2023 CS100 HW8

封面页

(Deadline: 2023-6-10 23:59)

注 意

- 不要被本说明的长度吓到。它之所以长达 **20** 多页，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来“教你做事”。
- 请为本次作业分配充足的时间（参考：至少 **3** 天[1]），并且不要到临近截止日期时才开始！
- 无论你是否准备现在开工，请去 **piazza** 上观看 **CMake** 使用教程，或阅读附录中的“首次运行”部分。**Windows** 下可以使用集成了生成工具的 **Visual Studio**，也可以单独下载 **Visual Studio** 生成工具后使用 **Clion** 或自行折腾配置 **VS code**。**MacOS** 下推荐使用 **Xcode**。请立刻下载所需的依赖，并且确保你能够正常编译并运行提供的代码框架。若遇到问题，尽早在 **Piazza** 上提问。
- 如果对任何部分有疑问，在你提问之前，请先在本说明中寻找答案，或试着游玩样例游戏并观察样例游戏的处理方式。对于任何细节，只要你的游戏与样例表现一致，你就不会失分。我们也不会检查任何未说明的、无法在样例游戏中复现的情况。
- 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。

【最终版本】

更新了【提交】部分。

支线目标中若干分值调整。增加了目标 **11**。

不会再进行大规模改动。如有其他补充说明将在 **piazza** 上发布。

[1] **3** 天的数值未经过任何数据收集或数学测算。若你参考本数据预留的时间不足以完成作业，**CS100 TA** 团队不为此负任何责任。

◆ 故事

你驻足于 TakuGames 公司的门前。阵风吹拂，你感慨万千。

一年前，成为一名游戏开发人员的梦想指引你来到这里，你却因为未能完成代号“破晓”项目（2022 年 CS100 HW8）而被辞退。

年轻气盛的你并不服气。你打算自立门户，坚信终会一鸣惊人。你执意标新立异，但已然一年，却从未有你所期望着的妙手偶得。

此刻，公司会议上的讨论声仿佛在你耳边响起。老旧！俗套！他们根本不懂年轻人喜欢什么！

“——做一个给年轻人的塔防游戏吧！”

灵光一现。机会转瞬即逝，但这一刻你抓住了它。无数的想法在脑海中汇集，此刻，你感觉你就是翻涌的银河中一颗璀璨的明星。

你充满了决心。你赶回家中，当机立断，写下了这份策划案。

◆ 游戏简介

代号“PvZ”（原型）是一个仿制游戏，仿自 PopCap Games 的著名游戏《植物大战僵尸》。本项目中的图片与动画资源均来自《植物大战僵尸》。

◆ 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（是的，30 帧，今年 5 月发售的一部伟大的游戏不也是 30 帧吗？）。

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（植物、僵尸、飞行的豌豆，甚至还包括背景）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[2]等），也可以用成员函数(member functions)来定义自己应当做的行为（比如豌豆射手如何攻击、僵尸如何移动或啃食植物）。

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

[2] HP 实际上来源于 Hit Point 的缩写，而不是大家更广泛以为的 Health Point。早期的很多简单游戏中，每受到一次打击，玩家就会减少一点 Hit Point，直至失去所有 HP 游戏结束。而随着伤害数值不再以一点为单位而开始细分，Hit Point 也从几点变为几十点、几百点，不再代表“受到打击的次数”。大家逐渐更认可 Health Point 的说法，正如我们也会将它叫做“生命值”或“血量”一样。

Update()函数。不同的对象对这个函数的内容需要各有实现：阳光可能只需要向下移动两个像素从而让它看起来像是正在匀速下落；豌豆射手需要确定是否要发射出一颗豌豆；背景图片或“铲子”图标很可能什么都不需要做。

在本游戏中，玩家会通过点击某个对象来完成种植/铲除植物等操作。实现的方式是OnClick()函数。当某个对象被点击时，它的OnClick()函数会被调用。多个对象重叠时，只有显示在最上方的对象会被点击。

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。

◆ UI，动画，还有隐形的兔子

UI (User Interface)即用户界面，包括按钮、菜单、显示的文字或数值等展示游戏内容和供玩家交互的部分。相信你已经很直观地感受到了UI对游戏的体验有多么关键：本游戏和《植物大战僵尸》相比，缺少了冷却时间显示、选取内容的光标显示、暂停菜单等细节。这些差别使得游戏的体验产生了很大的差距。

本游戏中的动画使用了非常简单的实现方式——逐帧动画。把每帧单独保存为一个图片文件会大大增加资源量、拖缓加载速度，所以我们将多帧动画放在同一张图片里，以每帧显示图片的一部分的方式播放动画。这样的图片被称为精灵动画表(sprite sheet)，应用于很多2D游戏中。

好吧，这个名字确实会让人一头雾水。Sprite直译为精灵，其实意思只是显示游戏里某个元素的一张二维图片。由于翻译成“精灵”可能会使人更困惑，一些游戏或动画软件的中文界面中会选择保留sprite这个单词不作翻译。

在你能看见的游戏背后，有很多看不见的东西支撑着游戏的运行。几乎每个游戏中都藏着一些“隐形的兔子”。这个典故来源于《魔兽世界》的早期版本，在那时，游戏中显示任务进度的代码只能判断有没有杀死指定怪物或获得指定道具。那“与NPC对话”或者“前往指定地点”的任务怎么完成呢？开发者们想到的方法是，为这样的任务创建一只隐形的兔子，在玩家与NPC对话或到达指定地点时将它杀掉。这样，无论多么稀奇古怪的任务要求都可以变成“杀死一只兔子”。《魔兽世界》就这样带着一只只被杀害的隐形兔子问世了。同样，今年《英雄联盟》更新产生的某个BUG让玩家发现，英雄“嘉文四世”大招创建的墙体实际上是由几十个隐形的“小兵”围成的。小兵的碰撞体积使得它像一片连续的不可通过的墙体。

在本游戏中，你们也需要发挥聪明才智，运用“隐形的兔子”来解决一些问题。草坪上本身是空的，那么在点击一片空地种下向日葵的时候，究竟点到了什么呢？

◆ 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（豌豆打到僵尸），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，豌豆射手希望在 **GameWorld** 中创建一颗豌豆）

