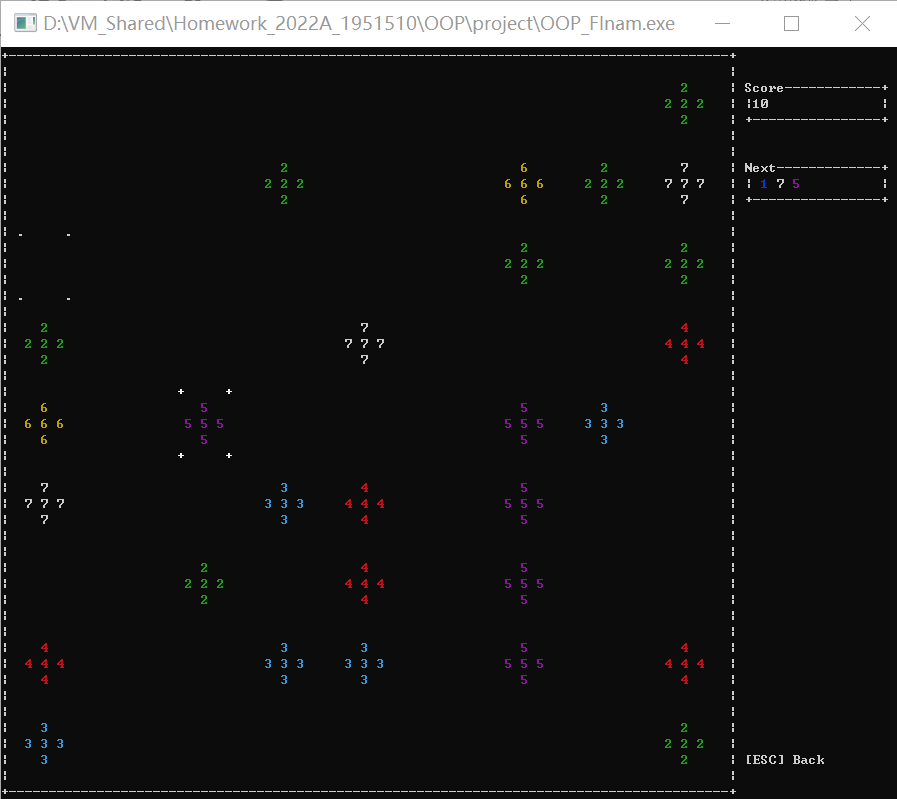
面向对象程序设计 42042002

课程项目文档

****

学号：1951510

姓名：姜文渊

2022年7月23日

1. **游戏使用说明**
   1. **编译该项目**
      1. **使用 VS2022 编译**

在安装有 VS2022 的平台上，直接打开项目目录下的 OOP\_Final.sln 解决方案，选择合适的构建方式（如 Release/x64），然后直接生成解决方案即可。在对应的目标文件夹下可以找到生成的可执行文件，即说明编译成功。

