项目说明文档

Today Is A Good Gay

——程序设计范式期末项目

项目成员：

2350944　　赵诣　（组长）

2350223　　戴昊晟

2351430　　胡晓天

2354100　　郝哲逸

专　　业：　软件工程

指导教师：　朱宏明 赵钦佩

2024-2025 学年第一学期

目录

1 项目简介

1.1 项目介绍

*Today Is A Good Day*游戏是一个基于 Cocos2d-x 3.17.2 （游戏引擎）开发的农场生活模拟游戏（类星露谷物语）。

游戏提供了丰富的基础功能和扩展内容，玩家可以在广阔的世界中探索多个区域，包括庄园、小镇、沙滩和矿洞等。

在庄园中，玩家可以管理自己的农场，种植各种作物（如浇水、施肥等）及收货、养殖动物，砍树，凿石头。在小镇中，玩家可以和 NPC 互动，可以去商店买需要种植的种子,还可以参加小镇上的活动,体验丰富多彩的社交生活。同时，玩家可以在矿洞中采矿、在沙滩场景的海里钓鱼。

游戏中的季节变化系统非常丰富，庄园和小镇的场景会随着季节的更替而变化，周围的环境也会相应调整。农作物的生长也与季节紧密相关，增加了游戏的真实感和策略性。此外，游戏还包含了动态的天气系统，进一步增强了沉浸感。

"Today Is A Good Day" 还支持**多人联机**游玩，玩家可以与朋友一起经营农场。

2 项目基础功能介绍

2.1 农场管理

在这款游戏中，玩家将携带初始工具，包括斧头、稿子、锄头、水壶和钓鱼竿，展开丰富的农场生活。玩家可以用锄头耕作土地，种植各种作物并收获丰富的农产品，同时还能用斧子砍伐树木、用镐子开采石头，获取建筑和制作材料。此外，玩家还可以进行动物养殖，体验从饲养到收获的完整过程。

作物拥有不同的生长阶段，成熟时间因种类而异，玩家需要根据季节选择合适的作物进行种植，否则过季植物将会枯萎。为了确保作物的健康生长，玩家可

以对作物进行施肥和浇水，施肥能够加速生长，而长时间不浇水则会导致作物枯死。通过精心管理，玩家将能够打造一个繁荣的农场，享受丰收的喜悦。

2.2 社区交互

在小镇场景中，玩家将体验到丰富多彩的社交与互动功能。首先，玩家可以在社区商店购买各种作物种子，为农场生活提供源源不断的种植资源。此外，玩家还可以与小镇中的 NPC 进行对话交互，通过与他们交流，了解小镇的日常动态、获取任务线索或解锁隐藏的故事情节。

特别值得一提的是，小镇在春季和冬季会举办盛大的节日活动。玩家可以在节日期间前往小镇，参与各种有趣的活动，例如集市、比赛、表演等，感受节日的欢乐氛围，同时还能获得丰厚的奖励或独特的节日道具。这些活动不仅丰富了游戏内容，也为玩家提供了与 NPC 和其他玩家互动的绝佳机会，让小镇成为游戏中一个充满生机与活力的社交中心。

2.3 探索与冒险

游戏中，玩家可以自由探索周边的多样化区域，包括宁静的庄园、热闹的小镇、美丽的沙滩以及神秘的矿洞。每个区域都拥有独特的环境和资源，等待玩家去发现和利用。

在庄园中，玩家可以专注于耕作和养殖，打造属于自己的田园生活；而在小镇，玩家可以参与社交活动、购买种子或与 NPC 互动，感受小镇的温馨氛围。沙滩则是一个放松的好去处，玩家可以在这里享受海风，同时在水域中进行钓鱼。游戏中的水域提供了丰富的鱼类资源，玩家可以钓到不同种类的鱼，每种鱼都有其独特的价值和用途。

此外，矿洞是一个充满挑战和机遇的地方。玩家可以在矿洞中进行采矿，获取煤矿、金矿、银矿、铜矿等多种矿石资源。每一层矿洞都会随机生成不同的矿石，玩家需要不断深入探索，才能发现更稀有和珍贵的矿物。矿洞的深度和复杂性为玩家提供了无尽的探索乐趣，同时也为游戏中的建造、制作和升级提供了重要的资源支持。

通过探索这些场景，玩家不仅可以体验到多样化的游戏玩法，还能收集丰富的资源，推动自己的农场和角色不断发展壮大。

2.4 角色成长与技能

在游戏中，玩家通过参与各种活动来提升自己的技能和等级。无论是砍伐树木、开采矿石、钓鱼还是耕作，这些行为都会为玩家积累经验值，从而提升等级。随着等级的提高，玩家将能够解锁和使用更高级的工具，例如更锋利的斧头、更坚固的镐子或更高效的锄头。这些高级工具不仅能够提高玩家的工作效率，还能增强攻击力，帮助玩家更轻松地应对游戏中的挑战。

此外，玩家可以通过按键 E 来方便地打开和关闭背包，查看并管理自己的物品。背包是玩家存放资源、工具和道具的重要空间，合理管理背包中的物品，能够帮助玩家更好地规划和利用资源。随着游戏的进行，玩家通过不断积累经验，提升等级，解锁更强大的工具和能力，逐步成长为一名经验丰富的农场主和冒险家。

3 项目 C++特性使用

3.1 auto 关键字

功能：允许编译器自动推导变量的类型。
代码示例：

```
  //切换场景调用函数
bool first_to_manor() {
    auto spring_scene = spring_manor::createScene();
    Director::getInstance()->replaceScene(TransitionFade::create(1.0f, spring_scene));
    return true;
}
```

3.2 Lambda 表达式

功能：允许在代码中内联定义匿名函数。
代码示例：

```
void GameTime::stopAutoUpdate() {
```

```
    // 停止定时器
    cocos2d::Director::getInstance()->getScheduler()->unschedule(
        "GameTimeUpdate",  // 使用相同的标签 "GameTimeUpdate" 停止定时器
        this // 表示定时器的目标对象
    );
    CCLOG("Auto update stopped.");  // 输出停止日志

}
```

3.3 强类型枚举 (enum class)

