厦門大学

Fishlime

——程序设计实践报告

姓　　名:	马成龙、吴雨凡、杨弈鸣
学　　号:	25120232201399、11920232202590、 11920232202611
院　　系:	计算机科学与技术系、软件工程系
专　　业:	计算机科学与技术、数字媒体技术
年　　级:	2023
指导教师:	郑宇辉

目录

第一章　知识拆解与分析

1.1 面向对象的设计步骤与应用

(一) 类的一般设计方法

1. **找名词；**
2. **找动词和形容词；**
3. **理关系；**
4. **再分组。**

1、**找名词：**

目的：提炼出【类】。

通过游戏原型，在游戏设计蓝图中，把所有出现的具体事物和抽象概念都列出来。这些名词是“类”的主要候选者。

在我们的游戏中：

“一个史莱姆角色在沙滩上移动，它可以在水边钓鱼。钓鱼时会进入一个 QTE 状态。游戏开始时有一个主菜单，可以开始游戏或查看帮助。”

从这段描述中，我们可以找出以下名词：

具体事物：

- 史莱姆（操作角色） → Player 类；
- 沙滩 → Background 类。

抽象概念：

- QTE 状态 → QTEManager 类（管理 QTE 逻辑）；
- 主菜单 → MenuState 类（主菜单界面”）；
- 帮助 → HelpState 类（帮助界面）；
- 游戏 → Game 类。

游戏设计的第一步，是分析游戏原型中的核心元素。游戏中的“名词”，无论是像“玩家”这样的实体，还是像“主菜单”这样的概念，都可以被直接映射为程序中的“类”。依此，可以逐个提炼、设计出 Player 类、MenuState 类等所有需要用到的类。

2、**找动词和形容词：**

目的：明确【方法】和【属性】。

确定了“有哪些事物”（类）之后，下一步是描述这些事物“能做什么”（方法）和“拥有什么特征”（属性）。

动词 → 对应类的方法，代表对象的行为。

形容词 → 对应类的属性，代表对象的状态和特征。

以 Player 类为例：

找到动词 (行为)：

- 角色可以移动 → Player::move(direction)；
- 角色可以钓鱼 → Player::startFishing()；
- 角色可以播放动画 → Player::updateAnimation()。

找到形容词/特征 (状态)：

- 角色的位置 → sf::Vector2f m_position；
- 角色的移动速度 → float m_speed；
- 角色的动画状态（静止，移动还是钓鱼） → Animation m_currentAnimation；
- 角色钓到的鱼的数量 → Inventory 类。

通过这一步，明确类里应该有什么，能做什么。

以上可以设计出一个最简单的 Player.hpp：

```
1  #include <SFML/Graphics.hpp>
2  #include "Inventory.hpp"                    // Inventory 类，存储鱼的数量
```

```
3
4   class Player {
5   public:
6       Player();
7       void handleInput();                                  // 处理输入
8       void update(sf::Time deltaTime);                     // 更新状态
9       void draw(sf::RenderTarget& target);                 // 绘制
10
11      void move(const sf::Vector2f& direction);
12      void startFishing();
13
14  private:
15      void updateAnimation();
16      sf::Sprite m_sprite;
17      sf::Texture m_texture;
18      sf::Vector2f m_position;
19      float m_speed;
20      PlayerState m_state;
21      Inventory m_inventory;
22  };
```

3、**理关系：**

目的：构建【类结构】。

类间有三种关系：组合、聚合与继承。

我们在游戏设计中组织类间关系时，主要用到两种关系：继承 和 组合。

继承： 表示一个类是另一个类的特殊化。例如，PlayState、MenuState、HelpState 都“是一种”State。这是状态机模式的基础，通过一个共同的基类 State 指针，就可以管理所有不同的游戏状态。

组合： 表示一个类包含另一个类的实例，并且两者的生命周期紧密绑定。如果容器对象被销毁，其包含的对象也会被销毁。例如，PlayState“拥有一个”Player 对象。当 PlayState 结束时，该状态下的 Player 实例也随之消亡。这通常通过将成员对象直接声明在类中来实现：Player m_player; 。

聚合： 表示一个类使用另一个类的实例，但并不拥有它。两者的生命周期是独立的。例如，PlayState“有一个”指向 ResourceManager 的引用。PlayState 需要使用资源管理器来加载纹理，但它不负责创建或销毁资源管理器。

4、**再分组：**

目的： 创造【管理器类】进行优化。

除了游戏的核心逻辑，还会有一些“公共服务”的需求，它们不属于任何一个特定的游戏对象。

比如：

谁来加载图片和字体资源？如果 Player 和 Background 都自己加载，会造成重复和浪费。

谁来播放背景音乐和音效？

谁来负责存档和读档的复杂文件操作？

解决方案：创建专门的管理器类。

依据：单一职责原则。

例如：

- 管理资源加载 → ResourceManager 类；
- 管理音频播放 → AudioManager 类；
- 管理存档读写 → SaveManager 类。

ResourceManager 专门负责所有资源的加载和缓存，这样 Player 类就不需要关心文件读写的细节，只需向

ResourceManager 请求自己需要的纹理即可。

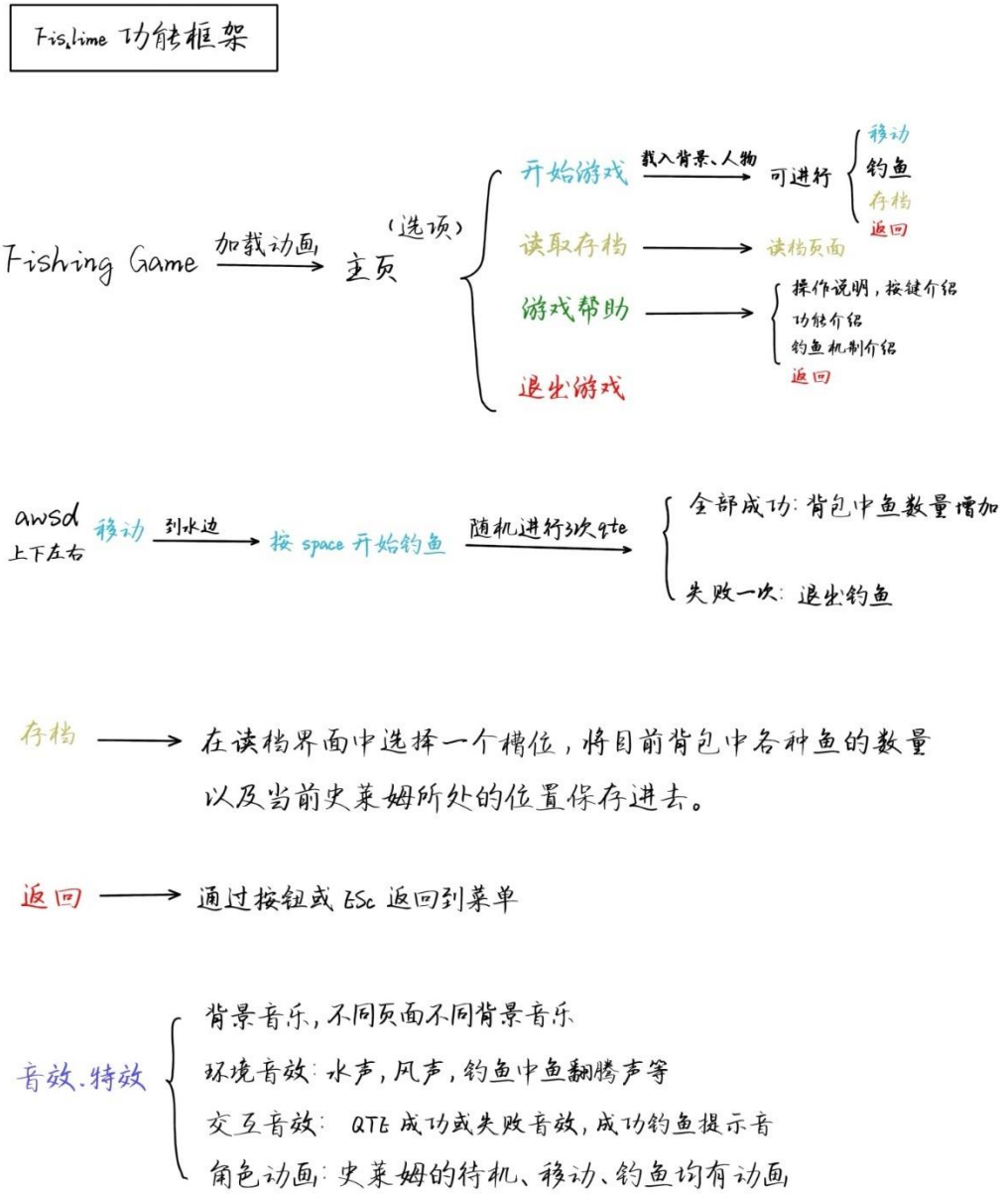

图 1：Fishlime 功能框架

根据游戏内容可总结出游戏的功能框架，再通过类的一般设计方法，即可得到游戏的类的设计，具体见下。

（二） 类设计概览

我们的游戏 Fishlime 采用了明确的面向对象(OOP)设计，将游戏各部分职责拆分为不同的类。首先，根据游戏原型中的名词识别出主要类，包括：Game（游戏核心）、Player（玩家角色）、Background（背景场景）、MenuState/PlayState/PauseState 等状态类、QTEManager

（QTE 逻辑管理）、Inventory/Item（物品与背包）、AudioManager（音频管理）、ResourceManager（资源管理）、Button（按钮 UI）等。接着根据动词和形容词确定每个类的方法和属性，例如 Player 具有移动、钓鱼、播放动画等方法，包含位置、速度、当前动画状态等属性。最后整理类之间的关系，通过**继承**和**组合**实现模块化设计。

（三）核心类与关系

Game 类是核心，负责初始化窗口和全局资源，持有主要管理对象。它包含渲染窗体（ sf::RenderWindow ）、纹理和字体资源管理器（ ResourceManager<sf::Texture> 和 ResourceManager<sf::Font> ）、游戏状态栈（ StateStack ）、全局玩家对象（ Player ）、以及辅助结构 Context 来共享这些资源。Game 在构造时创建窗口和资源，并设定初始状态为菜单。它通过成员函数接口集中调度全局行为，例如更换当前状态、显示临时消息等。

- Game 类是核心，负责初始化窗口和全局资源，持有主要管理对象。它包含渲染窗体(sf::RenderWindow)、纹理和字体资源管理器(ResourceManager<sf::Texture>和 ResourceManager<sf::Font>)、游戏状态栈(StateStack)、全局玩家对象(Player)、以及辅助结构 Context 来共享这些资源。Game 在构造时创建窗口和资源，并设定初始状态为菜单。它通过成员函数接口集中调度全局行为，例如更换当前状态、显示临时消息等。
- **状态类(State)**：采用**状态机**模式管理游戏流程。定义了抽象基类 State，提供 handleEvent()、update()、draw() 纯虚方法，以及生命周期钩子 onCreate(), onActivate() 等供子类覆写。具体状态如菜单 MenuState、游戏进行 PlayState、暂停 PauseState、背包 InventoryState、帮助 HelpState、存读档 SaveState/LoadState 等均继承自 State。状态类之间通过 **状态栈 (StateStack)** 协同工作，允许多个状态叠加（例如暂停界面覆盖在游戏状态上）。Game 持有一个 StateStack 实例，并提供模板方法便捷地更改或叠加状态。
- **游戏元素类(GameObject)**：抽象基类 GameObject 表示场景中可更新和绘制的对象，提供纯虚的 update() 和 draw() 接口。Player 和 Background 等游戏元素继承自 GameObject。GameObjectManager 则通过组合模式持有多个 GameObject 对象，用于在游戏状态中批量管理更新和绘制。例如在 PlayState 中，Background 被加入对象管理器统一更新、渲染。这种设计提高了**内聚性**和**可扩展性**——新添的游戏元素只需继

承 GameObject 并加入管理器即可参与游戏循环。

- **功能组件类**：将独立功能封装为小型类/结构，以通过组合使用。在我们的游戏中，Animation 结构保存帧列表、帧时长等动画数据；Animator 类管理一组动画，提供 play()、update()、draw() 等方法控制动画播放。MovementBoundary 提供多边形边界内检测的方法，用于限制玩家移动范围；QTEManager 管理钓鱼 QTE 事件逻辑（随机提示按键、计时和成功判定等）；AudioManager 采用单例模式集中管理声音资源和播放；ResourceManager 是模板类，统一加载和提供纹理、字体等资源，避免重复加载。
- **物品与背包类**：定义抽象基类 Item 表示可收集/使用的道具，只有获取名称和使用两个纯虚接口。具体道具如 FishItem 继承 Item，代表钓到的鱼，包含鱼的重量、稀有度、纹理 ID 等属性。FishItem::use() 可定义使用该鱼道具对玩家的效果(例如回血)。玩家的 Inventory 背包使用组合方式包含一组 Item（内部用 std::vector<std::unique_ptr<Item>>），提供添加、移除和访问接口。这种设计使得将来扩展更多道具类型时，只需派生新的 Item 子类而无需修改背包逻辑。

类关系结构：Game 拥有（组合）StateStack、资源管理器、玩家和窗口；StateStack 内部维护一个 State 指针栈并通过基类多态调用当前状态的方法；各具体 State 拥有（聚合）对 Game 的引用（访问全局资源和切换状态）以及需要的其他对象引用（如 PlayState 持有 Player 引用以更新玩家，SaveState/LoadState 持有 Player 用于存取数据）；Player 拥有动画管理 Animator、QTE 管理 QTEManager、背包 Inventory、边界检测 MovementBoundary 等组件；Player 和 Background 作为 GameObject 的子类被组合进 GameObjectManager 中，由 PlayState 管理。AudioManager 作为全局单例供各处按名称播放声音；ResourceManager 被 Game 持有并通过 Context 提供给各状态使用。按钮 Button 作为 UI 控件，不继承 GameObject 而是在需要的状态中以对象形式存在。所有这些类各司其职又通过指针/引用协作，实现了高内聚、低耦合的系统结构。

图 2：类关系结构

1.2 事件驱动机制原理与面向过程的区别

事件驱动简介：游戏采用了事件驱动的输入处理机制。与传统面向过程程序中持续轮询输入状态的方式不同，事件驱动通过**事件队列**异步地通知发生的用户交互，使程序以**回调/响应**模式执行相关逻辑。具体而言，SFML 提供了事件系统：每当窗口发生键盘、鼠标等交互，就生成一个 sf::Event 放入队列。游戏循环中调用 window.pollEvent() 按顺序取出事件并处理。这种方式将输入采集和响应解耦，提高了代码的组织性和可维护性。

在 Fishlime 中，主循环的第一步即**事件轮询**：在每帧开始时，使用 while (auto evOpt = m_window.pollEvent()) 获取当前所有待处理事件。对于每个事件，根据其类型分别处理：例如，若事件是窗口关闭，则调用 m_window.close() 退出游戏循环；若事件是键盘按下，则可以检查具体按键并执行对应逻辑（如数字键 1-9 用于音量控制，在代码中通过 sf::Event::KeyPressed 事件调整音量并显示提示消息）。处理完特定事件后，Game 会将事件转发给当前状态栈处理：m_states.handleEvent(ev)。这意味着当前活动的状态对象（如游戏界面或菜单界面）会进一步处理该事件，从而实现输入对不同游戏场景的**分发**。

区别于面向过程：在**事件驱动**框架下，**程序被动等待**事件发生并响应，逻辑的执行顺序取决于用户输入的触发顺序；而**面向过程**风格通常是**主动轮询**各项条件，例如在每帧中用 if 语句检查按键是否被按下、是否达到某条件，然后执行固定流程。这种区别体现在代码结构上：Fishlime 并没有在循环中直接写死对每个按键的检查逻辑，而是利用事件来驱动。例

如暂停游戏功能，当玩家按下 Esc 键时，会产生一个 KeyPressed 事件，PlayState::handleEvent 中捕获该事件进行暂停处理；而面向过程的做法可能是在每次循环都检查 if (Esc 按下) {...}。事件驱动让代码更简洁，并且更易于拓展新的事件处理，而无需修改主循环结构。

