CS100 HW7 — Runaway

注意

- **请为本次作业分配充足的时间**（参考：至少1.5天），并且不要到临近截止日期时才开始！

- **不要被本说明的长度吓到**。它之所以这么长，是因为在每个游戏对象的细节说明里，我们有意做了很多不必要的文字重复。我们希望你能够从这些重复的内容里提取出共同点与区分点，通过自己的构思与设计来解决继承结构等问题，而不是被我们用我们设计的结构束缚住。
- **本文档中只是包含样例实现的介绍**，你不必全部实现，也不必完全遵循其中要求。你的目标是做出「你自己」的游戏，而不是「与样例一致」的游戏。本文档只提供作为参考的大量的实现流程与数值等细节。
- 如果对任何部分有疑问，在你提问之前，请先在本说明和FAQ中寻找答案，或试着游玩样例游戏并观察样例游戏的处理方式。
- 请尽量在Piazza上公开提问而非私下询问TA。有相同疑问的其他同学也可以看到你的问题，你也能看到我们对更多问题的回答。这会更高效地解决你的疑问。
- **请一定经常上传和提交git以保存备份（无论离线或在线）**！如果你为了添加某一点功能使得整个程序崩溃或是在修复时越修越糟，简单地回退一版并比较不同，就可以快速解决问题。
- 如果你需要在保留当前版本的同时，开一版新的试错，那么不妨试试 `git checkout -b feature/newsomething` 其中newsomething可以是你新功能的名字，然后做好了测试通过以后，可以用 `git checkout develop && git merge feature/newsomething` 将新的功能合并到 `develop` 分支上

故事

昏暗的屋子里，老旧的CRT显示器泛着幽幽绿光，你缓缓睁开眼睛，周围的一切都让你感到陌生而又熟悉。墙上的日历清晰地写着“1986年”，破旧的房间里堆满了磁带机、打字机，还有几本泛黄的编程书籍。你有些茫然，最后的记忆停留在2025年那场灾难性的裁员。

“我不是在写OpenGL代码吗？这是什么鬼地方？”你低头，看见自己放在桌上的那台笔记本电脑。屏幕上依然显示着C++17代码和一段未完成的OpenGL渲染程序，这显然不属于80年代的科技。此刻，你才意识到一个惊人的事实——**你穿越了！**

2025年的你是一名经验丰富的程序员，曾参与多个大型3D游戏项目，尤其擅长C++17和OpenGL开发。然而，随着游戏行业的AI化浪潮，你的工作被自动化取代，最终被公司裁员。失业的你原本想用自己的技能开发一款属于自己的游戏，却因为资金和创意的枯竭而难以推进。而如今，命运给了你一次机会——回到1987年，一个属于8位游戏机和街机的年代。尽管技术落后，但你手里的工具却来自未来。

在这个时代，你意识到所有的技术都可以简化重构。你决定利用**C++17和OpenGL**的超前能力开发一款适配80年代街机的游戏。你以一个逃亡者的故事为背景，构想出《Runaway》这款游戏：一个主角被怪物哥布林追捕，在不断奔跑和战斗中寻找生存与自由的冒险故事。完成第一版Demo后，你找到一家小型街机厂商，请求他们帮忙将游戏部署到街机平台。当你展示《Runaway》的运行效果时，厂商的人瞪大了眼睛：“这画面是怎么做出来的？！还有这些敌人的动作也太流畅了吧！”你没有解释“未来的秘密”，只是微笑着告诉他们：

“技术突破，总是有方法的。”

当《Runaway》第一台街机被摆放在街头时，人们被那种流畅的动作、紧张的追逐感和动态的光影效果深深吸引。
“这游戏太厉害了！”，“就像在电影里一样紧张！”
人们排起长队，争相体验这款划时代的作品。最终，《Runaway》成为80年代街机市场的一匹黑马。

作为穿越者，你在游戏开发的过程中，也在思考如何应对未来。你逐渐意识到，这个时代的技术尽管落后，但却充满了创造力的可能性。你甚至开始怀疑，自己穿越的意义是否是为了重新点燃对游戏开发的热爱。

“或许这就是命运的安排。”
你站在游戏厅里，看着玩家们的欢笑与惊叹，内心涌起无尽的满足感。尽管你知道，自己无法彻底改变未来的游戏发展轨迹，但你创造了一款属于自己的作品，并真正让这个世界记住了你的名字——正所谓，**争者留其名。**

《Runaway》不仅是一个逃亡的故事，也是一个追逐梦想的故事。而这，仅仅是你的传奇的开端。

游戏简介

很久以前，曾经祥和的艾弗林小镇被一股神秘的哥布林潮席卷而过。这些哥布林原本只是生活在森林深处的小型种族，却在一夜之间变得异常狂暴、强大，甚至展现出前所未有的组织性。有人目击到一个身披黑袍的神秘身影潜入森林，怀疑他是这一切的幕后黑手。

你是一名年轻的发明家，偶然发现了一个古老的机械装置，隐藏着能够操控哥布林的力量。这一发现使得你成为了哥布林首领的头号目标。他派出无数哥布林与怪兽追捕你，为了夺回这个装置。你只能携带着装置，在森林中逃亡，寻找方法解开装置的秘密，并用它反击哥布林的侵略。

你要做什么

你会拿到一份游戏框架，框架为你处理了图片素材的显示和键鼠交互，但没有任何游戏内容。你通过代码创建的每个对象都可以被框架显示出来，因此你只需要通过代码为游戏编写内容。

与往年不同，今年的这次作业完全移除了OJ的代码测试，全部以线下check的形式进行。这意味着我们不再限制你对游戏内容的实现，你不必与我们的文档要求保持一致。你只需要做出「你自己的」一个「可玩」的游戏。并且通过代码注释或一份简单的文档说明自己写了哪些功能，就可以通过check。具体请参考grading部分。

游戏是如何运作的

你所看到的不断刷新的游戏界面其实也是一个逐帧播放的视频。在游戏中，一帧(frame)的时长被称为一刻(tick)。一刻即为游戏内部的一个周期。在每一刻中，游戏中所有内容的状态都会被更新，对玩家的输入做出反应，然后再被显示到游戏界面上。每一帧内不同的游戏内容可能会导致游戏以稍微不稳定的速率运行，但理想状态下，我们希望游戏运行中每一刻(tick)的时间是恒定的，比如每秒30帧。

面向对象编程(OOP)的思想在游戏中被广泛应用。我们可以将这个游戏的所有组成成分（玩家，哥布林，玩家子弹，哥布林的斧头，甚至还包括背景等）都抽象成一个个**“游戏对象”(GameObject)**。既然成为了一个C++中的对象，那么它便可以用成员变量(member variables)来储存属于自己的信息（比如所在的x, y坐标、剩余的HP等），也可以用成员函数(member functions)来定义自己应当做的行为（比如玩家如何攻击、哥布林如何移动和攻击）。

刚刚提到过，在每一刻里，游戏的所有内容都会更新。在我们的游戏里，这一行为实现在了Update()函数中。在一刻的时间中，我们可以让所有的游戏对象(`GameObject`)都执行一次`Update()`函数。不同的对象对这个函数的内容需要各有实现：玩家有可能会跳，也有可能会开枪，子弹会向右移动，而哥布林和它的斧子会向左移动，背景图片则什么都不需要做。

在本游戏中，玩家会通过点击屏幕（或者按`J`键）来完成开枪，而哥布林会隔一段时间丢出一个斧头。

当所有游戏对象(GameObject)都进行了一次`Update()`后，我们的游戏就应当把新的状态显示在屏幕上。原有的上一帧的游戏显示将会被清除，然后再根据每个游戏对象Update()更新过的状态（比如移动后的所在位置），重新将所有游戏对象再画出来。游戏对象具有的动画也会被逐帧播放。在一次`Update()`中或是逐帧动画的两帧之间，游戏对象通常不会移动太大的距离，从而使Update与显示不断循环下的游戏界面看起来像是平滑的动画。

