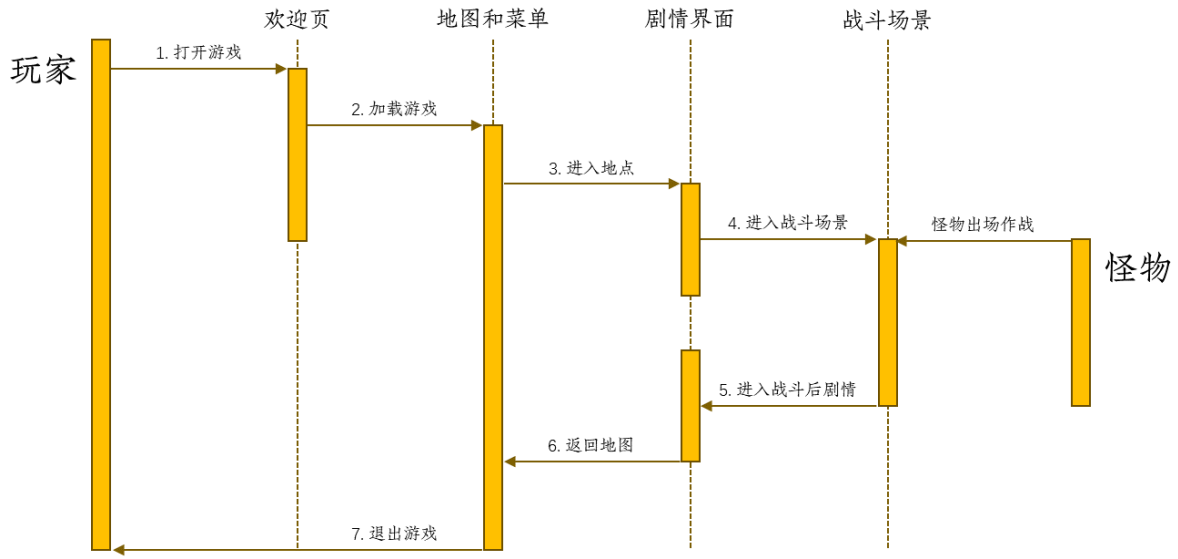
Player HP 100/100

玩家攻击区

打此处的词使对怪物造成伤害

战斗界面构想

跳出战斗形式的分析，按照我们的构想，当玩家在游玩一款游戏时，大致会有下图的一个流程：

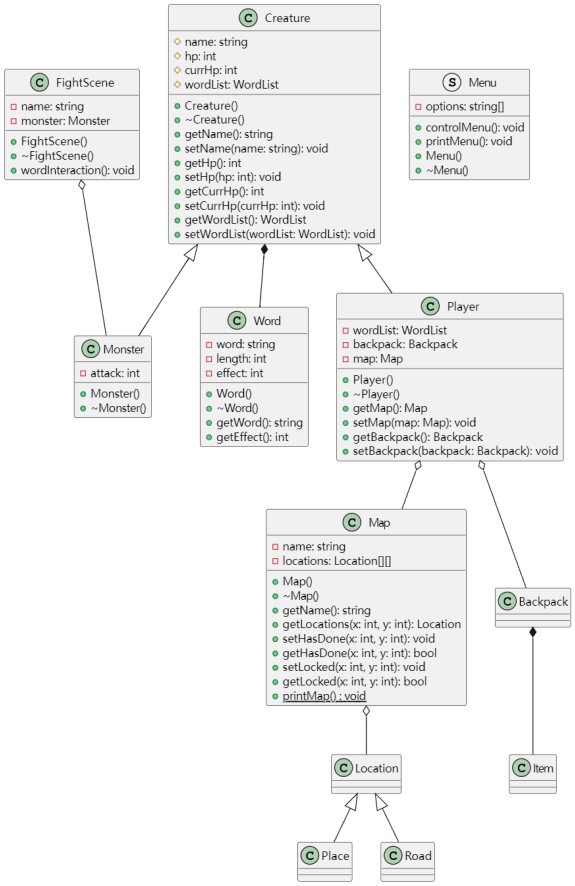


游戏大致流程

对上图进行分析，我们可以先确定“玩家”和“怪物”两个对象。考虑到玩家和怪物是同时进行相互对战的，它们都具有血量属性，因此我们让玩家和怪物都继承自同一“生物”基类。不同的玩家可能有不同的剧情进度、不同的地图探索度，因此地图也应该是玩家携带的一个属性。接着，玩家进入地图上的地点，开始战斗。根据我们特殊的战斗形式，无论是玩家或是怪物都应该有一个自己的“词库”，因此词库也应该是生物类的一个属性；但怪物会给玩家造成伤害，而玩家的伤害是绑定在自身词汇库上的，因此对于怪物需要单独有一个造成伤害值的属性。

然后是场景。场景可以大致分为“需要互动”和“不需要互动”两类。对于前者，经过分析可以把菜单、地图、战斗归为该类；而对于后者，可以把欢迎页、剧情、致谢页面、状态页面归为该类。更深入地想，一个界面实际上也可以细分成“需要互动”和“不需要互动”的两类组件。比如，地图的底图是不需要互动的，而在地图上切换地点这一操作才是属于地图需要互动的一部分。因此，对于前者需要互动的一类场景，在设计其类的时候可以单独设计一个静态函数用于输出界面，而其他的互动操作则单独设计成员函数或者非成员函数。