功能：通过强类型枚举避免了传统枚举的类型转换问题。
代码示例：

```
    // 定义主客机的枚举
    enum class Cooperation {
        Host,
        Guest,
    };
```

3.4线程支持库 (std::thread)

功能：提供了标准的多线程支持。
代码示例：

```
// 启动监听线程
void startListeningThread() {
    // 创建新线程来监听服务器消息
    std::thread listener(listenThread);
    listener.detach(); // 分离线程，使其独立运行
}
```

3.5 类和多态

功能：利用虚函数和 override 实现类和多态。

```
// 沙滩场景调用
class beach : public cocos2d::Scene
{
public:
    static cocos2d::Scene* createScene();

    virtual bool init();

    // a selector callback
```

```
    void menuCloseCallback(cocos2d::Ref* pSender);


    // implement the "static create()" method manually
    CREATE_FUNC(beach);

    // 重写 onEnter 方法
    virtual void onEnter() override;

    // 重写 onExit 方法
    virtual void onExit() override;
};
```

3.6 迭代器

功能：简化了遍历容器（如数组、std::vector 等）的语法。
代码示例：

```
Layer* currentCharacterLayer = nullptr;
 for (auto child : currentScene->getChildren()) {
     currentCharacterLayer = dynamic_cast<Layer*>(child);
     if (currentCharacterLayer) {
         break;
     }
 }
```

3.7 try-catch 异常处理

功能：Try-Catch 是一种异常处理机制，用于在程序中捕获和处理可能发生的异常或错误。在使用 Try-Catch 结构时，代码块被放置在一个 Try 块中，该块用于包含可能会抛出异常的代码。如果在 Try 块中发生异常，程序会立即跳转到对应的 Catch 块，这样就可以执行特定的异常处理代码，而不会导致程序崩溃。

```
// 将消息处理逻辑调度到主线程执行
postToMainThread([receivedMessage]() {
    try {
        if (receivedMessage.find("MATCHED") != std::string::npos) {
            handleMatchedMessage(receivedMessage);
        }
        else if (receivedMessage.find("FAILED") != std::string::npos) {
            handleFailedMessage(receivedMessage);
        }
        else if (receivedMessage.find("RELEASE") != std::string::npos) {
            handlePlayerReleaseMessage(receivedMessage);
```

```
        }
        else if (receivedMessage.find("MOVE") != std::string::npos) {
            handlePlayerMoveMessage(receivedMessage);
        }
        else if (receivedMessage.find("TREE") != std::string::npos) {
            handleTreeActionMessage(receivedMessage);
        }
        else if (receivedMessage.find("PLOUGH") != std::string::npos) {
            handlePloughActionMessage(receivedMessage);
        }
        else if (receivedMessage.find("STONE") != std::string::npos) {
            handleMineActionMessage(receivedMessage);
        }
        else if (receivedMessage.find("PLANT") != std::string::npos) {
            handlePlantActionMessage(receivedMessage);
        }
        else {
            throw std::runtime_error("Unknown message: " + receivedMessage);
        }
    }catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        CCLOG("Original message: %s", receivedMessage.c_str());
    }
```

3.8 单例模式

功能：确保一个类只有一个实例，并提供一个全局访问点来访问该实例。
代码示例

```
/* ---------- 地图管理层得到当前场景和地图 ----------*/
MapManager* MapManager::_instance = nullptr;

MapManager::MapManager() : _currentScene(nullptr)
{
    // 监听场景切换完成事件

Director::getInstance()->getEventDispatcher()->addCustomEventListener(Director::EVENT_A
FTER_SET_NEXT_SCENE, CC_CALLBACK_1(MapManager::onSceneChange, this));
}

MapManager::~MapManager()
{
    // 移除事件监听
```

```
Director::getInstance()->getEventDispatcher()->removeCustomEventListeners(Director::EVENT_AFTER_SET_NEXT_SCENE);
}

MapManager* MapManager::getInstance()
{
    if (!_instance)
    {
        _instance = new MapManager();
    }
    return _instance;
}
```

4 项目高级功能介绍

4.1 精美的游戏界面设计

游戏中，我们精心打造了多个风格迥异且细节丰富的场景，包括宁静的小镇、生机勃勃的庄园、温馨的家、神秘的矿洞、阳光沙滩以及繁华的商店，每一个场景都使用 Tiled 地图编辑器进行底层构建，确保了地图的精细度和多样性，并布置了大量与场景主题相符的装饰元素，如小镇的房屋、庄园的农田、家里的家具、矿洞的矿石、沙滩的椰树、商店的商品等，力求营造出沉浸感十足的氛围。为了呈现更广阔的游戏世界，每个场景的地图都设计得非常庞大，玩家实际看到的只是地图的一小部分，当角色在场景中移动时，底层地图会随之平滑滚动，无缝地展现更广阔的世界，让玩家能够自由探索，充分体验每个场景的独特魅力，仿佛置身于一个真实的、充满细节的世界之中。

另外，由于本游戏是多场景游戏，在一个场景中走到某些入口

4.2 掉落物掉落与自动拾取

在我们的游戏中，掉落物系统是构建一个动态、互动性强且充满探索乐趣的世界的关键组成部分。这个系统不仅仅简单地提供物品奖励，更是与玩家的日常操作紧密相连，从而驱动玩家不断地探索、交互和成长。当玩家投入到游戏中，进行诸如砍伐树木、开采矿石、收割农作物等一系列基础操作时，我们精心

设计的掉落物生成机制便开始发挥作用。每当玩家成功完成这些操作，相应的资源或物品便会以可视化的形式在玩家周围的地图上动态生成，这些掉落物的种类和数量会与玩家的操作类型和效率紧密相关。例如，砍伐一颗树木可能会掉落一定数量的木材，而开采矿脉则可能产生多种不同类型的矿石和宝石，收割农作物则会获得相应的农产品。这些掉落物不仅仅是单纯的图标，而是以精心设计的视觉元素呈现在地图之上，让玩家能够清晰地感知到它们的存在和价值，同时也能增强游戏的视觉吸引力。

