QT 大作业报告 —— 一羿孤行（横版射击游戏）

熊江凯 2000012515 刘明灏 2200012911 吴雨洋 2200012914

一、六大项目亮点

（1） 图像压缩技术

仅仅使用一张 png 图像（仅 10kb）就完成了所有游戏人物和游戏字体（英文）的绘制，使得整体游戏的大小减小了 80%——这个技术是受到超级马里奥的启发

（2） 丰富的技能种类

在游戏商店中提供了超过 18 种技能供玩家选择，技能不仅仅体现在底层的基础数据的改变，我们也设计了华丽的技能效果能直观呈现出来。

（3） 独立开发的游戏引擎

独立完成物理特性、碰撞体积、角色绘制、数学处理、图像处理等多项引擎功能，提供了丰富的接口供前后端调用。

（4） 扁平化的工厂函数模式

通过精简继承关系，通过扁平化的工厂模式产生游戏中的对象，在这个体量的游戏中提供了足够灵活的拓展性和足够丰富的拓展接口。

（5） 组合优于继承的设计模式

通过类的组合，将游戏物体的本身和复杂的行为逻辑分离再通过“行为容器”组合，大大简化了游戏行为的开发。

（6） 计算机生成音乐的设想

通过调用 Web Audio API，我们成功的将音乐设计拆分成了音符级别的计算机操作，通过随机生成的 bassline，组合多层次的配器，实现了背景音乐的生成——但是受限于 QT 的平台限制，该 API 并不能内置于游戏中（遗憾）。

二、项目模块架构和关联性

2.1 项目模块架构

（1） 层次介绍

本项目的主要文件分为五个层次，分别是：基础工具层、核心引擎层、对象实例层、运行逻辑层和画面展示层。总体呈现层层递进的逻辑关联，

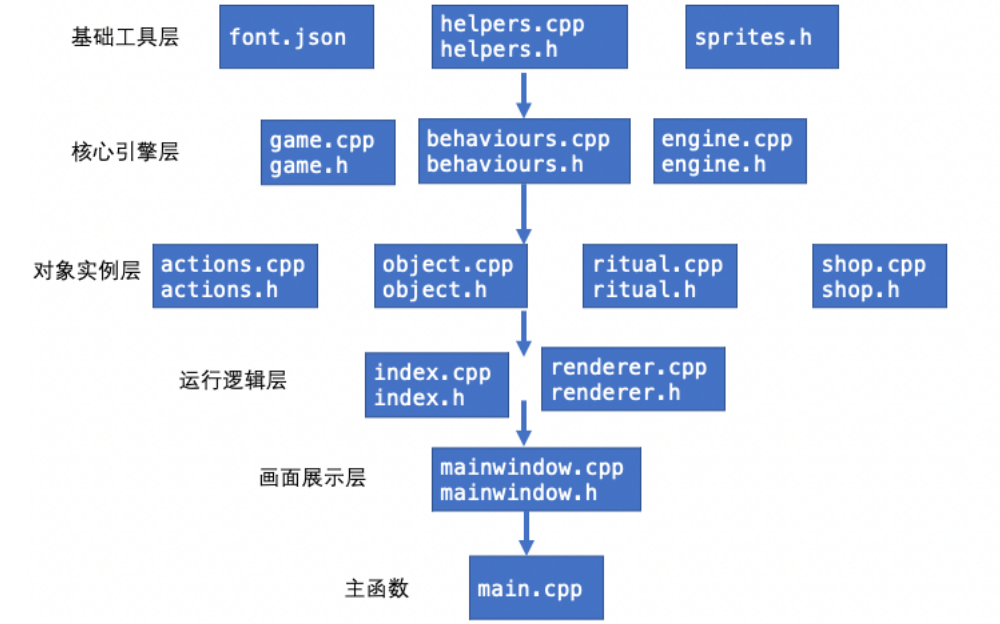

图 1：项目文件架构

2.2 文件功能介绍

文件	功能
helpers	提供基础的数学计算和随机性的函数和模版
sprites	提供游戏角色贴图坐标信息
font	提供字体坐标信息
game	游戏基本元素、游戏本身的类与方法
behaviour	游戏角色行为的类与方法
engine	游戏引擎，提供游戏画面更新和物理规则的约定
actions	组合角色行为形成更为复杂的运动
object	游戏中基础物体的类和相关方法
ritual	游戏中的技能的类和相关方法
shop	游戏商店的类的购买物品相关方法
index	游戏过程控制与游戏输入输出控制
renderer	游戏画面逻辑与函数
mainwindow	游戏画面呈现
main	主函数