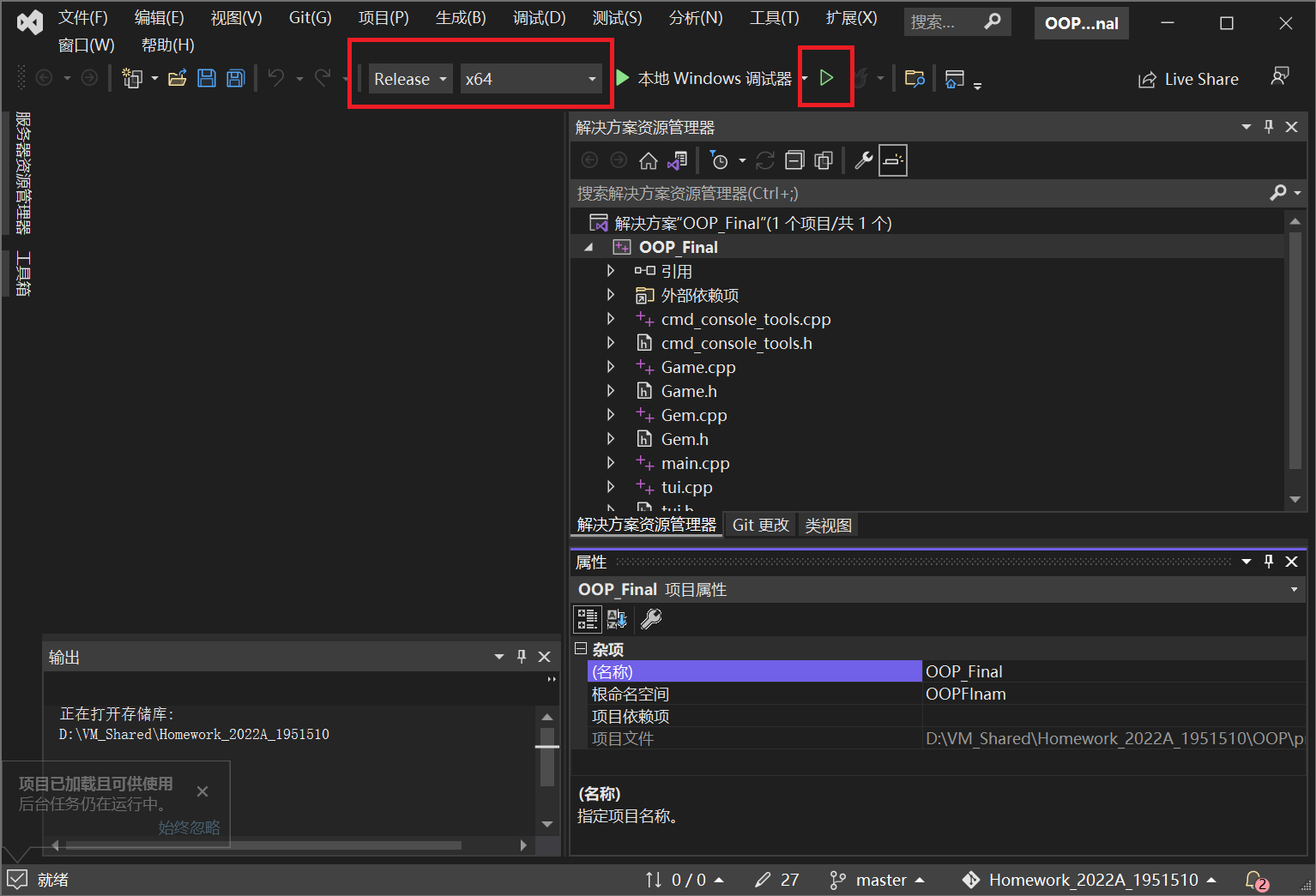


图 1：打开解决方案后的 VS 2022 设置

虽然笔者在自己的x64平台上 编译做到了 0 Error，0 Warning；但在编译过程中，可能会因为平台特异性问题产生一些 Warning ，只要不影响最后的生成，可以暂时忽略。

* + 1. **使用 Dev C++ 编译**

若用户更喜欢使用 Dev C++ 作为其 IDE ，或用户需要编译为静态链接的可执行文件（即可以在一些没有配置 VC++ Runtime 的平台上运行），则其可以使用 Dev C++ 进行编译。笔者推荐使用的 Dev C++ 版本为 5.11 ，推荐对应的 gcc 版本为 9.2.0 。

使用 Dev C++ 打开项目目录下的 OOP\_Final.dev ，选择合适的构建目标版本（推荐 32bit Release），然后按下 F12 进行 Rebuild All 动作，即可完成构建，可执行文件会生成在项目目录下。