在GameWorld类中管理你的游戏

`GameWorld`类就是你的游戏世界的缩影。在你的游戏中，无论是开始或结束一个关卡、添加或删除一个游戏对象，还是处理游戏对象之间的互动（子弹打到哥布林），都在`GameWorld`类中完成。因此，`GameWorld`类最重要的功能便是储存和管理游戏对象。

你的`GameWorld`类中需要有一个储存游戏对象的容器。你允许使用任何标准模板库(STL)容器，但我们非常建议你使用`std::list`。它内部的链表结构使它适用于需要频繁地添加或删除的应用场景，同时`std::list`还支持在使用迭代器(iterator)遍历的同时添加或删除元素，游戏中某些步骤可能需要此功能。（例如，玩家在鼠标被点击是希望在`GameWorld`中创建一颗子弹）

有了这个容器，就可以将游戏中的所有对象，无论属于什么类型，都放在这一个容器中。（把不同类型的对象用一个容器管理，需要用到"多态"的知识。所以，想想这个`std::list`内部的元素类型应该是什么？）你的GameWorld中可以只有这一个容器。游戏中每一刻(tick)需要让所有对象Update，我们便可以对这个`std::list`里的所有对象都调用一次`Update()`。

此外，你的GameWorld可能还需要储存除游戏对象之外的其他数据。比如，每隔一段时间就会生成一群哥布林，这个倒计时便需要`GameWorld`储存。

`GameWorld`类继承自提供的框架中的`WorldBase`类。你不应当更改`WorldBase`类，并且你需要用到`WorldBase`类中提供的一些方法。

以下三个`WorldBase`类中的方法被定义为纯虚，故你的`GameWorld`必须提供定义。同时注意，这三个方会被提供的游戏框架自动调用，你所写的代码中正常情况下不应当主动调用。

```
virtual void Init() = 0;
virtual LevelStatus Update() = 0;
virtual void CleanUp() = 0;
```

`Init()`函数即为关卡的初始化。每当一个关卡即将开始，游戏框架便会调用这个函数。在`Init()`中，你需要做好一个关卡开始的准备：初始化你的游戏世界中任何用于记录关卡的数据，把玩家放在屏幕上的正确位置，等等。细节实现见后文。

你可能会想，我们的游戏只有一关，并且需要的变量已经在构造函数里初始化过了，这里再做一遍不是重复了吗？当你输掉游戏再次尝试的时候，游戏框架的实现并不允许我们删除你的`GameWorld`再"重开"一个。于是就需要通过一次`CleanUp()`和一次`Init()`来清除上一局游戏剩余的对象，并重新开始一局新游戏。

`Update()`函数便是游戏过程中每一刻(tick)对游戏世界的更新。在关卡开始后，此函数每一刻均会被框架调用一次（频率约为1秒30次，因为游戏期望以30帧每秒(FPS)运行）。游戏世界`Update`过后的运行状态（正常运行还是输掉）将作为返回值传回游戏框架。

在GameWorld的`Update()`中，需要执行的步骤大致有这些（按重要程度排序而非实际应该执行的顺序，实际执行顺序见后文细节）：

- 让所有游戏对象(`GameObject`)均进行`Update()`。
- 检测关卡是否失败。
- 为`GameWorld`添加新的游戏对象，例如新生成的哥布林，哥布林扔出的斧头等。
- 删除`GameWorld`中应被删除的对象，例如超过屏幕的子弹、被击杀的哥布林等。

`CleanUp()`函数即为关卡结束时的清理步骤。当你的Update函数返回的状态表示当前关卡已经结束，游戏框架就会调用此函数。你需要清空当前关卡中所有游戏对象，不能出现内存泄漏。

请再次注意，上述三个函数会经过游戏框架处理，你不应当主动调用。

除此三个必须定义的纯虚函数外，你还可以为`GameWorld`自由添加新的成员变量、成员函数，或是调用基类WorldBase中提供的函数。

GameWorld类还继承了`std::enable_shared_from_this`。这可以允许你使用`shared_from_this()`来创建一个指向自己的智能指针以代替普通指针this。关于`shared_from_this()`的使用，FAQ中有更详细的讲解。

从GameObject基类开始创建每种游戏对象

你的GameWorld需要将所有游戏对象储存在同一个容器里，因此，只有让所有游戏对象都继承自同一个基类，才能利用多态(**polymorphism**)实现“在无需知道每个对象的具体类型的前提下让他们执行分化的操作”。你所需要的这个基类我们已经预先命名，叫做**GameObject**。

GameObject继承自游戏框架提供的更底层的基类**ObjectBase**。同时也继承了**std::enable_shared_from_this**，以允许使用**shared_from_this()**来创建一个指向自己的智能指针以代替普通指针**this**。你可能会想，为什么不直接把ObjectBase当作这个共同基类呢？实际上是出于设计的原因：我们希望把你需要完成的与游戏框架需要处理的事情清晰地划分开。如何将玩家和哥布林的动画显示在屏幕上不需要你去思考，而你是否选择“给你的游戏对象定义一个函数让它扣除HP”当然也不是游戏框架运行所必需的。

GameObject应当存什么，不应当存什么

你的GameObject从我们提供的基类ObjectBase中继承了5大基本属性，例如位置的x和y坐标等。你不需要再在你的GameObject中再储存一遍这些基本属性，而应当调用我们在ObjectBase中提供的函数去获得/修改他们。

除了下述的几个ObjectBase中定义的基本属性，如果你认为还有其他属性是所有游戏对象都必须具有的，就应当把它们定义在你的GameObject类中。如果一些属性是玩家有而哥布林没有的，就把它们定义在玩家中而非全都塞到GameObject中。

继承的基本属性介绍

`ObjectBase`中的基本属性包括：

imageID

- 表示这个对象对应的贴图素材编号。所有编号的定义在"`utils.hpp`"中。
- **imageID在对象生成时即需确定**
- 由于imageID与每个对象的实际类型对应，而多态的思想不应使用对象实际类型的信息，故imageID不提供访问函数(accessor)。在编写这个游戏时，“想要知道对象的imageID”或“知道它具体是什么东西”等想法都是不正确的，你应当使用具有分化实现的虚函数(virtual functions)来判断某些对象属于哪些大种类。阅读附录中的面向对象编程(OOP)小建议可能可以帮助你更快找到好的写法。

x和y

- 表示对象当前所在的坐标，以像素为单位。
- 屏幕左下角为坐标系原点(0, 0)，向右为x轴正方向，向上为y轴正方向，屏幕最右上角的坐标为(WINDOW_WIDTH – 1, WINDOW_HEIGHT – 1)。
- 具有访问函数**GetX()**与**GetY()**，以及同时更改x与y的修改函数**MoveTo(int x, int y)**。

layer

- 表示对象在屏幕上的显示层级，取值范围为[0, MAX_LAYER)。
- layer数值更低的对象将在显示时遮盖高层级数值的对象。

width和height

- 表示对象的（碰撞体）大小，类型为int。
- 具有访问函数**GetWidth()**, **GetHeight()**，和修改函数**SetWidth(int width)**, **SetHeight(int height)**。

AnimID

- 表示这个对象具有的动画。
- 具有访问函数**GetCurrentAnimation()**。
- 当对象需要改变动画时，使用**PlayAnimation(AnimID animID)**修改它，以从第一帧开始播放新的动画。