经过分析，我们可以大致画出一个通过 UML 图表示的一个 Use-Case 用例图，如下：



通过UML类图表示的Use-Case用例图

由于剧情和数值在分析期并不完全完善，我们先进行了游戏初步的开发。

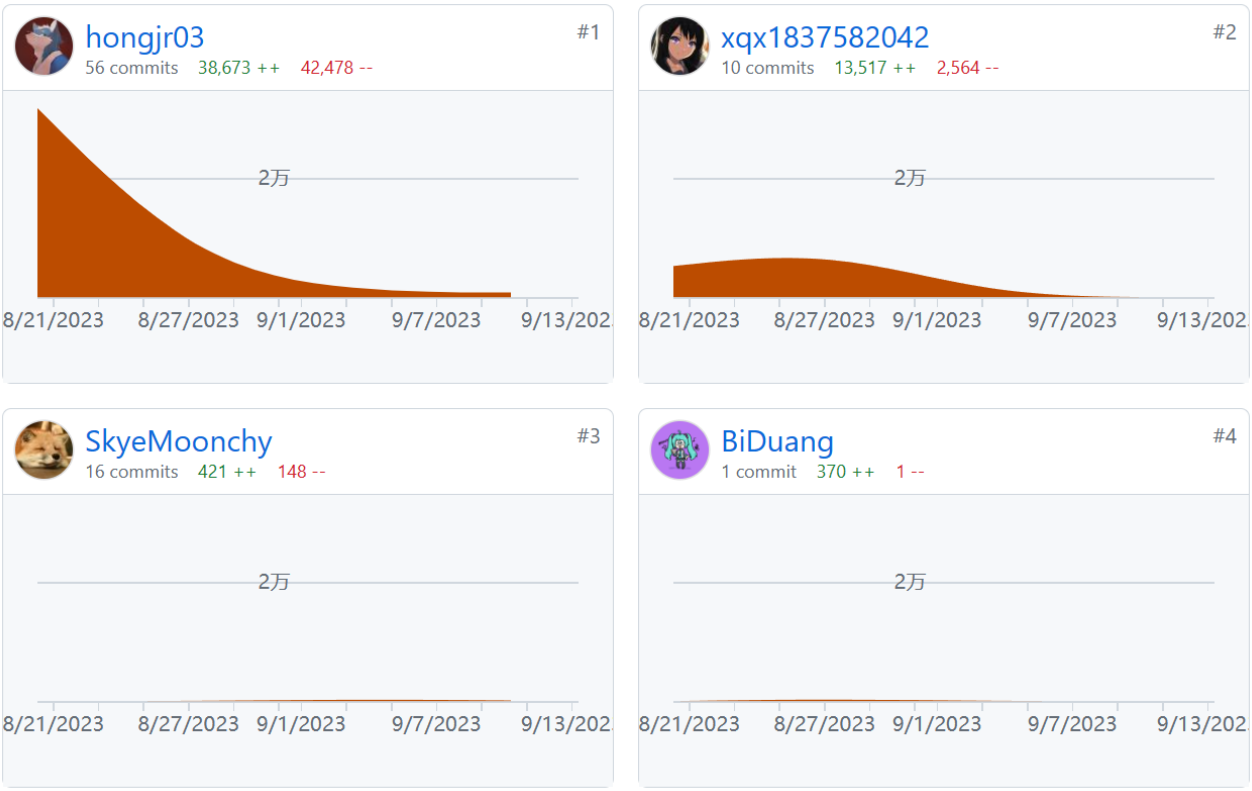
3 开发

3.1 版本控制

我们在 GitHub 上开源了本项目，并使用 Git 作为我们的版本控制工具。

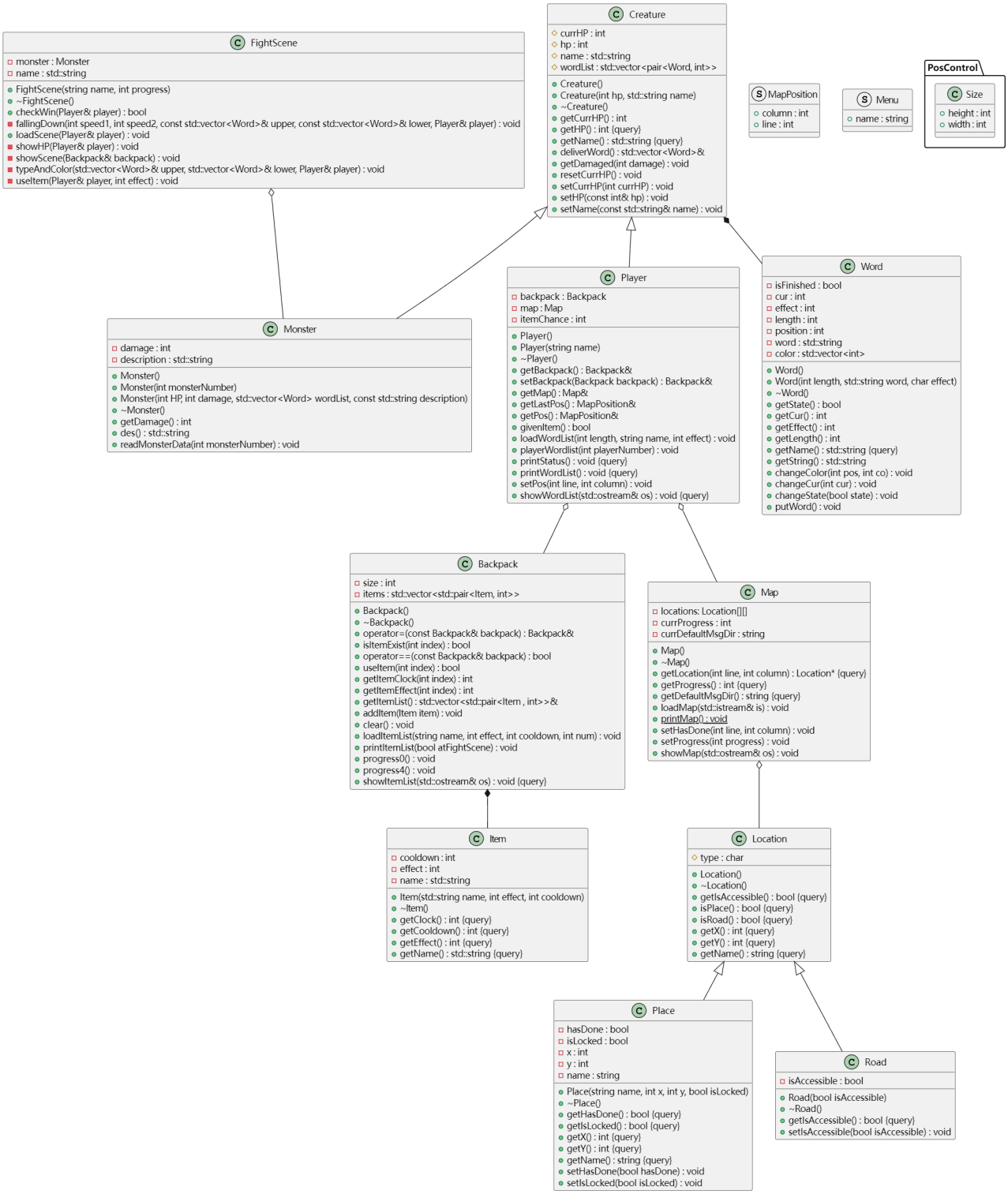
3.2 成员分工

- 组长：洪佳荣
  - 负责界面设计、存档、玩家行为、地图、进度管理，参与了战斗场景的思路设计和使用道具的实现
- 组员：薛全笑
  - 负责玩家、怪物类的完善，负责战斗的实现，包括打字判定和实时改色等
- 组员：高翊航
  - 负责剧情细节、词库和数值的编写，协助完善了战斗场景