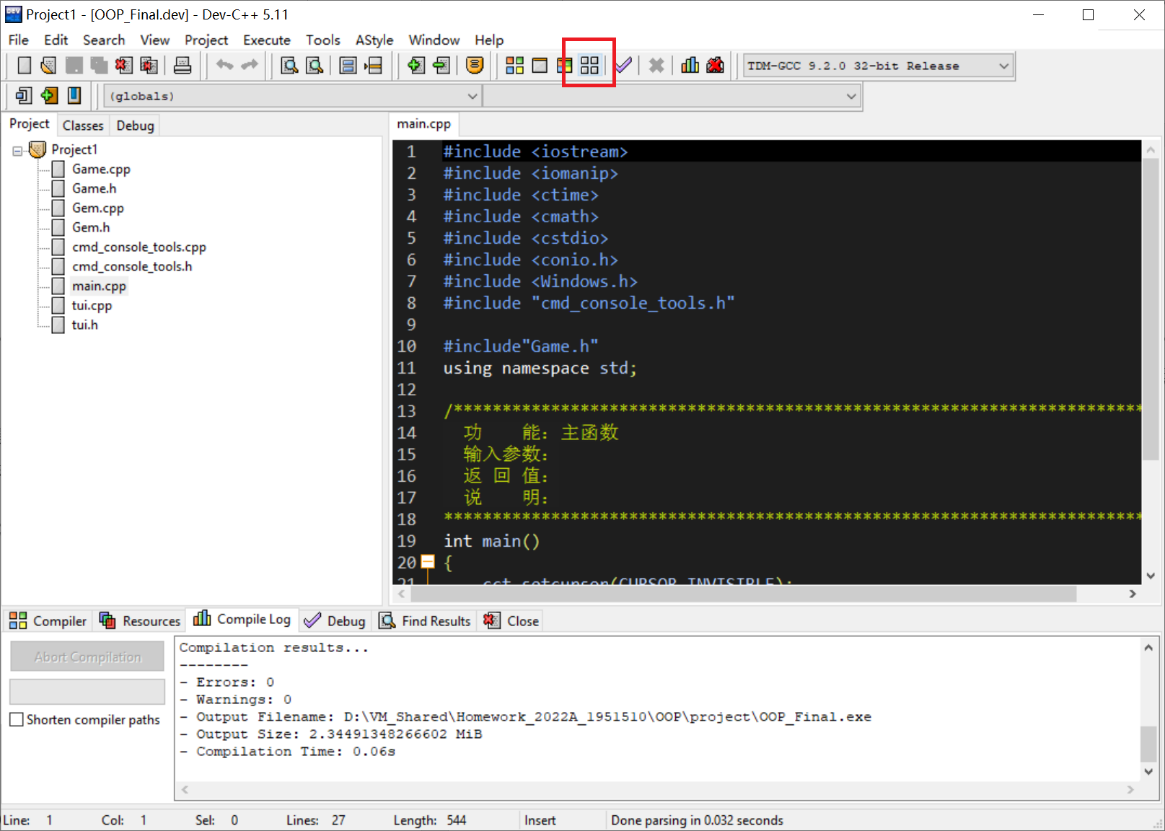


图 2：使用 Dev C++ 的编译方法

已知的 Dev C++ 在编译时会报一个关于宏的警告，这个警告在 VS2022 的编译过程中没有出现，初步认为是 gcc 和 msvc 的差异所致，因为不影响程序的总体功能，所以暂时不进行排查。

* 1. **运行项目**
     1. **运行前准备**

若您使用的是 VS 2022 编译生成的可执行文件，在没有配置 VS 2022 的设备上运行可能会出现缺少动态链接库的情况，此时需要根据具体的动态链接库版本进行配置，但最方便的方法是安装 VS 2022 ，这样一般不会出现动态链接库的版本冲突问题。

若使用的是 Dev C++ 产生的静态链接的可执行文件，则无需进行额外的链接库的配置。

* + 1. **终端设置**

由于本项目主要使用的是由 Windows 终端提供的鼠标服务进行的操作，故而在打开可执行文件后，需要进行如下的配置。首先在窗口的标题栏处右键，选择属性，然后按下图设置，去除“快速编辑模式”和“插入模式”，并且选择使用旧版控制台，重新打开可执行文件即可。