上述属性同时也是ObjectBase的构造函数需要提供的参数。因为ObjectBase不允许默认构造，所以在你的GameObject及其他子类的构造中，需要以**初始化列表(initializing list)**的方式为其中的基类ObjectBase提供这些参数。

> **注意：** "可是GameObject类也是一个抽象类，没有确定的imageID等属性的值，该怎么提供给ObjectBase呢？"GameObject类也可以同样地，要求所有继承自它的类都提供一个imageID。除了imageID，x和y等其他无法确定的部分也可以像这样，放在抽象类的构造函数里，继续向下传给具体的子类。当子类有了确定的属性值时（比如，玩家确定了它的imageID），再通过一层层调用基类构造函数逐级传回，最终提供给最底层的ObjectBase。

此外，ObjectBase**不允许拷贝/移动构造**，**不允许拷贝/移动赋值**。要直观地解释的话，管理游戏对象的权利应当只属于GameWorld，而不能让任何对象都能轻易地复制自己或其他对象。

每种对象通过分别继承GameObject表示

当你的GameObject类继承了ObjectBase之后，你便可以为每种特定的对象继续创建子类，定义它们的行为。每种游戏对象都可以不同地实现**Update()**函数。在**Update()**函数中，你的游戏对象可以移动，改变自身状态，甚至也可以"死亡"。

像超过屏幕的子弹、被击杀的哥布林、对玩家造成伤害的斧子等对象就需要"死亡"。注意，游戏中所有的对象都是由GameWorld里的容器管理的，任何对象，包括自己，都没有权利进行对象的清理。因此，一个对象"死亡"的方式是将自己标记为"已死亡"状态，或是以"HP为零"等方式判断。在所有对象进行了一次Update()后，GameWorld可以一并清理所有被这样标记为"死亡"状态的对象，即，从容器里删除。

那么，恭喜你，看到这里，你已经将最复杂的架构部分读完了。接下来的部分则是GameWorld与每一个对象分别的行为细节。每一个具体的对象都应当是`GameObject`的子类，但无需直接继承——如果你想加入多种不同的哥布林，试着写一个基类（可以叫`Goblin`），将他们的共同点包括在内；如果你想让发射的斧子只伤害玩家而不伤害自己的其他哥布林，那就在共同的GameObject基类里里加入一个虚函数(virtual function)，让玩家和哥布林对它的实现不同，从而区分开来。既可以简单地写一个叫做**IsGoblin()**的bool函数，也可以定义一种表示类型的枚举类(enum class)来表示对象的不同类型。

好的，所以这么多东西我怎么写?

可想而知，从零开始写出一个完整的游戏绝不是能一蹴而就的小工程。本游戏的大量内容需要逐步完成。我们建议你先阅读完整的题目说明，对这个游戏基本了解，之后再开工。"建议的开始流程"部分中提供了一个清晰的前几步流程，相信你完成这几步之后就会驾轻就熟。

在提供的文件之中，你不需要也不应当修改`src/Framework`路径下的文件。对于`utils.hpp`，在图片素材加载错误时你可能需要修改其中`ASSET_DIR`字符串，并且你可以在其中定义新的项目来添加属于你自己的内容（详见FAQ：怎么添加新内容）。

建议的开始流程

0. 先成功编译一次

在你开始写第一行代码之前，试着先成功编译并打开我们提供的代码框架。你应当能看到游戏的标题画面。在按回车开始游戏之后，你会看到只有背景的游戏画面。此时你看到的就是正在运行中的游戏，只是游戏里面什么也没有。

1. 背景

（为了方便大家理解，Background类的声明和实现已经为大家写好，作为example）

你要做的第一件事便是显示出“游戏画面”，也就是背景。你需要为背景创建你的第一个类，它将继承 `GameObject` 。在这时，你大可不必立刻规划好“什么成员属于 `GameObject` 基类，什么成员应当单独存储”。在你实现更多功能的过程中，这个问题会渐渐变得清晰。

创建完背景类后，你需要在 `GameWorld` 中生成一个背景。在 `Init()` 中生成一个临时变量当然是不行的，即使这个临时变量是给它动态分配内存的 `shared_ptr` ，也会因为离开 `Init()` 作用域（scope）时被销毁，然后分配的动态内存也会释放。
你应当把这个背景存储在 `GameWorld` 里。根据文档，所有 `GameObject` 都应存储在一个 `list` 里。考虑一下这个 `list` 的类型会是什么？

一切顺利的话，现在运行游戏你就能看到背景图片。如果你遇到编译错误或链接（linking）错误，看看在用到背景类的时候是否正确 `#include` 了它的文件？参考 FAQ 中的 Linking error 部分。

2. 放置玩家

接下来，你可以真正开始“写游戏”了。如果你感觉无从下手，就从先创建一个玩家开始，相信你在实现完之后就能明白游戏中任何部分的开发流程。

你需要先在 `Player.hpp` 里面**声明** `Player` 类，然后声明它的private成员变量（或许你这会还不知道需要哪些变量），但是没关系，最开始要做的工作是为 `Player` 声明public函数，比如构造函数，`void Update()` 等虚函数的重载。

然后再在 `Player.cpp` 里面完成对 `Player.hpp` 的成员函数的**实现**。

> 尽量把函数的实现放在 `.cpp` 文件里，这是因为这些函数的实现可能会牵扯到非常多的 `*.hpp` 文件，比如 `gameworld.hpp` ，而 `gameworld` 的实现又依赖于 `Player.hpp` ，因此这样会产生循环include，正确的做法是在 `*.hpp` 中只include有继承关系的头文件，在 `*.cpp` 中则可以include各种需要的头文件。因为被编译的只有 `*.cpp` 文件，生成的二进制可执行文件 `*.o` 之间是彼此链接的关系而不是include。

在正确完成 `Player` 的构造函数之后，玩家其实就可以显示在屏幕上面了，只需要在 `Gameworld` 的容器中像添加背景一样添加Player即可

如果你需要让他动起来（实则背景向后跑动），则需要正确完成 `Gameworld` 中对所有容器对象的遍历 `Update()` ，你就能看到你的玩家像是跑起来了。

3. 让玩家开枪

如果你成功地完成了上面的步骤，你应该对“ `GameObject` 需要存储什么成员”有了更多的想法。试试继续让这个游戏动起来吧：产生子弹。每次按下J或者鼠标左键时候即让 `Gameworld` 容器中加入一个子弹

4. 自由发挥

剩下的部分里，我相信你们已经不会遇到太多问题了，开始自由发挥吧！

下面是对游戏中每个组成部分的介绍。其中以符号开头的列表项是无关顺序的，数字或字母开头的列表项表示需要按顺序执行。

下面的实现，是样例的参考实现，你可以自定义你自己的数值，但是必须得合理

在看到下面的数值之前先叠个三级甲，数值的设置和每个人的操作的匹配情况都存在差异，如果觉得参考数值的难度过高可以手动修改数值降低难度。

> “可见，不管是哪个游戏，数值策划总是会被玩家抓出来狠狠骂的。”—— CS100 2022SpringHead、2023Spring、2025Spring TA，飞机大战和PvZ游戏框架的缔造者，现任米哈游员工，刘钰琦

GameWorld

GameWorld::Init()

你的`GameWorld::Init()`函数应当：

- 初始化任何用于记录关卡数据的成员变量。例如，记录游戏得分，游戏从开始运行的总帧数。
- 为需要显示的关卡数据创建文本显示。需要使用TextBase类，使用方法与你创建的所有GameObject均类似，详细用法参见FAQ。
    - 得分数推荐显示在`(WINDOW_WIDTH - 160, 8)`。
- 创建背景（已经帮你创建作为示例）。
- **创建玩家（x=200，y=120）**