GitHub 上的团队代码提交量图标

3.3 UML 类图



整个项目的UML类图

3.4 使用的设计模式

在开发过程中，由于经验匮乏，我们使用的设计模式并不规范。在设计几个主要的类时，我们使用了组合的设计模式以及模板方法的设计模式。

3.5 界面的开发

玩家进入游戏，第一个需要接触的就是界面，因此界面是否易懂、方便操作便成了玩家游玩体验在时间上的第一要素。

3.5.1 PosControl命名空间（属于 `Interface.h`）

对于一些复用程度高的控制光标的函数和清行的函数，我们单独写入了 `PosControl` 命名空间，防止和系统库函数冲突也方便调用。

```
    struct Size {
        int width;
        int height;
    } size = { .width: 1020,  .height: 860};

    void centerWindow() {
        HWND hwnd = GetForegroundWindow();
        SetWindowPos(hwnd,  hWndInsertAfter: HWND_TOPMOST,  X: 0,  Y: 0,  cx: size.width,  cy: size.height,  uFlags: 1);
        MoveWindow(hwnd,  X: (screen_width - size.width) / 2,  Y: (screen_height - size.height) / 2,  nWidth: size.width,  nHeight: size.height,
                   bRepaint: 1);
    }

// 回到坐标位置，坐标需要给定
    void setPos(int x, int y) {
        COORD coord{ .X: static_cast<SHORT>(y),  .Y: static_cast<SHORT>(x)};
        SetConsoleCursorPosition( hConsoleOutput: GetStdHandle( nStdHandle: STD_OUTPUT_HANDLE),  dwCursorPosition: coord);  //回到给定的坐标位置进行重新输出
    }


// 获取当前标准输出流位置
    void getPos(int &x, int &y) {
        CONSOLE_SCREEN_BUFFER_INFO b;           // 包含控制台屏幕缓冲区的信息
        GetConsoleScreenBufferInfo( hConsoleOutput: GetStdHandle( nStdHandle: STD_OUTPUT_HANDLE),  lpConsoleScreenBufferInfo: &b);    //  获取标准输出句柄
        y = b.dwCursorPosition.X;
        x = b.dwCursorPosition.Y;
    }


    void HideCursor() {
        CONSOLE_CURSOR_INFO cursor;
        cursor.bVisible = FALSE;
        cursor.dwSize = sizeof(cursor);
        HANDLE handle = GetStdHandle( nStdHandle: STD_OUTPUT_HANDLE);
        SetConsoleCursorInfo( hConsoleOutput: handle,  lpConsoleCursorInfo: &cursor);
    }
```

PosControl命名空间的函数

3.5.2 欢迎页面（属于 `Interface.h`）

考虑到这个层面，我们设计了一个很有意思的欢迎页。欢迎页主要由游戏 LOGO、菜单和背景剧情几个元素构成。对于游戏 LOGO，我们使用了 ASCII ART 生成工具 生成了美观的游戏 LOGO。考虑到游戏所谓“2077”的背景，我们决定加入一个文字跑马灯效果，伪代码实现如下：

```
获取显示标题前的光标坐标
while (1) {
    定位光标到显示标题前的坐标
    读取 logo 文件
    while (逐行读取 logo) {
        随机取一个颜色并染色
        Sleep(50);
        cout << logo << endl;
    }
}
```

同时，在标题循环显示后，需要给玩家继续下一步的提示。在这里，我们自己实现了一个按 `Enter` 键继续的效果，代码实现如下：

```
if (kbhit()) { // kbhit() 或 _kbhit() 函数仅在按下键盘时为真
    char c;
    c = getch();
    if (c == '\r' || c == '\n') {
        break;
    }
}
```

同理，也能实现按`Tab`跳过剧情。

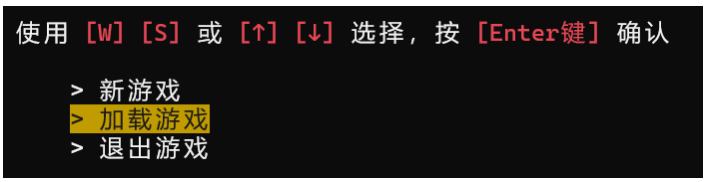
在开发过程中，我们发现，在不同的电脑上显示欢迎页面效果并不相同。为了解决这个问题，我们先后尝试了固定窗口分辨率（这个做法我们保留到了最后，但没能解决这个问题）和强制调整玩家电脑分辨率的做法，最后发现这一个“并不相同”造成的原因主要是因为 Windows 系统下的 DPI 缩放的关系。经过测试，在同一 DPI 下游戏界面显示没有异常。我们也上 GitHub 寻找了一些开源工具或者代码，试图“帮”玩家修改电脑的 DPI，比如 imniko/SetDPI 工具。但发现尽管 DPI 能被改对，最后倘若玩家没有以合法方式退出游戏（在游戏内退出而非强制退出），DPI 将无法恢复。于是我们最后决定在限制窗口分辨率大小的前提下，加入提示玩家缩放窗口文字大小的文字，以便玩家获得最好的游戏体验。

接着进入的是游戏的“注册”界面。考虑到存档文件以游戏玩家的用户名为名，而 Windows 下有些字符不能作用户名，我们使用了类似“白名单”的机制限制玩家输入的字符。

```
cout << "\33[2K";
cout << "请输入你的名字：";
cin >> name;
while (name.length() > 15 ||
       name.find_first_not_of( s: "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_") !=
       string::npos) {
    PosControl::setPos(x,  y: 0);
    // 清除当前行
    cout << "\33[2K" << endl;
    // 根据Windows文件名允许用的字符，判断name是否合法
    if (name.find_first_not_of( s: "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_") !=
        string::npos) {
        cout << "名字中含有非法字符，请重新输入：";
    } else {
        cout << "名字过长，请重新输入：";
    }
    cin >> name;
}
```