表 1:文件功能

2.3 文件关联介绍

游戏的核心抽象方式是将游戏拆分为对象，而对象将属性和行为分离——相当于对于 OOP 的广度优先的抽象方式，不再需要冗长的继承关系——这也是众多游戏所乐于采用的方法。

核心引擎层面的文件是组织整个游戏的关键，承担起了物理规则和游戏内所有对象的基类的作用，而由此引申出来的对象实例层，主要通过少继承多组合的思想形成浅而广的对象集合，在这个体量的游戏下，很利于增加其中的所需的游戏角色和角色的行为。

前端的逻辑相对简单，只需要调用后端提供的接口，不断调用 update 函数更新游戏内容即可。

基本分工：

人名	分工
熊江凯	后端逻辑和类的设计——核心引擎层和对象实例层的实现
刘明灏	前端逻辑方面和基础工具——基础工具层和部分运行逻辑层
吴雨洋	前端显示和报告撰写、视频拍摄——画面展示层和主函数

表 2:基本分工

三、 类的设计细节

3.1 角色图像类的压缩

值得说道的是，我们成功地将 83 个游戏物品和 96 个英文字符的贴图放在了一个 160x70 的 png 文件中，而这个文件仅仅只有 10kb。

而这个压缩主要通过 Photoshop，在事先准备好的底板上规划好每个贴图的区域，进而进行像素级的绘画

图 1：早期角色设计（和成品有所区别）

我们甚至为了把节约空间进行到底，选择放弃中文的字体，转而把所有英文字母和数字、常用符号通过像素画的方式放在了 png 图像的上方

3.2 角色图像移动的设计

主要通过补间(Tween)和粒子效果实现了整体游戏角色的动态效果，而贴图本质上都是静态的，这大大增加了游戏角色的精致程度

```
void updateTweens(double dt)
{
    // Implementation for updating the active tweens
    // 过滤未完成的 tweens
    tweens.erase(std::remove_if(tweens.begin(), tweens.end(), [&](Tween &tween)
                                {
        tween.elapsed += dt;
        double progress = helpers::clamp(tween.elapsed / tween.duration, 0.0, 1.0);
        double t = tween.ease(progress);
        double value = tween.startValue + (tween.endValue - tween.startValue) * t;
        tween.callback(value, t);
        return progress < 1.0; }),
                 tweens.end());
}
```

图 2：补间函数的实现（部分）

3.3 角色法术类（Spell）的设计

游戏的主要攻击方式就是玩家通过发射受重力作用的法术球实现的，因此我们在法术的丰富性上给出了大量的笔墨。

法术的基本攻击时很简单的抛物线，玩家选择一个角度就可以发射出子弹。但是玩家可以通过货币在商店购买技能实现多种多样的法术形式，19 种技能中有 13 种是具有特有粒子效果的，这大大增加游戏的观赏性和趣味性。

```
// 技能的函数（一部分）
extern game::Ritual Streak;
extern game::Ritual Bouncing;
extern game::Ritual Doubleshot;
extern game::Ritual Hunter;
extern game::Ritual Weightless;
// 击退技能（依赖击退法术类）
extern game::Ritual Knockback;
// 产生一个天花板的技能
extern game::Ritual Ceiling;
// 其他技能
extern game::Ritual Rain;           // 法术分裂为三个
extern game::Ritual Drunkard;       // 醉拳
extern game::Ritual Seer;           // 穿透（尸体）法术
extern game::Ritual Tearstone;      // 破釜沉舟（血量小于一半，攻击力翻倍）
extern game::Ritual Impatience;     // 降低召唤死灵的技能冷却时间
extern game::Ritual Bleed;          // 流血技能
extern game::Ritual Allegiance;     // 在复活之后集结死灵
extern game::Ritual Salvage;        // 尸体在游戏结束时换钱
extern game::Ritual Studious;       // 商店物品便宜一半
extern game::Ritual Electrodynamics; // 闪电法术
extern game::Ritual Chilly;         // 寒冰法术
extern game::Ritual Giants;         // 有机会召唤出巨人死灵
extern game::Ritual Hardened;       // 死灵的血量加1
```