游戏中的所有对象，无论属于什么类型，都必须放在这一个容器中。（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？）并且，你的 **GameWorld** 中只能有这一个容器。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，每隔一段时间就会掉落一颗阳光。这个倒计时便需要 **GameWorld** 储存。有一些游戏数据我们已经在框架中替你存储在了 **GameWorld** 的基类 **WorldBase** 中，分别是**当前波数**和**阳光数**，你可以通过 **WorldBase** 中的访问函数和更改函数来获取或修改它们。

GameWorld 类继承自提供的框架中的 **WorldBase** 类。你不允许更改 **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，这三个方法只允许被提供的游戏框架自动调用，而你所写的代码中不允许主动调用。

```
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，把种子按钮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，例如新掉落的阳光、新生成的僵尸等。
- 删除 GameWorld 中应被删除的对象，例如超过时间未被点击的阳光、被击杀的僵尸等。

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，上述三个函数均不允许被你所写的代码主动调用。

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。

◆ 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。如何将植物和僵尸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。因此，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。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，只有在坚果墙或铁桶僵尸更换贴图时，才需要使用 ChangeImage(ImageID imageID)修改它。
  - 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，如阳光位于 **layer 0** 而向日葵位于 **layer 3**，阳光在显示时就会遮盖向日葵。
- **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。

- “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，阳光确定了它的 **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。

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”。

像超过时间未被点击的阳光、被击杀的僵尸、被铲除的植物等对象就需要“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——如果你发现“普通僵尸”、“铁桶僵尸”和“撑杆跳僵尸”有很多共同点，试着写一个基类（可以叫 Zombie），将他们的共同点包括在内；如果你想让发射的豌豆只伤害僵尸而不伤害自己的植物，那就在共同的 GameObject 基类里里加入一个虚函数(virtual function)，让植物和僵尸对它的实现不同，从而区分开来。既可以简单地写一个叫做 IsZombie()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。因此，我们将推荐先实现的部分以蓝色字体标出。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。写码时建议从蓝色字体部分开始逐步进行。例如，你可以先实现背景，从而使“press Enter to start”后显示的不再是黑屏；然后可以让这个场景里开始掉落阳光；接下来当你成功种下向日葵的时候，应该就对这次作业的做法了然于胸了。

在提供的文件之中，你不允许修改 src/Framework 路径下的文件。对于 utils.hpp，你不允许修改除其中 ASSET_DIR 字符串的值以外的其他内容，若通过修改这些如提供的框架代码实现了需要的功能，会无法正常通过评测。

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。

◆ GameWorld

◆ GameWorld::Init()

你的 GameWorld::Init()函数必须：

- 初始化任何用于记录关卡数据的成员变量。例如，初始波数为 0，阳光为 50。
- 创建背景。
- 创建 45 个种植位，GAME_ROWS(5)行 GAME_COLS(9)列，如右图：
  - 首个（左下角）位于(x = FIRST_COL_CENTER, y = FIRST_ROW_CENTER)。
  - 种植位之间的横向间距为 LAWN_GRID_WIDTH，纵向间距为 LAWN_GRID_HEIGHT。
- 生成位于顶端的植物种子按钮：
  - 首个按钮为向日葵种子，位于(x = 130, y = WINDOW_HEIGHT - 44)。
  - 其余按钮按顺序为豌豆射手、坚果墙、樱桃炸弹、双发射手，横向间距为 60。
- 生成铲子按钮，位于(x = 600, y = WINDOW_HEIGHT - 40)。

◆ GameWorld::Update()

你的 GameWorld::Update()函数必须：