事件调度与封装：事件驱动机制结合 OOP 还体现为将事件处理责任分散给不同对象。Fishlime 中，各 UI 按钮、状态和玩家对象都实现各自的事件处理函数，只有在需要时才对特定事件作出反应。例如，Button 类封装了鼠标悬停和点击事件处理，如果当前鼠标位置进入按钮区域且发生左键按下事件，就调用预先绑定的回调函数。主循环并不知道按钮内部细节，只负责把事件传递给按钮对象，由按钮自行决定如何响应。这种设计避免了大量集中繁杂的 switch-case 判断，根据单一职责原则，各类仅处理与自己相关的事件，使系统更具模块化。

总结：事件驱动机制使 Fishlime 在输入处理上更灵活高效。程序流程不再是严格线性的，而是根据用户操作实时驱动。相比之下，面向过程的循环检查方式会浪费性能且难以管理复杂交互。在 Fishlime 中充分利用了 SFML 事件系统，实现了清晰的交互逻辑。

1.3 游戏图像显示机制

核心：渲染管线与双缓冲。

渲染管线流程：Fishlime 的主循环采用固定的渲染流水线顺序来更新画面，具体步骤为：清除上一帧内容、绘制当前帧场景、显示新帧。每帧逻辑更新完毕后，Game::run() 中执行渲染部分：

1. **清屏：**调用 m_window.clear()清除窗口的颜色缓冲,将其填充为背景色(默认为黑色)。这一步确保不会残留上帧图像，为新一帧绘制做好准备。
2. **绘制场景：**调用当前活动状态栈的 draw()方法，将游戏元素绘制到渲染目标上。例如在游戏状态下，StateStack::draw 会依次绘制背景、玩家、UI 等元素；在叠加状态下（如暂停），也可能绘制底层被遮罩的场景。需要注意的是，此时所有绘制操作实际上都画在后台缓冲区，并不会立即显示给玩家。
3. **绘制叠加内容：**在 Fishlime 中，还有一些临时内容或覆盖 UI 需要特殊绘制。例如在主游戏循环中，如果有临时消息（例如音量调整提示），则在主要场景绘制后，再绘制这条消息文本。对于状态叠加，StateStack::draw 方法也会在绘制下层场景后，添加一

个半透明黑色矩形作为遮罩，然后再绘制上层状态，从而实现下层场景变暗的效果。这些都发生在调用 display()之前。

4. **显示缓冲**：最后调用 m_window.display()将当前后台缓冲呈现在屏幕上。这一步完毕后，新帧就显示给玩家看，渲染过程进入下一循环。

双缓冲机制：SFML 底层采用双缓冲(Double Buffering)技术来避免屏幕闪烁和撕裂。简单来说，窗口维护两个缓冲区：**前台缓冲**（当前屏幕显示内容）和**后台缓冲**（正在绘制的新帧）。window.clear()和所有 draw 调用实际上作用在后台缓冲上，构建下一帧图像；当调用 window.display()时，后台缓冲和前台缓冲交换（Swap）。交换后，新帧成为前台显示内容，而之前的前台缓冲变为后台缓冲供下一帧使用。这一机制保证了一帧内所有绘制操作要么全部显示，要么完全不显示，避免了逐个绘制到屏幕造成的中间态，提高了显示稳定性和流畅度。

通过双缓冲，Fishlime 实现了平滑的动画和画面呈现。例如在玩家移动或动画播放时，各帧画面衔接自然，没有出现图像撕裂或闪烁。渲染管线严格按照“清除-绘制-显示”顺序，使每帧内容完整绘制后再一次性输出。值得注意的是，在开发中一旦忘记调用 display()，就会看不到画面更新；而若缺少 clear()则可能看到上一帧残影。Fishlime 遵循了正确的渲染流程，从而保证了**帧缓冲同步**和**画面稳定**。

1.4 输入处理机制（键盘输入）

SFML 提供了两种主要输入处理方式：**事件驱动式**（基于 sf::Event）和**状态查询式**（基于实时状态查询，如 sf::Keyboard::isKeyPressed()）。 Fishlime 针对不同场景选择不同的方式，在不同情况下选择最合适的输入处理方式，以实现既准确又流畅的输入响应。

事件式输入处理：事件式适用于一次性触发的操作，即用户进行某个输入动作的瞬间触发逻辑。Fishlime 大量使用事件机制处理这类情况。例如：

- **菜单/按钮点击**：通过捕获鼠标按钮按下事件来触发按钮的 onClick 回调。Button::handleEvent 中检测到sf::Event::MouseButtonPressed且左键点击发生在按钮区域时，立即调用绑定的动作函数（比如开始游戏、打开帮助等）。这样只有在点击瞬间触发一次，避免重复执行。
- **快捷键触发行为**：如在游戏进行中，按下 Esc 键会暂停游戏。这通过在

PlayState::handleEvent 中捕获 KeyPressed 事件并判断是否为 Escape 键来实现。一旦捕获到，立即调用 StateStack 推入暂停状态，从而在按键瞬间完成状态切换，而不会因为按键长按而多次触发。

- **钓鱼操作起点**：玩家按下空格键抛竿开始钓鱼，这是一个瞬时动作，也采用事件处理。在 Player::handleEvent 中，当检测到空格键按下事件且当前玩家状态为 Idle、位于水边，就切换玩家状态为开始钓鱼。同理，在 QTE 过程中，玩家按键的判断也基于事件。

事件式处理的特点是**响应精准**，只在输入发生的一刻响应一次，不会漏也不会连发。Fishlime 中涉及模式切换、一次性动作的输入（如暂停、开始/结束钓鱼、按钮点击等）均采用事件机制实现，确保了逻辑触发的准确性和及时性。

状态查询式输入处理：状态查询更适合**持续性**的输入检测，即根据输入设备当前状态每帧执行相应逻辑。Fishlime 将连续运动这类需求用状态查询实现。例如玩家的上下左右移动：

在 Player::update() 中，通过 sf::Keyboard::isKeyPressed 连续查询方向键 WASD 键是否被按住。如果任意移动键处于按下状态，则视为玩家正在移动，设定 isMoving = true 并切换玩家状态为 Moving，从而开始播放移动动画。然后根据按键组合计算移动方向向量，并持续更新玩家的位置。只要按键保持按下，每帧都会更新位置，实现平滑的移动。相反，一旦按键松开，对应的 isKeyPressed 返回 false，移动逻辑就会自动停止（玩家状态切回静止 Idle）。

这种连续检测方式适合需要**随时间积累效果**的输入：移动、射击连发、摄像机平移等。如果用事件处理移动，按下一次键只能移动一步，必须繁琐地处理长按。而状态查询能够天然支持长按：当键一直按住时，每帧都检测到 true，于是持续发生移动。

应用情境区别：

- **事件式**用于**离散动作**：菜单确认、模式切换、抛竿/收杆、暂停、背包开关等，这些操作一次按键只应触发一次逻辑，采用事件最合适。比如钓鱼 QTE 中，每次提示按键只需捕捉一次正确按键事件即可判定成功或失败。再比如按 Esc 暂停时，用事件避免了玩家长按 Esc 导致反复进出暂停的情况。
- **状态式**用于**连续状态**：如玩家移动。项目中还使用状态查询实现了移动时的音效处理——在玩家持续移动的动画中，每当动画帧循环到特定帧就播放脚步声，直到松开键停止

动画。这也依赖于每帧知道按键仍然按住。

两种方式相辅相成，使游戏操作既灵敏又平滑。需要强调的是，Fishlime 在实现时也注意了两者的协调：例如在玩家钓鱼(Fishing)状态下，Player::handleEvent 会优先把按键事件交给 QTEManager 处理而忽略移动按键，以免钓鱼过程中误触发移动。总的来说，事件式保证**动作**，状态式保证**过程**，正确场景下使用各自最佳。这样的输入架构提高了游戏操控体验的准确性和流畅性。

1.5 存档与读档机制

Fishlime 实现了游戏进度的存档和读档功能。项目中通过状态类 SaveState 和 LoadState 来承载存读档界面及逻辑，可将玩家位置和背包物品保存到文件，再从文件加载恢复。这部分可以视作“SaveManager”功能的实现，负责**序列化游戏数据**与**文件操作**。

SaveState 设计：当玩家在游戏中选择保存时，程序会推入 SaveState（存档状态）。SaveState 内部维护了若干槽位按钮（槽位 1~4），点击对应按钮即可将当前进度写入该槽位文件。核心保存逻辑封装在 SaveState::saveToSlot(int slot) 方法中：

- 它首先确保存档目录存在（ load 文件夹 ）并构建目标文件路径，如 load/slot1.txt。然后以输出流方式打开文件。
- **序列化数据**：按照约定格式，将玩家当前状态写入文件。格式分两行：第一行保存玩家坐标位置（以浮点数 .f 形式），两数表示 x 和 y；第二行保存玩家背包中各类鱼的数量。代码实现中，先获取玩家位置向量，将 pos.x 和 pos.y 写入并以换行结束。接着统计背包内每种鱼的数量：Fishlime 预设了 5 种鱼（ID 分别为 "fish1"..."fish5"），通过遍历玩家的 Inventory，对每个 FishItem 读取其纹理 ID，归类计数。然后将五种鱼的数量依次输出到文件第二行，各数量以空格分隔。例如，文件内容可能是：

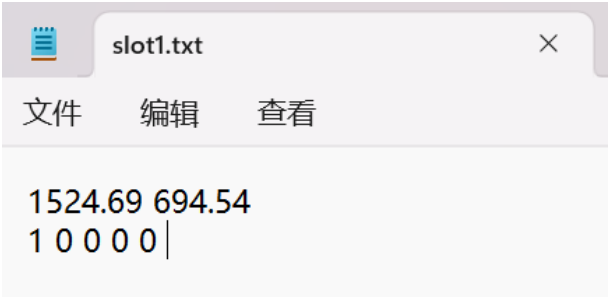

图 3：存档文件格式

表示玩家位置(1524.69, 694.54),拥有 1 条 fish1、0 条 fish2、0 条 fish3、0 条 fish4、0 条 fish5。最后关闭文件流并在控制台打印保存成功提示。

以上过程确保了保存文件是纯文本格式，便于调试查看。其中每种鱼的顺序和含义是固定的（需要与加载对应），避免存储额外的元信息。这种简洁文本序列化也使文件可扩展——若以后增加新鱼种，可在行中追加数字，方便快捷，体现 OOP 思想。

LoadState 设计：当从主菜单进入读档界面（LoadState）时，会显示各槽位的状态（空或有多少条鱼）。点击某槽位按钮则调用 LoadState::loadFromSlot(int slot) 从对应文件加载：

- 打开文件输入流，按保存时的格式读取。第一行读取两个浮点数作为玩家坐标。实现中通过 std::getline 读整行，再用 std::istringstream 提取 x 和 y。然后调用 Player::setPosition({x,y}) 将玩家移动到存档位置。
- 第二步，清空当前玩家背包。由于设计 Inventory 时，没有预设 clear 方法，游戏中采取循环 remove 第 0 项的方式反复删除,直到背包为空。清空背包可避免叠加旧数据。
- 第三步，读取第二行物品数量并重建道具。代码同样用 getline 获取整行，然后用 istringstream 逐个读出数字。按照与保存一致的鱼 ID 顺序，将每个数量对应的鱼道具添加回玩家背包：对于每个 fishId，读取一个数量值 cnt，然后循环 cnt 次创建该鱼对象并加入背包。FishItem 构造时使用鱼 ID 最后一位数字作为重量和稀有度（例如"id"为"fish3"，则 val=3，对应重量 3，稀有度 3）。若后续想完善游戏内容，还可以设计 id 与稀有度和重量不一一对应的鱼。这样就重现了保存时的道具集合。

完成上述步骤后，玩家位置和背包物品均恢复到存档时刻，接下来程序会切换回游戏进行状态(PlayState)，玩家即可从存档进度继续游戏。

调用逻辑：项目通过状态切换无缝整合了存读档流程：

在主菜单 MenuState 中，“读取存档”按钮被按下时压入一个新的 LoadState；游戏内“存档”按钮则推入 SaveState。进入存档/读档状态后，界面切换，玩家实际游戏暂停在后台（状态栈）。玩家在读档界面选择槽位加载时，LoadState 立即进行文件操作，然后调用 changeState<PlayState> 返回游戏。在读档成功后，LoadState 里调用 m_game.changeState<PlayState>(m_game.getPlayer())，清空状态栈并重新以玩家对象启动游戏状态。

数据完整性与反馈：LoadState 在进入时还会调用 updateSlotStatus() 检查每个槽位文件是否存在及其内容，更新显示的状态文本。如果槽位有存档，会读取鱼总数并显示“槽位 X + 鱼：总数”；无存档则显示“槽位 X + 空”。保存完成后，SaveState 也调用 updateSlotStatus()刷新各槽信息，用户可立即看到刚才保存的成果（如鱼总数变化）。这些设计提高了易用性，防止误操作。

总结：我们将 Fishlime 的存读档机制设计地简单实用，将玩家关键数据（位置和背包）以**文本**存储实现了**序列化**，利用状态界面与文件 I/O 配合实现了**游戏进度持久化**。尽管当前格式较基础，但已满足需求且便于后续扩展。通过 SaveState/LoadState 的封装，主游戏逻辑与文件操作解耦，不会因为存档流程而干扰正常游戏循环。

1.6 游戏主循环设计

游戏主循环是游戏引擎的核心，它负责在每帧执行输入处理、游戏更新和渲染。Fishlime 的 Game::run() 方法使用经典的主循环结构，并运用了 SFML 的时间工具类 sf::Clock 来控制更新频率，从而达到近似固定步长的逻辑更新效果（每秒 60 帧）。这确保游戏在不同性能机器上运行结果一致，并避免了逻辑因帧率波动而不同步。

Fishlime 中，Game::run() 主循环逻辑概要如下：

```
1   while (m_window.isOpen()) {
2       // 1) 事件轮询
3       while (auto evOpt = m_window.pollEvent()) { … }
4       // 2) 更新
5       sf::Time dt = clk.restart();
6       m_states.update(dt);
7       // 3) 渲染
8       m_window.clear();
9       m_states.draw(m_window);
10      m_window.display();
11      // 4) 延迟动作
12      if (m_scheduledAction) { … 执行延迟任务 … }
13      // 5) 应用挂起的状态操作
14      m_states.applyPending();
15  }
```

每次循环都会执行固定顺序的五个步骤：

- **事件处理**：如前所述，轮询并处理所有输入事件。这一部分完成后，当前帧的输入影响（如改变状态、触发动作）便已登记。
- **逻辑更新**：使用 sf::Clock 来获取上一帧到当前的时间差 dt，然后调用状态栈的 update(dt)。Clock::restart() 返回并重置计时器，相当于记录了帧间隔时间（1/60 秒）。各游戏对象会根据 dt 更新自己的状态，例如根据时间移动插值、播放动画过帧等。这种做法实现了帧率无关的运动：不论实际帧率高低，dt 补偿时间差，使运动速率恒定。例如玩家移动每秒 200 像素，无论 60FPS 或 30FPS，dt 不同但乘积近似保持每秒 200 像素移动。 Fishlime 将窗口帧率限制设置为 60，这使得 dt≈0.0167s 恒定，相当于固定步长模拟（每秒 60 步）。这既降低了 CPU 空转浪费，也方便逻辑简单处理。如果帧率偶尔波动，dt 也能适应，使游戏节奏不受影响。
- **渲染绘制**：如前所述，清除窗口并绘制当前状态场景和 UI，然后调用 display() 切换前后缓冲，输出画面。渲染与逻辑分离，使游戏逻辑更新和画面刷新同步进行，每帧一套完整逻辑+渲染。由于游戏逻辑较简单，没有单独采取更复杂的固定更新步长+插值渲染方案，60FPS 已足够流畅。
- **延迟任务执行**：这是 Fishlime 主循环中一个巧妙的设计点。Game 提供了 scheduleAction(std::function<void()>) 接口，用于安排在当前帧结束时执行一次回调。在主循环渲染完毕后，会检查是否有 m_scheduledAction 待执行，如有则调用之。这个设计巧妙在于，有些操作（特别是改变游戏状态栈，如 push/pop）不宜在处理中途立即执行，否则可能破坏当前迭代的稳定性。例如在状态的事件处理函数里调用 Game::changeState 会马上清空栈，造成当前状态自身上下文失效。通过 scheduleAction，可以把这些操作推迟到帧末统一执行，确保当前帧逻辑完整执行完，再安全地修改状态。这种延迟执行确保了状态改变的原子性和主循环流程的稳定。
- **应用挂起的状态操作**：紧随延迟任务之后，循环最后调用 m_states.applyPending()。StateStack 在处理状态切换请求（push/pop/clear）时，出于安全考虑并不直接修改栈，而是将请求暂存到一个 m_pending 列表。只有在每帧结尾，才统一应用这些更改。在 applyPending 中，对每个挂起操作进行：如果是 Push，则将新状态入栈并调用其 onCreate()（第一次创建）和 onActivate()（激活）；如果是 Pop，则销毁栈顶状态并调用相应钩子，让下层状态 onActivate()恢复；Clear 则清空整个栈。这样设计配合 scheduleAction，保证了状态栈修改总是在帧末执行，避免冲突。此外，applyPending

