FAQ

CMake / CMake-GUI 爆红字

如果是 warning 的话，有可能不必担心，看到 build 完成了可以 generate 即可。

如果是 error ，下面的几行里就包含报错信息。可以自己检索一下找找原因或者 Piazza 发帖。

MacOS 上常见问题

- Could Not find X11: X11是一个图形窗口库。试一下 `brew install xquartz --cask` 命令安装 xquartz ，或者搜索一下相关信息，安装一个 X11 。
- CMake build 不出来：有往年同学反映过是 toolchain 需要用 gcc 而不能用 clang 。先 `brew install gcc` 安装一个gcc，再在根目录的 `CMakeLists.txt` 的第3行之前添加：

  ```
  set(CMAKE_C_COMPILER "/opt/homebrew/bin/gcc-14")
  set(CMAKE_CXX_COMPILER "/opt/homebrew/bin/g++-14")
  ```

- 因为历年 MacOS 问题五花八门而经验也较少，建议开始之前去看一下CMake教学视频。

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。

VSCode 的 CMake Debug 功能的配置方式请看 VSCode 相关问题。

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 还都不知道对方的存在！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。

[ERROR] Error loading asset ***.png

这是程序没有找到对应路径的图片素材产生的报错。报错信息中的路径为程序尝试去寻找图片的路径，是 `utils.hpp` 中的 `ASSET_DIR` 。你可以将 `ASSET_DIR` 修改成正确的路径。

如果你使用相对路径， `ASSET_DIR` 应当是从你的项目工程文件（Windows 下的 `.sln` 文件）开始，指向 Assets 目录的路径。

MacOS 上，如果你使用 XCode 等开发工具，相对路径也可能是从你的工程文件目录或 /build 目录开始。如果命令行运行时，由于相对路径会跟运行程序的终端打开的目录有关，建议自己调试时使用绝对路径，或使用确定的相对路径并且如果打包给别人玩的话说明清楚。

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` 。

【最常见的问题】 使用了未定义类型 `GameWorld` / `GameWorld` is an incomplete type / undeclared identifiers / not overriding virtual functions / 为什么要把函数的声明和定义分开在 `.hpp` 和 `.cpp` 文件里？都写在 `.hpp` 里为什么报错？/ 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。

Linking error: 无法解析的外部符号

这个错误通常是链接器的报错，它类似于 GCC 的 ld 报的“undefined reference to xxxx”，通常意味着**调用了有声明但未有定义的函数**。也就是说，大概有这样两种情况：

1. 某个被使用了的函数的确只有声明而未有定义，可能是你忘记定义了。特别地，非纯的虚函数必须有定义，否则将影响一些必要的运行时信息的合成。
2. 某个函数被定义了，但是编译器/链接器没有看到这个定义。比如，
    - 你把某个函数的定义写在 `a.cpp` 中，但完全没有编译 `a.cpp` （没有将它加入 CMake 的任何一个 target ，或者在上层 `CMakeLists.txt` 中缺乏必要的 `add_subdirectory` ）；或者，
    - Target A 里的代码调用了一个定义在 target B 中的函数，但编译 A 时没有把 B 链接上去（你需要在相应的 `CMakeLists.txt` 中修改 A 的 `target_link_libraries` ）。

另一些 `#include` 错误：我没有循环引用， CMakeLists 也改了，还是 Linking error？

省流：最快的解决方案是使用以 `/src/` 为起始路径的，带文件夹名的路径。例如，将 `#include "Sunflower.hpp"` 改为 `#include "Sunflower/Sunflower.hpp"` 。问题的根本出在 `#include` 与 `target_include_directories` 。

万恶之源仍然是 `#include` ： `#include` 就是简单的文本复制粘贴。

`#include` 通过相对路径来寻找目标头文件。你的每个 `CMakeLists.txt` 中的 `target_include_directories` 列出了这个 target 能够“看得到”的 `#include` 路径。在上一条 FAQ 中的第二步中，你可能已经为会用到你的文件的项目添加了 `target_include_directories` ，然而，有时会忽视一些特殊的情况：

例如，你的 `GameObject.cpp` 中不出意外应该会有 `#include "GameWorld.hpp"` ，然而如果你的 `GameWorld.hpp` 中还有 `#include "Sunflower.hpp"` ，这行代码将会在被 `#include` 时一并被复制粘贴到 `GameObject.cpp` 中。此时， `GameObject.cpp` 虽然并没有依赖 `Sunflower` 项目，但仍然尝试 `#include "Sunflower.hpp"` ，他在“看得到”的 `#include` 路径中找不到，便产生了错误。