玩家创建时输入用户名处的代码

然后是选择界面，玩家应该在这时候选择是否新建游戏或者加载游戏，又或者退出游戏。



欢迎界面上的菜单

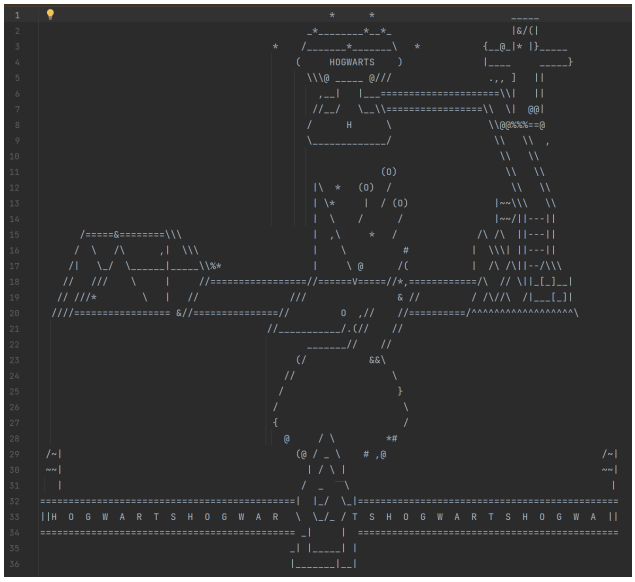
显然，最直观的想法就是使用一个数字编号在前，提示玩家输入数字编号回车选择的一个普通的菜单。但这样的菜单显然有着不够直观的问题。于是我们做了一个通过染色对应位置显示当前选中项的菜单，并且它是可以循环的。原理可以见以下伪代码：

```
while (1) {
```

```
        将坐标设置到上一个选项的位置
        复位上一个选项的染色
        将坐标设置到当前选中选项的位置
        高亮当前选项
        c = getch();
        switch (c) {
            case 'w':
            case 'W':
            case 72: // 72是上箭头
                prevChoice = choice;
                choice = (choice + length - 1) % length; // 循环的核心在于取模
                break;
            case 's':
            case 'S':
            case 80: // 80是下箭头
                prevChoice = choice;
                choice = (choice + 1) % length;
                break;
            case '\r':
            case '\n':
                return choice;
            default:
                prevChoice = choice;
                break;
        }
    }
```

3.5.3 地图界面（属于 `Map.h`，交互属于 `Behavior.h`）

地图界面其实就是将菜单的维度从一维拓展到了二维。但与菜单不同的是，它应当有更形象的界面来使玩家更具所谓“沉浸感”。于是在设计地图上，我们使用了字符画作为底图。为了让道路和地图的颜色有所区分，我们单独把道路拉出作为一个图层单独染色。



地图底图

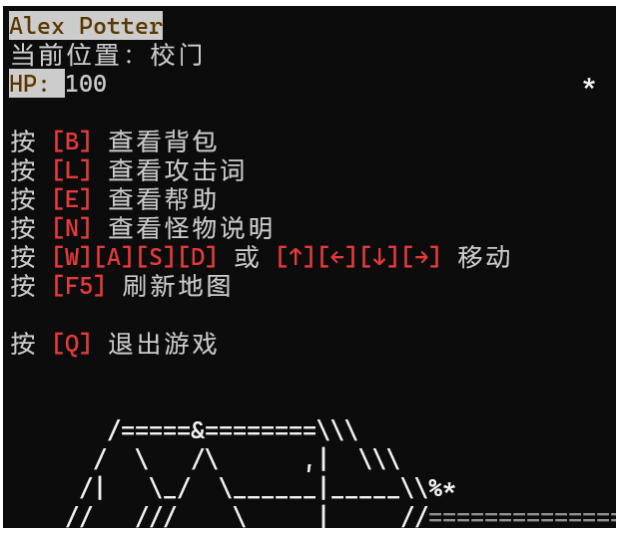
道路

通过在初始化地图时初始化各个地点的光标坐标，就能实现类似于菜单一样的移动操作。如以下初始化代码：

```
locations[8][4] = new Place("校门", 34, 50, false); // false 表示初始化为未上锁地点
locations[7][4] = new Road(true); // true 表示该路可通
locations[6][4] = new Place("迎新大厅", 26, 50);
locations[5][4] = new Road(true);
locations[4][4] = new Place("庆功宴会厅", 18, 50);
locations[3][6] = new Road(true);
locations[2][4] = new Place("圣坛", 8, 52);
locations[4][2] = new Place("战前准备营地", 19, 13);
locations[4][3] = new Road(true);
locations[4][5] = new Road(true);
locations[4][6] = new Place("秘密会议室", 18, 84);
locations[2][5] = new Road(true);
locations[2][6] = new Place("神秘的魔法墓地", 6, 84);
```

在游戏开发的初期，我们本来想通过限制路的“可通过性”来限制玩家的活动范围以达到分阶段的目的。后来考虑到地图本身并不复杂，所以没有将这个特点继续使用，但保留在了代码中。