放在循环最后，使得新的状态会在下一帧开始处理事件和更新，确保当前帧逻辑的一致性。

总结：Fishlime 的主循环遵循了事件->更新->渲染的典型模式，并通过 SFML Clock 控制帧步长。由于限制帧率为 60，每帧间隔基本固定，实现类似固定时间步长的效果。这简化了逻辑实现，无需复杂插值。此外，独特的延迟任务和挂起状态机制，解决了状态切换时序问题，使状态机与主循环配合顺畅。在开发和调试过程中，这种设计也方便了问题定位——每个阶段职责清晰，帧循环的执行顺序和结果是确定的。

1.7 状态栈与状态机的使用

该条是基于我们的游戏内容，对于知识点的拓展。

状态机架构：Fishlime 采用了**栈式状态机（State Stack）**架构来管理游戏的各种界面和模式，实现了不同游戏状态的**分离与切换**。核心在于抽象基类 State 和管理类 StateStack 的设计：

- 抽象类 State 定义了一系列接口函数：handleEvent() 处理输入事件，update() 更新逻辑，draw() 绘制内容，以及生命周期钩子 onCreate() (首次创建时调用)、onActivate() (每次激活时调用)、onDeactivate() (被覆盖或暂停时调用)、onDestroy() (最终销毁时调用)。这些使具体状态具备统一的行为接口，又允许不同状态有各自实现，符合**里氏替换原则**。例如，MenuState 和 PlayState 都继承自 State，它们能互相替代地被 StateStack 管理。
- StateStack 是状态容器与控制器。内部维护一个 std::vector<std::unique_ptr<State>> 作为栈，以及一个挂起操作列表。它提供 pushState<T>(Args...) 模板方法入栈新状态、popState() 出栈、clearStates() 清空等接口。但这些操作并不立即生效，而是记录为 m_pending 操作对，等待帧末统一处理（前节已详述其好处）。StateStack 的 handleEvent/update/draw 实现是简单地委托给当前栈顶状态，因此任意时刻只有**栈顶状态**在运行，其余状态要么已退出要么在栈底暂存（被遮罩）。

具体状态类功能：Fishlime 根据游戏需求实现了多个 State 子类，每个对应一种游戏情景：

- **MenuState（主菜单）**：显示游戏标题和四个按钮选项（开始游戏、帮助、读取存档、退出）。它的 onCreate()加载菜单所需的纹理和字体，创建并布局所有按钮。每个按钮通过设置回调实现不同功能：比如“开始游戏”按钮在点击时，调用 Game::changeState<PlayState>(player) 启动游戏；“帮助”按钮则 pushState<HelpState>() 叠加帮助界面；“读取存档”按钮 pushState<LoadState>(player)打开读档界面。MenuState 的 handleEvent()将事件分发给每个按钮，让按钮自行处理鼠标悬停和点击。MenuState 没有动态元素，update()为空实现，draw()按顺序绘制背景、标题和按钮。
- **PlayState（游戏进行）**：这是核心游戏状态，游戏主玩法（角色在沙滩上走动钓鱼）都在此状态发生。PlayState 在 onCreate()时完成场景初始化，包括：清空旧 GameObject 列表、防止重复添加；创建背景对象并加入管理器；加载玩家动画（Idle 等待、Move 行走、Start 抛竿、Fishing 等待、Finish 收杆共 5 组，每组若干帧）并设置初始动画 Idle；设置玩家移动边界（通过一组顶点定义允许活动的多边形区域）；创建界面按钮，如“返回菜单”按钮（点击后通过 changeState<MenuState>()返回主菜单）、“背包”按钮（点击 pushState<InventoryState>(player)打开道具界面）、“存档”按钮（点击 pushState<SaveState>(player)进入存档界面）。PlayState 的 handleEvent 负责处理游戏过程中输入：如检测 Esc 键按下以 pushState<PauseState>()弹出暂停菜单；将事件传递给玩家对象和场景对象管理器（例如 WASD 按键由玩家自己处理移动）；最后将事件传递给 UI 按钮处理鼠标交互。PlayState 的 update(dt)则每帧调用玩家 update 和对象管理器的 updateAll 来更新动画、移动等，draw()绘制场景元素、玩家和 UI 按钮。
- **PauseState（暂停）**：作为覆盖在 PlayState 之上的暂停菜单，它显示一个“游戏已暂停”标题和两个按钮（继续游戏、退出到菜单）。PauseState 在 onCreate()中初始化字体和文本，并创建按钮；“继续游戏”按钮的回调是调用 Game::popState()恢复游戏；“退出到菜单”按钮则通过 Game::changeState<MenuState>()返回主菜单。PauseState 的 onActivate()会重新布局文本和按钮位置居中屏幕。handleEvent()中，按钮自行处理鼠标点击，此外还捕捉 Esc 键按下作为快速继续游戏的快捷方式（与点击“继续”相同效果）。PauseState 的 update()无需内容，因为暂停界面没有动画或计时逻辑；draw()则绘制暂停标题和两个按钮。在状态栈中，PauseState 被设计为半透明覆盖：StateStack 的绘制逻辑会先绘制 PlayState 的画面，再加一层半透明黑幕，然后绘制 PauseState。因此暂停时玩家仍能看到暗淡的游戏背景，这是通过状态栈自动实现的。

- **InventoryState（背包界面）**：在游戏中按下背包按钮后叠加的状态，显示玩家背包中不同鱼类的数量。它在 onCreate()中加载字体、设置界面标题“背包”，创建一个关闭按钮（点 X 退出背包）。onActivate()时，先播放游戏背景音乐（因为切换可能从菜单过来，确保音乐正确）；然后设置背景和标题位置；接着统计玩家背包中每种鱼的数量，将对应 Sprite 和数字文本放入界面指定位置。布局上采用固定坐标摆放 5 种鱼图标，并在图标右下角绘制数量文本。handleEvent()仅需要处理关闭按钮点击（或者按 Esc 键关闭）。由于背包界面也是覆盖层，StateStack 绘制它时会先绘制下层 PlayState 并叠加**半透明效果**，让游戏画面变暗，再绘制 InventoryState 内容。InventoryState 没有持续更新逻辑，因此 update()实现为空。通过这种状态隔离，背包 UI 可以独立于 PlayState 实现，不用在 PlayState 代码里掺杂背包显示相关的布局与绘制逻辑。
- **HelpState（帮助界面）**：从菜单按“帮助”进入，显示游戏说明的图片轮播。HelpState 在 onCreate()中加载字体和帮助图片 ID 列表（共 9 张说明图预加载于资源管理中），创建“上一张”、“下一张”、“返回菜单”三个按钮。按钮回调通过更改当前图片索引刷新显示或返回主菜单（pop 状态）。onActivate()播放菜单背景音乐，并设定按钮悬停和点击音效。HelpState 的 handleEvent()与菜单类似，把事件交给按钮处理，并支持 Esc 快捷键返回。update()不需实现（无动画），draw()则绘制当前图片、页码计数文本和相应的导航按钮。HelpState 作为覆盖状态，下面的菜单也会被绘制并叠加半透明效果，效果同暂停和背包页面。
- **LoadState（读档界面）**：从菜单进入，用于选择槽位加载存档。在 onCreate()里创建返回菜单按钮和 4 个槽位按钮，每个槽位按钮本身透明、仅在悬停时高亮，以便在其位置显示背景图和状态文本。槽位按钮点击的回调调用 loadFromSlot(slot)，若文件存在则读取存档并调度在下一帧切换到 PlayState 继续游戏。LoadState 的 onActivate()布局背景和标题，并调用 updateSlotStatus()更新每个槽位文本（显示是否有存档及鱼总数）。它的 handleEvent()需要处理返回按钮和各槽按钮的鼠标事件，以及 Esc 按下返回菜单的快捷键。LoadState 绘制时，完全覆盖下层菜单，这是一个特殊设计：LoadState 是被 push 进状态栈顶而非 ChangeState，如果不对底层加全黑幕，菜单页面可能会影响 LoadState 的显示效果。于是，便采用了“StateStack 检测到 LoadState 作为 overlay，使用全黑幕，使下层菜单不可见”的方法。这样读档界面就像独立场景，不会透出菜单。
- **SaveState（存档界面）**：在游戏中按存档按钮进入，与 LoadState 类似但功能相反。

onCreate()中创建返回按钮和 4 个槽位按钮，槽位按钮在悬停时半透明高亮。每个槽位按钮点击时调用 saveToSlot(slot) 保存当前进度，并立即更新槽位状态文本。onActivate()与 LoadState 类似，布置背景和标题位置，调用 updateSlotStatus()读取已有存档信息显示。handleEvent()处理返回和槽位按钮事件，以及 Esc 快捷返回游戏。绘制上，SaveState 覆盖在游戏画面之上，使用半透明黑幕，所以游戏背景会变暗显示在存档界面下层。保存操作完成后，玩家按返回或 Esc 退出该状态回到游戏。

状态交互与切换：通过 StateStack，Fishlime 实现了灵活的状态控制：

Push 叠加：适用于临时覆盖当前游戏的界面，如暂停、背包、帮助、存档读档。这些状态以 pushState 压入，不会销毁下层状态。例如游戏中打开背包，PlayState 仍在下层暂停运行，InventoryState 完成后 popState 回到 PlayState 继续。叠加状态通过 StateStack 的绘制机制看到下层暗景，从而提供上下文联系。

Change 替换：适用于完全切换场景，如从菜单进入游戏或从游戏退回菜单。这通过 changeState<NewState>实现，内部是 clearStates()+pushState 即清空所有旧状态再入栈新状态。这样原来的状态被销毁，不保留历史。例如主菜单点“开始游戏”，调用 changeState<PlayState>(player)，菜单状态立即清除，新状态 PlayState 创建并激活，游戏场景加载。再如在 PauseState 选择“退出到菜单”，也使用 changeState<MenuState>()回到主菜单。Change 保证了不同场景间资源和数据隔离，不会出现回到菜单还留有游戏状态的数据。

Pop 恢复：用于简单返回上一个状态的场景。最典型的是 PauseState 和 HelpState，这些状态内部都在适当时刻调用 Game::popState()。StateStack 会弹出当前状态，销毁它并自动调用下层状态的 onActivate()恢复运行。例如 Esc 关闭 PauseState 后，PlayState 的 onActivate 将被调用。这样下层状态不需要关心自己何时被暂停或恢复，StateStack 自动管理了对应钩子。

Deferred 切换：正如前述，Fishlime 大量使用 scheduleAction 来延迟执行状态 push/pop 避免冲突。在上例中，PauseState 按 Esc 处理实际上是通过 m_game.scheduleAction([this]{ m_game.popState(); })提交，在帧末才真正 pop。MenuState 按钮等也类似。这保证在当前 handleEvent 函数结束后再改变状态，避免了例如正在遍历 UI 元素时栈内容被改变的不安全情况。

状态机优势：这种**栈式状态机**结构，使**游戏模块井井有条**，每个状态都是**相对独立**的**逻**

辑单元，彼此之间通过 Game 和 StateStack 通信（例如改变状态、访问共享资源），但 UI 元素、更新逻辑彼此隔离。增加新状态很方便，只需继承 State 实现接口，再在合适时机 push 即可，不影响现有代码。同时，状态类的划分让开发调试变简单：可以针对单一状态调试其行为，而不影响其他部分。例如调试帮助界面只需从菜单进入帮助状态测试，不会牵涉游戏逻辑。

通过状态栈，Fishlime 实现了**游戏流程控制**的清晰架构：菜单->游戏->暂停/背包->恢复->... 等流程都自然地对应栈操作。状态栈使游戏能够轻松管理复杂的界面和模式切换，为项目的整体框架奠定了稳固基础。

第二章　实践过程总结与分析

2.1 SFML 安装和配置

(一) 下载 SFML 3.0 SDK

访问 SFML 官网下载页（`https://www.sfml-dev.org/fr/download/sfml/3.0.0/`），选择 “Visual C++ 17 (2022) - 64 bit” 版本并下载安装包。

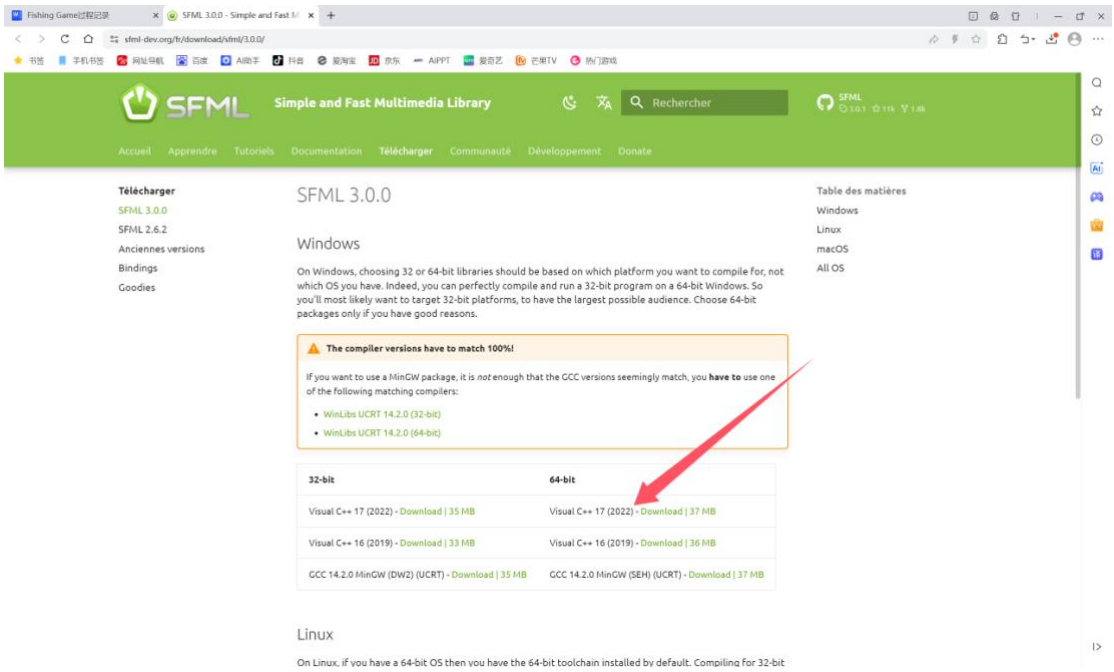

图 4：SFML 3.0 下载

(二) 解压下载文件，并放到特定目录中待用

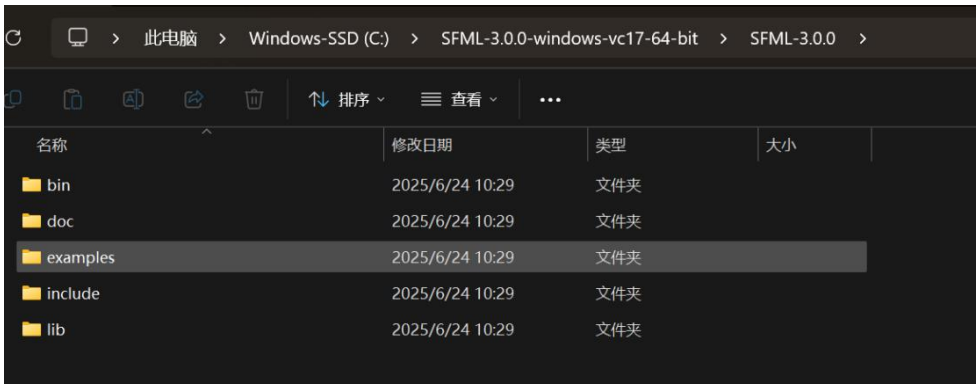

图 5：SFML 解压效果

（三） 新建 VS2022 项目

在 “解决方案资源管理器” 中右键项目 → “属性”，确保“平台”是 x64（如果不是 x64，如可能是 ARM64，需要在配置管理器中修改平台为 x64，以防止有不可预料的兼容性问题），配置选 Debug，然后选择平台为“所有平台”，使得后面统一配置到 “所有配置”。