GameWorld::Update()

你的`GameWorld::Update()`函数应当：

1. **为游戏生成新的哥布林，植物，野猪，飞鸟。** 你可以自定义生成的时间与位置。（推荐植物的高度为100，哥布林与野猪的高度为120）。
2. 遍历所有游戏对象(GameObject)，并依次调用它们的Update()函数。
3. 检测碰撞。注意不要让同一碰撞被触发多次。可能发生的碰撞有：
    - 子弹击中哥布林。
    - 斧头打到玩家。
    - 植物碰到玩家。
    - 野猪碰到玩家。
    - 飞鸟碰到玩家。
    - 哥布林碰到玩家。
4. 再次遍历所有游戏对象，将需要删除的对象从你的存储容器中移除。
    - 需要删除的对象应被标记为"死亡"或"HP为零"等状态，GameWorld才能找到。
5. 判断是否失败。若玩家死亡，则返回`LevelStatus::LOSING`。
    - 在失败的显示画面上，我们预留了一个显示你分数的空位。你可以在此时创建一个文本来显示出你的游戏结果。建议以白色(RGB = 1, 1, 1)创建在(360, 50)位置。关于文本颜色的细节请参考FAQ。
6. 更新你在游戏中创建的文本显示(TextBase)的内容，以正确显示需要的信息，例如分数、HP等。

7. 返回`LevelStatus::ONGOING`，表示当前关卡在正常运行。

GameWorld::CleanUp()

你的`GameWorld::CleanUp()`函数必须清空你所使用的容器。若你保存了其他对象，也需要将它们正确清除。

背景(Background)

当被创建时

- 背景的贴图编号为`IMGID_BACKGROUND`。
- 背景的位置为`(x = WINDOW_WIDTH , y = WINDOW_HEIGHT / 2)`。
- 背景的所在层级为`LAYER_BACKGROUND`。
- 背景的宽度为`2*WINDOW_WIDTH`，高度为`WINDOW_HEIGHT`。（特殊的宽度和位置设计也是背景能不断向后跑的奥秘）
- 背景不具有动画，即，动画编号为`ANIMID_NO_ANIMATION`。

当Update()时

背景每帧向后移动3个像素，若背景的横坐标小于等于0，则向右移动`WINDOW_WIDTH`

玩家(Player)

当被创建时

- 玩家的贴图编号为`ImageID::PLAYER`。
- 玩家的起始位置固定，`(x=200，y=120)`
- 玩家的所在层级为`LayerID::PLAYER`。
- 玩家的宽度为20，高度为48。
- 玩家初始具有`AnimID::IDLE`动画。

当Update()时

1. **玩家需要首先检查自己是否已经死亡。** 若已经死亡，它将等待GameWorld清理。如果有显示玩家HP的文字则需要擦除，然后立刻返回，无视下述步骤。
2. 若按键J或鼠标左键按下，玩家则会开枪，子弹会从玩家坐标右边30个像素射出（**子弹是从枪里面射出来的**，而不是像某个瓦开头的游戏一样是从脑袋射出来的）
    - 开枪存在冷却，冷却为10帧
3. 若按键K或空格被按下，玩家则会跳起
    - 跳跃是一个竖直上抛运动，存在向上初速度和重力加速度
    - 初速度的大小是23像素/帧，重力加速度的大小是2像素/帧^2
    - 在跳跃状态下需要切换动画为`AnimID::JUMP`
    - 若玩家正在跳跃，按下跳跃键将触发二段跳，速度变为跳跃初速度23像素/帧，不会触发三段跳
    - 不限制跳跃过程中是否能开枪
    - 落地后需要切换回`AnimID::IDLE`

思考一下： 要做到开枪（创建对象），玩家需要知道自己所在的`GameWorld`，也就意味着需要储存这样一个成员。这个成员的类型应当是什么？GameWorld创建子弹时，怎样把"所在的GameWorld"这一信息告诉它？除了玩家，其他的类也可能需要知道所在的世界，那么这个信息应当存在哪儿？

子弹(Bullet)

当被创建时

- 子弹的贴图编号为`ImageID::BULLET`。
- 子弹的位置由创建它的代码决定。
- 子弹的所在层级为`LAYER_PROJECTILES`。
- 子弹的宽度为10，高度为10。
- 子弹不具有动画。

当Update()时

1. 子弹需要首先检查自己是否已经死亡。若已经死亡，它将等待GameWorld清理。它的Update()应当立刻返回，无视下述步骤。
2. 子弹将向右移动10像素。
3. 如果子弹飞出了屏幕右边界(x >= WINDOW_WIDTH)，子弹需要死亡。

当与哥布林碰撞时

子弹将会对与它碰撞的哥布林造成1点伤害，然后死亡。

斧子(Axe)

当被创建时

- 斧子的贴图编号为`ImageID::AXE`。
- 斧子的位置由创建它的代码决定。
- 斧子的所在层级为`LAYER_PROJECTILES`。
- 斧子的宽度为25，高度为25。
- 斧子不具有动画`AnimID::NO_ANIMATION`。

当Update()时

1. 斧子需要首先检查自己是否已经死亡。若已经死亡，它将等待GameWorld清理。它的Update()应当立刻返回，无视下述步骤。
2. 斧子将向左移动10像素。
3. 如果斧子飞出了屏幕左边界(x <= 0)，斧子需要死亡。

当与玩家碰撞时

斧子将会对与它碰撞的玩家造成1点伤害，然后死亡。

植物(Plant)

当被创建时

- 植物的贴图编号为`ImageID::PLANT`。
- 植物的位置由创建它的代码决定。
- 植物的所在层级为`LAYER_PROJECTILES`。
- 植物的宽度为25，高度为25。
- 植物不具有动画`AnimID::NO_ANIMATION`。

当Update()时

1. 植物需要首先检查自己是否已经死亡。若已经死亡，它将等待GameWorld清理。它的Update()应当立刻返回，无视下述步骤。
2. 植物将向左移动3像素。
3. 如果植物飞出了屏幕左边界(x <= 0)，植物需要死亡。

当与玩家碰撞时

植物将会对与它碰撞的玩家造成1点伤害，然后死亡。（样例的植物没法被子弹打中）

飞鸟(Bird)

当被创建时

- 飞鸟的贴图编号为`ImageID::BIRD`。
- 飞鸟的位置由创建它的代码决定。（推荐215左右，不要低于人物）
- 飞鸟的所在层级为`LAYER_ZOMBIES`。
- 飞鸟的宽度为25，高度为25。
- 飞鸟初始具有`AnimID::IDLE`动画。
- 飞鸟具有1点HP。

当Update()时：

1. 飞鸟需要首先检查自己是否已经死亡。若已经死亡，它将等待GameWorld清理。它的Update()应当立刻返回，无视下述步骤。
2. 它将向左移动14像素。

当发生碰撞时

- 如果与子弹发生碰撞，飞鸟将会受到来自子弹伤害数值的伤害，并让那颗子弹死亡。
- 如果与玩家发生碰撞，飞鸟将会立马死亡，并对玩家造成1点伤害。

野猪(Pig)

当被创建时

- 野猪的贴图编号为`ImageID::PIG`。
- 野猪的位置由创建它的代码决定。（建议100，避免野猪在天上飞或沉入地底）
- 野猪的所在层级为`LAYER_ZOMBIES`。
- 野猪的宽度为20，高度为48。
- 野猪初始具有`AnimID::IDLE`动画。
- 野猪具有5点HP。
- 野猪的初速度为3。
- 野猪的将在15ticks后发起冲刺。