1. 为游戏掉落新的阳光。掉落第一个阳光的时间为游戏开始后的第 180 个 tick（6 秒）。之后，每 300 个 tick（10 秒）掉落一个阳光。生成的阳光：
   - x 坐标为[75, WINDOW_WIDTH - 75]区间内的一个随机 int。
     - 我们已在 utils.hpp 中提供了生成随机闭区间 int 的函数 randInt()。
   - y 坐标为 WINDOW_HEIGHT - 1。
   - 将开始下落。
2. 为游戏生成新的僵尸。僵尸以波的形式出现，第一波僵尸将在游戏开始后的第 1200 个 tick（40 秒）生成，之后每波僵尸的生成间隔将加快。具体策略如下：

- 当前生成的僵尸波数记为 `wave`。（游戏初始 `wave` 为 `0` 而首波僵尸为 `wave 1`）
- 随机生成$\left\lfloor\frac{(15+wave)}{10}\right\rfloor$个僵尸。
- 下一波僵尸将在 `max(150, 600 - 20 * wave)`个 `tick` 后生成。
  - 简单的验算：第 23 波将在第 22 波后 160 个 `tick` 生成，而之后每波的间隔固定为 150 个 `tick`。

3. 若在上一步中确定生成僵尸，则需要进一步随机决定僵尸的种类，对于每个生成的僵尸，策略如下：
   - 令 `P1 = 20`,
     `P2 = 2 * max(wave - 8, 0)`,
     `P3 = 3 * max(wave - 15, 0)`，则：
   - 有 `P1 / (P1 + P2 + P3)`的概率，生成的僵尸为普通僵尸。
   - 有 `P2 / (P1 + P2 + P3)`的概率，生成的僵尸为撑杆跳僵尸。
   - 剩余的 `P3 / (P1 + P2 + P3)`的概率，生成的僵尸为铁桶僵尸。
   - 生成的僵尸 `x` 坐标为`[WINDOW_WIDTH - 40, WINDOW_WIDTH - 1]`区间内的一个随机 `int`，`y` 坐标为随机一行的 `y` 坐标（首行为 `FIRST_ROW_CENTER`，纵向间距为 `LAWN_GRID_HEIGHT`）。
     - 当你心里在抱怨“计算好麻烦”的时候，你又在骂数值策划了。果然，无论在什么游戏里，策划都会挨骂……
4. 遍历所有游戏对象(`GameObject`)，并依次调用它们的 `Update()`函数。
5. 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：
   - 豌豆击中僵尸。
   - 爆炸影响到僵尸。
   - 僵尸啃咬植物。
     - 撑杆跳僵尸的特殊碰撞检测会在其`Update()`内完成。
6. 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。
7. 判断是否失败。若有任何僵尸的 `x` 坐标 `< 0`，则返回 `LevelStatus::LOSING`。
8. 额外检测一遍僵尸是否在与植物发生碰撞。若某个僵尸没有和植物发生碰撞而在啃食植物，则使其正常走动。
   - 这种情况的发生是由于多个僵尸在某一帧啃死了同一棵植物。先啃的僵尸并不知道植物会死亡，只能保持啃食状态，所以需要额外判断一次以使其正常移动。
9. 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 / 2, y = WINDOW_HEIGHT / 2)。
- 背景的所在层级为 LAYER_BACKGROUND。
- 背景的宽度为 WINDOW_WIDTH，高度为 WINDOW_HEIGHT。
- 背景不具有动画，即，动画编号为 ANIMID_NO_ANIMATION。

◆ 当 Update()时

背景什么也不需要做。

◆ 当被点击时

背景什么也不需要做。

◆ 阳光(Sun)

◆ 当被创建时

- 阳光的贴图编号为 IMGID_SUN。
- 阳光的起始位置并不固定，而由创建它的代码决定。(因此，它的构造函数中必须包含此部分。)
- 阳光的所在层级为 LAYER_SUN。
- 阳光的宽度为 80，高度为 80。
- 阳光具有 ANIMID_IDLE_ANIM 动画。
- 阳光具有掉落时间。
  - 从天上掉落的阳光的掉落时间为[63, 263]之间的随机 int。这个掉落时间保证了它将掉在草坪上。
  - 向日葵产生的阳光将以抛物线落地，掉落时间为 12。

◆ 当 Update()时

1. 若阳光还未落地，则：
   - 如果它是从天上掉落的阳光，则它每刻向下移动 2 像素。
   - 如果它是向日葵产生的阳光，则它每刻向左移动 1 像素，并在竖直方向上做竖直上抛运动：其初速度（第一帧）为向上的 4 像素/刻，加速度为-1（方向向下）。

- 这样的阳光运动轨迹是斜向左上的抛物线。在第 5 个 tick，它的 y 方向速度为 0，位于抛物线顶端。

2. 若阳光已经落地，它将不再移动，并在落地后 300 个 tick（10 秒）消失。

◆ **当被点击时**

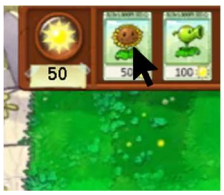
被点击时，阳光需要消失，并通知 GameWorld 获得 25 点阳光。

- 思考一下：要做到这点，阳光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 创建阳光时，怎样把“所在的 GameWorld”这一信息告诉它？除了阳光，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？

◆ 种植位(Planting Spot)

种植位是位于草坪上的“隐形的兔子”。点击它可以把手上的植物种在它的位置。

◆ **当被创建时**



- 种植位的贴图编号为 IMGID_NONE，没有贴图。
- 种植位的位置并不固定，而由创建它的代码决定。（因此，它的构造函数中必须包含此部分。）
- 种植位的所在层级为 LAYER_UI。
- 种植位的宽度为 60，高度为 80。略小于草坪格，与植物大小一致。