画出地图后，我们意识到地图的左上角其实很空旷。于是继续沿用叠加图层的想法，我们在上面继续叠加了一层玩家状态和辅助说明的文字。



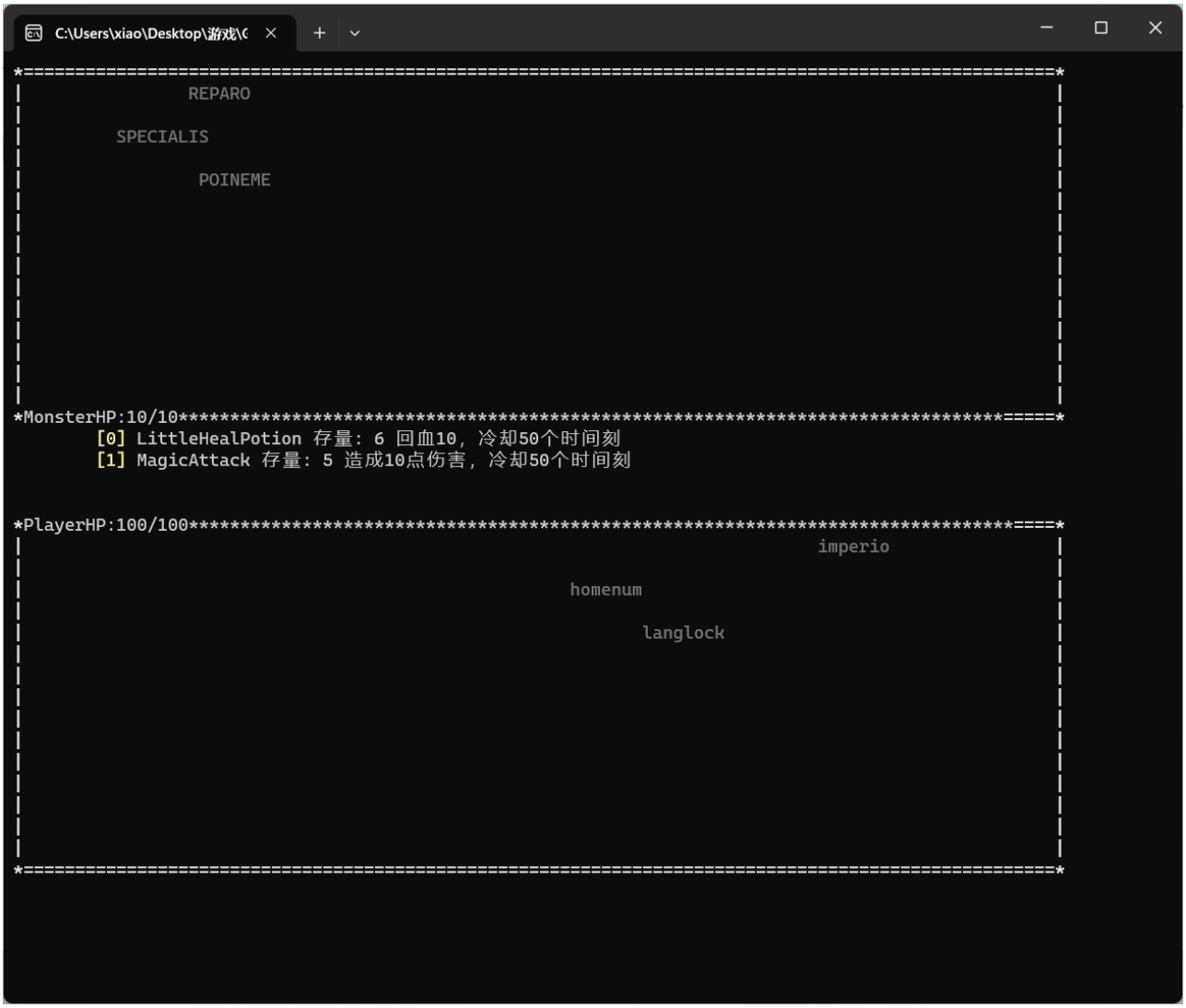
地图界面上的玩家状态

而地图需要有根据对应当前位置匹配的读剧情或进入战斗场景的功能，共同点是它们都通过`Enter`触发。根据不同的剧情需要，我们可以通过当前位置对应的`Place`的是否完成属性和是否上锁属性确定进度。考虑到“移动”是玩家的行为而非地图的行为，地图类中并没有实现`move()`函数。起初`move()`是交给玩家的一个成员函数，在实现过程中发现这样会导致一些循环包含的关系，不是很合理，因此单独写了一个`onMap(Player&)`函数，实现移动和坐标的记录。现在以一个开发完的状态回首，更倾向于将这个函数写成一个`Player`的静态成员函数。

3.5.4 战斗界面（属于`FightScene.h`）

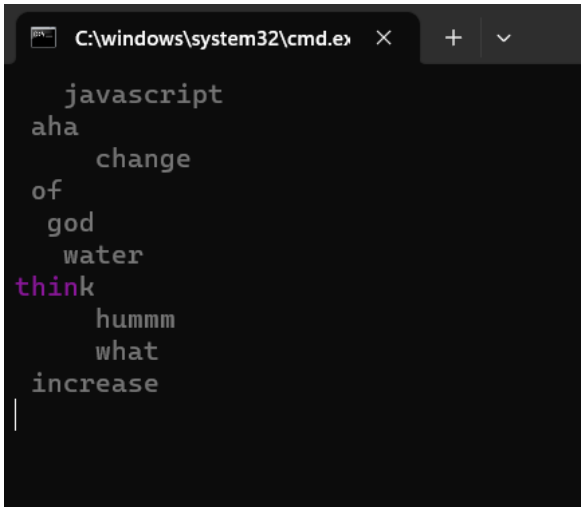
战斗界面不同于地图界面，是一个更加抽象的概念。起初我们打算将地点与战斗界面划等号，但在实际操作中发现从地图进入一个战斗场景并不是一个连贯的思路，因为战斗界面只是对战斗的一种抽象的表示，而非与这一地点的地形等性质想关联，于是我们新创出了FightScene类，它不属于任何类，当玩家进入某地点时，会根据外部的资源文件生成符合当前进度的场景。

起初只是想做一个打字游戏，后来为了提高创新性，决定开辟两块空间来进行战斗交互，一方防守，另一方攻击。在后续的讨论中又加入了中间的区域，用来显示过场剧情和战斗时的道具。最后我们又加入可视化血条，直接在原先的分割线上加以改动，最终呈现出游戏中的界面。



战斗界面

关于打字和染色的判定可谓是本游戏最核心的部分，起初我们并没有头绪，于是在网上查找关于键盘判定和字符染色的资料，过程中发现了C语言实现简单打字游戏这篇文章，给了我们很大启发。我们学习到了基本的键盘判定和染色的思路，以及对一个词语进行深度判定的方法，制作了测试小样，实现了基本的功能。



测试 Demo

但到了实际开发中，单区域变为双区域、词语不是生成而是通过词库传入。这些都是需要解决的难题。经过多次讨论过后，我们梳理出了基本的交互思路：

```cpp
void FightScene::loadScene(Player &player) {//最主要的函数，万物的起源
    showScene( &: player.getBackpack());//打印一下边框

    PosControl::HideCursor();//隐藏光标

    std::vector<Word> upper, lower;//核心单词表，直接源自monster和player，分别代表上下
    upper = monster.deliverWord();
    lower = player.deliverWord();

    showHP( &: player);//显示初始血量
```

loadScene函数

进入战斗后先打印界面，随后传入单词。词库分为上下两部分，分别对应上下两块区域。

```cpp
    while (player.getCurrHP() > 0 && monster.getCurrHP() > 0) {//战斗循环，只要都没死就一直进行
        int j = 0;
        for (int i = 0; i < upper.size(); i++) {...}//258~296作用是将词库打乱；
        for (int i = 0; i < upper.size(); i++) {...}//将词库初始化
        fallingDown( speed1: 1,  speed2: 1, upper, lower,  &: player);//speed暂时没有用，
        //fallingDown是控制词往下掉的函数。

    }
```