为了保证玩家能够流畅而便捷地收集这些掉落物，我们还设计了一套高效的拾取机制。当玩家控制的角色在地图上自由移动时，游戏引擎会实时计算每个掉落物与玩家角色之间的距离。这个距离计算不仅仅是简单的坐标差值，而是考虑到角色和掉落物的大小，以及潜在的障碍物，以确保拾取逻辑的准确性。当掉落物进入预设的拾取半径内，即玩家角色周围的指定区域时，系统便会自动执行拾取操作。这个自动拾取的过程无需玩家额外的操作，极大提升了游戏的操作流畅度。拾取之后，掉落物会从地图上消失，并转化为玩家背包中的物品，此时，玩家便可以自由地使用这些收集到的资源来制作工具、建造房屋、烹饪食物或者进行其他各种游戏内的操作，这让收集的价值得到了充分的体现。

那些没有被玩家及时拾取的掉落物并不会立即消失，而是会以可视化的形式长时间存在于地图之上，这为玩家提供了足够的探索空间，玩家可以随时回到之前活动过的区域，拾取遗漏的物品。这种机制既避免了玩家因为错过一些掉落物而产生的遗憾，同时也增强了玩家探索地图的动力，同时也增加了游戏世界的持久性和活力，让玩家在不同的时间探索同一个区域时，都会有不同的感受。

此外，我们还考虑到了游戏的平衡性和公平性。掉落物的生成概率和数量是经过精心设计的，避免出现某些区域掉落过于频繁或稀有物品掉落过少的问题，以确保不同类型的玩家，都能获得公平的游戏体验。我们还通过数据分析不断调整这些参数，以保证游戏具有良好的可玩性和挑战性。这个掉落物和拾取系统不仅是游戏的基础功能，更是一种引导玩家体验游戏，探索世界的手段。通过这些机制，玩家能够更加投入到游戏的世界中，感受到每一次砍伐，挖掘，收获，探索所带来的快乐。

4.3 总控游戏的设置

4.3.1 设置界面与背景音乐

游戏中所有场景的左上角都会放置一个设置按钮，点击该按钮会弹出一个设置面板，面板中包含一个背景音乐的开关，玩家可以通过切换开关来控制背景音乐的播放与停止，这个开关的状态会保持跨场景有效。设置按钮和面板都属于 UI 元素，并利用 Cocos2d-x 的音频引擎来实现背景音乐的控制，当开关处于开启状态时，会播放当前场景的背景音乐，处于关闭状态则会停止播放。这个模块的目标是为玩家提供个性化的游戏体验，使得玩家可以根据自己的喜好调整游戏中的背景音乐。

游戏内还实现了背景音乐的预加载和播放：在游戏启动时预加载背景音乐，并在需要时播放或暂停。以及音量调整：调整背景音乐的音量。并且实现了全局界面与背景音乐的交互：在全局界面中添加音量控制按钮，并确保它们能够正确地控制背景音乐的播放和音量。

4.3.2 时间流逝与动态季节变化

游戏中，每个场景的右上角都会实时显示当前的游戏年份、季节、天数和当天时间，为玩家提供清晰的游戏进度信息。季节会随着时间动态变化，并且会直接影响农作物的生长周期和速度。此外，每天的天气也是随机生成的，雨天会免去玩家的浇水操作，而连续干旱的天气则需要玩家进行灌溉以保证农作物的正常生长。季节的变换还会对游戏地图的视觉效果产生影响，并相应地改变可种植的农作物种类，形成一个动态且富有变化的游戏生态系统。这些相互关联的机制共同构成了游戏的核心体验，要求玩家根据环境变化进行策略性的规划和操作。

4.4 不同层级遮挡关系的实现

在我们的游戏中，为了确保玩家在复杂的场景中始终拥有清晰的视觉感知和流畅的操作体验，我们精心设计了一套基于层级和动态透明度调整的遮挡处理机制。这一机制的核心目标是解决游戏中常见的人物与场景元素（如树木、房屋等）之间的遮挡问题，避免遮挡导致玩家无法看清自身角色，从而影响游戏体

验。

当玩家角色在场景中移动时，本游戏会实时检测玩家角色与场景中的可遮挡元素之间的空间关系，尤其是人物与树木之间。在这种情况下，我们设定了特定的遮挡逻辑：如果人物走到树木后面，即树木在视觉上遮挡了人物，游戏会动态地将树木的显示模式调整为半透明模式。这种半透明效果并非对所有物体都生效，而是有选择性的，主要针对那些可能对玩家视觉产生较大干扰的物体。

这种处理方式的主要优势在于，能够让玩家始终清晰地看到自己在游戏场景中的位置，即便被高大的树木或其他场景元素遮挡。半透明模式的出现并不会完全遮挡树木的视觉效果，而是让树木变得透明，既保留了场景的整体感，又使得玩家的角色始终可见，确保了玩家的视觉清晰度，从而避免了因为遮挡而导致的迷失感。

实现这一效果的技术细节包括：游戏内会维护一个图层系统，通过图层的前后顺序决定绘制顺序。人物角色通常会处于一个较高的图层，而树木等遮挡物会处于较低的图层。当角色与遮挡物发生空间关系时，我们通过动态调整遮挡物图层的透明度，实现半透明效果。这个透明度的调整是实时进行的，随着角色位置的变化而动态变化，从而保证了效果的流畅性。

此外，我们还对半透明的程度进行了精确控制，避免了透明度过高导致场景元素过于虚无，或是透明度过低导致遮挡依然存在。通过多次测试和调整，最终确定了一个既能有效解决遮挡问题，又能保持游戏画面美观度的透明度值。

这种遮挡处理机制应用于树木，还其目的始终是确保玩家能够清晰地了解自己所在的位置，并流畅地进行游戏操作。这一机制不仅仅是一个简单的技术实现，更是我们为了提升玩家游戏体验所做的努力，通过细节的优化，让玩家能够更加沉浸于游戏世界之中。

4.5 联网模式的实现

4.5.1 硬件支持

服务器是基于在阿里云购买的轻量应用服务器，通过 Vscode 相关插件通过 SSH 远程连接服务器并实现在服务器上编辑和运行代码

4.5.2 核心功能

(1) 客户端连接管理

服务器支持多个客户端同时连接，并为每个客户端创建独立的线程处理请求。通过 connected_clients 列表管理当前连接的客户端。

(2) 匹配机制

客户端发送的请求（如房间邀请码）会被服务器检查是否存在于文件 clients.txt 中。如果请求存在于文件中，说明有另一个客户端已经创建了相同的房间，服务器会通知这两个客户端匹配成功。匹配成功后，服务器会向两个客户端发送 MATCHED! 消息，表示它们可以开始协作游戏。