- 种植位不具有动画，即，动画编号为 ANIMID_NO_ANIMATION。

◆ **当 Update()时**

种植位什么也不需要做。

◆ **当被点击时**

若玩家正拿着一个还未种下的种子，则“种下”它，即在种植位的位置生成一个对应的植物。

- 种下的植物由于位于更靠前的层级，将覆盖种植位。也就是说，有植物的种植位将不会再被点击到。
- 思考一下：既然在点击时才需要生成植物，那么玩家拿在手里的就并不是植物，当然也不是种子包（指左上角的按钮），毕竟手中的这个东西并没有贴图。而且，种植位需要根据“手里的东西”决定种下的是什么植物，说明这一信息存储在“手里的东西”上。这个东西究竟是什么？提示：另一种隐形的兔子！

◆ 向日葵种子(Sunflower Seed)

◆ **当被创建时**

- 向日葵种子的贴图编号为 IMGID_SEED_SUNFLOWER。

- 向日葵种子位于第一个种子包位置(x = 130, y = WINDOW_HEIGHT - 44)。
- 向日葵种子的所在层级为 LAYER_UI。
- 向日葵种子的宽度为 50，高度为 70。
- 向日葵种子不具有动画。
- 向日葵种子的价格是 50 阳光。
- 向日葵种子的冷却时间是 240 个 tick（8 秒）。

◆ **当 Update()时**

向日葵种子的 Update()并没有必须要做的事情，然而，基于你的实现，你也可以在 Update()中做些什么……

◆ **当被点击时**

1. 如果玩家正拿着铲子，或正拿着一个还未种下的种子，则点击无效。
2. 如果向日葵种子正处于冷却状态，则点击无效。（如果你实现了冷却遮盖物，则遮盖物会遮挡住点击）
3. 询问 GameWorld 是否有 50 阳光。如果有，则花费 50 阳光，进入冷却状态，在原地（自身 x, y 坐标位置）生成一个冷却遮盖物，并在下一次点击种植位时种下向日葵。
   - ◆ 花费阳光后，你并不能直接生成向日葵，因为只有点击种植位后才能知道它应当生成在哪里。
   - ◆ 那么，点击种子后要做什么呢？（没有想到的话，再去读读上一页）

◆ 向日葵(Sunflower)

◆ **当被创建时**

- 向日葵的贴图编号为 IMGID_SUNFLOWER。
- 向日葵的位置并不固定，而由创建它的代码决定。（因此，它的构造函数中必须包含此部分。）
- 向日葵的所在层级为 LAYER_PLANTS。
- 向日葵的宽度为 60，高度为 80。
- 向日葵具有 ANIMID_IDLE_ANIM 动画。
- 向日葵具有 300 点 HP。
- 向日葵第一次产生阳光的时刻是[30, 600]的随机 int，之后产生阳光的间隔为 600 刻。
  - ◆ 也就是说，运气好的话，刚种下 1 秒多就可以获得阳光。

◆ **当 Update()时**

1. 向日葵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. 如果向日葵在此刻产生阳光，它将在原地（自身 x, y 坐标处）生成一个“由向日葵产生的阳光”。

◆ **当被点击时**

如果玩家正拿着铲子，那么向日葵需要死亡。放下铲子。

◆ 冷却遮盖物(Cooldown Mask)

◆ **当被创建时**

- 冷却遮盖物的贴图编号为 IMGID_COOLDOWN_MASK。
- 冷却遮盖物的位置由创建它的代码决定。
- 冷却遮盖物的所在层级为 LAYER_COOLDOWN_MASK。（会遮挡种子）
- 冷却遮盖物的宽度为 50，高度为 70。
- 冷却遮盖物不具有动画。
- 冷却遮盖物需要在对应的种子冷却后消失。

◆ **当 Update()时**

冷却遮盖物需要在对应的种子冷却后消失。

◆ **当被点击时**

冷却遮盖物什么也不需要做。

◆ 豌豆射手种子(Peashooter Seed)

◆ **当被创建时**

- 豌豆射手种子的贴图编号为 IMGID_SEED_PEASHOOTER。
- 豌豆射手种子位于第 2 个种子包位置，每个种子包的横向间隔为 60。
- 豌豆射手种子的所在层级为 LAYER_UI。
- 豌豆射手种子的宽度为 50，高度为 70。
- 豌豆射手种子不具有动画。
- 豌豆射手种子的价格是 100 阳光。
- 豌豆射手种子的冷却时间是 240 个 tick（8 秒）。

◆ **当 Update()时**

豌豆射手种子的 Update()并没有必须要做的事情，然而，基于你的实现，你也可以在 Update()中做些什么……

◆ **当被点击时**

1. 如果玩家正拿着铲子，或正拿着一个还未种下的种子，则点击无效。
2. 如果豌豆射手种子正处于冷却状态，则点击无效。（如果你实现了冷却遮盖物，则遮盖物会遮挡住点击）
3. 询问 GameWorld 是否有 100 阳光。如果有，则花费 100 阳光，进入冷却状态，在原地（自身 x, y 坐标位置）生成一个冷却遮盖物，并在下一次点击种植位时种下豌豆射手。

◆ 豌豆射手(Peashooter)

◆ **当被创建时**

- 豌豆射手的贴图编号为 IMGID_PEASHOOTER。
- 豌豆射手的位置由创建它的代码决定。
- 豌豆射手的所在层级为 LAYER_PLANTS。
- 豌豆射手的宽度为 60，高度为 80。
- 豌豆射手具有 ANIMID_IDLE_ANIM 动画。
- 豌豆射手具有 300 点 HP。
- 豌豆射手每 30 tick（1 秒）发射一颗豌豆。

◆ **当 Update()时**

1. 豌豆射手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. 如果豌豆射手的攻击能力正在冷却，则降低 1 冷却时间，它的 Update()返回。
3. 如果豌豆射手可以攻击，它将询问 GameWorld，自己的一行上，自己的右边是否存在僵尸。若存在，则在豌豆射手自身坐标的右方 30 像素、上方 20 像素位置生成一颗豌豆，然后进入 30 tick 的冷却时间。

◆ **当被点击时**

如果玩家正拿着铲子，那么豌豆射手需要死亡。放下铲子。