图 3：控制台的设置

* 1. **使用说明**
     1. **进入与退出游戏**

设置完成后，确保此时电脑的输入法为 英语（美式键盘），然后进入游戏的开始界面，如下图所示：

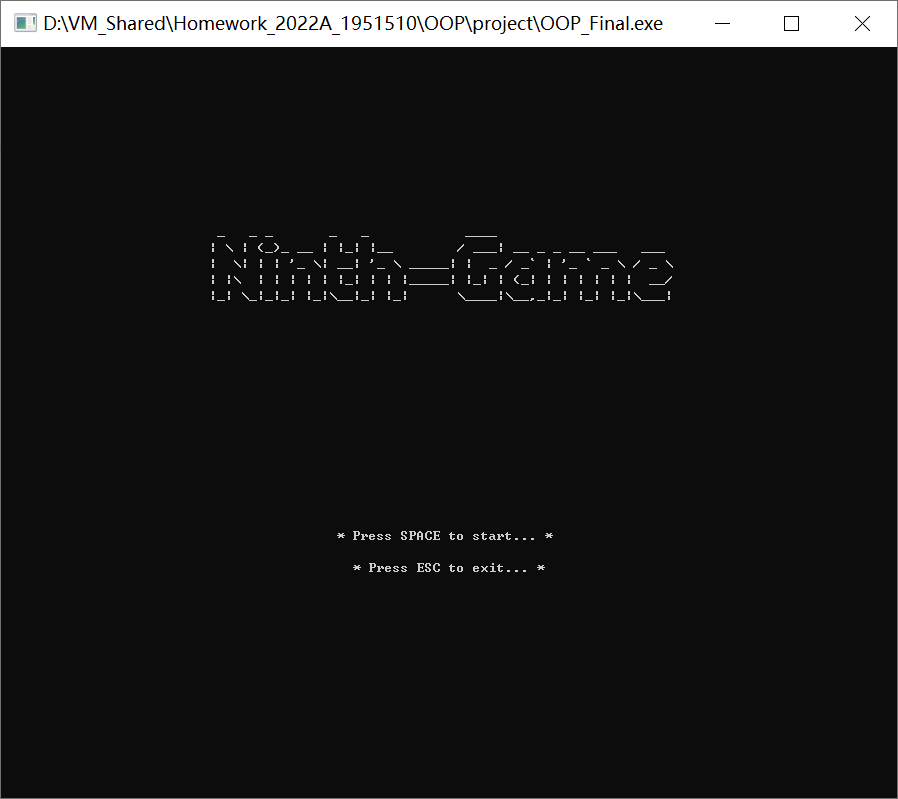


图 4：初始的游戏界面

按照提示，按下空格键进入游戏，按下 ESC 键退出游戏，在进入游戏后，按下 ESC 键，即可结束本局游戏，查看结果后按下空格键即可返回开始界面。

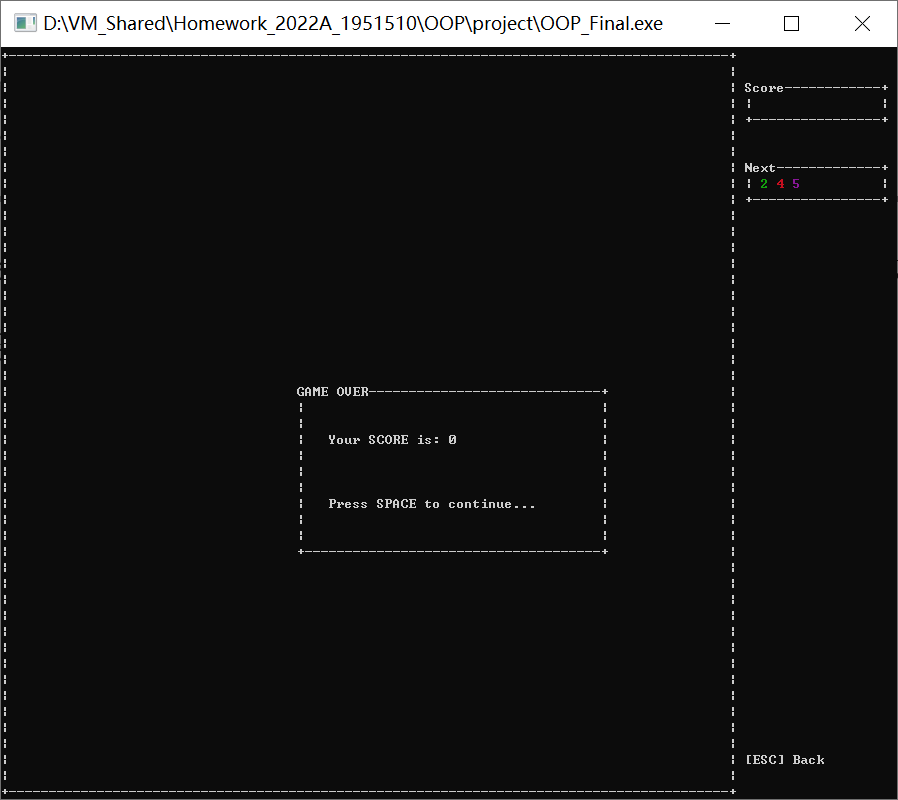


图 5：游戏结束界面

* + 1. **玩法说明**

游戏中，使用左键选择珠子并用左键指定移动位置，选择的珠子可以用右键取消选择。

游戏规则如下：

1，一次只允许移动一个珠子。要求必须存在可移动路径时方可移动珠子，即在移动路径上存在珠子阻挡就无法移动。

2，每移动一颗珠子以后，如果不满足同色的5颗珠子相连，将会出现3个随机颜色珠子分布到棋盘任意空置的位置上。如果同色的珠子能有5颗连在一起排成横向、纵向或者斜向时，这5颗珠子从棋盘上消失，不产生3颗珠子。同时游戏者可以得10分。

3，当同色的珠子有6颗连在一起排成横向、纵向或者斜向时，游戏者可以得12分。同时6颗珠子从棋盘上消失。（即：在同一方向上连在一起的珠子每增加一颗，游戏者多得2分，依此类推。）

4， 如果移动一个珠子之后，有两个方向都可以同时消除（即：任何单一方向上的同色珠子数至少为5颗），则两个方向的所有珠子都消除。按每个珠子2分获得分值。

5，两个方向同时消除的规则，同样适用于三个或四个方向。