(3) 持久化存储

新的客户端请求会被保存到文件 clients.txt 中，以便下次启动服务器时加载。这种机制确保了房间信息不会丢失，即使服务器重启。

(4) 广播机制

如果客户端发送的消息不是六位数邀请码中，服务器会将消息广播给其他所有连接的客户端。这种机制用于实现多人协作中的消息同步（如玩家动作、状态更新等）。

4.5.3 客户端实现

a.模块化设计

场景管理：游戏分为多个场景（Scene），每个场景负责不同的功能模块。

scene_coop：合作模式选择界面，玩家可以选择创建房间或加入房间。

scene_create：创建房间界面，玩家输入邀请码并创建房间。

scene_join：加入房间界面，玩家输入邀请码并加入房间。

ConnectingScene：连接中界面，显示连接状态。

功能模块：每个场景负责特定的用户交互和逻辑处理。

按钮回调函数（如 onCreate、onJoin）用于处理用户点击事件。

网络通信模块（如 listenThread、sendMessageToServer）负责与服务器交互。

b. 网络通信

Socket 通信：使用 Winsock 库实现客户端与服务器的 TCP 连接。

connectToServer：客户端连接到服务器。

listenThread：在后台线程中监听服务器发送的消息。

sendMessageToServer：向服务器发送消息。

消息处理：客户端根据服务器发送的消息类型执行不同的逻辑。

c.线程管理

后台线程：

listenThread：在后台线程中监听服务器消息，避免阻塞主线程。

startListeningThread：启动监听线程。

主线程调度：

使用 postToMainThread 将需要在主线程执行的逻辑（如 UI 更新）调度到主线程。

4.5.4 联网功能实现

a. 协作流程

(1) 客户端连接

客户端通过 TCP 连接到服务器的 12345 端口。服务器为每个客户端创建一个独立的线程，开始处理客户端的请求。

(2) 创建房间

客户端 A 发送一个唯一的邀请码（如 123456）到服务器。服务器检查文件 clients.txt，发现该邀请码不存在，于是将邀请码保存到文件中。服务器等待另一个客户端加入相同的房间。

(3) 加入房间

客户端 B 发送相同的邀请码（如 123456）到服务器。服务器检查文件 clients.txt，发现该邀请码已存在，说明有另一个客户端已经创建了相同的房间。服务器向客户端 A 和客户端 B 发送 MATCHED! 消息，表示它们匹配成功。客户端 A 和客户端 B 可以开始协作游戏。

(4) 协作游戏

客户端 A 和客户端 B 可以发送游戏状态或动作消息到服务器（如 MOVE、PLANT、MINE 等）。

服务器将这些消息广播给所有连接的客户端，确保所有玩家的状态同步。

例如：

客户端 A 发送 MOVE A（表示向左移动）。

服务器将 MOVE A 广播给客户端 B。

客户端 B 更新自己的游戏状态，显示客户端 A 向左移动。

(5) 断开连接

如果客户端断开连接，服务器会从 connected_clients 列表中移除该客户端。服务器会通知其他客户端该玩家已断开连接。

b.协作原理

(1) 匹配机制

通过文件 clients.txt 存储房间邀请码，实现客户端之间的匹配。匹配成功后，服务器会通知两个客户端开始协作游戏。

(2) 状态同步

服务器作为中转站，负责将客户端发送的消息广播给其他客户端。

例如：

客户端 A 发送 PLANT（表示种植作物）。

服务器将 PLANT 广播给客户端 B。

客户端 B 更新游戏状态，显示作物已种植。

(3) 持久化存储

通过文件 clients.txt 存储房间信息，确保房间信息不会丢失。

即使服务器重启，已创建的房间仍然有效。

(4) 线程安全

使用 threading.Lock() 保护共享资源（如 connected_clients），避免多线程竞争问题。

5 项目代码逻辑与实现

5.1 全局监听 UI 事件

本游戏所有界面 UI 时间均有统一的监听器接受然后下发。PlayerControlLayer 是负责管理整个用户交互的层，所有鼠标与键盘的用户输入均会被识别，进入本层后利用不同逻辑实现相应功能。

4.4.1 PlayerControlLayer 代码实现

```
// 专门负责主角移动和键盘输入的层
class PlayerControlLayer : public Layer
{
public:
    CREATE_FUNC(PlayerControlLayer);
    bool init() override;
    void onKeyPressed(EventKeyboard::KeyCode keyCode, Event* event);
    void onKeyReleased(EventKeyboard::KeyCode keyCode, Event* event);
    void onMouseDown(Event* event); // 新增鼠标点击回调函数
    void update(float dt) override;
    void setPlayer(Person* player);
    void movePlayer(float dt);
    void onExit() override; // 添加 override 关键字
private:
    EventListenerKeyboard* _keyboardListener;
    EventListenerMouse* _mouseListener; // 新增鼠标事件监听器
    Person* _player; // 指向要控制的主角
    bool _moveLeft;
    bool _moveRight;
```

```
    bool _moveUp;
    bool _moveDown;
    float _playerSpeed;
    // 移动标志位
    bool _isRunning;
    std::string _currentDirection; // 定义 _currentDirection 变量
};
```

5.2 Map 类设计与实现

5.2.1 地图公用函数

工具：获取瓦片指定角落的像素坐标

```
Vec2 getTilePixelPosition(const Vec2& tileCoord,const Size& tileSize, const Size& mapSize, TileCorner corner = CENTER);
```