◆ 豌豆(Pea)

◆ **当被创建时**

- 豌豆的贴图编号为 IMGID_PEA。
- 豌豆的位置由创建它的代码决定。
- 豌豆的所在层级为 LAYER_PROJECTILES。
- 豌豆的宽度为 28，高度为 28。
- 豌豆不具有动画。

◆ **当 Update()时**

1. 豌豆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. 豌豆将向右移动 8 像素。
3. 如果豌豆飞出了屏幕右边界(y >= WINDOW_LENGTH)，豌豆需要死亡。

◆ **当被点击时**

豌豆什么也不需要做。

◆ **当与僵尸碰撞时**

豌豆将会对与它碰撞的僵尸造成 20 点伤害，然后死亡。

◆ 坚果墙种子(Wallnut Seed)

◆ **当被创建时**

- 坚果墙种子的贴图编号为 IMGID_SEED_WALLNUT。
- 坚果墙种子位于第 3 个种子包位置，每个种子包的横向间隔为 60。
- 坚果墙种子的所在层级为 LAYER_UI。
- 坚果墙种子的宽度为 50，高度为 70。
- 坚果墙种子不具有动画。
- 坚果墙种子的价格是 50 阳光。
- 坚果墙种子的冷却时间是 900 个 tick（30 秒）。

◆ **当 Update()时**

坚果墙种子的Update()并没有必须要做的事情,然而,基于你的实现,你也可以在Update()中做些什么……

◆ **当被点击时**

1. 如果玩家正拿着铲子，或正拿着一个还未种下的种子，则点击无效。
2. 如果坚果墙种子正处于冷却状态，则点击无效。（如果你实现了冷却遮盖物，则遮盖物会遮挡住点击）
3. 询问 GameWorld 是否有 50 阳光。如果有，则花费 50 阳光，进入冷却状态，在原地（自身 x, y 坐标位置）生成一个冷却遮盖物，并在下一次点击种植位时种下坚果墙。

◆ 坚果墙(Wallnut)

◆ **当被创建时**

- 坚果墙的贴图编号为 `IMGID_WALLNUT`。
- 坚果墙的位置由创建它的代码决定。
- 坚果墙的所在层级为 `LAYER_PLANTS`。
- 坚果墙的宽度为 `60`，高度为 `80`。
- 坚果墙具有 `ANIMID_IDLE_ANIM` 动画。
- 坚果墙具有 `4000` 点 HP。

◆ **当 `Update()`时**

1. 坚果墙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. 如果坚果墙的 HP 小于总 HP 的 `1/3`，将它的贴图更换为“受损的坚果墙” `IMGID_WALLNUT_CRACKED`。

◆ **当被点击时**

如果玩家正拿着铲子，那么坚果墙需要死亡。放下铲子。

◆ 樱桃炸弹种子(Cherry Bomb Seed)

◆ **当被创建时**

- 樱桃炸弹种子的贴图编号为 `IMGID_SEED_CHERRY_BOMB`。
- 樱桃炸弹种子位于第 `4` 个种子包位置，每个种子包的横向间隔为 `60`。
- 樱桃炸弹种子的所在层级为 `LAYER_UI`。
- 樱桃炸弹种子的宽度为 `50`，高度为 `70`。
- 樱桃炸弹种子不具有动画。
- 樱桃炸弹种子的价格是 `150` 阳光。
- 樱桃炸弹种子的冷却时间是 `1200` 个 `tick`（`40` 秒）。

◆ **当 `Update()`时**

樱桃炸弹种子的 `Update()`并没有必须要做的事情，然而，基于你的实现，你也可以在 `Update()`中做些什么……

◆ **当被点击时**

1. 如果玩家正拿着铲子，或正拿着一个还未种下的种子，则点击无效。
2. 如果樱桃炸弹种子正处于冷却状态，则点击无效。（如果你实现了冷却遮盖物，则遮盖物会遮挡住点击）
3. 询问 `GameWorld` 是否有 `150` 阳光。如果有，则花费 `150` 阳光，进入冷却状态，在原地（自身 `x, y` 坐标位置）生成一个冷却遮盖物，并在下一次点击种植位时种下樱桃炸弹。

◆ 樱桃炸弹(Cherry Bomb)

◆ 当被创建时

- 樱桃炸弹的贴图编号为 IMGID_CHERRY_BOMB。
- 樱桃炸弹的位置由创建它的代码决定。
- 樱桃炸弹的所在层级为 LAYER_PLANTS。
- 樱桃炸弹的宽度为 60，高度为 80。
- 樱桃炸弹具有 ANIMID_IDLE_ANIM 动画。
- 樱桃炸弹具有 4000 点 HP。（以保证它不会在爆炸前被僵尸吃掉）

◆ 当 Update()时

樱桃炸弹需要在种下后的第 15 帧死亡，并在原地生成一个爆炸特效。

◆ 当被点击时

如果玩家正拿着铲子，那么樱桃炸弹需要死亡。放下铲子。

◆ 爆炸(Explosion)

◆ 当被创建时

- 爆炸的贴图编号为 IMGID_EXPLOSION。
- 爆炸的位置由创建它的代码决定。
- 爆炸的所在层级为 LAYER_PROJECTILES。
- 爆炸的宽度为 3 * LAWN_GRID_WIDTH，高度为 3 * LAWN_GRID_HEIGHT。
- 爆炸不具有动画。
- 爆炸的存在时间为 3 帧。

◆ 当 Update()时

爆炸需要在被创建的 3 帧后消失。

◆ 当被点击时

爆炸什么都不需要做。

◆ 当与僵尸碰撞时

在爆炸存在的 3 帧内，它将杀死所有与它碰撞的僵尸。

◆ 双发射手种子(Repeater Seed)

◆ 当被创建时

- 双发射手种子的贴图编号为 IMGID_SEED_REPEATER。
- 双发射手种子位于第 5 个种子包位置，每个种子包的横向间隔为 60。
- 双发射手种子的所在层级为 LAYER_UI。
- 双发射手种子的宽度为 50，高度为 70。
- 双发射手种子不具有动画。
- 双发射手种子的价格是 200 阳光。
- 双发射手种子的冷却时间是 240 个 tick（8 秒）。

◆ **当 Update()时**

双发射手种子的 Update()并没有必须要做的事情，然而，基于你的实现，你也可以在 Update()中做些什么……

◆ **当被点击时**