解决方法有以下几种：

- 使用以 `/src/` 为起始路径的，带文件夹名的路径。例如， `#include "Sunflower/Sunflower.hpp"` 。因为每个项目的 `target_include_directories` 中都包含 `/src/` ，他们一定能“看得到”这个文件。
- 为这种情况下的 `GameObject` 项目，将 `Sunflower.hpp` 的位置添加到 `target_include_directories` ，即使他们不以 `target_link_libraries` 互相依赖。
- `GameWorld.hpp` 中真的需要 `#include "Sunflower.hpp"` 吗？有些情况下确有需求，但如果没有需求，只让 `GameWorld.cpp` 去 `#include` 它，就不会有这些麻烦。

VS（中文版）的默认字体好难看，影响我编程了

确实难看并且不适合作为代码字体。工具-选项-环境-字体和颜色，将“文本编辑器”的字体设置为适合编程的等宽字体，例如系统内置的Consolas或Cascadia Mono等。

~~也可以现在卸载 Windows，使用 Linux。~~

我想把我的游戏打包出来给别人玩，可以吗？

当然可以，你需要做的是像示例程序一样，将程序的可执行文件，必要的库（Windows dll）和图片素材文件夹整理出来。

你可以在 build/Debug 或 build/Release 中找到你的可执行文件，在 Windows 下，你也可能会找到一些 dll 库。这些文件都是必要的。

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。（注意与上一条的区别，在 project 里运行时会被解析为从项目工程文件开始的路径）

Debug 和 Release 是什么？有什么区别？

你的项目能以 Debug/Release 两种方式生成 (build) 。Debug 模式下可以接入调试器 (debugger) ，利用断点或堆栈检查来调试代码，而 Release 模式则会进行一些编译器优化等，获得更快的运行速度。**并且， Release 模式下编译时会定义 `NDEBUG` 宏，这会使得你程序里的所有 `assert` 都被去除。**

你的项目的默认生成模式一般是 Debug 。你可以在你的开发环境（IDE）的顶端状态栏看到并调整生成模式。其实 CMake 提供了 `CMAKE_BUILD_TYPE` 这个变量，在最初 build 项目时设定生成模式，但是这个设定一般会被 IDE 接管覆盖。

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每棵植物的 HP 或冷却时间等功能。

我想做一份自己的“杂交植物”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。

VSCode 相关问题

在开始之前，请确保当前工作区的绝对路径上没有任何非 ASCII 字符、空格、可能引发问题的标点符号。像给文件夹命名为“作业”、“大一下”这样的中文名的习惯**建议趁早改**，因为不支持中文路径的软件非常多。

要使用 VSCode 配合 CMake 进行项目构建，首先需要安装 **CMake**、 **CMake Tools** 和 **CMake Language Support** 这三个扩展（“扩展”又名“插件”）。

你还需要了解一个**非常基本的知识**：对于当前工作区（即，现在打开的最外层文件夹）的配置，是通过在当前工作区下的 `.vscode` 文件夹中的若干配置文件来完成的。这个 `.vscode` 文件夹必须直接位于当前工作区，不能在某个更里面的文件夹中。一般来说， `settings.json` 包含了绝大多数的设置项， `tasks.json` 通常负责项目的编译， `launch.json` 通常负责程序的运行，而某些扩展可能会约定属于它自己的配置文件，例如 `c_cpp_properties.json` 是 “C/C++” 这个扩展自己的特殊配置文件。在本学期初的搭环境视频中，我们其实已经解释了这些配置文件的一些细节，可能那时对代码十分陌生的你还未能很好地理解它们，但是经过一个学期的学习之后，你应该逐渐认识它们并学会编写它们了。

接下来要明确的是，我们需要解决以下问题：

1. 使用 CMake 生成 build system 所需的文件，通俗点说就是生成 `build` 文件夹以及里面的一大堆东西。
2. 调用 build system 来根据 `build` 文件夹里的内容对项目进行编译。
3. 编译成功后，运行、调试可执行文件。
4. 配置 IntelliSense ，使得你写代码的时候可以获得补全、查找、重命名、实时报错等功能。

一个热知识： Ctrl + Shift + p （在 Mac 上是 Command + Shift + p）是 VSCode 的“万能按键”，在这里你可以输入一些内容， VSCode 会帮你找到最匹配的配置或功能。