5.2.2 MyObject、GrowObject、DeadObject 类

MyObject 是一个抽象的基类，继承自 Sprite 类,派生出 GrowObject 和 DeadObject 类。这些抽象类是为了分门别类地管理地图上的各种物体。

```
class MyObject : public Sprite {
protected:
    Vec2 tilePosition;  // 物体的瓦片坐标
    int health = 10;          // 物体的血量
public:
    // 构造函数
    MyObject(){}
    // 获取瓦片坐标
    Vec2 getTilePosition() const { return tilePosition; }
    // 获取血量
    int getHealth() const { return health; }
    // 减少血量
```

```
    virtual void reduceHealth(int damage) {}
    // 死亡动画
    virtual void deathAnimation() = 0;
    // 产生掉落物
    virtual void generateDrops(){}
};
class GrowObject : public MyObject {
protected:
    int growthDays = 0;              // 生长天数
    Stage stage = Stage::Childhood;  // 当前阶段
public:
    // 构造函数
    GrowObject(){}
    // 更新生长天数（增加一天，并可能改变阶段）
    virtual void update() = 0;
    Stage getStage() { return stage; }
};
class DeadObject : public MyObject {
public:
    // 构造函数
    DeadObject(){}
};
```

5.2.3 Tree 类

Tree 类继承自 GrowObject 类，用于管理地图上的树。树有不同的种类、成长阶段、季节变化等。

```
class Tree : public GrowObject {
private:
    TreeType type; // 树的种类
public:
```

```
    // 构造函数
    Tree(TMXTiledMap* tileMap, Layer* objectLayer, Vec2 tile, TreeType ty, Stage st = Stage::Childhood);
    // 在瓦片地图上随机生成 num 个树苗
    static void randomGenerate(TMXTiledMap* tileMap, Layer* objectLayer, int num, Stage stage);
    // 静态方法：遍历 objectLayer 的所有子节点并调用 update
    static void updateAll(Layer* objectLayer);
    // 更新树的贴图
    void update() override;
    // 死亡动画
    void deathAnimation() override;
    // 产生掉落物
    void generateDrops() override;
    // 设置树的透明度
    void reduceOpacity();
    // 恢复树的透明度
    void restoreOpacity();
};
```

5.2.4 Stone 类

Stone 类继承自 DeadObject 类，有随机生成和在矿洞随机生成的静态方法。

```
class Stone : public DeadObject {
private:
    StoneType type; // 石头的种类
public:
    // 构造函数
    Stone(TMXTiledMap* tileMap, Layer* objectLayer, const Vec2& tile, StoneType ty);
    // 在瓦片地图上随机生成 num 个石头
```

```
    static void randomGenerate(TMXTiledMap* tileMap, Layer*
objectLayer, int num, StoneType type);
    // 在矿洞里随机生成 num 个石头
    static void randomGenerateInMine(TMXTiledMap* tileMap, Layer*
objectLayer);
protected:
    // 死亡动画
    void deathAnimation() override;
    // 产生掉落物
    void generateDrops() override;
};
```

5.2.5 Crops 类

Crops 类继承自 GrowObject 类，是一个抽象基类，用于进行作物管理。派生出的作物有不同的生长阶段、时间、掉落物等。

```
class Crops : public GrowObject {
protected:
    // 作物的生长季节
    Season cropsSeason;
    // 作物类型
    CropsType type;
    // 存储每个阶段的图片路径
    std::string stageImages[3];
    // 生长阶段阈值
    std::vector<int> growthStageThreshold;
    // 构造函数
    Crops(){}
    // 静态成员变量，用于记录需要移除的对象
    static std::vector<Node*> nodesToRemove;
public:
```

```
    // 更新生长天数
    void grow();
    // 死亡动画
    void deathAnimation() override;
    // 产生掉落物
    void generateDrops() override;
    // 生长
    void update() override;
    // 收获
    void harvest();
    // 扣血
    virtual void reduceHealth(int damage);
    // 静态方法：遍历 cropsLayer 的所有子节点并调用 update
    static void updateAll(Layer* cropsLayer);
};
// 酸菜类
class Withered : public Crops {
public:
    Withered(TMXTiledMap* tileMap, Layer* cropsLayer, Vec2 tile, Stage
st = Stage::Childhood);
    void update() override {};
};
// 胡萝卜类
class Carrot : public Crops {
public:
    Carrot(TMXTiledMap* tileMap, Layer* cropsLayer, Vec2 tile, Stage
st = Stage::Childhood);
};
// 大蒜类
```

```
class Garlic : public Crops {
public:
    Garlic(TMXTiledMap* tileMap, Layer* cropsLayer, Vec2 tile, Stage
st = Stage::Childhood);
};
// 土豆类
class Potato : public Crops {
public:
    Potato(TMXTiledMap* tileMap, Layer* cropsLayer, Vec2 tile, Stage
st = Stage::Childhood);
};
// 玉米类
class Corn : public Crops {
public:
    Corn(TMXTiledMap* tileMap, Layer* cropsLayer, Vec2 tile, Stage st
= Stage::Childhood);
};
// 甜瓜类
class Melon : public Crops {
public:
    Melon(TMXTiledMap* tileMap, Layer* cropsLayer, Vec2 tile, Stage st
= Stage::Childhood);
};
// 西红柿类
class Tomato : public Crops {
public:
    Tomato(TMXTiledMap* tileMap, Layer* cropsLayer, Vec2 tile, Stage
st = Stage::Childhood);
};
```

```
// 小白菜类
class Cabbage : public Crops {
public:
    Cabbage(TMXTiledMap* tileMap, Layer* cropsLayer, Vec2 tile, Stage
st = Stage::Childhood);
};
// 茄子类
class Eggplant : public Crops {
public:
    Eggplant(TMXTiledMap* tileMap, Layer* cropsLayer, Vec2 tile, Stage
st = Stage::Childhood);
};
// 南瓜类
class Pumpkin : public Crops {
public:
    Pumpkin(TMXTiledMap* tileMap, Layer* cropsLayer, Vec2 tile, Stage
st = Stage::Childhood);
};
```