当Update()时：

1. 野猪需要首先检查自己是否已经死亡。若已经死亡，它将等待GameWorld清理。它的Update()应当立刻返回，无视下述步骤。
2. 检查是否为冲刺状态，如果不在冲刺阶段，它将向左移动3像素，如果在冲刺阶段，你需要先播放动画`AnimID::RUSH`，它的速度将被改为7，并将向左移动7像素。

当发生碰撞时

- 如果与子弹发生碰撞，野猪将会受到来自子弹伤害数值的伤害，并让那颗子弹死亡。
- 如果与玩家发生碰撞，野猪将会立马死亡，并对玩家造成1点伤害。

哥布林(Goblin)

当被创建时

- 哥布林的贴图编号为`ImageID::GOBLIN`。
- 哥布林的位置由创建它的代码决定。(建议高度为125-130)
- 哥布林的所在层级为`LAYER_ZOMBIES`。
- 哥布林的宽度为20，高度为48。
- 哥布林初始具有`AnimID::IDLE`动画。
- 哥布林具有5点HP。

当Update()时：

1. 哥布林需要首先检查自己是否已经死亡。若已经死亡，它将等待GameWorld清理。它的Update()应当立刻返回，无视下述步骤。
2. 它将向左移动3像素。
3. 每100ticks丢出一个斧子，特殊的是，丢出斧子需要在动画播放20ticks后实现：你需要先播放动画`AnimID::THROW`，等20ticks后在哥布林的坐标处创建一个斧子。

当发生碰撞时

- 如果与子弹发生碰撞，哥布林将会受到来自子弹伤害数值的伤害，并让那颗子弹死亡。
- （可选）如果与子弹发生碰撞，哥布林会播放受伤动画`AnimID::HURT`，时长为5帧，可以选择加入打断施法或减速等特性。
- 如果与玩家发生碰撞，哥布林将会立马死亡，并对玩家造成1点伤害。

碰撞处理

坐标系统说明

本游戏使用中心点坐标系：

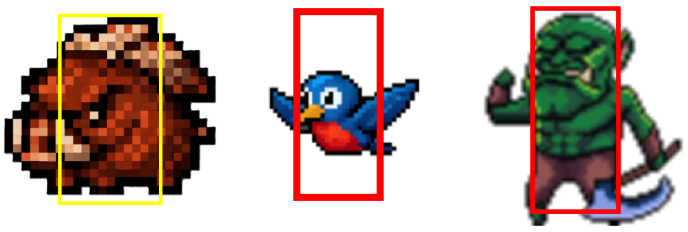
- 每个游戏对象的`(x, y)`坐标表示该对象的**中心点位置**，而不是左上角！
- 屏幕左下角为坐标系原点`(0, 0)`，向右为 x 轴正方向，向上为 y 轴正方向。
- 屏幕最右上角的坐标为`(WINDOW_WIDTH - 1, WINDOW_HEIGHT - 1)`。

碰撞体定义

每个对象的碰撞体均为**以`(x, y)`为中心**，横向宽度为`width`，纵向高度为`height`的长方形。

也就是说，对象的实际边界位置为：

```
左边界 = x - width / 2
右边界 = x + width / 2
下边界 = y - height / 2
上边界 = y + height / 2
```



碰撞检测算法

若两对象的碰撞体长方形发生重叠，则认为两对象发生碰撞。

碰撞检测的直观理解

实现建议

为了避免代码重复，强烈建议在 `GameObject` 基类中实现一个碰撞检测辅助函数：

```
// 在 GameObject 基类中
protected:
    bool IsCollidingWith(std::shared_ptr<GameObject> other) const;
```

然后在各个子类中使用：

```
void Goblin::OnCollision(std::shared_ptr<GameObject> other) {
    if (other->GetType() == GameObject::Type::ProjectilePlayer) {
        if (IsCollidingWith(other)) {

        }
    }
}
```

附录

首次运行

当你按下面的方法生成你的项目后，如果你能够运行，并且可以看到标题界面（按回车进入游戏后是黑屏），你就成功完成首次运行的设置了。在你开始逐步开工之后，每实现一个部分，记得运行一下游戏，看看新写的部分有没有为游戏带来bug。不要等到写完所有部分才开始运行游戏！

环境配置

我们默认使用WSL或者macOS来完成本次项目，如果你坚持使用windows，请自行想办法编译安装freeglut库

以下是linux上需要安装的必要依赖，macOS则无需安装下面的两个库

安装OpenGL库请输入如下指令

```
sudo apt-get install libgl-dev libglu1-mesa-dev libx11-dev libxrandr-dev libxi-dev
```

安装freeglut库请输入如下指令

```
sudo apt install freeglut3-dev
```

安装完成后就可以在你的assignment目录下输入指令进行编译了（如果没有`make`，就`sudo apt install make`就可以了）,-j表示用多核并行编译

```
make -j
```

最后你就可以运行游戏了

```
make run
```

或者你可以直接编译完直接运行(如果不加-j也可以在没有编译的时候先编译，但是-j可以多核并行编译)

```
make run -j
```

面向对象编程(OOP)小建议

在你设计你要写的对象的结构时，试着考虑一下接下来的几条小建议。这些建议不仅会帮助你写出更规范的面向对象程序，也会降低你在本次作业中出错的概率。“尽可能”遵从这些建议即可，不必过分拘泥于教条。在细枝末节、无关紧要的设计上纠结只会浪费精力。切记，纸上得来终觉浅，绝知此事要躬行。

1. 不要通过类型转换确认对象类型

不要通过任何形式的类型转换来确认一个对象的类型，而应当添加成员函数来检测是否有某类型对象的通用性质或行为。同时，注意也不要为每种对象单独定义一个"IsSomeClass()"的方法，而应当用一种方法来检查多个对象的通用性质。

❌ 不要如此做：

```
for (auto& item : familyMart) {
    if (dynamic_cast<CocaCola*>(item) != nullptr ||
        dynamic_cast<Pepsi*>(item) != nullptr ||
        dynamic_cast<Tea*>(item) != nullptr ||
        dynamic_cast<Milk*>(item) != nullptr) {
        me.Buy(item);
    }
}
```

❌ 也不要如此做：

```
for (auto& item : familyMart) {
    if (IsCocaCola(item) || IsPepsi(item) || IsTea(item) || IsMilk(item)) {
        me.Buy(item);
    }
}
```

✅ 而应当如此做：

```
for (auto& item : familyMart) {
    if (IsBeverage(item)) {
        me.Buy(item);
    }
}
```

2. 封装成员变量

不要将成员变量声明为public，尽可能少地声明为protected，而尽量都声明为private。

即使public的成员变量可以节省时间，也请尽量使用private成员。对于private的成员，可以选择是否提供public的访问函数与修改函数。

3. 合理设置成员方法的访问权限

如果某个成员方法只供自己或子类调用而不会被外部调用，那么可以将它声明为**protected**或**private**。

4. 公共属性放在基类

如果两个相关的子类都需要定义一个用法相同的成员变量，不要分别定义，而是将定义放在它们的公共基类里，并提供public的访问函数与修改函数。

5. 使用模板方法模式

如果两个相关的子类都需要实现某个方法，而在此方法中既有相同的部分又有不同的部分，不要分别实现而将相同的部分重复一遍，而应当定义另一个virtual的辅助函数来将不同的部分区别开来。

❌ 不要如此做：

```
class SomeStudent : public Student {
public:
    virtual void DoSomething() {
        Eat();  // 相同的部分
        TakeClasses();
        GoToFamilyMart();
        Sleep();  // 相同的部分
    }
};

class OtherStudent : public Student {
public:
    virtual void DoSomething() {
        Eat();  // 相同的部分
        PlayGames();
        DoHomework();
        Sleep();  // 相同的部分
    }
};
```