1. 如果玩家正拿着铲子，或正拿着一个还未种下的种子，则点击无效。
2. 如果双发射手种子正处于冷却状态，则点击无效。（如果你实现了冷却遮盖物，则遮盖物会遮挡住点击）
3. 询问 GameWorld 是否有 200 阳光。如果有，则花费 200 阳光，进入冷却状态，在原地（自身 x, y 坐标位置）生成一个冷却遮盖物，并在下一次点击种植位时种下双发射手。

◆ 双发射手(Repeater)

◆ **当被创建时**

- 双发射手的贴图编号为 IMGID_REPEATER。
- 双发射手的位置由创建它的代码决定。
- 双发射手的所在层级为 LAYER_PLANTS。
- 双发射手的宽度为 60，高度为 80。
- 双发射手具有 ANIMID_IDLE_ANIM 动画。
- 双发射手具有 300 点 HP。
- 双发射手每 30 tick（1 秒）发射两颗豌豆。

◆ **当 Update()时**

1. 双发射手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. 如果双发射手的攻击能力正在冷却，则降低 1 冷却时间，它的 Update()返回。

3. 如果双发射手可以攻击，它将询问 GameWorld，自己的一行上，自己的右边是否存在僵尸。若存在，则在双发射手的右方 30 像素、上方 20 像素位置生成一颗豌豆，进入 30 tick 的冷却时间，在之后的第 5 tick 再次在同一位置生成一颗豌豆。

    - 换言之，在发射第二发豌豆后，双发射手的攻击能力还需冷却 25 tick。

- **当被点击时**

  如果玩家正拿着铲子，那么双发射手需要死亡。放下铲子。

◆ 铲子(Shovel)

- **当被创建时**
  - 铲子的贴图编号为 IMGID_SHOVEL。
  - 铲子位于(x = 600, y = WINDOW_HEIGHT − 40)。
  - 铲子的所在层级为 LAYER_UI。
  - 铲子的宽度为 50，高度为 50。
  - 铲子不具有动画。
- **当 Update()时**

  铲子什么也不需要做。
- **当被点击时**

  1. 如果玩家正拿着一个还未种下的种子，则点击无效。
  2. 如果玩家正拿着铲子，则放下铲子。

◆ 普通僵尸(Regular Zombie)

- **当被创建时**

  - 普通僵尸的贴图编号为 IMGID_REGULAR_ZOMBIE。
  - 普通僵尸的位置由创建它的代码决定。
  - 普通僵尸的所在层级为 LAYER_ZOMBIES。
  - 普通僵尸的宽度为 20，高度为 80，如图。这样设定是为了防止它同时与两棵植物碰撞。
  - 普通僵尸初始具有 ANIMID_WALK_ANIM 动画。
  - 普通僵尸具有 200 点 HP。
- **当 Update()时：**

  1. 普通僵尸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. 若普通僵尸当前正在行走，它将向左移动 1 像素。若普通僵尸当前正在啃食植物，它将不会移动。

◆ **当被点击时**

普通僵尸什么也不需要做。

◆ **当发生碰撞时**

- 如果与一颗豌豆发生碰撞，普通僵尸将会受到 20 点伤害，并让那颗豌豆死亡。
- 如果与爆炸发生碰撞，普通僵尸将会立即死亡。
- 如果与任何植物发生碰撞，普通僵尸将会：
  - ◆ 如果当前正在行走，则进入啃食状态，并播放 ANIMID_EAT_ANIM 动画。
  - ◆ 对那棵植物造成 3 点伤害。（这是每帧 3 点的持续伤害，即每秒 90 点）

◆ 铁桶僵尸(Bucket Head Zombie)

◆ **当被创建时**

- 铁桶僵尸的贴图编号为 IMGID_BUCKET_HEAD_ZOMBIE。
- 铁桶僵尸的位置由创建它的代码决定。
- 铁桶僵尸的所在层级为 LAYER_ZOMBIES。
- 铁桶僵尸的宽度为 20，高度为 80。
- 铁桶僵尸初始具有 ANIMID_WALK_ANIM 动画。
- 铁桶僵尸具有 1300 点 HP。（铁桶 1100 点，僵尸 200 点）

◆ **当 Update()时：**

1. 铁桶僵尸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. 若铁桶僵尸当前正在行走，它将向左移动 1 像素。若铁桶僵尸当前正在啃食植物，它将不会移动。
3. 若铁桶僵尸的 HP 小于等于 200 点，它将失去铁桶。将它的贴图更换为普通僵尸的贴图。

◆ **当被点击时**

铁桶僵尸什么也不需要做。

◆ **当发生碰撞时**

- 如果与一颗豌豆发生碰撞，铁桶僵尸将会受到 20 点伤害，并让那颗豌豆死亡。
- 如果与爆炸发生碰撞，铁桶僵尸将会立即死亡。
- 如果与任何植物发生碰撞，铁桶僵尸将会：

- 如果当前正在行走，则进入啃食状态，并播放 ANIMID_EAT_ANIM 动画。
- 对那棵植物造成 3 点伤害。（这是每帧 3 点的持续伤害，即每秒 90 点）

◆ 撑杆跳僵尸(Pole Vaulting Zombie)

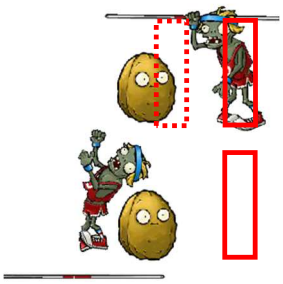
撑杆跳僵尸是《植物大战僵尸》的设计师 George Fan 最喜欢的一种僵尸，因为玩家第一次遇见它时常常会发生很有趣的节目效果：试着用坚果墙挡住它却被跳了过去。

◆ 当被创建时

- 撑杆跳僵尸的贴图编号为 IMGID_POLE_VAULTING_ZOMBIE。
- 撑杆跳僵尸的位置由创建它的代码决定。
- 撑杆跳僵尸的所在层级为 LAYER_ZOMBIES。
- 撑杆跳僵尸的宽度为 20，高度为 80。
- 撑杆跳僵尸初始具有 ANIMID_RUN_ANIM 动画。
- 撑杆跳僵尸具有 340 点 HP。

◆ 当 Update()时：

1. 撑杆跳僵尸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. 若撑杆跳僵尸当前正在奔跑，它将检测左方 40 像素位置是否有可以跳过的植物：