6， 如果系统随机产生的珠子正好能凑成了同色的5颗及以上一起排成横向、纵向或者斜向，则这几颗同向的珠子自行消除，游戏者得分。

7，当棋盘被珠子占满时游戏结束。

1. **整体设计思路**
   1. **项目结构**

本项目的文件组织结构如下图所示：

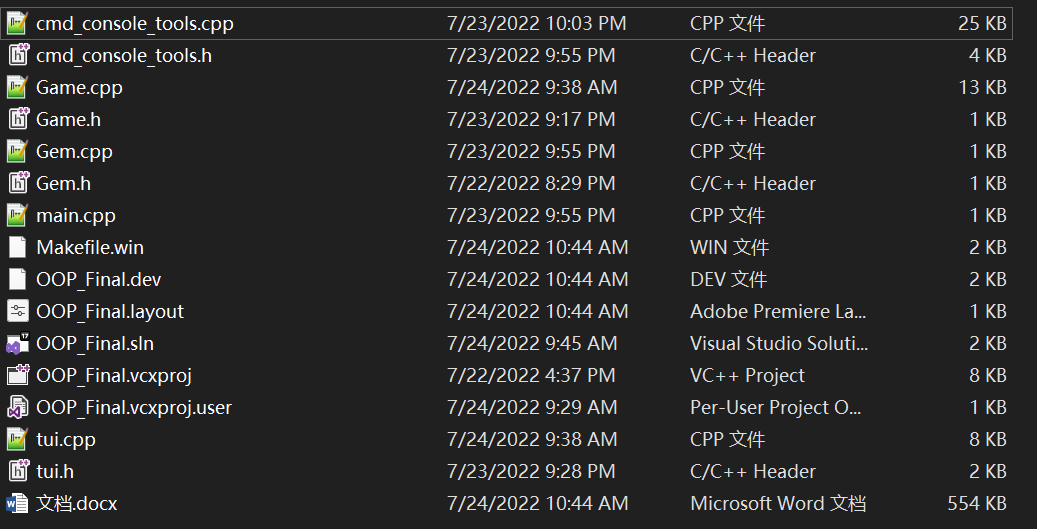


图 6：项目文件

各 \*.cpp 和 \*.h 文件均为源码文件，除了 main.cpp 外， 其余的 \*.cpp 与 \*.h 均成对出现，用于实现和声明类或者函数。

* 1. **类的设计**
     1. **基本元素**

为了描述棋盘上的珠子，笔者设计了一个 Gem 类用于存储与珠子相关的信息，如其坐标，颜色和消除后的得分等。

为了方便判断两个珠子的颜色是否相等，笔者重载了 Gem 类的 == 运算符，使之颜色相等即相等。

* + 1. **游戏逻辑**

为了统一管理游戏逻辑，笔者设计了一个 Game 类用于控制游戏进行的过程。

Game 类中的私有方法用于实现游戏内部的逻辑，例如寻找移动的最短路径或判断游戏是否结束等。

* + 1. **界面渲染**

为了统一管理游戏图形界面的绘制，笔者设计了一个 tui （textual user interface）类，用于提供绘图的接口并统一管理界面的渲染。

Tui 类提供了一个缓冲区，用以存储即将绘制的界面；除了可以供其他对象直接操作缓冲区外，tui 类还提供了一系列使用的函数用来方便的绘制缓冲区。缓冲区会在绘制到屏幕上后被重置。