战斗在一方死亡前是不会停止的。我们将战斗的最小单位划分为“词库从头到尾出现过一次的时间”，每次战斗前都会将词序打乱，并进行初始化，随后调用`fallingDown()`函数实现单词的下落。

```cpp
void FightScene::fallingDown(int speed1, int speed2, const std::vector<Word> &upper, const std::vector<Word> &lower,
                             Player &player) {
    //分两部分：怪物攻击侧的check及结算，玩家侧的check及结算。

    Word blank( length: -1,  word: "",  effect: ' ');//特殊白板，长度为-1，与常规白板作区分
    std::vector<Word> shownUpper, shownLower;
    for (int i = 0; i < 15; i++) {
        shownUpper.push_back(blank);
        shownLower.push_back(blank);
    }//先给上下都填上空白，之后再单双填词，以实现隔一行的效果
    int cur = 0, offset = 0;//这个cur用来判断当前进行到哪一个单词了
```

进入`fallingDown()`后，第一个问题就是无法将词库映射到界面上。首先，词的数量是不确定的，其次为了显示出下落效果，每个词肯定都是从最上方一格开始掉落的。考虑之下我们引入了新的`vector`容器`shown`系列。

```cpp
    while (cur < upper.size()) {//这是第一部分，词还没有完全进入shown区域内。
        if (shownUpper[shownUpper.size() - 1].getState() == 0 && shownUpper[shownUpper.size() - 1].getLength() > 0) {
            player.getDamaged( damage: monster.getDamage());
            showHP( &: player);
        }//人物扣血的判定
        for (int i = shownUpper.size() - 1; i > 0; i--) {...}//替换词，向下一行
        offset = (offset + 1) % 2;//offset在0和1之间变换，用来实现每两回合加入一个单词的设定
        if (offset) {...}
        for (int i = offset; i < shownUpper.size(); i += 2) {...}
        for (int i = 0; i < shownUpper.size(); i++) {...}//输出一遍当前的shown
        typeAndColor( &: shownUpper,  &: shownLower,  &: player);//核心函数，判断打字用的
```

以屏幕上半方区域为例，所有属于上半方的对象的单词都存在名为`upper`的容器中。那么，跑完一轮词库可以分成两个过程：

1. 所有词都进入屏幕；
2. 是所有词都离开屏幕。

我们先将`shown`填入对应数量的白板词（数量与该区域行数相同），在过程一中，每两次滚动将`upper`中的一个词传入`shown`（两次滚动是因为设计了空行避免过于拥挤），而每次滚动都将`shown`自动向下替换一轮。这样就可以实现词从词库传入区域中，同时区域模拟下落效果；词库读取完后，就不再需要传入`upper`，只需要执行`shown`的向下替换直到`shown`中所有内容为空。这样就完成了一次最小战斗。

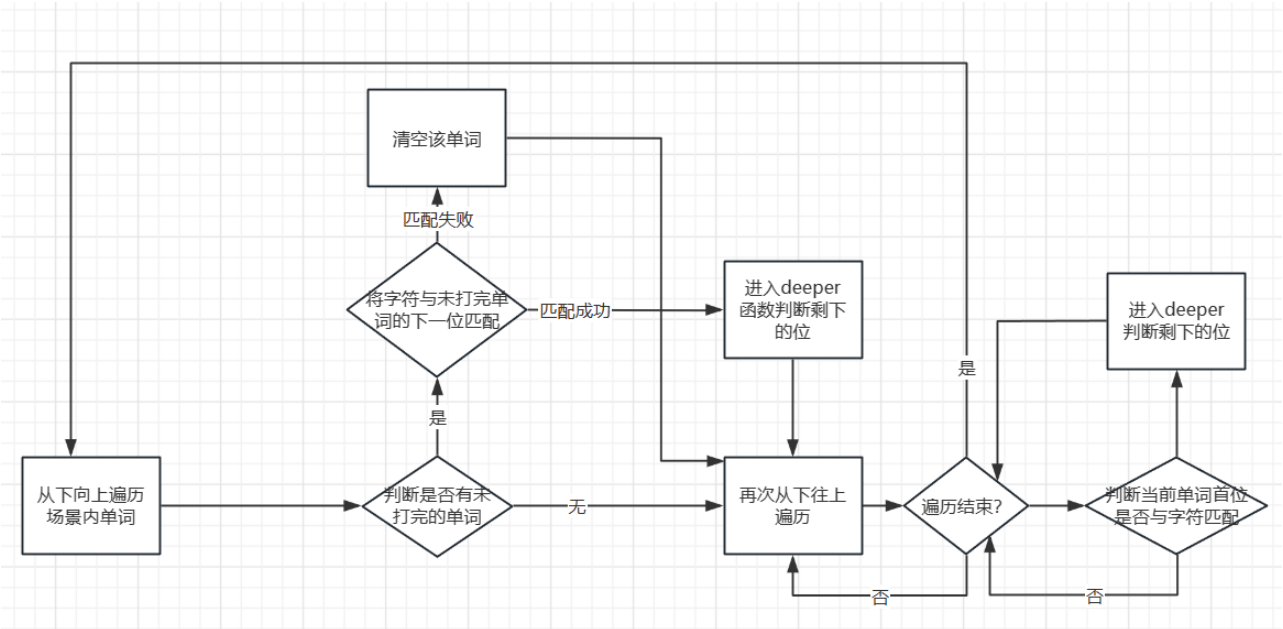
在战斗中，需要做的就是实时判定，并给予反馈，这部分由`typeAndColor()`函数进行。起初我们的做法是：每次进入函数时，调用`Sleep`函数给予停顿效果，同时读取键盘信息，再将键盘信息与场上单词进行比对，最后再清屏更新。

```
void FightScene::typeAndColor(std::vector<Word> &upper, std::vector<Word> &lower, Player &player
    int t = 100;//hjr的妙手，通过多段短暂的sleep在一次刷屏间隙中能打多个字母
    char next = ' ';//next是下一个键盘敲得字符
    int itemClock[10] = {0};
    while (t-- >= 0) {
        Sleep( dwMilliseconds: 5);
        next = ' ';
        if (_kbhit()) {//get键盘字符
            next = _getch();
        }
```