5.2.6 Drop 类

Drop 类实现了地图上掉落物的管理，包括掉落物的生成、拾取。

```
// 掉落物组类型结构
struct DropItem {
    std::string type;         // 掉落物的类型
    int id;                   // 唯一标识符
    std::string texturePath; // 掉落物的贴图路径
    Vec2 position;            // 掉落物的当前坐标
};
// 掉落物类型结构
class Dropper : public Sprite{
```

```
public:
    std::string type;
    Vec2 tilePosition;
    Dropper(Vec2 tile) :tilePosition(tile){}
    // 获取掉落物的瓦片坐标
    Vec2 getTilePosition() { return tilePosition; }
};
class Drop {
protected:
    Vec2 dropPosition;                          // 掉落物的初始坐标
    std::vector<DropItem> dropItems;            // 掉落物的类型和贴图列表
    float spreadRadius = 5.0f;                  // 掉落物散开半径
    Layer* targetLayer;                         // 目标层
    int nextDropId = 0;                         // 用于生成唯一 ID
    TMXTiledMap* tileMap;                       // 地图指针
    // 随机数生成器
    static std::mt19937 gen;
    // 随机生成掉落物数量
    int generateRandomCount(int min, int max);
public:
    // 构造函数
    Drop(const Vec2& position, Layer* targetLayer, TMXTiledMap*
tileMap);
    // 添加掉落物
    void addDropItem(const std::string& itemType, const std::string&
texturePath);
    // 生成掉落物
    void generate();
    // 删除掉落物（根据 ID）
```

```
    void removeDropItemById(int id);
    // 返回掉落物的类型
    std::string getType();
};
class TreeDrop : public Drop {
public:
    // 构造函数
    TreeDrop(const Vec2& position, Layer* targetLayer, TMXTiledMap*
tileMap);
};

class StoneDrop : public Drop {
private:
    // 石头类型到掉落物类型和贴图路径的映射表
    static const std::unordered_map<StoneType, std::pair<std::string,
std::string>> stoneDropMap;
public:
    // 构造函数
    StoneDrop(const Vec2& position, Layer* targetLayer, StoneType type,
TMXTiledMap* tileMap);
};
class CropsDrop : public Drop {
private:
    // 农作物类型到掉落物类型和贴图路径的映射表
    static const std::unordered_map<CropsType, std::pair<std::string,
std::string>> cropsDropMap;
public:
    // 构造函数
    CropsDrop(const Vec2& position, Layer* targetLayer, CropsType type,
```

```
TMXTiledMap* tileMap);
};
```

5.2.7 Plough 类

Plough 类继承自 Sprite 类，用于管理耕地。耕地可以进行浇水、施肥等操作。

```
    class Plough : public Sprite{
private:
    LandState state;
    int non_watered_count;
    Vec2 tilePosition; // 瓦片坐标
    // 静态成员变量，用于记录需要移除的对象
    static std::vector<Node*> nodesToRemove;  // 使用 std::list，避免迭代器失效的问题
public:
    // 构造函数
    Plough(TMXTiledMap* tileMap, Layer* ploughLayer,const Vec2 tile, LandState state = LandState::Tilled);
    // 天数更新，遍历当前图层所有耕地更新
    void update();
    // 浇水
    void water();
    // 施肥
    void fertilize();
    // 还原（耕地）
    void restore();
    // 获取瓦片坐标
    Vec2 getTilePosition() const { return tilePosition; }
    // 获取状态
    LandState getState() const { return state; }
```

```
    // 静态方法：遍历 ploughLayer 的所有子节点并调用 update()
    static void updateAll(Layer* ploughLayer);
};
```

5.3 Person 设计与管理

5.3.1 Person 类的总体设计

Person 类是游戏中的玩家角色类，包含了背包管理、移动功能以及工具使用等功能，为实现功能，在 Person 里面还包含了 Bag 类和 Tool 类。

与 Map 不同的是，Person 的最终设计并未大量使用继承功能。在早期设计时，我们曾经设想过将 Person 设置为 cocos2d:;sprite 的子类，但最终选择了将 Sprite 设置为 Person 的一个成员。前一种设计的优点主要在于以下几点：

一是简单直观，因为 Person 类直接继承自 Sprite，可以直接使用 Sprite 的所有功能。

二是生命周期管理简单，因为 Person 的生命周期与 Sprite 的生命周期一致，无需手动管理。

三是代码简洁，使得可以直接在 Person 类中实现显示和动画。

这种设计模式的缺点在于 Person 和 Sprite 紧密耦合，灵活性低。而在本次项目中，游戏需要用户拥有如背包，工具等其他精灵。如果需要为 Person 添加工具的 Sprite,那么会形成一个 Sprite 的子类包含了另一个 Sprite 的情况，使得继承出现混乱。如果将两个都设置为 Sprite 的子类，那人物与工具的交互又会变得复杂。最终，面向对象设计中的一个重要的原则——组合优于继承，我们最终采用了将 Sprite 设置为 Person 的一个子类的设计方法。通过这个设计模式，可以避免继承带来的耦合问题，同时提高代码的灵活性和可维护性，使得类的职责分离，Person 负责游戏内在逻辑， Sprite 类负责屏幕上的显示，代码职责清晰明确。

Person 总体上主要包括了人物属性的私有成员：

```
std::string _name;
std::string _farmName;
int _sex;                                                    //玩家性别
int _level;
int _energy;
int _HP;                                                     // 玩家生命值
int _money;

int experience_all;
```

以及关于这些属性的功能函数

```
// 构造函数
 Person();
    // 初始化函数
 virtual bool init();
int toolLevel() { return _level > 200 ? 2 : 1; }
 //扣血
 void Person::decreaseHP(const int attack);
 //判断人物是否死亡
 bool isDead() { return _HP <= 0; }
 void dead();
 void levelUP();

 void moneyUP(int deltaMoney);
```