* 1. **运行流程**

为了统一管理游戏图形界面的绘制，笔者设计了一个 tui （textual user interface）类，用于提供绘图的接口并统一管理界面的渲染。

Tui 类提供了一个缓冲区，用以存储即将绘制的界面；除了可以供其他对象直接操作缓冲区外，tui 类还提供了一系列使用的函数用来方便的绘制缓冲区。缓冲区会在绘制到屏幕上后被重置。

1. **程序功能与二次开发的描述**
   1. **修改游戏参数**

游戏中关于一些常量的设置在各头文件的宏定义中，例如：颜色的数量，起始的珠子数量，每次新增的珠子数量等；如果需要修改这些参数，则可以在源码中对对应的宏定义进行修改。

各常量如下列表所示：

1. Gem.h #define COLORS 7 定义珠子颜色的总数
2. Game.h #define START\_GEMS 3 起始珠子数量
3. Game.h #define STEP\_GEMS 3 每步新增珠子数量
4. Game.h #define ELIM\_GEMS 5 至少触发消除的珠子数量
5. Game.h #define ELIM\_SCROE 2 每个珠子的消除默认分数
6. tui.h 开头的定义与界面绘制相关
   1. **添加图形界面元素**
      1. **直接在缓冲区内绘制**

由于 tui 类使用的是单例模式（即整个程序运行时），故而在绘制图形界面前，需要先获取 tui 类的单例的引用：

**tui& tui0 = tui::get\_instance();**

此后，利用 tui 的公共缓冲区，将需要绘制的内容加入到缓冲区中。缓冲区中的每一个元素都是一个结构体 pixel ，存储了字符和颜色。一个例子如下：

tui0.frame\_buf[TUI\_BAR\_Y + 1][TUI\_BAR\_X + i + 1].ch = buf[i];

在下一次 render 时即可将放入的内容。

* + 1. **使用 tui 类提供的函数进行绘制**

除了直接操作缓冲区外，可以使用 tui 类提供的方法来方便的操作缓冲区，一些方法的使用方式如下：

1. void add\_gem\_to\_buf(int x, int y, int c); 将某个颜色的宝石放置到缓冲区。
2. void add\_rect\_to\_buf(int x, int y, int w, int h, bool clear = false); 在缓冲区中绘制方框，并选择是否清空框内内容。
3. void add\_rect\_to\_buf(int x, int y, int w, int h, const char\* s, bool clear = false); 绘制带标题的方框。
4. void add\_text\_to\_buf(int x, int y, const char\* s); 在缓冲区中绘制字符串。
5. void add\_focus\_to\_buf(int x, int y); 在缓冲区中绘制鼠标焦点。
   * 1. **绘制并刷新缓冲区**

在缓冲区中加入内容后，需要调用 tui 的 render 方法才可以将内容增量更新到终端屏幕，而 render 会根据传入的参数对缓冲区进行重置，例如：

在 Game::render 里，操作完成缓冲区后会调用 tui0.render(); ，使用其默认的参数对缓冲区进行重置，此时 tui 会对屏幕进行绘制，并且将缓冲区重置为游戏的界面，等待下面对缓冲区的操作。