但是这样有两个问题：首先，频繁的清屏给人一种不连续的感觉；其次，按这种思路每下落一次只能更新一个字母。讨论之后，我们想出了绝妙的解决办法：将一次长时间的`Sleep`效果分割为多份短时间的`Sleep`，每次`Sleep`都读取键盘信息并执行`check`操作。而想要做到实时字母上色且不频闪，只需要以覆盖代清屏，除目标单词外的信息就完全不受影响。这样，我们也可以把每次大循环的`Sleep`分割出来的每个小`Sleep`即上图中循环的`Sleep(5)`作为我们的一个时间刻，每 100 时间刻词语下落一次。

```
if (next <= 'z' && next >= 'a') {...}
else if (next <= 'Z' && next >= 'A') {...}
else if (next <= '9' && next >= '0') {...}
```

`check`的操作非常复杂，总的来说先根据当前读入的字符类型判断玩家在打哪一区域的单词，进而进入对应的判定区域，如下图：



每个区域内判定方法大同小异，总的来说就是进行两次遍历（第一次是为了给未打完的单词更高的优先级），每次遍历时从下往上找匹配，找到后就进一步对后续字符进行判断，直到打错归零或打完结算。这样做的不足是，当多个词有着相同的开头时，玩家只能打较低的一个词才能被记为有效，而较高的词不会被检测。

3.6 玩家类、背包类和词语类的开发

3.6.1 玩家和怪物类的 UML 类图

3.6.2 玩家类（属于 `Player.h`）

对于玩家类而言，很重要的一个属性是`map`。地图上不同地点是否解锁、是否完成，地图当前的阶段乃至玩家在地图上的坐标，这些属性共同构成了玩家当前的游戏进度。因此地图是玩家的一个成员属性。除此之外，在实现玩家类的过程中使用了大量引用成员，便于调用这些引用成员的接口。

除了和怪物对战，玩家还需要在地图上移动、退出游戏需要存档。因此玩家类中有一些记录坐标的方法以及`load`和`get`的方法。

3.6.3 词语类（属于 `Word.h`）

攻击的核心元素词语是由`Word`类管理的。对于一个`Creature`，或玩家或怪物，都有着自己的一个`wordList`。在实现上，`wordList`使用`vector<Word>`容器管理。而对于每一个`Word`类对象，它有着基础属性`string`类型的词语本身和`int`类型的词语长度。结合`FightScene`需要，我们继续给它加入了伤害点数`effect`、屏幕显示位置`position`（其实就是输出词之前的随机空格长度）、`bool`类型的词语状态标识词语是否被打完以及用于变色的`cur`当前打出长度和`vector<int>`类型的一个词中各个字母的颜色属性。

```
std::random_device rd;
std::mt19937 gen( sd: rd());
std::uniform_int_distribution<> distr( a: 0,  b: 100 - length - 1);
position = distr( &: gen);
```

词语 position 属性的初始化

词语 `position` 属性的初始化是随机的，因此为了避免其超出界面边框，我们需要限制它的范围在 [0, 100-length - 1] 之间。

3.6.4 背包类和物品类（分别属于 `Backpack.h` 和 `Item.h`）

物品实际上也类似于一个词，但它的触发并非是打词实现的，而是直接输入其物品编号。为了避免冲突，物品编号是数字。

```
[0] LittleHealPotion 存量: 6 回血10，冷却50个时间刻
[1] MagicAttack 存量: 5 造成10点伤害，冷却50个时间刻
```

战斗状态下的物品栏

因此实现使用物品其实并不复杂，逻辑上和判断词的输入并无不同。而物品上比较特殊的是有一个 cd 钟的设计，在实现上其实就是在每 tick 末刷新一下 `itemClock`。代码实现如下：

```
// next是当前读入的字符
...
else if (next <= '9' && next >= '0') {
    if (itemClock[next - '0'] == 0 && player.getBackpack().isItemExist(next - '0')) {
        if (!player.getBackpack().useItem(next - '0'))
            useItem(player, player.getBackpack().getItemEffect(next - '0'));
    }
    if (player.getBackpack().isItemExist(next - '0')) // itemClock就是物品的cd钟
        itemClock[next - '0'] = player.getBackpack().getItemClock(next - '0');
}
for (int i = 0; i < 10; i++) {
    if (itemClock[i] > 0) itemClock[i]--;
}
```

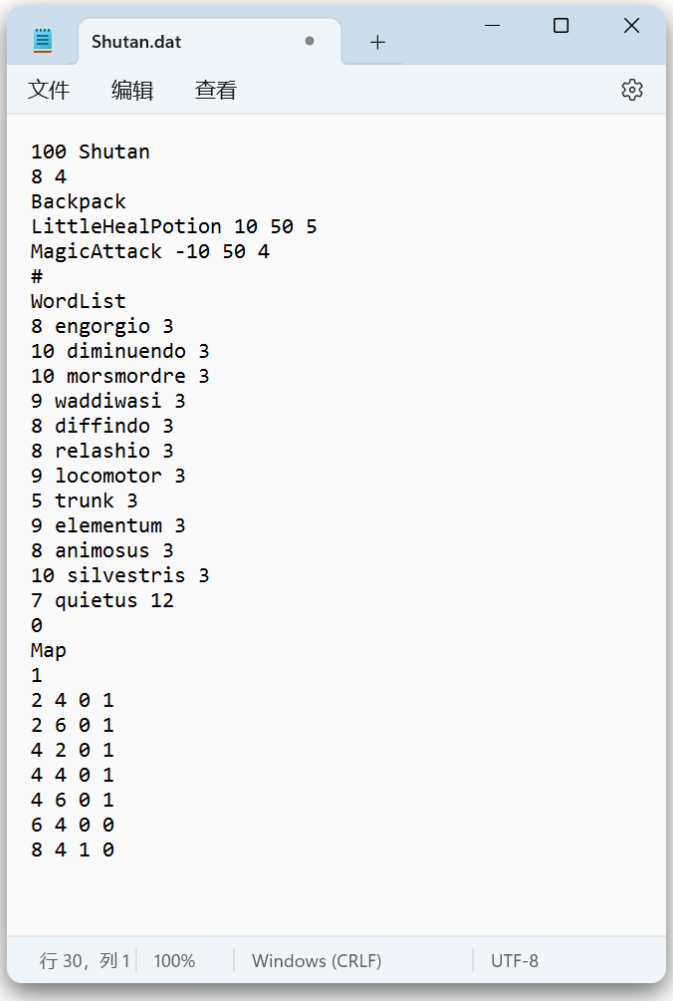
而对于背包类，它核心的容器是一个 `vector<pair<Item, int>>`。使用可变长度的 `vector` 能够做到清空、插入等操作。而 `vector` 中的元素使用的是一个 `pair<Item, int>`。在第一个位置放的是物品，相当于物品的类别；在第二个位置放的是物品的数量。在调用时，这一部分因为实现得仓促，所以在游戏的第一版 demo 中，这里出了无限使用物品的恶性 bug。这也提醒了我们考虑边界条件的重要性。