Person 类主要有 3 个子功能：Move，Bag，Tool。

5.3.2 Bag 类的总体设计

首先，Bag 类负责管理背包，包括添加、移除和显示物品。在实现上，它使用了 3 个 std::vector 来存储物品，物品对应的精灵和标签，并且在显示时根据预定义的位置排列物品的精灵和标签。作为对背包的管理，Bag 类有 addItem 和 removeItem 的方法，以及 displayBag 和 closeBag 的方法。同时，还有一个函数 ChangeBag 和成员变量 isOpen 用于判断当前背包是否打开，从而便于调用不同的函数。对于背包中存储的物品，定义了一个 item 结构体，具有 name，num 和 value。定义了结构体的两个构造函数，默认构造函数将 item 初始化为“nothing”，带参构造函数则需要输入名称，默认参数 num=1，value 根据预先定义的 std::map 来获取。

a.物品管理:

addItem(const item& MyItem): 检查背包中是否已有该物品，若有则合并数量，否则添加新物品。遍历_items 向量，找到相同名称的物品，增加数量，若未找到则在第一个“nothing”位置添加新物品。

removeItem(const item& MyItem): 减少物品数量，若数量<=0 则移除该物品。遍历_items 向量，找到相同名称的物品，减少数量，若数量<=0 则将其设置为“nothing”。

b.显示背包:

displayBag(): 创建背包背景精灵，遍历_items 向量，为每个物品创建精灵和标签，并按照 BAG_LEFT_LOCATION, BAG_UP_LOCATION 等常量定义的位置排列显示。

closeBag(): 移除所有物品精灵和标签，关闭背包背景。

```
class Bag : public Node
{
```

```
public:
    // 构造函数
    Bag();
    //监听器，按E打开背包
    void changeBag();
    // 添加工具或材料
    void addItem(const item& MyItem);
    // 移除工具或材料
    int removeItem(const item& MyItem);
    // 显示背包内容
    void displayBag();
    //关闭背包
    void closeBag();
    // 更新物品信息
    void updateItemInfo(cocos2d::Vec2 position);
    std::vector<item> getItems() { return _items; };
private:
    //物品列表
    std::vector<item> _items;
    //物品精灵列表
    std::vector<cocos2d::Sprite*> _itemSprites;
    //物品标签列表
    std::vector<Label* >_itemLabels;
    int _selectedItemIndex;
    bool isOpen;
};
```

5.3.3 Tool 类的总体设计

Tool 类是玩家使用的工具基类，管理工具的名称、等级和攻击力，并负责工具的使用和升级。私有成员包括名字，等级和攻击力。Person 具有私有成员 Tool* currentTool。通过 getTool 可以获得当前使用的工具，并对工具进行使用。

工具使用:

useTools(): 检查当前工具，显示工具精灵并运行动画序列，处理工具等级提升。

startFishing(): 特殊处理钓鱼动作，显示钓鱼竿动画，在整个钓鱼过程中维持。

endFishing(): 结束钓鱼动作，运行旋转和淡出动画。

其他功能的设计与实现逻辑

属性管理:

提供角色属性的 getter 和 setter 方法，如 getName、setName 等。

生命值管理:

decreaseHP 方法扣减生命值，并检查是否死亡。

死亡处理:

dead 方法处理角色死亡后的状态，如减少生命值、金钱和能量，并可能切换场景。

等级提升:

levelUP 方法根据经验值提升角色等级。

物品收集:

collectItems 方法检查角色当前位置是否有掉落物，若有则添加到背包中。

5.3.4Move 功能的总体设计

然后是移动功能，Person 类中有关于角色移动的动画创建和播放，还有移动逻辑的判断和处理。createAnimations 方法根据方向创建不同的动

画，PersonMove 方法根据移动方向切换动画，并判断是否可以移动，如果可以则移动地图以保持角色在屏幕中心，如果已到地图边缘则移动人物。

动画创建:

createAnimations(const std::string& direction): 根据方向创建对应的动画，加载相应图片帧并设置动画参数，如延迟时间和循环次数。

createAnimate(): 创建并存储四个方向的动画对象，并在需要时运行相应的动画。

移动判断与处理:

canMove(float deltaX, float deltaY, TMXTiledMap* currentMap): 计算目标位置，转换为瓦片坐标，检查瓦片地图中是否有阻挡物，通过获取地板层的 GID 值判断是否可以移动。

PersonMove(float deltaX, float deltaY): 根据移动方向切换动画，判断是否可以移动，若可以则移动角色或地图，保持角色在屏幕中心。

moveTileMap(const cocos2d::Vec2& playerPosition, TMXTiledMap* tileMap): 移动瓦片地图，使角色保持在屏幕中心。

```
Animation* createAnimations(const std::string& direction);
void createAnimate();


void collectItems();


void Person::useTools();
// 辅助方法：移动瓦片地图
void moveTileMap(const cocos2d::Vec2& playerPosition, TMXTiledMap* tileMap);
```

```
// 移动函数
void PersonMove(float deltaX, float deltaY);
void PersonStop(float deltaX, float deltaY);
//判断是否可以移动
bool canMove(float deltaX, float deltaY, TMXTiledMap* currentMap);
```

5.3.5 其他功能的设计

等级提升:

levelUP 方法根据经验值提升角色等级。

物品收集:

collectItems 方法检查角色当前位置是否有掉落物，若有则添加到背包中。在每次 move 时调用。

角色全局变量与依赖

Person leading_charactor; 全局变量，表示当前玩家角色。

extern NPC* npc1; 全局变量，表示 NPC 角色。

5.4 共有部分实现

5.4.1 设置界面与背景音乐

音乐的相关预加载和播放函数放置在 music.cpp 中，函数声明在 setting.h 内供 setting.h 调用。setting.cpp 用于实现设置界面以及音乐开关控制的实现。

```
// 预加载背景音乐
int preload_BGM()
//播放背景音乐
void play_BGM()
// 创建音量按钮
cocos2d::MenuItemToggle* GlobalLayer::createVolumeButton()
// 音量按钮的回调函数
void GlobalLayer::toggleBGM(cocos2d::Ref* pSender)
//全局界面中的时间显示
```

void GlobalLayer::updateTimeDisplay(float dt)

5.4.2 工具函数类别

本程序所有共有工具都统一放置在 totaltools.cpp 文件中。所有工具可以在其他需要使用时直接调用。其中包含以下工具：

Vec2 tile_change_screen(Size mapsize, Size tilesize, Vec2 original, float scale);

工具：瓦片坐标转世界坐标

Vec2 convertWorldToTileCoord(const Vec2& worldPosition, const Vec2& Tiledposition);

工具：世界坐标转瓦片坐标

bool random_bernoulli(double p);

工具：生成随机数

MyObject* getSpriteOnMap(Vec2 tilePosition);

工具：得到当前位置上是否有东西

void people_remove_change();

工具：将人移除出当前场景

void people_change_scene(const Vec2 change_Vec2);