✅ 而应当如此做：

```
class Student {
public:
    virtual void DoSomething() {
        Eat();  // 相同的部分（在基类里）
        DoDifferentStuff(); // 不同的部分，虚函数不同实现
        Sleep();  // 相同的部分（在基类里）
    }
private:
    virtual void DoDifferentStuff() = 0;
};

class SomeStudent : public Student {
private:
    void DoDifferentStuff() override {
        TakeClasses();
        GoToFamilyMart();
    }
```

```
};

class OtherStudent : public Student {
private:
    void DoDifferentStuff() override {
        PlayGames();
        DoHomework();
    }
};
```

6. 控制对象访问权限

在你的GameWorld中，不要将装有对象的List或List的iterator作为返回值或是public的部分。只有GameWorld才能访问和储存这些对象，而不能交与其他GameObject。你应当让GameObject通知GameWorld，来处理与自己和其他对象相关的请求。

❌ 不要如此做：

```
class FamilyMart {
public:
    std::list<std::shared_ptr<Item>>& GetItems() {return m_allItems;}
    // Bad! 不要直接把自己private的容器交出去!
};

class Me : public Student {
public:
    void GoToFamilyMart() {
        for (auto& item : GetFamilyMart()->GetItems()) {
            // Student类竟然可以管理FamilyMart的所有物品？
            if (IsMyFavorite(item)) {
                Buy(item);
            }
        }
    }
};
```

✅ 而应当如此做：

```
class FamilyMart {
public:
    void BuyFavoriteItem(std::shared_ptr<Student> student) {
        for (const auto& item : m_allItems) {
            if (student->IsMyFavorite(item)) {
                student->Buy(item);
            }
        }
    }
};

class Me : public Student {
public:
    void GoToFamilyMart() {
        GetFamilyMart()->BuyFavoriteItem(shared_from_this());
        // Student 应当把自己交给FamilyMart去处理，而不是反过来。
    }
```

```
};
```

FAQ

我的游戏运行起来有 bug 但看不出在哪怎么办?

你之前的所有 debug 经验，在这个项目里都可以使用。你可以通过 `std::cout` 向控制台打印一些输出，可以将一些没有贴图的物体换上贴图以便显示，当然也可以使用 debugger ，在指定代码处打上断点或逐行运行。

GameWorld.hpp 中的 `std::enable_shared_from_this` 是什么?

省流：这是对 `this` 指针的"智能指针"版替代。当你需要在 `GameWorld` 里获得一个指向自己的 `std::shared_ptr` 的时候，可以换成调用 `shared_from_this()` 获得一个智能指针，类型为 `pGameWorld` ，也就是 `std::shared_ptr<GameWorld>` 的别名。

一般地，通过让一个类 `X` 继承自 `std::enable_shared_from_this<X>` ，我们将能够在 `X` 的内部调用 `shared_from_this()` 来安全地获得一个管理 `*this` 的 `std::shared_ptr<X>` 。注意，调用 `shared_from_this()` 意味着**现在存在某个 `std::shared_ptr` 正在管理 `*this` ，且我们想要获得一个与它共享 `*this` 的 `std::shared_ptr`** 。什么时候会出现这种情况？考虑下面的代码。

```
struct X;

void do_something(std::shared_ptr<X> ptr);

std::vector<std::shared_ptr<X>> gObjects;

struct X {
  void some_action() {
    do_something(std::shared_ptr<X>(this)); // 这将导致灾难!
    // shared_ptr<X>(this) 创建了一个新的管理 *this 的 shared_ptr ，
    // 但其实现在 *this 正在被某个 shared_ptr （也就是 gObjects[0] ）管理着，
    // 而且这两个 shared_ptr 还不知道对方的存在！它们自己都以为自己独占了这个对象。
  }
};

int main() {
  gObjects.push_back(std::make_shared<X>()); // 创建了一个 shared_ptr<X>
  gObjects.front()->some_action();
}
```

要解决这个问题，需要这样做：

```
struct X;

void do_something(std::shared_ptr<X> ptr);

std::vector<std::shared_ptr<X>> gObjects;

struct X : public std::enable_shared_from_this<X> {
  void some_action() {
    do_something(shared_from_this()); // 没问题。这将正确地创建一个与 gObjects[0] 共享 *this 的
shared_ptr。
  }
```

```
};

int main() {
  gObjects.push_back(std::make_shared<X>()); // 没问题。 std::make_shared<X> 会认识到： X 是一个继承自 std::enable_shared_from_this 的类型！它将会做一些特殊处理。
  gObjects.front()->some_action();
}
```

注意：

1. **并非**需要将所有的 `this` 都替换为 `shared_from_this()` 。不要胡乱使用 `shared_from_this()` 。
2. 调用 `shared_from_this()` 的前提是**现在** `*this` **正被某个** `shared_ptr` **管理着**。如果不是这样，则 `shared_from_this()` 会抛出 `std::bad_weak_ptr` 异常。

我删除 `std::list` 内元素的时候，为什么程序会崩溃?

省流：对于 `std::list` ，一边遍历一边删除一些指定元素的正确方法是使用迭代器作为循环变量，调用 `erase()` ，而不是使用 range-for 遍历每个元素。**请你耐心地点开这个链接读一下** `erase` **的用法，特别是注意它的返回值。你会需要的。**

Range-for 本质上也是在使用迭代器遍历你的 `list` 。当你删除 `list` 中元素时，会**无效化** (invalidate) 指向被删除元素的迭代器，也就是 range-for 在背后使用的那个，于是引发错误。这其实很好理解：所谓"链表"就是在每个元素的身上记录它前一个和后一个元素的地址。如果你把 `iter` 指向的元素删除了， `++iter` 还怎么执行呢？它怎么知道自己指向谁、下一个又是谁呢？

当然，你也可以考虑使用 `std::list<T>::remove_if` 。我个人更推荐这个。

使用了未定义类型 `GameWorld` / 循环依赖与 forward declaration

省流： `#include` 就是简单的文本复制粘贴。两个头文件互相包含就会产生错误。一些情况下，将声明和定义分开在 `.hpp` 和 `.cpp` 文件里可以消除一些循环依赖问题。如果你忘记了什么是声明、什么是定义，请去复习 Lecture 16 。

`#include` 就是简单的文本复制粘贴。假如 `A.hpp` 和 `B.hpp` 中分别定义了类 `A` 和类 `B` 。当 `A.hpp` 与 `B.hpp` 互相 `#include` 时，若 `A` 类与 `B` 类彼此互相依赖，都需要对方提前给出自己类的完整定义，一定会有一个类因为"出现在前面"而不满足依赖。

实际上，这种情况一般不太会出现：如果 `A` 类有一个 `B` 类型的成员， `B` 类肯定不能有一个 `A` 类型的成员，否则它们所占的空间将是无穷大。常见的情况是其中一方需要另一方的指针 `*` 或引用 `&` （也包括智能指针和容器等）。只创建类 `A` 的指针或引用是不需要 `A` 的完整定义的，仅需要在事先有一个对 `A` 的声明，也就是只需要 `class A;` 这样一条语句，**而不需要** `#include "A.hpp"` **来拿到** `A` **的定义**。这样就解决了两个文件互相依赖的问题：其中一方只用到指针或引用时，可以将 `#include` 改为声明。

然而，像 `class A;` 这样的声明只告诉了编译器 `A` 是一个类，编译器并不知道关于 `A` 类的任何信息。在类 `A` 的定义（即，包括它所有成员的**声明**）出现之前，任何依赖于其定义的行为都无法编译，比方说调用 `A` 的某个成员函数、创建 `A` 类型的对象等等。要想找到定义，就必须 `#include "A.hpp"` ，而这又回到了循环依赖的问题。

所以，头文件之间是不能随便互相 `#include` 的。但是，通常你可以在一个 `.cpp` 文件里随意 `#include` 你需要的东西，因为 `.cpp` 文件一般不会被别人 `#include` ，不会有循环依赖问题。所以，前文提到的依赖于某个类的定义的行为，一般都应该放在 `.cpp` 文件里。

我想把我的游戏打包出来给别人玩，可以吗?