玩家可以通过战斗以及推进剧情来获得更多物品。

3.7 游戏流程中的其他开发

3.7.1 存档功能的开发（属于 `Data.h`）

经梳理，玩家需要存档的内容有当前的地图进度（各个地点的解锁度、各个地点的完成度）、玩家退出游戏的地图坐标、玩家的背包物品和习得的词汇列表，以及玩家自身的基础属性血量和名字。因此，我们使用多个成员函数对玩家存档进行写入和读取。



存档文件

在这里可以提升的是使用`yaml`或者`json`文件存档，显然比纯文字存档来得更高效。在开发过程中，我们也在GitHub 上查找了相关项目，尝试使用 jbeder/yaml-cpp 和 nlohmann/json。但考虑到这些数据格式大多只支持 UTF-8 解析，倘若有中文输入可能会有一定影响（游戏初期开发时允许中文用户名），于是作罢。但后期决定限制英文用户名时，没再尝试修改，也是比较遗憾的一点。

3.7.2 地图和剧情

游戏最终的地图改自起初《达拉崩吧》版本游戏的地图，仅对名称作了修改和路径上的简化。由于故事背景是《哈利波特》的未来魔法世界，因此保留了《哈利波特》故事中最核心的霍格沃茨学院，还有各种魔法元素。为了更贴合游戏背景，我们将起始的出生点设置在“校门”。剧情推进过程中加入了一些西方魔幻作品中的常见元素，如：探险、个人英雄主义解决危机、寻找失落的真相等。整个剧情主线也是按照传统西幻冒险作品来设置。科技元素则是主要参考诸多作品，比如死亡后玩家会看到亚历克斯发动“斯安威斯坦”，这就是《赛博朋克2077》中的典型装备，还有最终 Boss 的心灵控制仪，参考了《红警2：尤里的复仇》等。这些设计除了为了给作品添加科幻元素，也是在设置一些小彩蛋，希望让玩家在发现并理解它们时候能多一些惊喜感。

3.7.3 数值的确定

为了避免游戏难度过难，一开始我们取了一个小一些的数字作为初始战斗数值，也避免游戏后期数值过大。因此根据游戏中要打的攻击、防御词语复杂度与词语的下落速度，最后决定玩家攻击力和敌人血量的比例为1：10。所以取初始攻击力为1，敌人初始血量为10。每打完一关后，玩家都会获得攻击力更高的新词语。如果新词语数值过高会让前面的攻击词毫无意义（当然，词语也是随机出现的），而数值过低会让游戏难度曲线过于平缓，游戏难度几乎没有变化。所以为了让玩家的攻击力适当增加，取不同等级的攻击词攻击力为1，3，5，10，20。由于获得了更高攻击的词语后，每个词的平均攻击力是低于最高攻击力的，因此在后续关卡敌人hp的设计中，最高攻击力与敌人hp的比

例要逐渐升高。这样设计能让难度尽量线性增加。而暴击概率也是考虑玩家大概要攻击10到30词甚至更高才能赢得战斗，为了适当缩减战斗过程并增加趣味性，故设计为1/15概率获得2到6倍单次攻击。

3.7.4 项目文件管理、构建和兼容性

在游戏开发初期，避免游戏剧情、数值等“外部元素”过度和代码耦合影响代码美观度和可维护性，我们决定单独使用一个`Assets`文件夹存放各种游戏素材。其与源代码分开存储，并在代码中通过文件流读取来获得指定资源文件。如此操作同时也实现了类似于“热重载”的操作。但这样做带来的弊端也是显然的，玩家可能可以通过修改数值而改变游戏难度（但考虑到游戏的娱乐属性这未必是一件坏事）。

编译时，我们使用 CMake 通过`CMakeLists.txt`将整个项目构建起来。考虑到很多普通玩家的电脑中并没有配置`C/C++`环境，或者有的人用的是`msvc`而非我们开发所用的`mingw`，运行库可能也不尽相同，我们使用了`-static`参数让游戏静态编译，保证在不同电脑上的可运行性。

但由于不同系统版本的默认终端和控制台并不完全相同，我们发现仍然是 Windows 11 下使用新终端的环境能获得游戏的最佳体验。

4 测试

在我们自己测试的过程中，我们使用了单元测试、集成测试和系统测试等多种测试方法。但由于对一些边界条件的疏忽与代码复用时的不正确写法，我们初步写出来的游戏仍然有不少 bug。在这里非常感谢参与测试我们游戏的每一位同学。

GitHub上的一个Issue，由徐坤提出

GitHub上的Pull Requests记录

5 总结

5.1 优点

1. 项目模块化程度较高，方便对代码的合理管理和可拓展性。

2. 项目代码的命名风格统一，注释丰富。

3. 项目版本管理与开发流程较为规范，更加贴近实际生产过程。

4. 游戏完成度较高，存档读档功能、装备道具、非固定的战斗系统等功能完善。

5. 游戏玩法直观易懂，区别于常规的 MUD 游戏，有着游戏方式上的新颖度。

5.2 缺点

1. 在协作过程中由于沟通和规划上的问题一定程度上拖累了项目进度。

2. 代码仍没能做到完全“面向对象”，有一些地方的实现仍然颇有“殊途同归”的感觉。

3. 游戏的可玩性仍然有待提升，打字的方式相对单调，道具获取的方式相对单一。

4. 由于没有进行设计模式的系统学习，没能贯彻合理的设计模式进行游戏设计。

6 致谢

参与测试并给出游戏建议的林舒坦同学、徐坤同学，涨价 30% 的《霍格沃兹之遗》。

还有中国海洋大学西海岸校区外面蔚蓝的海。