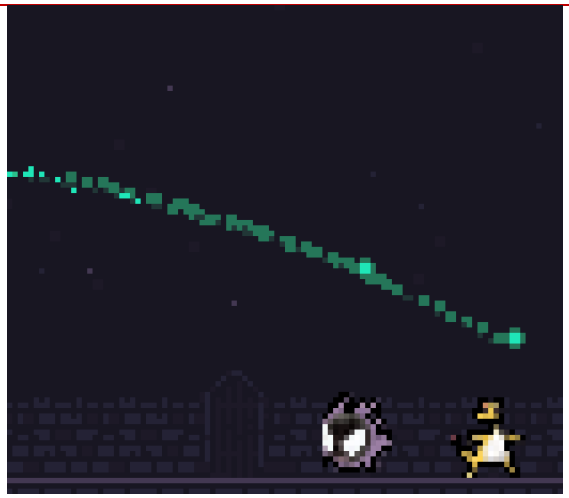

图 3:所有技能的设计和玩家基础攻击方式

3.4 游戏物体类（GameObject）的设计

我们将游戏中所有的物体都定义在了游戏物体类上，并且没有对这个类进行任何的继承，从而形成了一个足够扁平的类的结构——这是受到了著名的 Youtube 视频 "Is There More to Game Architecture than ECS?"的影响。比如，我们将游戏中的基础小怪设计为工厂函数，通过调用函数，就可以返回一个所需的游戏角色的对象，其相关属性在函数内部都已经调整完成

```
// 村民对象函数实现
game::GameObject objects::Villager()
{
    game::GameObject unit; // 创建游戏对象
    std::vector<Sprite> tmp = {sprites::villager_1, sprites::villager_2, sprites::villager_3, sprites::villager_4};
    unit.sprite = helpers::randomElement(tmp);          // 随机设置精灵图像
    unit.friction = 0.8;                                // 设置摩擦力
    unit.mass = 75;                                     // 设置质量
    unit.x = game::game.stage.width;                    // 设置初始位置
    unit.tags = LIVING | MOBILE;                        // 设置标签
    unit.hp = unit.maxHp = 1;                           // 设置生命值和最大生命值
    unit.updateSpeed = 600;                             // 设置更新速度
    unit.addBehaviour(new behaviours::March(unit, -16)); // 添加行军行为
    unit.corpseChance = 0.75;                           // 设置尸体生成概率
    unit.souls = 5;                                     // 设置灵魂数量
    return unit;                                        // 返回游戏对象
}
```

图 4:工厂函数的设计模式 1

实际上，不同的工厂函数也可以相互调用，实现类似于继承的能力，比如，下面这个角色就在上面的工厂函数提供的对象的基础上进一步更新了属性和方法：

```
// 小丑对象函数实现
game::GameObject objects::Piper()
{
    game::GameObject unit = objects::Villager();                          // 创建村民对象
    unit.sprite = sprites::piper;                                         // 设置精灵图像
    unit.updateSpeed = 500;                                               // 设置更新速度
    unit.hp = unit.maxHp = 15;                                            // 设置生命值和最大生命
    unit.addBehaviour(new behaviours::Summon(unit, objects::Rat, 2000)); // 添加召唤行为（召唤老
    unit.souls = 100;                                                     // 设置灵魂数量
    return unit;                                                          // 返回游戏对象
}
```