工具：将人加入至当前场景

std::vector<Dropper*>* getDrops(Vec2 personPosition);

工具：得到掉落物

void treeBlock(Vec2 personPosition);

工具：判断碰撞关系

void updateTreeBlock(Vec2 personPosition);

工具：更新碰撞关系

bool is_have_plough(Vec2 tilePosition);

工具：判断当前位置是否有耕地

bool is_have_object(Vec2 tilePosition);

工具：判断耕地上是否有种植

bool isFiveTool(const std::string& name);

工具：是否为五大工具种类

```
void harvest(Vec2 tilePosition);
```

工具：收货

工具：管理当前场景与当前地图，许多场合需要知道当前所在场景与当前地图，调用一下方法可以得到当前场景与地图。

```
class MapManager
{
public:
    // 获取单例实例
    static MapManager* getInstance();
    // 获取当前场景的地图
    TMXTiledMap* getCurrentMap();
    //获取当前场景
    Scene* getCurrentScene();
    //获取当前地图大小
    Size getCurrentMapSize();
    //获取当前瓦片大小
    Size getCurrentTileSize();
private:
    // 构造函数和析构函数
    MapManager();
    ~MapManager();
    // 场景切换事件回调
    void onSceneChange(EventCustom* event);
private:
    // 单例实例
    static MapManager* _instance;
    // 当前场景
    Scene* _currentScene;
```

```
};
```

5.4.3 时间类的实现

GameTime 类实现了游戏中的时间总控，包括：时间的增加、日期的变换、天气的变化、季节的更替。

```
class GameTime {
private:
    int year;          // 当前年份
    int day;           // 当前天数
    Season season;     // 当前季节
    Weather weather;  // 当前天气 随机天气还没实现
    int totalDays;     // 总天数
    int time[2];       // 当前时间：time[0] 表示小时，time[1] 表示分钟
    // 单例模式：私有构造函数
    GameTime();
    // 单例模式：静态成员变量
    static GameTime* instance;
public:
    // 单例模式：获取单例实例
    static GameTime* getInstance();
    // 更新一天
    void updateDay();
    // 更新一天的时间
    void updateTime();
    // 启动自动更新
    void startAutoUpdate();
    // 停止自动更新
    void stopAutoUpdate();
    // 获取当前年份
    int getYear() const;
```

```
    // 获取当前天数
    int getDay() const;
    // 获取当前季节
    Season getSeason() const;
    // 获取当前天气
    Weather getWeather() const;
    // 获取总天数
    int getTotalDays() const;
    // 打印时间，调试用
    void printTime() const;
    // 返回时间：小时+分钟
    std::vector<int> getTime() const;
    //下雨天
    ParticleRain* RainLayers();
};
```

5.5 联网模式实现

在 cooperation.cpp 里面处理联网相关的初始界面制作以及联网后用户交互的逻辑实现。

5.5.1 监听线程与通信函数

```
//在后台线程中监听服务器消息，并处理接收到的消息。
void listenThread();
//启动监听线程，持续接收服务器消息。
void startListeningThread();
//向服务器发送消息。
void sendMessageToServer(const std::string& message);
//连接到服务器，返回连接是否成功。
bool connectToServer();
```

```
//清理 Winsock 资源，关闭套接字并释放相关资源。
void cleanupWinsock();
```

5.5.2 消息处理函数：

```
void handleMatchedMessage(std::string receivedMessage);
//处理匹配成功的消息，创建合作场景。
void handleFailedMessage(std::string receivedMessage);
//处理匹配失败的消息，显示失败提示并返回主界面。
void handlePlayerReleaseMessage(std::string receivedMessage);
//处理玩家按键释放的消息，更新合作者的动画状态。
void handlePlayerMoveMessage(std::string receivedMessage);
//处理玩家按键按下的消息，更新合作者的动画状态。
void handleTreeActionMessage(std::string receivedMessage);
//处理砍树动作的消息。
void handleMineActionMessage(std::string receivedMessage);
//处理采矿动作的消息。
void handlePlantActionMessage(std::string receivedMessage);
//处理种植动作的消息。
void handlePloughActionMessage(std::string receivedMessage);
//处理耕地动作的消息。
```

5.5.3 场景初始化函数

```
bool scene_coop::init();
//初始化合作界面，设置背景、标题和按钮。
bool scene_create::init();
//初始化创建界面，设置背景、标题和输入框。
bool scene_join::init();
//初始化加入界面，设置背景、标题和输入框。
bool ConnectingScene::init();
//初始化连接界面，显示“Connecting...”提示并启动监听线程。
```

6 项目功能测试

6.1 系统测试

6.1.1 时间系统测试

6.1.1.1 树/农作物生长测试



6.1.1.2 农作物死亡测试

6.1.1.3 节日测试

6.1.1.4 换季测试

6.1.2 随机天气系统测试

6.1.3 场景切换测试

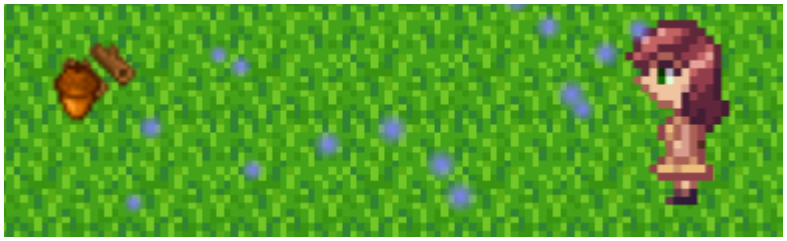
6.1.4 背景音乐测试

6.2 工具测试

6.2.1 耕地测试

6.2.2 种植测试

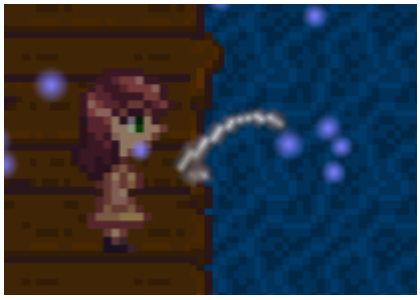
6.2.4 掉落物拾取（收获）测试



6.2.5 砍树测试

6.2.6 凿矿测试

6.2.7 钓鱼测试



6.2.8 浇水测试