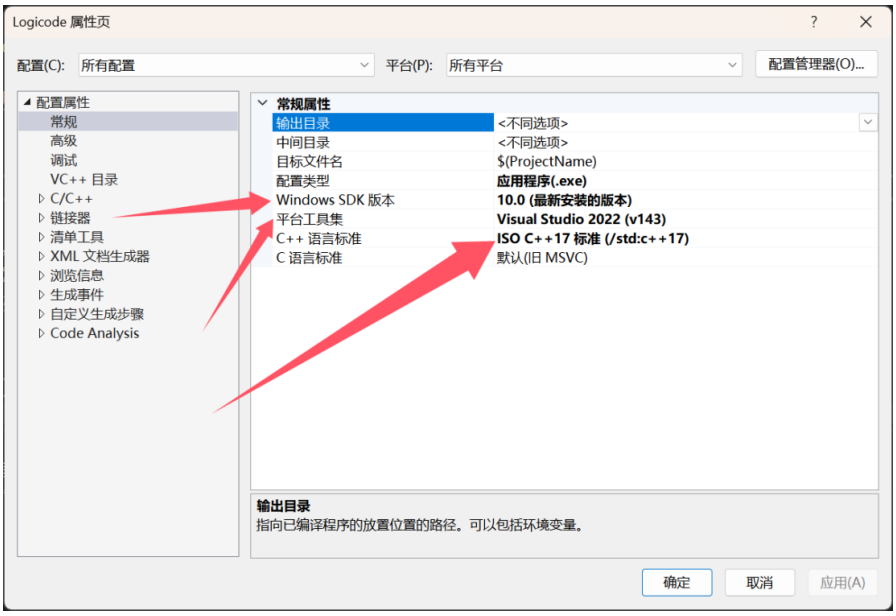

图 6：解决方案资源管理器 - 属性

注意，请确保 C++语言标准为 /std:c++17 标准，否则会在运行项目时出现报错！（因为 SFML 3.0 的头文件里用到了 C++17 的特性，如果使用 C++14 及以前的标准，就无法正常运行，出现如下所示的报错。）

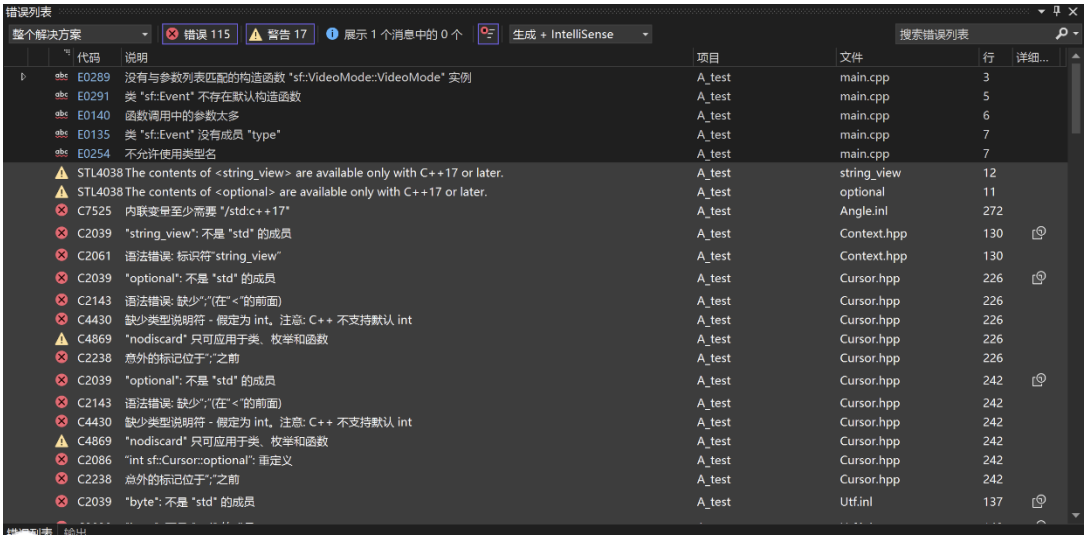

图 7：C++标准问题报错

（四） 配置包含目录

在项目属性 → VC++目录 → 包含目录中，添加刚刚解压得到的 include 文件夹，格式为：xxx\SFML-3.0.0\include。

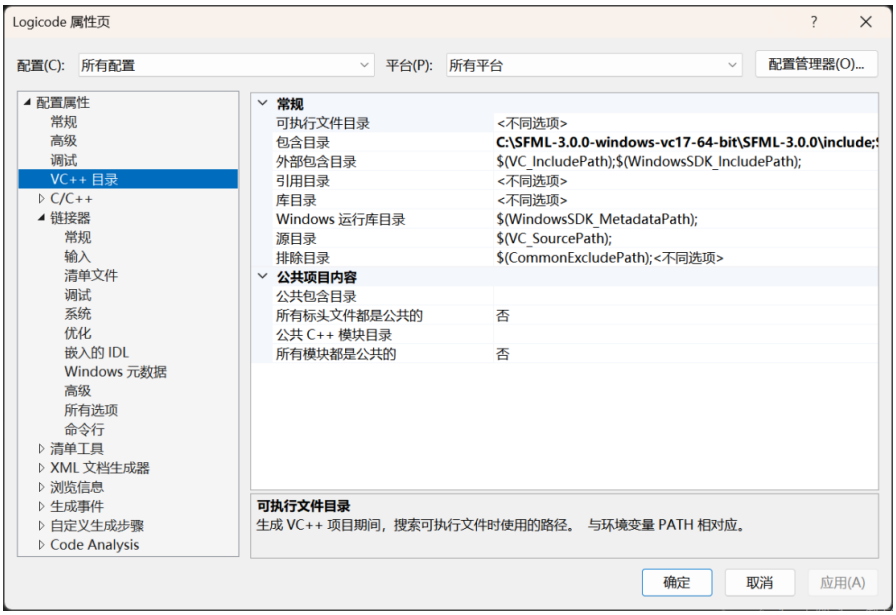

图 8：VC++目录 - 包含目录

（五） 配置库目录

在项目属性 → 链接器 → 常规 → 附加依赖项中，添加刚刚解压得到的 lib 文件夹，格式为：xxx\SFML-3.0.0\lib。

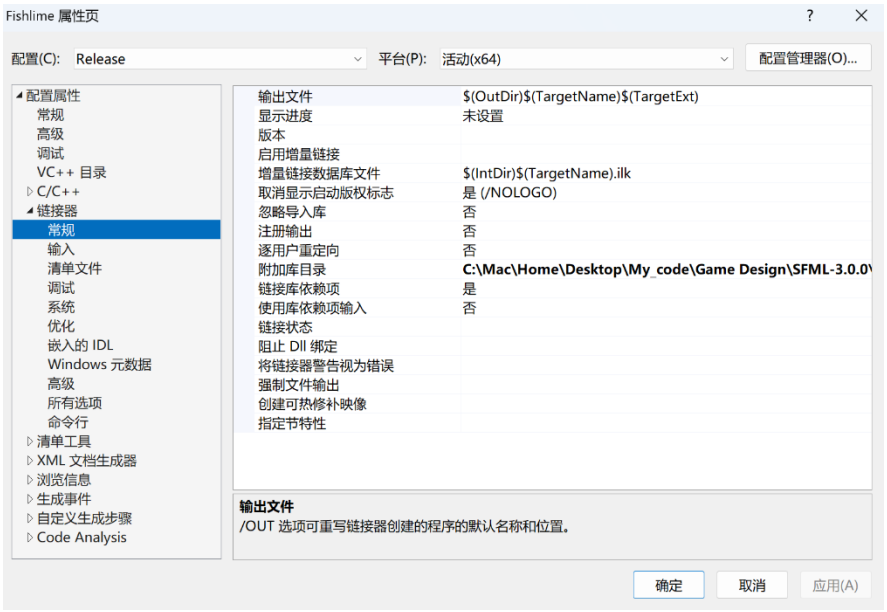

图 9：链接器 – 常规

(六) 添加依赖库

在项目属性 → 链接器 → 输入 → 附加依赖项中，添加：

1	sfml-system-d.lib
2	sfml-window-d.lib
3	sfml-graphics-d.lib
4	sfml-audio-d.lib
5	sfml-network-d.lib

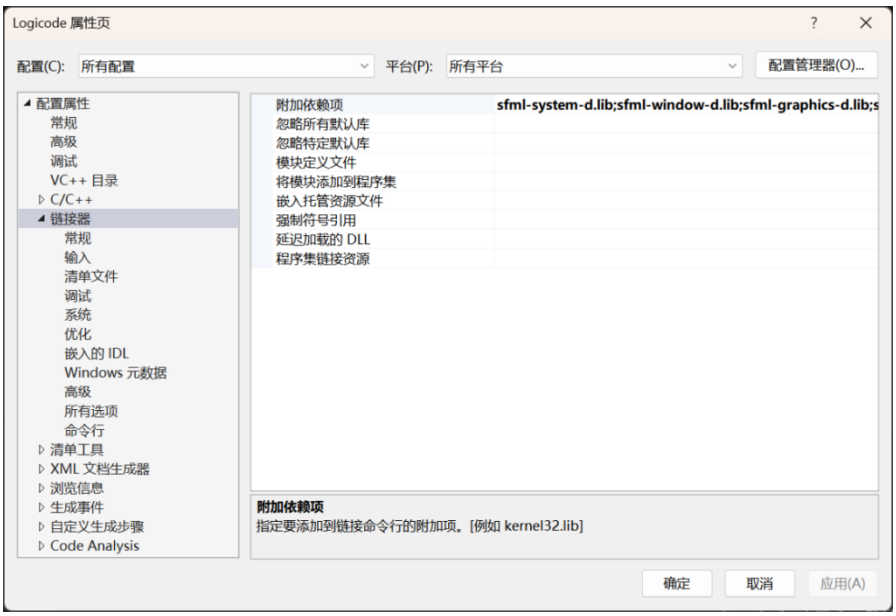

图 10：链接器 – 输入

(七) 复制 DLL 到输出目录

复制解压后得到的 SFML-3.0.0\bin 中的所有 `sfml-*-d-3.dll` 文件，到程序根目录（非解决方案目录）下的 x64\Debug 文件夹中（如果没有可以先自行建立文件夹，因为这个文件夹是在编译后才会生成）。

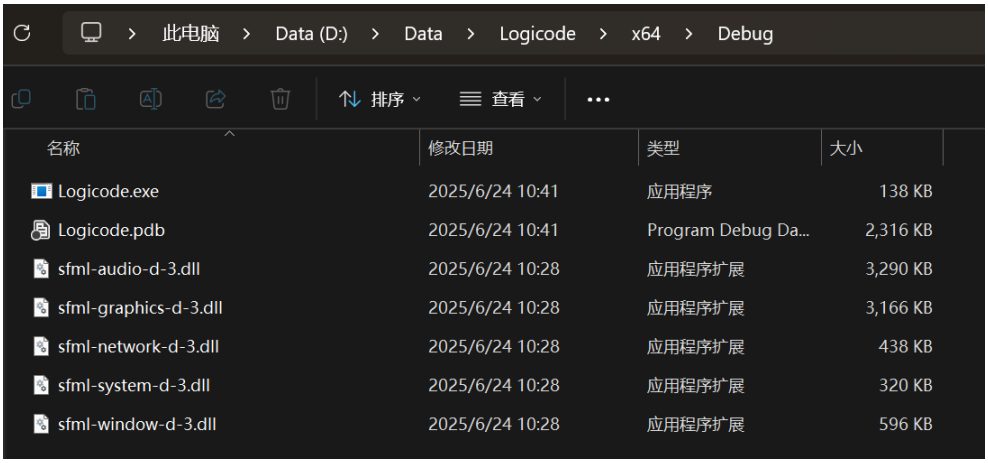

图 11：Debug 文件夹

如此以上，SFML3.0 配置完毕（为方便后续其他新项目的创建，可以将该项目导出模板，这样后续创建新 SFML3.0 项目时，不用再手动配置）。

2.2 开发与调试过程总结

在开发 Fishlime 的过程中，我们（团队“逻辑写不队”）经历了从搭建框架、功能实现到不断调试完善的迭代过程。下面总结重要困难、解决方法，以及项目分工和各系统的设计调优经验。

（一） 重要困难及解决

窗口标题、UI 中文乱码：如果要在窗口标题以及内部显示中文，由于源代码文件的编码格式与编译器解析字符串字面量的方式不匹配，需要把 UTF-8 编码的字节串准确地转换成 SFML 内部的 UTF-32，否则会显示乱码。有两种解决办法：其一，是先准备一个 UTF-8 编码的 std::string，然后用 fromUtf8 将字节流转成 sf::String（UTF-32），再使用转换后的字节串，即可正常显示中文，部分代码如下(从创建窗口开始)：

```
1  // 1. 创建窗口
2      // 1. 构造 VideoMode
3  sf::VideoMode vm({ 800, 600 });
4      // 2. 准备好一个 UTF-8 编码的 std::string
5  std::string utf8 = u8"SFML 3.0 示例";
6      // 3. 用 fromUtf8 将字节流转成 sf::String (UTF-32)
```

```
7   sf::String title = sf::String::fromUtf8(utf8.begin(), utf8.end());
8       // 4. 用转换后的 title 构造窗口
9   sf::RenderWindow window(vm, title);
```

其二是使用宽字符串字面量（L"..."），这个方法利用了 Windows 对宽字符（wchar_t）的原生支持，只需要将代码中的所有中文字符串修改为使用 L 前缀即可，示例如下：

```
1       sf::RenderWindow window{vm, (L"SFML 3.0 示例") };
```

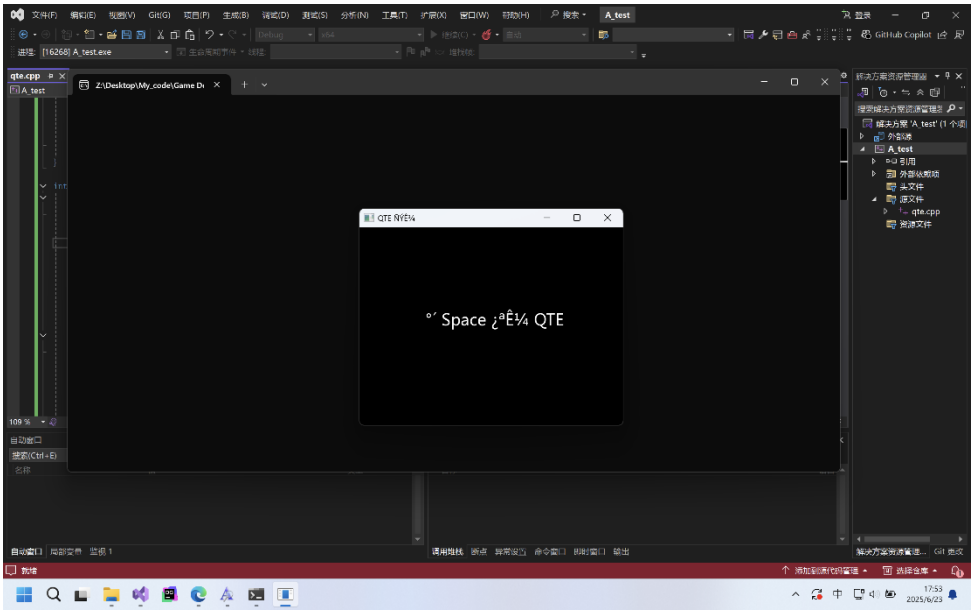

图 12：输出中文乱码

图 13：正常输出中文

架构搭建：初期需要决定整体架构，我们选择了 State 状态机+OOP 的模式。为验证架构合理性，先实现了一个最小原型：角色可走动的沙滩场景和简易菜单。早期曾尝试用单一