图 4:工厂函数的设计模式 2

3.5 法术技能类（Ritual）的设计

由于玩家在游戏中每一种技能仅仅只能购买一次（买两次没意义），所以，我们使用了某种意义上的单例模式对技能进行设计——因为后续调用只会使用我们实例化的 Ritual 对象，而不会再重新初始化新的 Ritual。

```
// 技能的结构体
struct Ritual
{
    std::string name;
    std::string description;
    int tags;
    int exclusiveTags;
    int requiredTags;
    bool recursive;
    Rarity rarity;
    void (*onFrame)(float dt);
    std::function<void()> onActive;
    std::function<void(GameObject *)> onCast;
    std::function<void()> onResurrect;
    std::function<void(GameObject *)> onResurrection;
    void (*onDeath)(Death death);
    std::function<void()> onLevelEnd;
    void (*onLevelStart)();
    std::function<void()> onShopEnter;
```

```
game::Ritual Weightless{
    NONE,
    "Weightless",
    "Spells are not "
"    affected by gravity",
    [](game::GameObject *spell)
    {
        spell->mass = 0;
        spell->friction = 0;
        spell->bounce = 1;
    }};
```

图 5:游戏技能的对象设置

我们通过各类二进制掩码的 tags 完成技能之间依赖和互斥关系的判别，比如：exclusiveTags 进行“与”操作如果不为 0，则说明这两个技能不能同时拥有，如果 requireTags 的“与”操作不为 0，则说明这两个技能有依赖关系。

3.6 角色行为类（Behaviour）的设计

将角色属性和行为分离，大大简化了类的复杂程度，这样的分离或许和我们在课上学习的 OOP 思想有所不同，但是在游戏设计中确实比较常见且有效的设计手段。

每个游戏物体都有一个行为数组，这些行为响应游戏过程中发生的各种事件，但是角色行为类本身是一个十分简单的类

```
// 行为的类
class Behaviour
{
public:
    virtual ~Behaviour() {}
    Behaviour(GameObject *object) : object(object) {}
    int turns = 1;
    int timer = 0;
    Sprite sprite;

    std::function<void()> onAdded;
    std::function<void()> onRemoved;
    std::function<std::optional<bool>()> onUpdate;
    std::function<void()> onBounce;
    std::function<void(struct Damage *)> onDamage;
    std::function<void(Death)> onDeath;
    std::function<void(double)> onFrame;
    std::function<void(GameObject *)> onCollision;
    GameObject *object;
};
```

图 6:行为类的设计

通过对这个类的继承，我们衍生出了一系列角色的行为——其中大部分是为了给技能设计作铺垫。每个对象都有一个内部更新速度，以在不同类型的敌人之间产生差异，反过来，每个行为都有一个回合计时器。当父对象更新时，计时器就会启动，当计时器 > 规定好的启动时间时，我们会调用 onUpdate 方法更新事件。

其中 invulnerable 保证了角色不会受到伤害，March 保证了敌人行进的逻辑等等。具体举一个例子：Summon 行为的存在，是的我们之前提到的小丑角色能够在移动过程中不断召唤老鼠对象，而巫师也可以在移动过程中不断放置传送门。

四、项目总结和反思

4.1 游戏音乐的遗憾

我们最初决定让计算机通过调用 API 运算产生音乐，游戏中的音乐随着角色在敌人的浪潮中前进而构建。音乐的事件开始于一个在相同音符上反复的单独的“风琴”，然后加入随机生成的贝斯线，最后是底鼓。我尝试生成主音合成器线、踩镲和军鼓层，但它们最终都有太多的程序性差异，以至于我不愿意保留它们。

```
sequence([A4, H, A4, H], -36, synths.kick);
sequence([A4, E, A3, E], -36, synths.ambientOrgan);
sequence(createBassline(), -24, synths.bass);
```

图 7:音乐类的设计

比如这段代码，第一个参数是音符索引和音符长度的连续数组（例如 A4 为 0，表示 440hz，H 是半音符，持续小节长度的一半）；接下来是一个失谐参数，可用于将整个乐句上下移动，这使得在 3 个八度音阶上生成模式变得更容易（我选择 A3 到 G5），然后根据需要将它们向上或向下移动将用哪种乐器来演奏它们。