   a) 它会暂时将自己向左移动 40 像素。
   b) 询问 GameWorld，它是否在与一棵植物碰撞。如果是，他将：
      - 停止移动（为了和动画保持一致），播放 ANIMID_JUMP_ANIM 动画。在 42 帧后转为行走(ANIMID_WALK_ANIM)并向左“瞬移”150 像素（还是为了和动画保持一致，如右图所示）



   c) 不要忘了再向右移回 40 像素。
3. 若撑杆跳僵尸当前正在奔跑，它将向左移动 2 像素。若撑杆跳僵尸当前正在行走，它将向左移动 1 像素。若撑杆跳僵尸当前正在啃食植物或跳过植物，它将不会移动。

◆ 当被点击时

撑杆跳僵尸什么也不需要做。

◆ 当发生碰撞时

- 如果与一颗豌豆发生碰撞，撑杆跳僵尸将会受到 20 点伤害，并让那颗豌豆死亡。
- 如果与爆炸发生碰撞，撑杆跳僵尸将会立即死亡。
- 如果与任何植物发生碰撞，撑杆跳僵尸将会：

- 如果当前正在行走，则进入啃食状态，并播放 **ANIMID_EAT_ANIM** 动画。
- 对那棵植物造成 **3** 点伤害。（这是每帧 **3** 点的持续伤害，即每秒 **90** 点）

◆ 附录

◆ 碰撞处理

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

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。

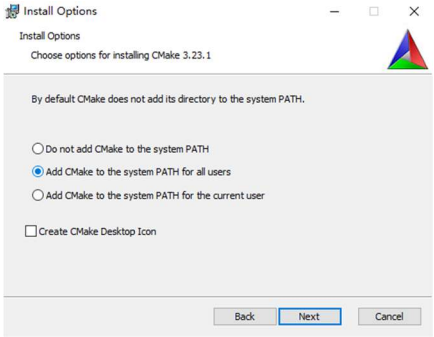
◆ 关于项目

我们提供的文件是一个 CMake 项目。CMake 是一个跨平台生成工具，可以用同样的源文件在不同平台(Windows, MacOS, Linux)下生成项目。

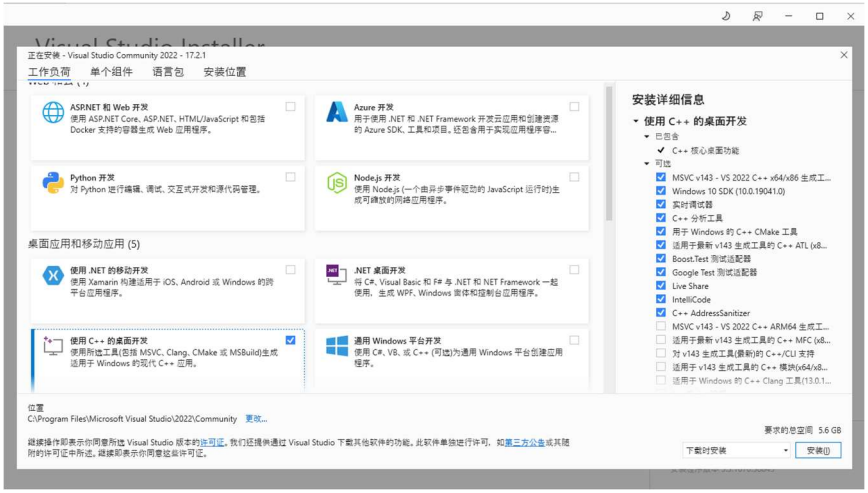
当你按下面的方法生成你的项目后，如果你能够运行，并且可以看到标题界面（按回车进入游戏后是黑屏），你就成功完成首次运行的设置了。在你开始逐步开工之后，每实现一个部分，记得运行一下游戏，看看新写的部分有没有为游戏带来 bug。不要等到写完所有部分才开始运行游戏！

◆ Windows 下首次运行

在 https://cmake.org/download/ 下载 CMake，选择 Windows x64 Installer。如果你希望（现在或以后）通过命令行使用 CMake，可以选择添加到当前用户或所有用户的环境变量。



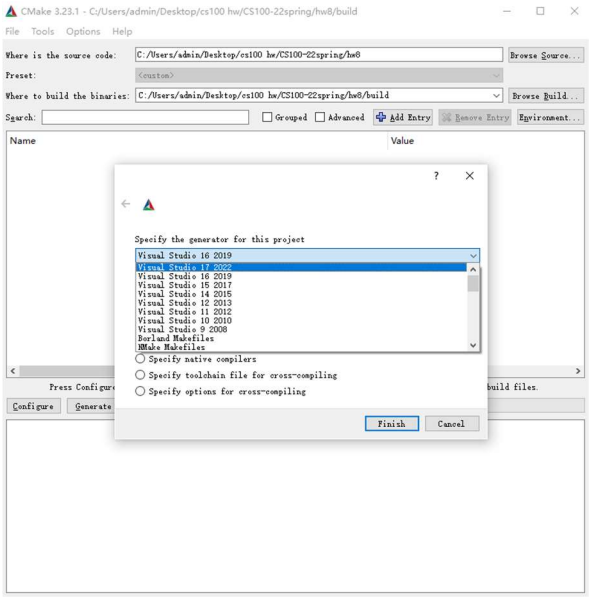
CMake 在 Windows 下的默认生成工具是 Visual Studio，生成的输出是一个 Visual Studio 工程文件(*.sln)。因此，如果你的电脑上没有安装 Visual Studio，请在

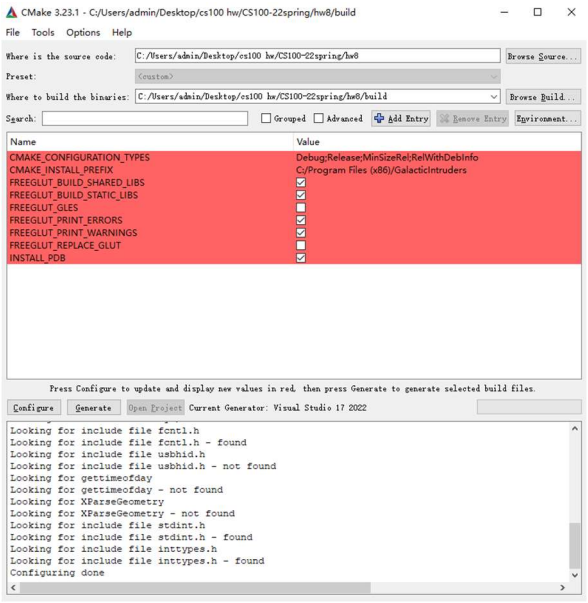


https://visualstudio.microsoft.com/下载，最新版本为 **Community 2022**。如果你已安装了 **2019** 或 **2017** 版，不必重新安装最新版。安装时，选择“使用 **C++**的桌面开发”。

安装后，运行 **cmake-gui**，在 **Where is the source code** 中填写项目的路径（**CMakeLists.txt** 所在路径），在项目里新建**/build** 文件夹，并填入 **Where to build the libraries**。

点击左下方 **Configure**。在下拉菜单中选择你的 **Visual Studio** 版本。设定完成后再依次点击 **Generate** 和 **Open Project** 打开 **Visual Studio** 项目。生成的这个项目为位于 **build** 目录下的 **PvZ.sln**，之后可以直接双击打开。





在右侧的解决方案资源管理器中右键点击项目**"PvZ"**，将它设为启动项目。单击右图所示按钮（仅 **2022** 版），或使用快捷键 **Ctrl+F5** 运行程序。如果你在代码编辑器中为你的程序中打了断点，其左侧的“本地 **Windows** 调试器”按钮（快捷键 **F5**）可以在 **debug** 模式下调试，还有逐语句**/**逐函数运行等用法。

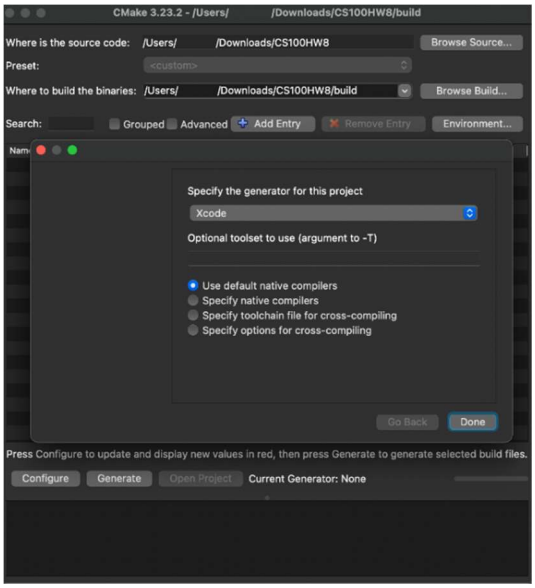
◆ **MacOS 下首次运行**

- Xcode 运行

若希望使用具有图形化界面的 CMake 或使用 Xcode 生成，可前往 https://cmake.org/download/安装 CMake。若也希望安装用于命令行的 CMake，可以执行安装后左上角菜单栏上“How to Install For Command Line Use”指令。

运行 cmake，在 Where is the source code 中填写项目的路径（CMakeLists.txt 所在路径），在项目里新建/build 文件夹，并填入 Where to build the libraries，点击左下方 Configure。在下拉菜单中选择 Xcode（仅测试过使用 Xcode，其他选项可自行尝试）。设定完成后再依次点击 Generate 和 Open Project。

打开项目后需要在标题栏“红色靶心”图标处将启动项目从 ALL_BUILD 改为 PvZ。另外，若程序运行时有"Error Loading [某个贴图]"输出，可以在 utils.hpp 中将 ASSET_DIR 改为 asset 文件夹的绝对路径。



- 命令行运行

前往 https://brew.sh/安装 homebrew，并终端输入“brew install cmake”。

首次运行在终端中打开游戏目录（该目录包含 assets，CMakeLists.txt，src 和 third_party）。输入以下命令创建 build 文件夹，并编译项目：