循环处理所有逻辑，但随着需求增加发现混杂不便，最终确立了当前的状态划分方案。这解决了界面和游戏逻辑交织的问题，通过模块化状态，开发后期新增 Pause、Help 等非常顺利。

事件与逻辑解耦：调试中曾遇到过输入处理顺序的问题。例如最初直接在事件循环中进行状态切换，导致当前状态被删除，后续代码访问无效对象。为解决此类错误，我们引入了 scheduleAction 延迟执行机制，确保敏感操作在帧末统一处理。这一改动非常关键，经过测试验证了状态切换的稳定性，再未发生类似崩溃。这算是将异步事件安全应用于同步循环的一个经验。

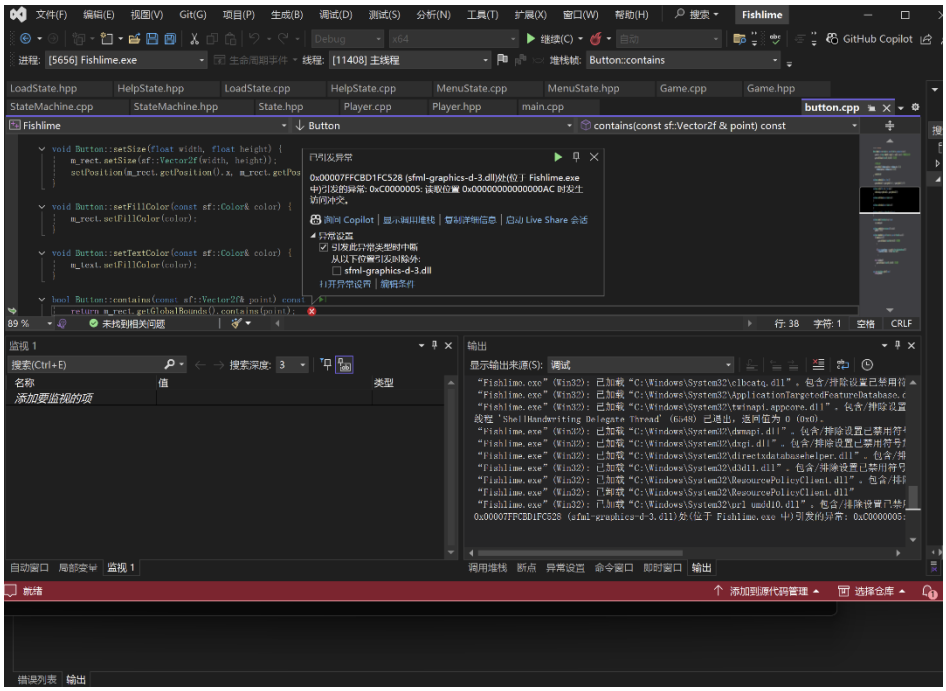

图 14：状态删除后，后续代码访问无效对象导致内存错误

未知异常：代码更新后，直接点击开始调试可能会报未知异常，此时尝试一下项目 - 清理可能可以解决问题。

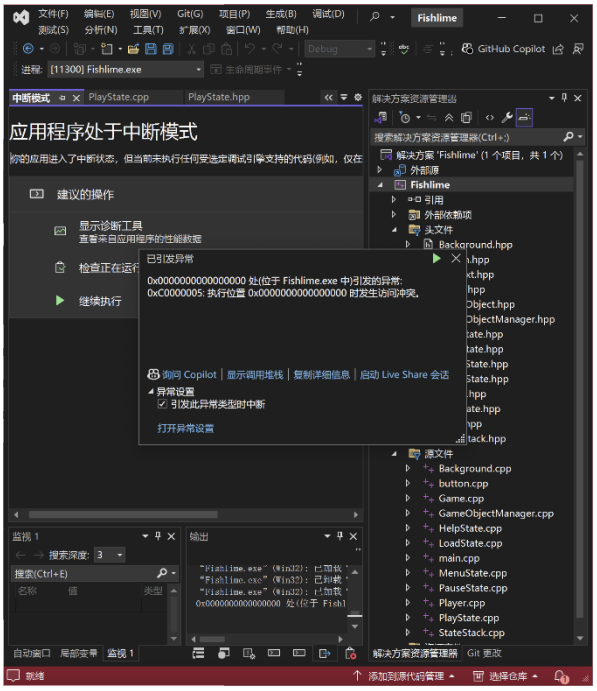

图 15：代码报错：已引发异常

动画与帧同步：实现角色动画时，需要调整帧率与移动速度匹配，避免动画滑步。开始时行走动画帧切换过慢或过快，通过调整 Animation.frameDuration（采用 0.1 秒一帧）并在 Player 更新逻辑中根据移动状态才切换动画。这些细节调优花了一些时间，通过不断运行观察调整参数，最终达到了角色移动和动画节奏协调一致，视觉效果较好。

QTE 手感调试：钓鱼 QTE 玩法需要平衡难度与反馈。我们设定随机 2~3 秒触发一次按键提示，并要求连续按对 3 次成功。初版 QTE 可能过于简单或难，我们通过测试调整了按键提示频率(随机 2-3 秒)和超时时间(3 秒内未按则失败)。还增加了音效反馈:提示音(QTEStart)让玩家警觉，按对键播放不同成功音效（第 2 次成功额外音效），失败或成功结束也各有音效。这些音效资源的加入极大增强了 QTE 的临场感。调试 QTE 时也曾发现一个问题：玩家在钓鱼状态时仍可移动,我们通过在 Fishing 状态下优先让 Player 把事件交给 QTEManager 处理并中止其它输入来解决，保证钓鱼时角色不乱动。

Player 松开键瞬移到 (0,0)：出现问题后，按以下思路调试解决：重新梳理 Idle ↔ Moving 状态机：

1) 按键检测决定 isMoving

2) 切 Moving 时立即 playAnimation("Move")

3) 松键先设 mPendingIdleSwitch，等动画循环到帧 0 再切 Idle —— 位置不再重置。

史莱姆奔出场景：为防止玩家走出沙滩范围，我们设计了不规则多边形边界。用射线法实现的 MovementBoundary::contains 算法开始调试时有误判，曾一度出现玩家贴边抖动或卡在边界上的 bug。通过在各顶点之间多次测试，修改了判断条件（确保边界条件处理正确），最终 contains() 能精确判断点是否在多边形内。同时在 Player 移动代码中，对新位置进行 contains 检查后再赋值，彻底杜绝了角色出界的问题。但是目前由于多边形的顶点固定为一个坐标集合，而该坐标集合是基于 1920*1080 窗口大小确定，所有游戏目前缺乏对窗口大小的适配性，在游戏中无法手动拉伸窗口并让其自动适应，否则会出现各种问题，如窗口显示异常，Player 移动界限异常等等。

生成独立 .exe 报错：在 VS2022 中使用 Release 生成解决方案时，需要在项目属性页中单独配置 Release - 链接器 - 输入 - 附加依赖项 为无-d 后缀的.lib 文件，需要与 Debug 配置区分开。并且，在项目根目录 - x64 - Release 文件夹中，需要复制所有-3 后缀的.dll 文件（注意，非-d-3 后缀）。此后，才可正常使用 Release 生成解决方案并得到可执行的.exe 文件。若要运行.exe 可执行文件，也需要把项目解决方案目录中的资源复制到.exe 同目录下。

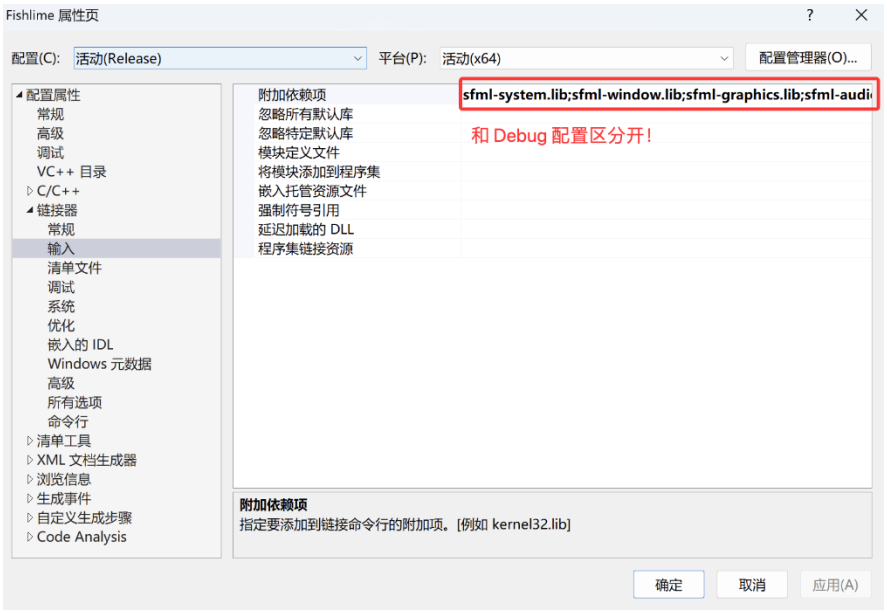

图 16：Release 配置须与 Debug 配置区分开

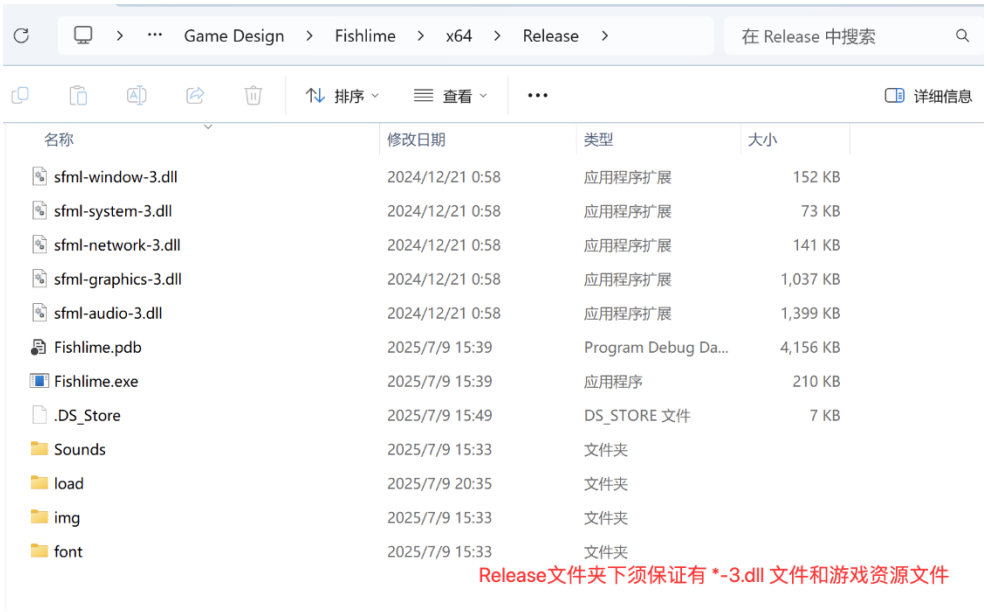

图 17：.exe 文件可执行的基础

（二） 资源管理和性能

我们使用了统一的 ResourceManager 来加载纹理和字体，使得相同资源**只加载一次**并**在各处共享**。调试中我们发现，如果每次切换状态**重新加载**资源会明显**卡顿**，于是决定**在 Game 构造时预加载大部分资源**。事实证明这是正确的——游戏需要用到的所有贴图（除动画贴图）我们在进入 Game（第一个页面为 MenuState，所以体现在第一次进入游戏主页时）时就通过 Game 预载入，进入时直接取用，使得界面切换非常流畅，没有卡顿等待。所有动画贴图在第一次进入 PlayState 时统一加载，并且保存在 ResourceManager 中，后续再进入游戏页面就不需要等待。音频方面，AudioManager 在启动时也加载了所有音效和音乐文件。虽然占用一些内存，但换来的是切换页面时切换 BGM 和播放音效时毫无延迟。我们实现了运行时音量调节（1-9 键），这给了玩家更多个性化选择。当觉得某音效过大，可以按键调整，以适应不同玩家对不同音量的需求。

（三） 团队分工

本项目在小组协作下完成。我们大致按类型分配任务：一名成员负责主框架和状态机（搭建 StateStack、Game 类、主循环等），并且整合所有游戏逻辑，确保游戏的整体流程控制；一名成员专注游戏的各种逻辑功能的实现，如存档/读档的具体实现、音频效果的实现等；另一名成员侧重 UI 界面和资源，各种美术素材的绘制，音效的收集，以及部分逻辑功能的

实现，如页面的切换逻辑等。开发过程中大家交叉测试对方的模块，提出改进意见；最后，由一人进行所有代码的整合。这种明确分工又密切配合的方式，使开发效率很高。具体见下：

成员	分工内容
马成龙	负责主框架和状态机（搭建 StateStack、Game 类、主循环等），并且整合所有游戏逻辑，确保游戏的整体流程控制
吴雨凡	专注游戏的各种逻辑功能的实现，如存档/读档的具体实现、音频效果的实现等
杨弈鸣	UI 界面和资源，各种美术素材的绘制，音效的收集，以及部分逻辑功能的实现，如页面的切换逻辑等

（四） 调优过程

随着主体功能完成，我们进行了多轮细节打磨：

在**动画系统**上，调整了角色各动画的起始结束衔接。特别是抛竿(Start)动画和等待(Fishing)动画的衔接，我们通过在 Player::update 里检测动画帧循环结束切换状态来实现无缝过渡。曾遇到过动画循环判断不严导致多循环的问题，后来加了标志位 mStartDone/mFinishDone 确保只切换一次状态。这些标志逻辑经过调试，现在能稳定地在动画播完一圈时切换钓鱼/收杆状态。

UI 细节：对齐和美观上也花了一些工夫调整。比如菜单和各界面大量使用了手动计算坐标：MenuState 中按钮整齐纵向排列，我们计算了起始 Y 和间距来放置；帮助界面图片根据窗口 0.7 倍大小缩放，并垂直居中偏上 50 像素；背包中各鱼类图片和鱼类数量的显示位置，以及存读档界面按钮的排布、文字的排布等等。调试这些坐标需要不断运行查看效果，再调整参数。虽然过程繁琐，但最终界面元素布局井然，使得 UI 在 1920*1080 下效果很好。

回顾开发历程，我们深刻体会到逐步构建、持续调试的重要性。每新增一个模块，就尽早测试基本功能是否正确，再叠加复杂交互。这使我们能及时发现问题并修正。例如存档系统在第一次测试时发现文件路径错误无法创建，通过调试信息迅速改正了目录处理。又如多人协作时通过版本管理，我们有效地整合了各自的代码，解决了资源引用等集成问题。 总的来说，开发 Fishlime 既有遇到难题时的困惑，也有攻克 bug 时的欣喜。在不断调优过程中，游戏更加完善，团队成员的开发能力和协作默契也得到了提高，收获的成就感也无可比拟。

第三章　总结与展望

经过对 Fishlime 项目的设计与实现回顾，我们可以看到合理的架构使开发事半功倍，但仍有很多改进和拓展的空间。在总结本次项目得失的同时，也展望未来可以进一步优化的方向：