当然可以，但你的游戏如果要在win11上运行，那么就需要去在win11上编译，你需要从github上clone下来并编译安装一个叫**freeglut**的图形库（而linux直接一行apt install就搞定了，甚至mac系统直接有这个库），而他用的编译工具是cmake，这可能比较复杂，建议私聊TA。

要注意的是，你定义的 `ASSET_DIR` 如果是相对路径的话，在你双击 `.exe` 运行游戏时，它应该从到 `.exe` 可执行文件所在位置到 Assets 目录的相对路径。

如何添加文字?

在 `Framework` 目录下有 `TextBase.hpp` 文件。其构造函数中，必须提供的参数是 `x` 和 `y` 。其初始文字内容默认为空字符串、颜色默认为 `0, 0, 0`（黑色），居中模式默认为 `true` 。

你可以通过 `TextBase.hpp` 中提供的方法来修改它的位置、内容、颜色。

当你想要创建一个文字时，我们建议你以对待 `GameObject` 类似的方式创建它，即使用指针去动态地分配。与 `GameObject` 不同的是，你不必将所有文字都存放在 `GameWorld` 里的某个容器中，因为游戏对象每帧都需要 `Update()` 做出不同的行为，而文字不需要。通过将文字与一些 `GameObject` 绑定，你还可以做到比如显示玩家的 HP 或子弹冷却时间等功能。

我想做一份自己的“哥布林首领”的图片资源显示进游戏，应该怎么做?

我们欢迎你在有余力的情况下这样做，因为视觉效果的不同是游戏对玩家的很重要的反馈。

想要获得新的图片资源，对游戏中现有的图片素材进行 PS /调色是最简单的方法。将新的资源放进 assets 文件夹后，你还需要：

- 在 `utils.hpp` 里新建一个属于你的 `IMGID` ，如果你加入了新的动画，也可以创建新的 `ANIMID` 。
- 在 `Framework/SpriteManager.cpp` 里，加载你的新素材。 15-50 行中的代码用来完成这些，并且 hard-code 录入了素材的大小和帧数信息。你可以复制一行，参考它对应的图片素材与它的写法，填入你的信息。每个参数的意义是：

```
{ EncodeAnim(/* 你的 IMGID */, /* 你的 ANIMID */), SpriteInfo{/* 文件名 */,
 /* 图片总宽 */, /* 图片总高 */, /* 每帧宽 */, /* 每帧高 */,
 /* 每行帧数，默认 1 */, /* 动画长度，默认 1 */} }
```

- 【注意】你加入的上面这行中，至少有两处需要更改，分别是 `EncodeAnim` 的 `IMGID` 和 `SpriteInfo` 的文件名。

协作与代码风格（Git + clang-format）

本作业建议两位同学使用 Git 协作开发。下面是一套简单且实用的分支策略与工作流，目标是：**主分支永远可运行**、开发分支便于集成、个人分支减少互相干扰。

分支建议：main / develop / 个人分支

- `main`：稳定版（**随时可运行、可演示、可交付**）。只从 `develop` 合并“确认稳定”的版本进来。
- `develop`：开发集成分支（功能基本完成、基本测试能过，但可能仍有隐藏 bug）。
- `feature/<name>` 或 `user/<name>`：个人开发分支（两位同学各自的日常工作分支）。

- 平时写代码尽量只在自己的分支上提交。
- 功能完成后再合并到 `develop`，避免两个人直接在 `develop` 上互相打架。

推荐工作流（命令示例）

1. 从 `develop` 拉出自己的功能分支：

```
git checkout develop
git pull
git checkout -b feature/my-feature
```

2. 在自己的分支上开发、提交（小步提交更容易合并与回滚）：

```
git add -A
git commit -m "Implement xxx"
```

3. 功能写完后，先把 `develop` 的最新改动同步到自己的分支（减少合并冲突）：

```
git checkout develop
git pull
git checkout feature/my-feature
git merge develop
# 解决冲突 -> git add -> git commit
```

4. 再把自己的分支合并回 `develop`（建议由一人负责集成，或发起 PR）：

```
git checkout develop
git merge feature/my-feature
git push
```

5. 当 `develop` 累积到一个“稳定可演示”的节点，再合并到 `main`：

```
git checkout main
git merge develop
git push
```

> 关键原则：
>
> - `main` 只放“稳定版”；
> - `develop` 用于“集成版”；
> - 两位同学尽量不要直接在 `develop` 上写代码。

---

clang-format：统一格式，减少合并冲突

多人协作时，**格式化风格不统一**会导致 PR/merge 出现大量无意义的 diff，且极易触发冲突。建议使用 `clang-format` 统一格式。

安装（Ubuntu/WSL）

```
sudo apt install clang-format
```

VSCode 配置建议

1. 安装 VSCode 插件：`clang-format`（或同类支持 clang-format 的格式化插件）。
2. 把默认格式化器切换为 clang-format（避免每个人用不同 formatter）。
3. 养成保存即格式化/提交前格式化的习惯（团队一致即可）。

这样 Git 合并时更大概率只关注“真实逻辑改动”，而不是处理一堆格式差异带来的冲突。

本仓库的 `.clang-format` 配置说明（简要）

仓库根目录下的 `.clang-format` 会被 clang-format 自动读取，用于统一 C++ 代码风格。当前配置大意如下：

- `Language: Cpp`：该配置用于 C++。
- `BasedOnStyle: LLVM`：以 LLVM 风格作为基础模板（很多细节沿用 LLVM）。
- `IndentWidth: 4`：缩进宽度 4 空格（团队统一，减少 diff）。
- `AccessModifierOffset: -4`：`public/private/protected` 相对类体缩进向左“顶出来”4 格（更醒目）。
- `ColumnLimit: 100`：单行建议最长 100 字符，超过会倾向于自动换行（减少横向滚动与超长行）。

建议两位同学都使用同一个 formatter（clang-format）并保持同一个配置文件，这样 Git 合并时更少出现“只改了缩进/换行”的冲突。

Grading

DDL 与线上 check 说明

- **阶段性 DDL：1.3 23:59 前提交**
  - 你可以看到自己的分数；如果没有满分可以继续修改再提交。
  - 如果在该阶段性 DDL 前拿到**满分**，可获得**额外 5%** 的“开荒者行动”奖励。
- **最终 DDL：1.11 23:59**
- **check 形式：线上 check**
  - 你不需要准备任何额外材料或展示。
  - 助教会直接查看你提交的文件；**你的代码是分数的唯一判据。**

基础功能和扩展功能中的各项不再设部分分，**如果有 bug 则视为未实现**，所以请务必测试你实现的功能。

功能实现 (100%)

以下的功能为加分项，功能应该正确实现没有 bug ，并且拥有合适的贴图与动画显示。

1. (5%) 背景能动起来，而且看起来和样例大体一致
2. (5%) 玩家能原地跑起来
3. (5%) 玩家能正确的跳起来，落地位置和原位置一致
4. (5%) 玩家能正确进行二段跳
5. (5%) 哥布林能正确产生，并血量归零后消失
6. (5%) 哥布林会对玩家近身造成伤害