合成器本身是振荡器、增益节点、滤波器和效果器的更复杂的组合，具有大量手动调整的斜坡以保持听起来流畅

```
function Kick(): Synth {
  let synth = Synth();
  synth.filter.type = "lowpass";
  synth.filter.frequency.value = 80;
  synth.osc.frequency.value = 150;
  synth.play = time => {
    synth.osc.frequency.setValueAtTime(150, time);
    synth.gain.gain.setValueAtTime(1, time);
    synth.filter.frequency.setValueAtTime(80, time);
    synth.osc.frequency.exponentialRampToValueAtTime(0.001, time + 0.5);
    synth.gain.gain.exponentialRampToValueAtTime(0.001, time + 0.5);
    synth.filter.frequency.linearRampToValueAtTime(0.001, time + 0.5);
  };
  return synth;
}
```

图 8:合成器类的设计

除了整个游戏中轨道的逐渐分层之外，还有其他事件需要改变合成器和排序，而这最终导致编排起来很棘手。例如，当你走进一家商店时，底鼓会停止，以减轻一些压力，当你到达游戏的结局时，一个完全不同的曲目开始，合成器在战斗的不同阶段返回。我基本上是通过让所有东西一直播放并使用单独的增益节点将乐器带入和带出来实现这项工作的。

尽管做了这些努力，但是因为 QT 对于 API 支持的限制，我们没办法将 Web Audio API 移植到 QT 中，因此只能留作遗憾。

4.2 游戏平衡性的遗憾

游戏在技能设置上虽然十分丰富，但是某些强力技能如果在前期获得了，那么只要熬过了中期的难搞的 boss——小丑，那么后期就只需要无聊地点击屏幕就好了——所以如果能够在前期降低强力技能的出现概率，而不是在仅仅在整个游戏过程中保持相同的概率（不同技能之间还是有概率区别），游戏的难度和可玩性会进一步提升。

对于游戏的赚钱机制，也存在问题——当敌人死亡（无论是死于法术还是骷髅）时，角色都会获得固定数量的金钱（根据敌人类型）。在每个关卡结束时，角色可以使用这些金钱来购买技能，这是游戏唯一的升级系统。

问题是，赢得金钱太容易了，在游戏结束时，你通常已经赢得了大量的金钱，以至于你可以购买所有的技能。对于 QT 大作业来说，这是一个不得不做的取舍，人们可能只会玩我们的游戏几分钟，我们希望他们看到游戏玩法的多样性。

4.3 小组合作的反思

（1） 前后端接口的沟通问题

尽管我们很明确的进行了前后端的分工，但是双方的沟通还存在一些不及时的问题——导致前后端的接口出现了一些错配，比如，在指针和对象复制的问题上，后端按照指针传递的逻辑进行，而前端则认为容器中传递的是对象本身，这就导致了整体的框架上的方向区别——进而导致了一次大范围的修改。

因此，我们认为，前后端沟通最为重要的不是自身内部的复杂逻辑，而是要强调接口的兼容性和通用型。

（2） 团队内部的任务规划问题

我们往往采用较大目标的方式规划一大段时间内的任务，但是这样的方式并不能灵活的应对学期中的突发事件，导致很多时候我们是没有办法在规定时间内完成任务的——这会在一定程度上打击团队的士气。

所以在之后的团队合作中，随着我们对于编程任务的经验进一步加深，我们会有能力和意识去进一步细化任务，做到“小步快走”，逐步完成整个项目。

4.4 对程设课程老师和助教的感谢

这个学期的程序设计实习课程设计精良,选材适度, 老师的讲解清晰精彩,使我们对程序设计有了比较系统和深入的理解。各种练习和作业的设计也很到位,不仅增强了我们的编程技能,也锻炼了独立思考的能力。在学习过程中遇到的困难,老师和助教们总是很快给予解答和帮助,这让我们对这门课程充满信心。

老师的精心教学和课程设计,助教们的辛勤帮助,使这门课程得以顺利进行,也使我们学生受益匪浅。我们由衷感谢老师和助教们这个学期的付出与贡献。老师的教诲和助教们的努力,将持续鼓舞我在未来的学习与工作中追求更高的技能和水平。

这门课程的设计和教学都很出色,我们很幸运能在老师和助教们的悉心指导下学习程序设计。谢谢老师和助教们这个学期的辛勤工作！！！