3.1 设计得失反思

本项目的 OOP 架构清晰，实现了**高内聚、低耦合**。状态机模式很好地组织了菜单、游戏、暂停等流程，各模块分工明确。得益于此，我们在开发过程中较少出现因模块交叉导致的 bug，大部分错误都能在模块内解决。比如输入处理按模块划分，几乎没有冲突。再如存档/读档设计为独立状态，避免了在 PlayState 混入文件操作逻辑，使主游戏代码保持简洁。这体现了架构设计的成功。

在类设计上，我们基本遵循了 **SOLID** 原则（具体介绍可见 stage2 的文档）。例如单一职责：Animator 只管动画，AudioManager 只管声音，Player 负责协调而不亲自处理输入事件细节等。接口隔离：GameObject 只定义 update/draw 接口，没有强迫所有子类实现不需要的方法。这些使得修改一个部分通常不影响其他部分。例如后来增加背包界面时，只需新增 InventoryState 并 push，不用改 PlayState 逻辑；又如调整键位只涉及 InputHandler 部分（我们将事件处理集中管理，没有在其它类散乱检测键盘）。当然，也有一些不足值得改进：

- **类耦合性方面**，Game 对各子系统仍有较强依赖，它直接持有 Player、ResourceManager 等。虽然这在小项目中问题不大，但严格来说，可以通过接口或依赖注入进一步解耦，让 Game 更抽象。
- **全局单例的使用**：AudioManager 作为单例使用方便，但违反了依赖倒置原则（高层模块依赖低层实现）。理想情况下，我们可以注入一个 IAudioPlayer 接口给需要播放声音的类，而不是直接调用 AudioManager 静态方法。
- **魔数与布局**：目前界面布局用了大量硬编码数值，如像素坐标、缩放比例。这些参数缺乏自适应性，如果更换分辨率或资源，可能界面错位。今后可引入 UI 布局系统或按相对百分比定位，亦或提供配置文件调整这些参数，使 UI 调整更灵活。
- **输入控制**：我们使用 SFML pollEvent，这在 PC 单机上足够。但如果游戏需要更复杂的输入管理（如可配置按键等），当前方案就显得简陋。后续可考虑引入 InputMapper，将硬件事件映射为游戏动作，更易扩展。

- **性能调优**：本游戏逻辑简单，帧率始终稳定 60 以上。但我们未对异常情况做太多处理，例如资源加载失败只简单输出错误，不做备用方案；游戏循环没有 delta 时间上限控制，极端情况下（如窗口拖拽导致 dt 很大）可能出现运动跳跃。这些尚未造成明显问题，但值得留意。

3.2 最大挑战回顾

团队一致认为，实现钓鱼 QTE 是开发中最具挑战的部分。因为它涉及多状态切换、随机、计时、人机交互等，多方面逻辑交织。开始设计 QTE 时，我们在如何通知玩家按键、如何判断成功上考虑了多种方案，也参考了其它游戏的 QTE 实现。最终通过 QTEManager 类，结合动画状态机和音效，我们构建了一个相对完善的钓鱼小游戏。调试 QTE 时遇到的同步问题（提示与动画同步、输入与移动冲突）让我们印象深刻，好在通过状态机分离和优先级控制解决了。这部分开发锻炼了我们处理异步事件序列的能力，也是代码中较复杂的一段逻辑。

3.3 可行的后续优化

若有更多时间和资源，Fishlime 可以向以下方向加强和扩展：

- **丰富游戏内容**（增加地图和场景）：目前只有一个沙滩钓鱼点，将来可以扩展多个地图区域，如河流、湖泊等，每个场景有不同鱼类、风景。可以在地图上实现角色自由走动切换场景，甚至加入小镇购买装备等内容。这需要引入地图管理，甚至基本的地图编辑和碰撞检测。我们已有 MovementBoundary，可以扩展用于地图碰撞。不同地图的背景、音乐等可通过状态机方便切换。
- 引**入简单 AI**：例如在场景中加入其他钓鱼者（NPC），他们可以随机抛竿收杆，让场景更生动。初步分析该功能的实现思路，这需要一个 AI 控制的 Character 类，让其继承 GameObject，由 GameObjectManager 管理。AI 逻辑或许很简单（随机间隔开始 QTE，自动成功或失败），但可以丰富氛围。甚至可以设定比赛模式，看谁钓鱼多。此外，鱼本身也可拟作 AI 实体，如水中影子随机游动，在玩家抛竿时靠近。这将使钓鱼过程更有真实感。实现 AI 需要一定系统改动，实现逻辑较复杂，但实现后效果会很好。我们现存的代码框架允许添加此类逻辑，不会影响现有的玩家逻辑。
- **改良 QTE 玩法**：目前 QTE 只有按键反应一种形式，时间固定三次。未来可增加其他形式，如连续按键（快速连击某键）、节奏圈（在正确时机按下）等。初步分析该功能的

实现思路，这可通过策略模式扩展 QTEManager：定义 IQteStrategy 接口，不同 QTE 玩法不同实现。QTEManager 持有接口指针，根据需要切换具体策略，实现对扩展开放。这样可以在后续版本中加入新的钓鱼小游戏模式而不改动现有代码。此外，可以根据鱼的稀有度调整 QTE 难度：比如稀有鱼需要 4 次成功或者更短反应时间。QTE 结束后的反馈也可更丰富，当前只是打印一行消息，将来可播放特殊动画或弹出 UI 展示鱼的模型。

- **完善 UI 和用户体验**：UI 方面还有很多可提升之处：增加菜单动画效果，按钮悬停时文字/图标变化更明显；加入设置菜单，允许配置音量、键位等；帮助界面可更加完善美观；游戏中可有 HUD 界面，比如显示玩家体力值、饥饿度等。特别是当内容扩充后，良好的 UI 能显著改善玩家体验。
- **性能和平台**：如果将来鱼的数量、NPC 增多，需关注性能。SFML 对这种规模的实现支持良好。可能更实际的是兼容不同分辨率或者发布 Web/移动版。SFML 主要针对 PC，如需发布 Web 游戏，需要确保代码中没有依赖桌面特性的部分（可能音频需要改）。我们现有代码大多可移植，但文件存档在 Web 端不可用，届时存档可改用数据库等替代。
- **多人游戏和联网**：这属于非常远期的想法。如果把 Fishlime 拓展为一款多人竞赛钓鱼游戏，可以考虑通过网络让玩家实时比赛钓鱼。但这涉及同步和服务器，实现复杂性激增。

总结：Fishlime 中，我们成功实现了从菜单到钓鱼的一整套流程，代码结构清晰，运行稳定。在此基础上做扩展是顺理成章的事情。最大挑战在于始终保持代码质量和架构稳定——随着功能增加，必须谨慎避免“修修补补”的临时方案，继续遵循 OOP 原则和良好实践，这样才能让项目持续演进而不过度增加维护成本。

本次项目让我们深刻体会到游戏开发的魅力和难点。从零开始设计架构并最终实现出可玩的游戏，是一段宝贵的经验。我们在过程中学会了合理划分问题、运用合适的设计模式，也品尝到调试优化后游戏逐渐完善的成就感。当然，还有许多可以改进的细节未能在有限时间内实现，但留待将来也是一种动力。总而言之，Fishlime 项目达到了预期目标 —— 提供了一个轻松有趣的钓鱼小游戏，同时在技术上锻炼了我们的编程能力。展望未来，我们希望能以此为起点，继续优化 Fishlime，或者应用所学再挑战更复杂的游戏开发项目。游戏的世界广阔无垠，而我们才刚刚扬帆起航！

附录：类功能详细剖析

A. Game 类（游戏主控）

职责：

Game 是游戏应用的入口和主控制类。它负责初始化窗口和全局资源、管理主循环，以及提供全局访问接口（如资源、状态和玩家）。Game 协调了游戏运行期间的大部分全局行为。

主要属性：

- m_window：主渲染窗口 (sf::RenderWindow)，用于显示游戏画面。Game 在构造时创建窗口，设置了分辨率 1920x1080 和标题“Fishlime”。还设定帧率限制 60，以稳定游戏帧率。
- 资源管理器 m_textureManager 和 m_fontManager：ResourceManager<sf::Texture> 与 ResourceManager<sf::Font> 模板实例，用于加载和缓存纹理、字体资源。Game 在构造时预加载了一些关键资源（背景、UI 图片、字体等）并提供 textures()和 fonts()方法供其他对象获取资源。
- m_states：状态栈 (StateStack)，管理游戏状态的栈结构。Game 通过它来实现状态切换/叠加。Game 构造时将自身和资源封装进 Context，并用于初始化 StateStack。
- m_player：全局玩家对象 (Player)，代表玩家控制的史莱姆角色。该对象在 Game 初始化时创建一次并贯穿游戏始终，可在各状态间保持连续（如背包内容不会因切换状态而丢失）。Game 提供 getPlayer() 访问它。
- 其他：例如 m_context（Context 结构，包含指向 window 和资源管理的指针，用于传递给 StateStack）；m_scheduledAction（延迟执行函数对象）；m_transientText（临时文本用于显示提示消息），m_showMessage 标志及计时 Clock 等。

主要方法：

- run(): 游戏主循环方法，前节已详述。它在退出循环后（窗口关闭）方法结束，Game 生命周期也结束。
- 状态管理接口：changeState<>(), pushState<>(), popState()。这些封装调用了 StateStack 的对应功能。changeState 模板接受可变参数转发给新状态构造，比如 changeState<PlayState>(player)将当前状态清空并压入一个以 player 引用构造的 PlayState。pushState 则保留现有状态压入新状态（如 pushState<PauseState>()）。

popState 无参数，从栈弹出当前状态。Game 通过这些接口使其他类（如状态）能够方便地请求切换状态，而不需要直接接触 StateStack 内部细节。

- 资源访问：textures()和 fonts()返回对资源管理器的引用，让各状态通过 Game 获取共享的纹理字体资源。例如 PlayState 会用 game.textures().get("background")来取得背景图纹理。
- 玩家访问:getPlayer()返回对 Game 持有的 Player 引用。这样状态对象可以操控玩家。例如 MenuState 开始游戏前，会通过 auto& player = m_game.getPlayer();拿到玩家对象进行背包清空、位置重置。
- 临时消息：showTransientMessage(sf::String, sf::Color)用于在屏幕上部中央显示一条 3 秒后自动消失的提示文本。实现是设置 m_transientText 内容和颜色，启动 m_messageClock 计时，并打开 m_showMessage 标志。主循环渲染阶段会检查该标志，在 3 秒内绘制文本。这个功能用于各种需要提示玩家的信息，如改变音量时在屏幕上显示百分比或钓鱼成功获得某条鱼。
- scheduleAction(std::function<void()>): 前述延迟任务接口。将给定函数存入 m_scheduledAction，在主循环帧末执行并清空。Game 提供它供状态或其他地方调用，以安全地延迟执行敏感操作。实际上，大部分状态切换都是通过 Game::scheduleAction 包裹 Game::pushState/changeState 调用来完成的。

Game 在系统中的作用：可以说 Game 是“导演”，安排着舞台上各元素的登场退场。它本身不处理具体游戏逻辑细节，但通过组合各管理器和状态机，提供了一个稳定的运行环境。Game 确保窗口和资源的生命周期长于一切状态和对象，使大家可以放心引用；Game 又在合适时机调用各模块的接口，使整个游戏按序运行。Game 与 StateStack/Player 等紧密协作：StateStack 需要 Game 引用（比如创建状态时传入 Game&以供状态用），Player 需要 Game 驱动更新渲染。Game 也持有 AudioManager 单例（虽然 AudioManager 是全局，但 Game 初始化时负责加载音频）。Game 相当于全局调度中心，但并未膨胀成上帝类，而是巧妙地将具体功能分散出去，仅保留必要的统筹方法。这体现了良好的单一职责和控制反转原则，让 Game 灵活而不臃肿。

B. Player 类（玩家角色控制）

职责：

Player 类封装了主角史莱姆角色的一切行为和属性，包括移动、动画状态、钓鱼流程、背包管理等。它既是游戏逻辑的主体，又是渲染表现的主要对象。Player 通过组合各种组件，实现了复杂的行为协作。

继承关系：

Player 继承自 GameObject，因此实现了 update() 和 draw() 方法，可由 GameObjectManager 统一管理。在 PlayState 中，尽管 Player 未被加入 GameObjectManager（因为 PlayState 直接保存了 Player 引用并手动调用其 update/draw），但作为 GameObject 接口的一部分，它符合统一结构。

主要属性：

- 位置与移动：mPosition 表示玩家逻辑坐标位置（Vector2f），初始为沙滩左下起点 (1440, 780)。mMoveSpeed 是移动速度（像素/秒），构造时指定 200.0f。还有一个内部枚举 State {Idle, Moving, StartFishing, Fishing, FinishFishing} 表示玩家当前动作状态。mState 初始 Idle 静止。另外 mPendingIdleSwitch、mStartDone、mFinishDone 等 bool 用于细化状态切换控制（如行走停止时等一帧动画循环再 Idle，抛竿/收杆动画完成标志）。
- 动画相关：组合了一个 Animator mAnimator。Animator 管理玩家所有动画序列。Player 在初始化时会利用 Game 的 ResourceManager 加载各动画帧纹理并通过 Animator.addAnimation 添加到 Animator 中，然后通过 Animator.play(name)切换动画。Animator 内部保存当前动画帧索引和精灵(Sprite)，Player 每帧调用 mAnimator.update(dt) 推进帧序列。Player 包含了一个动画帧资源表 m_fishInfos/m_fishIds 用于钓到鱼时选择哪种鱼。此外有一个标志 m_moveSoundPlayed 用于控制行走动画的脚步声播放（每循环播放一次）。
- 输入/QTE：组合了一个 QTEManager mQTEManager。QTEManager 管理钓鱼小游戏的状态（何时提示按键、成功计数等）。Player 会在钓鱼开始时调用 mQTEManager.start()重置 QTE 流程；每帧更新时，把 dt 传给 QTEManager 更新计时；在玩家事件处理时，如果正在 Fishing 状态，则先让 QTEManager 处理按键事件。这建立了一种协作关系：Player 管宏观状态，QTEManager 管细节，二者相互调用完成钓鱼逻辑。
- 背包系统：组合了一个 Inventory m_inventory。Inventory 内部是 std::vector<std::unique_ptr<Item>>。Player 提供 inventory()接口获取背包，以便

其他类操作道具。钓鱼成功时，Player 会创建 FishItem 加入 m_inventory。背包中的物品可用于恢复等，但目前 use()逻辑未具体实现（预留接口）。

- 移动边界：组合了一个 MovementBoundary mBoundary。PlayState 在创建时设置了边界多边形顶点给 Player。Player 的移动代码在计算新位置后，会用 mBoundary.contains(newPos)判断是否合法，只有在多边形内才更新位置。这样玩家不会走出地图。边界检查算法在 MovementBoundary 实现，通过射线交点判断点在多边形内。
- 回调：FishCallback onFishCaught 是一个函数回调，用于当成功钓到鱼时通知外部。PlayState 利用它在钓到鱼后弹出提示消息。Player 在添加鱼后会检测 if(onFishCaught) onFishCaught(fishId)。这种设计降低了 Player 和 UI 之间的耦合。

主要方法：

- loadAnimation(name, folder, frameCount, secondsPerFrame, textures, scale): 加载指定目录下的若干帧图片，创建 Animation 并添加进 Animator。返回 bool 表示是否成功。这封装了帧资源加载和组装，使得 PlayState 添加动画时调用几次 loadAnimation 即可完成初始化。
- playAnimation(name): 切换当前动画为指定名称。内部调用 Animator.play()并将 Animator 精灵位置设置为 Player 当前 mPosition。这一点很重要，因为 Animator 的 sprite 独立管理位置，而 Player 记录了逻辑位置，两者需要同步。每次切换动画都先对齐位置，保证不会因动画切换出现角色跳动。
- handleEvent(const sf::Event&): 处理玩家相关的输入事件。逻辑包括：在 Fishing 状态中，如果空格键按下则立刻中断钓鱼（改变状态为 FinishFishing 并播放收杆动画）——这是允许玩家放弃钓鱼的选项；然后，如果当前状态是 Fishing（钓鱼进行），则把事件传递给 QTEManager 处理，它会检查按键是否正确；除此之外，检测 Idle 状态且位于左边界时按下空格，触发开始钓鱼：状态切为 StartFishing，播放抛竿动画并播放音效。可以看到，handleEvent 仅处理跟钓鱼状态切换相关的输入，玩家的移动并不在此处理——移动用状态查询方式在 update 中完成（下一条）。
- update(sf::Time dt): 玩家每帧逻辑更新函数，是内容较多的一个方法。可以分为两个部分：钓鱼流程和移动流程。
  - 钓鱼流程管理： 根据 Player 的 mState 推进钓鱼状态机：