7. (5%) 玩家可以开枪，子弹能正确射出
8. (5%) 射出的子弹将会对第一个碰到的怪物造成伤害，碰撞后子弹将消失。
9. (5%) 哥布林能扔出斧子，哥布林扔斧子动画正确
10. (5%) 哥布林的斧子能正确造成伤害，碰撞后斧子会消失
11. (5%) 玩家起跳的数值足够规避地面怪物的攻击
12. (5%) 植物能正确产生，并血量归零后消失
13. (5%) 植物会对玩家近身造成伤害
14. (5%) 野猪能正确产生，并血量归零后消失
15. (5%) 野猪能进行冲刺，野猪经过前摇后冲刺动画正确
16. (5%) 野猪会对玩家近身造成伤害
17. (5%) 飞鸟能正确产生，并血量归零后消失
18. (5%) 飞鸟会对玩家近身造成伤害
19. (5%) 游戏难度将随时间进行不断提高
20. (5%) 玩家血量与伤害数值设置合理，玩家血量归零后结束游戏

代码规范（20%）

此部分为扣分项，扣满 20% 为止。

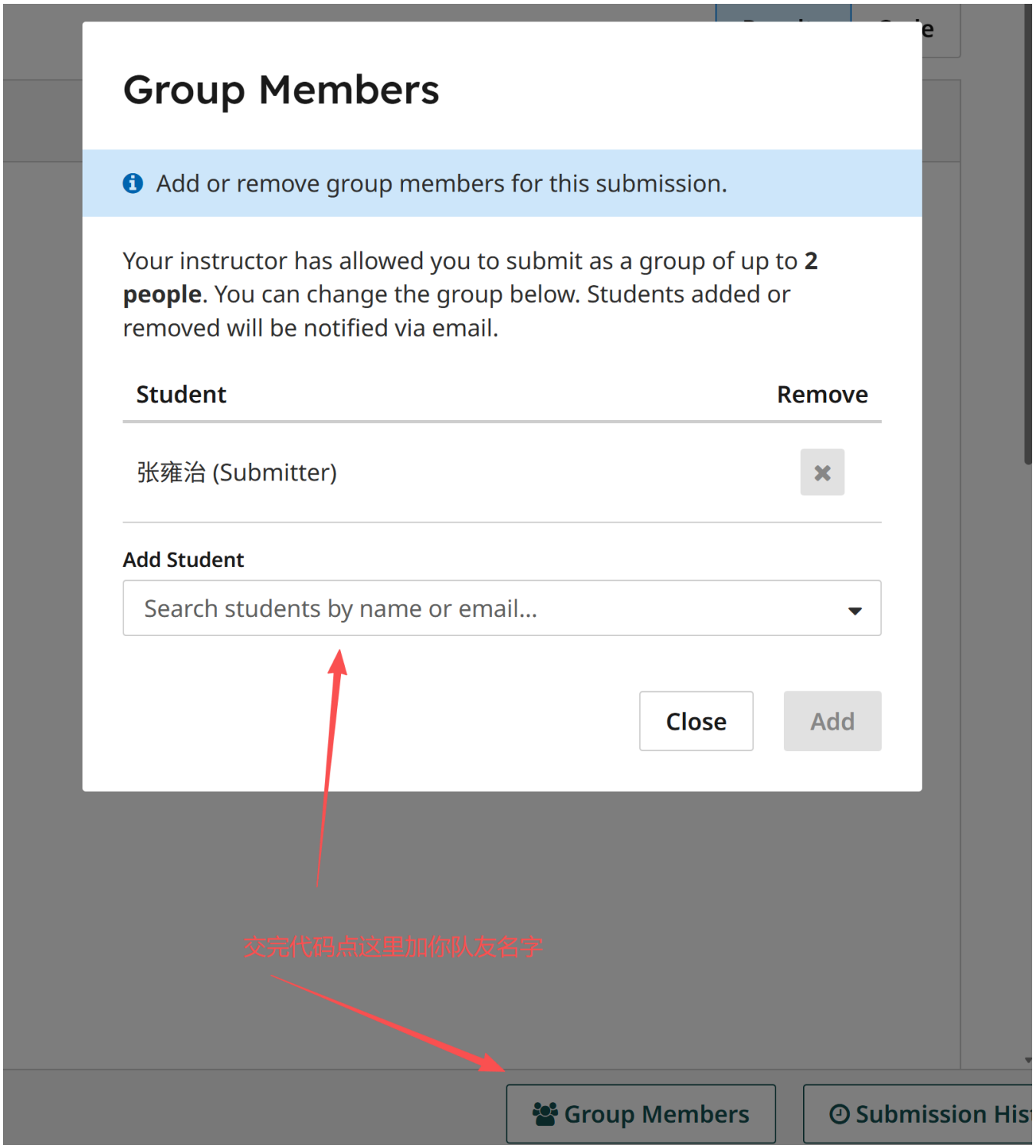
1. **(20%)** 编译没有来自于你自己的代码的 warning 。
2. (5%) 代码中不应手动管理内存，而是应该正确使用智能指针和标准库容器。
    - “正确使用智能指针”还包括**正确使用** `std::enable_shared_from_this` **相关设施**，例如，应该只在必要时调用 `shared_from_this()` ，而不是粗暴地将所有 `this` 都替换为 `shared_from_this()` 。
3. (5%) 将代码按类别分在不同的文件里。按类别把文件分在不同的子目录里。
4. (5%) 对于相似类的类似功能，不应复制代码，而是应该在基类里统一实现。
5. (5%) 正确使用枚举 (`enum`) 类型，并且应该使用限定作用域的枚举类型（即 `enum class` ），除非有特殊原因（请解释）。不应使用神秘数字 (magic numbers) 或字符串来完成本该由枚举类型完成的功能。
6. (5%) 正确使用 `utils.hpp` 中定义的 `const` 变量，而一些类内常量比如重力加速度，则可以在类内用`static const int GRAVITY = 2;`的static成员变量，而不是使用神秘数字。
7. (5%) 通常情况下，所有数据成员几乎都属于“实现细节”，应当设置为 private 。注意：
    - 一些需要被子类访问的数据成员应该设置为 protected 。但是你不能滥用 protected ，因为 protected 成员的封装性实际上和 public 是一样弱的，详见《Effective C++》条款 22 。
    - 的确存在一些足够简单的类，比如 `std::pair` ，它的数据成员（ `first` 和 `second` ）就是它的接口，此时直接将它们设为 public 即可，无需编写那些 trivial 的 getters/setters 。如果你的代码中的确存在这种情况，你需要给出解释。
8. (5%) 正确使用 `static` 成员函数、 `const` 成员函数和虚函数。
    - 与 `this` 无关的成员函数应当是 `static` 的。
    - 逻辑上不修改当前对象状态的 non-`static` 成员函数应当是 `const` 的。
    - 一些函数需要是虚函数，但不应将所有函数都设为虚函数。
9. (5%) 合理设计各个类之间的继承关系。
10. (5%) 尽可能使用构造函数初始值列表。
11. (5%) 非必要不使用显式类型转换（用来构造临时对象的表达式 `Type(args...)` 或 `Type{args...}` 可以被允

许）。如若需要，不使用 C 风格的类型转换运算符 `(Type)expr`，而是使用 C++ 的有名字的类型转换运算符 `what_cast<Type>(expr)`。

12. (5%) 变量应在即将使用时才被声明，特别是不应随意使用全局变量（常量除外）。

提交

请编写. gitignore, 确保你提交的文件中不会包含build, .vscode文件， 提交方式与HW4相同，即创建一个**私有**仓库，在gradescope上传你要提交的该仓库的某分支



查重

特别说明：本题将查重，**甚至与往年代码一起查重**。和信院其它代码课类似，抄袭被发现的后果**非常严重**，并且抄袭者与被抄袭者是同等处罚的，请不要抱有侥幸心理。