生成 `build`

按 Ctrl + Shift + p 输入 `cmake configure` ，最优匹配应当是“CMake: Configure”，回车。接下来如果让你选 Kit （第一次执行通常需要）， Windows 上应该选 Visual Studio 17 2022 amd64 ， Mac 上选最高版本的 gcc 。特别是对于 Windows ，如果这个选项不存在，有可能是没有正确安装 Visual Studio ，请先去安装最新版的 Visual Studio ；如果已经装好了，可以选 [Scan for kits] 刷新一下，应该就有了。

默认情况下， CMake 插件会将当前工作区中（最外层文件夹）的 `CMakeLists.txt` 作为根。如果这个文件不存在，它会报错。修改这一行为的方式是在 `settings.json` 中加入一项 `"cmake.sourceDirectory"` ，例如

```
    "cmake.sourceDirectory": "${workspaceFolder}/attachments"
```

就会让 CMake 插件去当前工作区下的 `attachments` 文件夹中寻找 `CMakeLists.txt` 作为根。这里我们使用了一个变量 `${workspaceFolder}` ，这也是在 VSCode 配置中十分常见的。

在生成的过程中你需要密切关注的，是它采用的 generator 是什么，特别是 Windows 上可能会有问题。在你启动“CMake: Configure”时， Output 面板中会显示一些日志，其中最前面应该有一段 `[proc] Executing command:` 开头的内容，它表示究竟执行了什么指令，找到 `-G` 后面紧跟着的东西，它就是现在 CMake 在使用何种 generator 。对于 Windows ，这一项应该是 `Visual Studio 17 2022` ，如果是 `MinGW Makefiles` 则在本次作业中大概率会引发错误（具体原因未知）。你也可以通过生成的 `build` 文件夹中的内容来看： Visual Studio 对应的是某个 `.sln` 文件，而 MinGW Makefiles 对应的是一个名为 `Makefile` （无后缀名）的文件。可以在 `settings.json` 中设置 `"cmake.generator"` ，例如

```
    "cmake.generator": "Visual Studio 17 2022"
```

如果之前已经生成了 `build` ，换了一个 generator 之后要重新生成，你需要把 `build` 文件夹删除。

编译

按 Ctrl + Shift + p 输入 `cmake build` ，最优匹配应当是“CMake: Build”，回车。如果之前没有启动过“CMake: Configure”，“CMake: Build” 会启动“CMake Configure”生成 `build` ，在此过程中遇到问题请看上一节。

编译的过程中，如果在 Output 面板中显示的中文有乱码，去全局设置（按 Ctrl + 逗号）找“Cmake: Output Log Encoding”，将其值改为 `UTF-8` 。

编译完成后，可以留意一下生成的可执行文件的位置，它可能在 `build/bin` 、 `build/bin/Debug` 或者什么类似的目录里。

运行、调试

编译成功后，按 Ctrl + Shift + p 输入 `cmake run without debug` ，最优匹配应当是“CMake: Run Without Debugging”，回车，就是非调试地运行。如果输入 `cmake debug` ，最优匹配应当是“CMake: Debug” ，就是调试。

无论是调试还是非调试运行，我们可能都需要将运行时的工作目录设为 `build` ，以正确加载 assets 。这一项的配置是 `"cmake.debugConfig"` 的 `"cwd"` 子项：

```
    "cmake.debugConfig": {
        "cwd": "${workspaceFolder}/build"
    }
```

其它的事情跟正常 debug 没啥区别，你仍然可以打断点、看 local variables 和 call stack 等。

配置 IntelliSense

一般情况下， C/C++ 的 IntelliSense 是由插件“C/C++”负责的，你可以在当前工作区的 `.vscode` 中编写 `c_cpp_properties.json` 来对其进行配置。

现在我们有 CMake 了，影响是什么呢？是我们的编译方式变复杂了，而且编译指令是由 CMake 生成的相关文件给出的，想要直接获得编译指令并翻译为 `c_cpp_properties.json` 的配置项是非常困难的。幸好，有一个简单的办法，那就是让 CMake 相关插件直接和 C/C++ 插件沟通：

```
// c_cpp_properties.json 的内容
{
    "configurations": [
        {
            "name": "CMake",
            "configurationProvider": "ms-vscode.cmake-tools",
        }
    ],
    "version": 4
}
```

这里， `"name"` 一项并不重要，它可以是任何值。关键之处是 `"configurationProvider"` 一项，设为 `"ms-vscode.cmake-tools"` 就是让 CMake Tools 插件告诉 C/C++ 插件应该怎样配置。就这样。

现在你可以试一下，比方说随便删掉代码里的一个分号，看看 VSCode 有没有给你画红线，以及按 ctrl 左击某个 `#include` 的文件是否能正常跳转。