- 如果状态是 StartFishing（抛竿动画播放阶段），则持续 update 动画；当动画播完一圈（通过 Animator 帧索引回到 0 来判断）且尚未标记完成，就切换到 Fishing 状态开始正式钓鱼。此时播放第二段音效并调用 mQTEManager.start()启动 QTE。
- 如果状态是 Fishing（钓鱼中），则 update 等待动画和 QTEManager。检查 QTEManager 是否结束：如果结束且成功，则随机生成一种鱼（用随机数引擎 m_rng 和分布 m_fishDist 得到 0-4 序号），取出对应鱼的信息（weight、rarity、textureId），创建 FishItem 加入背包。打印获得鱼的信息并调用 onFishCaught 回调通知 UI。无论成功与否，一旦 QTE 结束，都将玩家状态设为 FinishFishing，并开始播放收杆动画。
- 如果状态是 FinishFishing（收杆动画阶段），则持续 update 动画；当动画播完一圈（同样检测帧归零）且之前未标记完成时，切换回 Idle 状态，重新播放 Idle 待机动画。这样一次完整的钓鱼循环结束。
- 整个流程严谨地通过有限状态机实现，保证状态转换在正确时间点发生，不会错乱。例如通过 mStartDone 等标志避免重复切换状态。

- 移动与动画更新： 如果玩家当前不在钓鱼相关状态（Idle 或 Moving），则进入移动逻辑。首先检测 W/A/S/D 或方向键是否按下，用 isKeyPressed 设置局部变量 isMoving。如果有任何移动键，这一帧认为玩家在移动：若之前状态是 Idle，则立刻切换状态为 Moving 并播放移动动画；还重置了脚步声播放标志，以便动画循环时重新触发声音。反之如果没有按键且当前状态是 Moving，则置 mPendingIdleSwitch=true，表示准备切回 Idle，但要等动画循环结束再正式切换。接下来，无论移动与否，都调用 mAnimator.update(dt)推进当前动画帧。如果正处于 Moving 动画，还会检查当前帧索引：当达到特定帧（这里判断 frame == 2）且脚步声未播，则播放行走音效“Moving”，并标记已播放以免本循环余下帧重复触发；当动画帧回到 0，则重置标志以便下一循环再次播放。然后处理 Idle 切换：如果先前检测到停止移动(mPendingIdleSwitch=true)，且此时动画帧索引归 0(说明移动动画正好播完一轮)，则将 Pending 标志清除、状态设回 Idle 并播放 Idle 静止动画。最后才真正计算移动：如果当前帧检测到按键（isMoving=true）且速度>0，则求和方向向量（根据按键组合上下左右得到 dir）并归一化；以 dt 和速度计算位移增量加到当前 Animator 精灵位置得到 newPos。然后利用边界对象检查

newPos 是否在允许范围内，若是则调用 setPosition 更新玩家位置（这同时更新 mPosition 和 Animator 精灵位置）。因此玩家移动由“按键 -> 状态变化(动画) -> 位置更新”这三个环节组成。合理的状态切换让动画过渡更自然，例如松开键时不会立刻静止，而是等当前步伐动作完才停。

- draw(sf::RenderTarget& target) const: 绘制玩家，实际上就是调用 Animator 的 draw 将当前帧精灵画出。另外在钓鱼状态下，还要调用 QTEManager.draw，在玩家精灵位置上绘制当前提示文本（提示“按 X!”的那个字）。QTEManager 的 draw 通过绘制预置的 sf::Text 来显示提示。这样玩家在钓鱼时，角色身上方会出现需要按的键字母，实现直观的 QTE 提示。
- setPosition(sf::Vector2f pos): 设置玩家位置。它将给定坐标保存到 mPosition，并调用 Animator.setPosition 保持动画精灵同步。由于游戏中移动基本通过 update 逻辑完成，直接用 setPosition 的场景主要是加载存档时恢复玩家位置。另外抛竿/收杆动画切换时也用 playAnimation 内部 setPosition 确保动画开始位置正确。
- 其他接口：setMovementBoundary(vertices): 设置边界多边形（PlayState 调用）。loadQTEFont(filepath, size): 加载 QTEManager 用于提示的字体及字号（PlayState 调用一次）。setFishCaughtCallback(cb): 设置鱼获回调（PlayState 在 onCreate 绑定，用于显示 UI 消息）。

Player 与其他类协作：Player 几乎是游戏内各种系统交汇之处：

- 与 Animator：组合关系。Player 驱动 Animator 的动画播放，并根据不同状态选择动画和调整位置。Animator 完全由 Player 控制，不自行移动。
- 与 QTEManager：组合关系+回调。Player 根据自身状态决定何时开始、更新、结束 QTE，而 QTEManager 内部完成具体随机和判定逻辑。Player 的 handleEvent 和 update 会多次调用 QTEManager 的方法。QTEManager 反过来通过 Player 通知结果（成功或失败音效由 QTEManager 内部播，鱼获物品则 Player 负责添加）。两者互相信任配合完成钓鱼小游戏。
- 与 AudioManager：Player 负责在合适时机调用音效播放。例如开始抛竿播放“StartFishing”音、抛竿结束播放“StartFishing2”音、行走时播放“Moving”音，这些都是直接通过 AudioManager 单例接口完成。同时 QTEManager 也会播放 QTE 相关音效。这说明音效调用分散在逻辑各处但集中由 AudioManager 调度。

- 与 Inventory/Item：Player 管理自身的 Inventory。当钓鱼成功，Player 创建 FishItem 并 inventory().add() 添加。InventoryState、SaveState 等通过 Game.getPlayer().inventory()拿到背包数据进行展示或保存。如果将来有其他道具，Player 的 inventory 机制也能复用。
- 与状态 State：Player 对象由 Game 全局持有，PlayState 对它进行主要操作。在 PlayState 的 update 中调用 player.update(dt)、draw 中调用 player.draw(rt)、handleEvent 中调用 player.handleEvent(ev)。PauseState/InventoryState 叠加时，Player 虽然暂停移动，但 Player 对象依然存在于 PlayState 中。StateStack 暂停 PlayState 的 update 调用，使 Player 停止更新，相当于冻结状态。恢复时再继续 update 即可，无需额外处理，这归功于状态机的自动管理。

总结：Player 类设计体现了组合优于继承和单一职责原则：它没有继承 Animator 或 QTEManager，而是将它们作为成员，各司其职。Player 像一个导演，协调动画、输入、物品、音效，让角色“活起来”。通过精心设计的状态机，玩家的各种行为无论正常行走还是钓鱼特殊流程，都被纳入统一更新逻辑，避免了杂乱的条件判断。Player 的代码相对复杂，但逻辑清晰分块，易于维护和扩展（如将来加新动作状态）。这一类的实现为游戏玩法奠定了扎实基础。

C. State 与 StateStack 类（游戏状态管理）

State（抽象基类）职责：

定义游戏状态接口，作为状态机中各具体状态的父类。它提供了统一的方法，让 StateStack 能够以多态方式管理不同状态。State 本身并不实现具体逻辑，但约束了子类必须实现的行为。

State 主要内容：

- 生命周期钩子：onCreate(), onActivate(), onDeactivate(), onDestroy()。这些为空虚函数，由子类按需重写。StateStack 会在状态 push/pop 时自动调用：首次 Push 调用 onCreate 和 onActivate；当状态遮挡另一个状态时，对被遮挡者调用 onDeactivate，当弹出遮挡状态时对下面状态调用 onActivate 恢复；Pop 或 Clear 销毁状态时调用 onDestroy。Fishlime 的各 State 子类有选择地实现了这些钩子。例如

MenuState::onActivate 用于每次显示菜单时布局元素，PauseState::onActivate 重新定位菜单选项；而 onDestroy 目前基本未用，但设计上允许释放资源等。

- 核心接口：handleEvent(const sf::Event&), update(sf::Time), draw(sf::RenderTarget&) const 纯虚函数。每个状态必须实现自己的输入处理、更新和渲染逻辑。这样 StateStack 就能在不知道具体子类的情况下，通过基类指针调用这些方法完成状态功能。这体现了开闭原则，StateStack 面对新加的状态类无需修改，只要他们实现这些接口即可良好运作。
- 其他：State 保存了对 StateStack 本身的引用 m_stack。这是通过构造函数注入的依赖。有了这个引用，State 子类可通过 getStack()访问其管理栈，从而自我操作栈，如 Push 新状态或 Pop 自身。实际开发中，我们更常通过 Game 提供的接口操作状态，但 State 持有栈引用在更通用框架下是必要的（Fishlime 中没直接用 m_stack，但例如可以做 getStack().pushState<...>()等）。

StateStack 职责：

管理多个状态对象的容器，提供栈的 Push/Pop/Clear 操作，以及每帧驱动栈顶状态运行和绘制。它实现了状态机控制逻辑。

主要属性：

- m_stack: 存放实际状态对象的容器，类型为 std::vector<std::unique_ptr<State>>。栈顶元素对应当前活动状态。
- m_pending: 存放暂存的状态操作列表，类型为 std::vector<std::pair<Action, std::unique_ptr<State>>>。其中 Action 是枚举{Push, Pop, Clear}。任何要变动栈的请求都压入这个列表，等待时机统一执行。
- m_context: 保存 Context 结构，里面是 Game 传入的窗口和资源管理器指针等。StateStack 本身并未用到 context 太多，仅在构造时存下以便给新状态构造传参。Fishlime 的状态构造函数通常签名为(StateStack&, Game&, ...)，StateStack::pushState 在创建状态对象时即传入*this 和 Game、其他参数。

主要方法：

- 模板方法 pushState<T>(Args&&...): 创建一个类型为 T 的状态对象并请求压入栈。实现上，它用 std::make_unique<T>(*this, std::forward<Args>(args)...)构造对象，

传入自己引用和其他参数，然后将(Action::Push, 新状态指针)加入 m_pending。这里用了完美转发 Args，以支持任意构造参数（Fishlime 中典型的是 Game&或者 Game&和 Player&）。由于采用 pending 机制，pushState 并不立即修改 m_stack。

- popState(): 请求弹出当前状态。实现就是 m_pending.emplace_back(Pop, nullptr)。不需要附带对象指针，因为 pop 不涉及创建新对象。
- clearStates(): 请求清空栈。实现 m_pending.emplace_back(Clear, nullptr)。
- applyPending(): 应用所有挂起的栈操作。它遍历 m_pending 列表，对每个操作类型执行：
  - **Push**: 将持有的 unique_ptr 状态对象移动进 m_stack 末尾。然后对新状态调用 onCreate()和 onActivate()。这样新状态初次创建和激活过程在此完成。
  - **Pop**: 检查栈非空，则对栈顶状态调用 onDeactivate()和 onDestroy()（这里假定 pop 总是用于主动退出，不考虑栈空情况）。然后 pop_back 弹出销毁之。如果弹出后栈不为空，说明恢复到下面一个状态，则调用新栈顶的 onActivate()。这样保证暂停过的下层状态在重新成为顶层时能重新激活资源（如音乐）。
  - **Clear**: 遍历整个栈，对每个状态调用 onDeactivate 和 onDestroy，然后清空向量。清空通常用于从游戏返回主菜单的场景。
  - 最后清空 m_pending 列表。值得注意的是，applyPending 是在 Game 主循环最后调用。这意味着在一帧内可能累积多个 push/pop 请求，全部在帧末一次性处理。如果 push/pop 相互交织，比如先 push 再 pop，也能处理有序（按记录顺序执行）。Fishlime 场景下 pending 列表通常最多一个操作，比如 scheduleAction 包裹的只有一个 push/pop。但框架支持多个。
- handleEvent(const sf::Event& e): 转发事件给栈顶状态。实现简单：如果栈不空，m_stack.back()->handleEvent(e)。只有顶层活动状态接收输入，其它被覆盖状态静默。
- update(sf::Time dt): 同理，只更新栈顶状态。这实现了状态暂停特性：非顶层状态不会更新，等再次被激活时才继续（相当于逻辑停留）。例如 PlayState 进入 PauseState 后，不再调用 PlayState.update，所以玩家停在原地不会走动时间也冻结。
- draw(sf::RenderTarget& target) const: 绘制状态栈内容。实现稍复杂以处理覆盖状态视觉效果：
  - 若栈为空直接返回。
  - 如果栈大小大于 1，则可能存在 Overlay 状态。首先判断栈顶是不是 LoadState（通

过 dynamic_cast)。为什么特别判断 LoadState？因为加载存档界面打算完全遮住下面菜单，不需要透出下层内容，于是对 LoadState 的处理不同。isLoadOverlay 为 true 表示顶层是 LoadState。

- 接下来，如果不是 LoadState（如 Pause, Help, Inventory 等属于半透明叠加），则绘制栈顶之下的那个状态。也就是倒数第二个状态的画面，会作为背景。
- 然后创建一个覆盖全窗口的矩形 fade，颜色设为黑色但透明度不同：如果是 LoadState 覆盖，则 alpha=255（全不透明黑，彻底遮住下层）；否则 alpha=150（半透明黑）。将这个 fade 绘制到 target 上，结果就是下层状态画面被盖上一层暗幕。如果顶层是 LoadState，暗幕完全不透明=直接盖黑底。
- 最后，再绘制栈顶状态。这样上层状态内容清晰可见，下层状态要么完全不可见（load），要么隐约可见当背景(pause 等)。

此绘制机制使 Fishlime 的状态叠加具有视觉区分度：暂停/背包/帮助界面下能看到游戏背景变暗，强化了上下文联系；而读档界面完全黑底避免了与菜单的混淆。实现上只需在 StateStack::draw 中处理一次，所有状态都自动获得这种效果，十分方便。

状态机交互：

- 状态切换调用链： 例如在 MenuState 点击“开始游戏”按钮，会调用 m_game.scheduleAction([this]{ m_game.changeState<PlayState>(player); });。这导致 Game 在帧末执行 changeState， Game.changeState 内部调用 StateStack.clearStates()+pushState<PlayState>。pushState 挂起 Push 操作；clearStates 挂起 Clear 操作。applyPending 执行时，会先 Clear: 把 MenuState 销毁干净，再 Push: 建立 PlayState，对其 onCreate+onActivate。下一帧开始 PlayState 即在栈顶运行。整个切换过程对开发者是透明的，无需手动处理细节，StateStack 自动管理了旧状态资源释放和新状态初始化。
- 状态自管理： 某些状态会在内部条件满足时退出自己。例如 PauseState 检测到 Esc 键则 resume 游戏，通过 Game.scheduleAction(popState) 实现。最终 StateStack.applyPending 会 Pop 掉 PauseState，自动激活 PlayState。State 类也可以直接使用 getStack()操作，比如 pop 自己或 push 别的状态（Fishlime 没直接用，但设计上允许）。因此每个 State 既受 StateStack 管理，也能主动要求 StateStack 变化，实现双向控制：高层模块 Game/菜单可以控制状态转换，状态自身也可按逻辑需要终止

或叠加其他状态（如 GameOver 状态可能自己 push ScoreBoard 状态等）。

- 资源共享： StateStack 持有 Context，因此每个 State 构造时都能拿到 Context 里的 window 和 ResourceManager 指针。Fishlime 中的 State 构造函数普遍形式如 PlayState(StateStack&, Game&, Player&)，Game 中又含 ResourceManager 成员，所以状态可以通过 Game 访问资源。但在一般架构中也可以让 Context 直接提供资源指针以减少耦合度。StateStack 的设计留有余地，不过当前实现简单地依赖 Game 对象。