```
mkdir build && cd build
cmake .. && make -j
```

之后，在命令行中输入下面这行命令即可运行游戏：

```
./bin/PvZ
```

每当你修改完你的代码，只要在 **build** 目录下输入**"make -j"**进行编译，并输入**"./bin/PvZ"**运行游戏即可。

◆ 多文件项目以及多个 CMake 编译目标

在本次作业中，你可能需要写出大量的代码。为了更好地管理自己的代码而不至于杂乱无章地堆在一起，我们要求你将代码分成多个(.hpp 与.cpp)文件，将不同部分的源代码分开存放。

在创建新的源代码文件时，若不修改 CMakeLists.txt，新的文件将无法加入项目中。我们也要求你将代码按类别分别放在不同的目录（文件夹）下，然后以“子目录”的形式将其添加至本项目的 CMake 目标。

- 在每个你创建的目录下，都需要新建一个 CMakeLists.txt 文件。在这个文件中，你需要使用 add_library 命令，将目录内的源代码编译成一个静态(STATIC)库，你可以随意为创建的库命名。
- 同时，若一个子目录内的源代码需要#include 来自其他子目录的头文件，将需要的子目录的路径写在 target_include_directories 命令中。
- 若一个子目录依赖其他子目录的库，将依赖的库的名字写在 target_link_libraries 命令中。
- 最后，打开根目录的 CMakeLists.txt，使用 add_sub_directory 命令将你的子目录添加进项目。
- 你也可以参考 Framework，GameObject，GameWorld 等文件夹里为你写好的 CMakeLists.txt 的写法。
- 针对 CMake 的使用，请观看我们发布的教学视频。

◆ 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。例如：
   - 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. 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. 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. 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**(accessor)**与修改函数**(mutator)**。

5. 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。例如：

   - 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. 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。例如：

   ◆ 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。
  };
```

◆ **提交**

请前往 OJ 提交，提交方式见 OJ 页面说明。

本次作业的分值分为两部分，OJ 评测（上限 80 分）、支线目标（下限 0 分，上限 20 分）。

OJ 评测提供了大于 120 分的单元测试，超过 80 分的部分不计。

- 这就意味着你不必完成本文档中的所有项目。例如，你没有完成“樱桃炸弹”，那么只要保证你的游戏能够正常运行，你就可以得到其他正确部分的分数。

支线目标是若干个 Yes/No 的问答题，你将需要根据你是否完成了目标如实回答。每个问答题的分值分为三个档位，例如+3/+1/-3，规则如下：

- 若你的回答为 No，你将获得+1 分。
- 若你的回答为 Yes，并且经过线下 check，你如实完成了目标，你将获得+3 分。
- 若你的回答为 Yes，而线下 check 中发现你并未完成目标，你将获得-3 分。

你须对你在支线目标中的回答负全部责任，包括正确理解目标内容。“我忘记检查以为我做到了所以误选了是”等类似理由将不被接受。

在截止日期结束后，我们会立刻查重，并组织线下 check。（别紧张，说白了就是让你们当场来玩自己做出来的游戏）

若你不参加线下 check，视为你同意我们有权利决定你将获得的分数。例如，视查重情况与对支线目标的检查等，你可能获得你 OJ 上得分的 0%至 100%。

◆ **支线目标：**

1. (+3/+2/-1)我使用的指针全部为智能指针。

2. (+3/+1/-2)我的 GameWorld 中只有一个存储 GameObject 的容器，并且没有把它交给其他类管理（没有返回一个容器的成员函数）。（可以有其他单独存储的对象，并且这些单独存储的对象类型可以是 GameObject 的子类。）
3. (+3/+0/-1)我将代码按类别分在了不同的文件里。
4. (+2/+0/-1)我按类别分了不同的子目录，并为每个子目录创建了 CMakeLists.txt，以 add_subdirectory 的方式添加进了项目。
5. (+3/+1/-2)在我写的代码中搜索 shared_ptr<，尖括号里面的类型名只有 GameObject 和 GameWorld。（需要目标 1 回答为 Yes，make_shared<>不需要符合该要求。提示：使用 make_shared 来创建新的智能指针，而非 shared_ptr()）
6. (+5/+2/-4)我没有将成员变量定义成 public。我写的所有类的所有成员变量均为 protected 或 private。
7. (+3/+2/-2)我没有存储或获取 ImageID。在我写的代码中搜索== IMGID 得不到结果。
8. (+3/+1/-3)我的代码里没有使用神秘数字(magic numbers)来区分类型。（例如，1 代表植物、5 代表铁桶僵尸、0 代表背景等，不论神秘数字代表的是大类型还是具体对象。如果你有，可以使用 enum class 来替代。）
9. (+3/+1/-1)对于相似类的类似功能（例如，各种种子的 OnClick、各种僵尸的 Update），我没有重复代码，而是在基类里统一实现。
10. (+1/+0/-1)在碰撞检测中，我始终没有判断过对象的具体类型，包括豌豆与爆炸。（提示：豌豆和爆炸的层级相同，所做的事情也相似，是否可以归为同一基类？）
11. (+1/+0/-1)我使用了模板类解决“隐形的兔子”问题。