State/StateStack 与其他部分关系：

Game 创建了 StateStack 并调用其 handleEvent/update/draw，让状态机融入主循环。StateStack 需要 Game 提供 Game&给状态构造，这有点依赖反转的意思，即高层 Game 把自身注入状态，以便状态调用 Game 接口（如 scheduleAction 等）。这虽然有少许耦合，但在这个项目体量下是方便的做法。理想上可引入接口 IState 或 IStateContext 削弱耦合，但非必要。

状态与状态之间通过栈耦合：上层状态知道存在下层但不直接相互引用，通过 StateStack 协调交互。例如 PauseState 退出后，谁来恢复 PlayState？答案是 StateStack 在 pop 时自动处理，不需要 PauseState 去调用 PlayState 的方法。类似地，InventoryState 关闭后游戏继续，也是不需要显式通知 PlayState“你可以继续了”，而是状态机机制保证的。这种松耦合提高了扩展性，一个状态的改动不影响其他状态。

综上，State/StateStack 实现了一个简洁而功能完备的状态管理系统。它使游戏不同场景逻辑分隔开，代码清晰；同时通过栈可以灵活叠加 UI。StateStack 的 deferred 操作确保安全，fade 效果提升体验。从架构上看，这部分代码很有复用价值，可适用于很多游戏项目。

D. 其他辅助类

AudioManager 类（音频管理）：

采用单例模式（静态实例）提供全局音频支持。AudioManager 在 Game 初始化时通过 AudioManager::get() 获取实例，并加载所有需要的声音文件。它内部使用 std::unordered_map 保存 SoundBuffer 和 Sound 对象，以名称索引。提供 loadSound(name, file)将文件读入 SoundBuffer 并创建 Sound；loadMusic(name, file)注册音乐文件路径；

playSound(name)查找 Sound 并播放；playMusic(name, loop)打开音乐文件并播放，可设置循环。还有 stop/pause/resume 音乐，以及 setMasterVolume/setMusicVolume/setSfxVolume 控制总音量和分类音量。

AudioManager 被整个游戏共享：各种地方直接通过 AudioManager::get().playSound("X")调用，无需传递对象引用。由于其内部自行保证实例唯一，这样使用很方便。但这也引入全局依赖，好在 Fishlime 项目规模可控，并无多大问题。音频播放的大部分逻辑在 AudioManager 里完成，开发者无需关心底层。例如双音效“QTESuccess2times”第二次成功音，只需提前用不同名字载入，一个条件满足时调用即可。AudioManager 也让音量管理变简单，通过一个地方控制和 Clamp 范围。总之，AudioManager 封装了 SFML Audio 的繁琐细节，使各模块以解耦方式使用声音。

ResourceManager 模板类（资源管理）：

这是一个通用资源缓存类，用于纹理和字体管理。内部用哈希表存已加载资源的 unique_ptr，提供 load(id, filepath)和 get(id)接口。load 如果资源未加载就创建并加载，从而保证每个资源只加载一次。Game 创建了两个 ResourceManager 实例，用于纹理和字体。各处需要资源时，调用如 game.textures().get("fish1")获得 Texture 引用。如果资源未被加载过，ResourceManager 会尝试通过构造函数加载文件。这简化了资源获取流程，也避免重复加载耗时。Fishlime 中大多数资源在 Game 构造里就用 load 预载入了（注意 ResourceManager::load 这里被用作两阶段，Game 先调用 load 返回 Resource 引用以预载入，然后状态里再 get 时已经有缓存）。ResourceManager 简洁地运用了 C++泛型，达到了在不重复代码情况下管理不同类型资源的目的，体现了代码复用性和抽象能力。

Button 类（按钮 UI）：

封装交互式按钮元素，广泛用于菜单和 UI 界面。Button 并未继承 GameObject，因为它不在 GameObjectManager 中统一管理，而是由各状态自己管理。Button 的构造接受显示文本、字体和字号。内部包含：m_rect 矩形底板（sf::RectangleShape）、m_text 文本(sf::Text)、可选的 m_sprite 贴图（sf::Sprite）。还有状态标志 m_isHovered 表示鼠标是否悬停在按钮上，和存储的字符串 m_buttonText 作为文本内容。Button 提供一系列设置接口：设置位置 setPosition()会同时调整矩形、文本和贴图的位置，确保文本居中对齐；setSize()改变按钮大小，内部会调用 setPosition 重新调整文本居中。此外可设置默认背

景色和悬停背景色、文本颜色等。支持为按钮添加贴图：setTexture(texture)将矩形换成 Sprite 显示，并根据按钮大小缩放贴图；也可用 setTextures(normal, hover)同时设置普通和悬停两种状态贴图（Fishlime 里“背包”按钮用同一图，未区分 Hover，setTextures 接口提供扩展可能）。

按钮的交互通过 handleEvent(const sf::Event&, const sf::Vector2f mousePos)实现：

- 每次调用时传入当前鼠标窗口坐标转换到世界坐标的 mousePos（各 State 通过 window.mapPixelToCoords 获取鼠标位置）。Button 用 contains(mousePos)检测该点是否在自身矩形范围内。
- 如检测到鼠标刚进入按钮区域（此前未 hover，此次 containsNow 为 true），则设置 m_isHovered=true，将矩形填充色改为预设的 hoverColor，并播放悬停音效（默认音效名称为 "BtnHovered"）。如果鼠标刚离开按钮区域，则还原颜色并重置 hover 标志。
- 接着如果当前处于 hover 状态，且事件是鼠标按下（MouseButtonPressed）且左键，则触发按钮点击逻辑：播放点击音效（默认 "BtnPressed"），然后如果开发者绑定了 m_onClick 函数则调用之。onClick 是在外部通过 setOnClick(std::function<void()>)设置的回调。Fishlime 中所有按钮的具体功能都通过 setOnClick 设置，例如 MenuState 用 lambda 实现切换状态。

Button 的渲染通过 draw(sf::RenderTarget&) const 完成：它根据是否设置 Sprite 决定绘制 Sprite 还是矩形，然后如果当前 hover 则绘制半透明遮罩矩形（m_hoverMask，半透明黑色覆盖在按钮上，让贴图变暗效果），最后绘制文字。这个顺序确保了无论贴图或底色，文字都在最上层清晰可见。 Button 类的实现考虑了 UI 控件的多方面需求：外观（颜色、贴图、大小、文字）可配置，布局有自动居中，交互有声音反馈且通过回调灵活定义行为。开发中我们曾调整过 Button 默认颜色，尝试用贴图装饰等。值得一提的是帮助界面的“上一张/下一张”按钮并没有贴图，只是透明矩形，在 hover 时才半透明显示，我们通过设置 TextColor 透明、FillColor 透明、hoverFillColor 半透明白来实现。这体现了 Button 的可定制性。Button 和 AudioManager 结合，使 UI 交互有声有色，在调试体验上提升很大。

QTEManager 类（钓鱼 QTE 逻辑）：

负责处理钓鱼时的快速反应小游戏逻辑。它不是游戏对象，不自行 update，而是由

Player 驱动使用。QTEManager 内部维护 QTE 的一次流程状态，包括：

- 提示按键种类 Key 的枚举（A/W/S/D 四种）；当前目标按键 m_currentKey；已经成功按对的次数 m_successCount；是否当前有提示激活 m_promptActive；QTE 流程结束标志 m_finished 及成功/失败标志 m_success。
- 随机机制：使用 std::mt19937 m_rng 和两个分布 m_intervalDist(2.f,3.f)、m_keyDist(0,3)用于生成下一次提示的延迟秒数以及随机键索引。
- 定时器：sf::Clock m_timer 和 sf::Time m_nextTime 记录下一次出提示的时间间隔；常量 m_timeout=3s 设定每次提示等待玩家输入的时限。
- 文本显示：包含一个 sf::Font m_font 和 sf::Text m_text，用于绘制提示按键的大字。构造函数 QTEManager()初始化了 Text 默认字体（需后面 loadFont 设置实际字体）和颜色。loadFont(path, size)加载字体并设置 Text 的 Font 和字符大小、颜色等。特别地，将 Text 的 origin 设为自身中心，方便后续绘制时居中定位。

QTEManager 流程方法：

- start(): 开始一次新的 QTE 流程。它将 m_promptActive 设为 false（表示目前没有提示在屏幕）、m_finished=false、m_success=false、m_successCount=0。重启计时器 m_timer 并调用 scheduleNextQTE()决定下次提示时间。这个函数在钓鱼开始时调用，每次重新钓鱼都重置状态，使上一次残留数据不影响新一次。
- scheduleNextQTE(): 随机决定下一提示出现的时间间隔，将 m_nextTime 设为一个 2~3 秒范围的随机值。下次调用 update 时若计时达到这个值就触发提示。
- update(sf::Time dt): 每帧调用，用于管理 QTE 定时和判定。如果 QTE 流程已结束(m_finished=true)，则什么也不做直接返回。否则分两种情况：
  - 如果当前已有提示(m_promptActive=true)，则检查计时器是否超过 m_timeout 3 秒。若超时说明玩家没有及时按键，则认定失败：设置 m_finished=true, m_success=false。之后 Player.update 检测到 isFinished 会转入 FinishFishing 状态收杆。
  - 如果当前没有提示(m_promptActive=false)，则累计计时，当 m_timer.getElapsedTime() >= m_nextTime 时触发一次新的按键提示。触发时将 m_promptActive=true，随机生成一个按键 m_currentKey = randomKey()，播放提示音效"QTEStart"提醒玩家。然后设置提示文本内容：根据 currentKey 是 A/W/S/D

设置相应宽字符 "按 A！"等。重新调整 Text origin 为中心（因为改变了字符串长度）。最后重启计时器，用于计算玩家反应时间。这样，QTEManager 在一段静默等待后激活提示，等待玩家输入。

- handleEvent(const sf::Event& event)：由 Player 在 Fishing 状态时调用，用于判断玩家按键对错。如果 QTE 已结束或当前没有提示激活（promptActive=false 意味着还没提示就按键了）则直接返回什么也不做。否则，当捕获到 KeyPressed 事件时：取出按下的键码 code；将它与当前目标键 toSfKey(m_currentKey)比较。如果正确：则 m_successCount++，播放成功音效"QTESuccess"；如果这是第 2 次成功（successCount==2），额外播放"QTESuccess2"音效。然后判断成功次数是否达到 3：如果未达到 3，则还没结束：将 m_promptActive=false 准备下一个提示，并调用 scheduleNextQTE()重新定下次随机触发时间，重启计时；如果这次成功使 successCount>=3，则表示三连成功完成钓鱼，把 m_finished=true，m_success=true，并播放"FinishFishing"完成音效。
- 如果按错：则直接失败，m_finished=true，m_success=false，播放失败音效"FishingFailed"。
- 处理完正确或错误，在这次 QTEPrompt 上都算结束提示，函数返回。后续状态切换由 Player.update 监测 m_finished 进行。
- draw(sf::RenderTarget& target, const sf::Vector2f& position) const：在给定位置绘制当前提示文本。只有在 promptActive=true 时才绘制，以免平时无提示却画出空文本。实现上，为了不改变 const 对象内部状态，采用复制一份 sf::Text，然后将这份临时文本 tmp 的位置设置为指定 position，再 draw。在 Fishlime 中，这个 position 由 Player.draw 调用时传入玩家动画精灵的位置。这样提示字将出现在角色身上方（因为 sf::Text 默认绘制锚点在字符区域中心，我们设置 origin 为中心，所以 position 其实就是文字中心位置）。实际效果是字母出现在角色稍偏上的位置，看起来像头顶指示。

QTEManager 与其他部分关系：

- 仅由 Player 持有和使用，不与其他类直接交互。Player 在 StartFishing 时 mQTEManager.start()；Player.update 中 mQTEManager.update(dt)；Player.handleEvent 中 mQTEManager.handleEvent(event)；Player.draw 中

mQTEManager.draw(target, position)。因此 QTEManager 完全依赖 Player 驱动。

- QTEManager 用 AudioManager 播放音效，资源名由我们事先在 Game 加载音效时定义（QTEStart/QTESuccess/...）。
- QTEManager 的生命周期随 Player，由 Player 构造时自动构造。没有动态分配、没有指针管理烦恼。

通过 QTEManager，我们将钓鱼小游戏的逻辑封装起来，使 Player 无需关心按键随机与计时等细节，只需监视结果。代码清晰地表达出成功按三次或超时失败的规则，易于调整参数。它也符合开放封闭原则，若以后增加新的 QTE 类型，可通过多态或策略扩展。但在当前简单需求下，一个类已足够完成功能。

MovementBoundary 类（移动边界判定）：

用于多边形区域的点内检测。在 Fishlime 用它限制 Player 移动范围（例如只允许在沙滩区域）。MovementBoundary 内部存储顶点列表 m_vertices。构造可传入初始顶点，也提供 setVertices(vector<sf::Vector2f>) 更新。主要算法实现是 contains(const sf::Vector2f& point) const：采用射线法判断点在多边形内。简单来说，从测试点水平向右发射一条射线，数一数它与多边形每条边相交次数，若为奇数则点在多边形内，偶数则在外部。代码具体做法：对顶点数组每相邻两点(vi, vj)组成的边，检查射线与边是否相交。里的判断条件 (vi.y > point.y) != (vj.y > point.y) 确保测试点水平位置在两个顶点 y 坐标之间(即射线可能穿过边);然后计算射线与边交点的 x 坐标 (vj.x - vi.x) * (point.y - vi.y) / (vj.y - vi.y) + vi.x，如果测试点的 x 小于交点 x，则说明射线在此边发生穿越。每发现一次交点就取反 inside 布尔变量。最后根据 inside 真/假返回。这个实现是经典的点-多边形检测算法，也叫“射线交叉法”，时间复杂度 O(n)对于单次判断足够快。

MovementBoundary 本身不知道使用场景，Fishlime 中 PlayState 在 onCreate 时定义了顶点数组并传给 Player.setMovementBoundary。顶点坐标我们根据沙滩形状手动取了几组（略作简化成多边形）。经测试 MovementBoundary.contains 准确判定，Player 移动中调用它避免出界。MovementBoundary 体现了算法封装思想，将通用几何算法封在类中，Game 逻辑部分无需管实现细节，只调接口得到结果。这提高了代码可读性，也方便复用或扩展（若我们地图有多个多边形区域，可实例化多个 MovementBoundary 应用于不同角色或 AI）。

Item / FishItem / Inventory 类（道具及背包）：

Item 是抽象类，只有两个纯虚方法：getName()和 use(Player&)。Fishlime 目前只有一个子类 FishItem，用于表示钓到的鱼。FishItem 构造时传入鱼重量、稀有度和纹理 ID。getName()返回字符串如"Fish 3"；use(Player&)在当前实现为空（可以拓展为吃鱼回血等效果）。FishItem 还提供 getTextureId()返回纹理名称字符串，方便 UI 界面拿对应贴图显示。

Inventory 类是一个简单容器，内部保存 std::vector<std::unique_ptr<Item>> m_items。方法有 add(std::unique_ptr<Item>)添加道具、remove(size_t index)删除某位置道具、get(size_t index)返回指针供查询使用。还有 size()返回当前数量。Fishlime 的设计假定每个 Item 都是独立 slot（物品不堆叠，每条鱼一个 Item）。Inventory 没有自己的 draw，显示逻辑由外部决定（InventoryState 根据背包内容统计各鱼数量来展示图标）。

在 Player 中组合一个 Inventory，这样玩家就拥有背包能力。钓鱼得到鱼时，创建 FishItem 加入 inventory。存档时数背包中各 FishItem 统计数量。读档时根据数量批量 create FishItem 加入 inventory。Inventory 相当于内聚了物品集合的操作，方便在不同地方调用，不用每次都处理底层容器。因为 Item 基类的存在，将来可扩展其它道具类型（比如渔具、金币），Inventory 不用修改。符合开闭原则和里氏替换：背包里的对象都以 Item 接口操作，FishItem 只是其中一种实现。