如果时重新绘制主界面或其他界面，则重置缓冲区使用的参数不为默认参数，详情参见源码中的注释部分。

* 1. **游戏功能对应的代码**

实现具体的游戏逻辑的代码主要集中在 Game 类的实现中，按游戏规则列举如下：

1. 一次只允许移动一个珠子。要求必须存在可移动路径时方可移动珠子，即在移动路径上存在珠子阻挡就无法移动。这部分代码主要涉及包装一次移动的 Game::operation 、寻路算法 Game:: path\_finder 和绘制移动过程并真正执行移动的 Game:: move\_gem 。
2. 每移动一颗珠子以后，如果不满足同色的5颗珠子相连，将会出现3个随机颜色珠子分布到棋盘任意空置的位置上。如果同色的珠子能有5颗连在一起排成横向、纵向或者斜向时，这5颗珠子从棋盘上消失，不产生3颗珠子。同时游戏者可以得10分。记录分数的变量为 Game 类的私有变量 score ，在一次操作中，Game::operation 会负责调用 Game::mark\_elim 、 Game::elim\_gem 等函数，并生成新的珠子，从而完成整个过程。
3. 当同色的珠子有6颗连在一起排成横向、纵向或者斜向时，游戏者可以得12分。同时6颗珠子从棋盘上消失。（即：在同一方向上连在一起的珠子每增加一颗，游戏者多得2分。依此类推。）这个功能主要时在 Game::mark\_elim 中实现的，通过对每个珠子的四个方向进行遍历，从而判断该珠子是否需要消除，记录下位置后 Game::operation 会调用其他函数完成消除的工作。对于 Game::mark\_elim 的修改可以更改消除的触发条件。
4. 如果移动一个珠子之后，有两个方向都可以同时消除（即：任何单一方向上的同色珠子数至少为5颗），则两个方向的所有珠子都消除。按每个珠子2分获得分值。同样主要是靠 Game::mark\_elim 实现的。
5. 两个方向同时消除的规则，同样适用于三个或四个方向。同 4.
6. 如果系统随机产生的珠子正好能凑成了同色的5颗及以上一起排成横向、纵向或者斜向，则这几颗同向的珠子自行消除，游戏者得分。在 Game::operation 生成完成后，再进行一次消除即可实现该功能。
7. 当棋盘被珠子占满时游戏结束。这个判断是由 Game::is\_end 进行的。
   1. **修改逻辑功能**

修改逻辑功能时，一般需要调整 Game::operation 和 Game::session 中的内容，具体的各功能的实现需要参考源码进行修改，这里旧不赘述了。

1. **各模块的体系结构设计和描述**
   1. **类 Gem**
      1. **类的结构**

该类主要用于提供存储珠子的数据结构，并不承担很多实际的计算逻辑，其成员如下图所示：



图 7：Gem 的类图

* + 1. **设计细节描述**

主要功能的实现在其构造函数中，默认构造函数为生成随机位置随机颜色的珠子，但也提供了一个重载，允许用户指定其各项参数。重载的 == 运算符用于方便比较相同颜色的珠子。

除了上面的功能外，笔者还预置了一个 score 私有成员，用以方便后续开发人员扩展功能，为不同的珠子的消除附上不同的得分，不过在当前的版本中，这个功能并没有实装。

* 1. **类 Game**
     1. **类的结构**

该类是游戏主要逻辑的承担者，除了少量的公共接口外，大部分函数都是为了内部实现逻辑而使用的私有成员，其成员如下图所示：



图 8：Game 的类图

* + 1. **设计细节描述**

其构造函数中，可以通过指定随机化种子来使得生成的内容具有一致性；默认的构造函数会用系统时间作为随机数种子。

Game::arr 是存放 Gem 对象的重要数据结构，每次对其操作都需要使用 Game 的私有方法进行，例如 get\_gem ，mark\_gem ， elim\_gem 和 move\_gem 等；这些私有方法保证了 arr 中的坐标不重复，并且提供了对 Gem 类的一个较好的封装。注意，这里对 Gem 类的抽象并非是在 Gem 类中实现的，而是在使用 Gem 的类中实现的，即若是后续版本需要添加一种新的珠子，只需继承 Gem 类，而不需要修改 Game 类中的实现，使得开发可以增量地进行。

Game类中的公共方法 render 是在终端绘图功能的实现，render 会调用后面提到 tui 类进行绘图，通过调用 tui 类的方法或直接操作缓冲区来实现绘制。

Game类中的公共方法 operation 是每次用户操作后的逻辑处理部分，通过 operation 来对 Game 类中的大量私有方法进行一个封装（类似 facade pattern），提供给外部唯一的接口。

Game类还附带了一个 session 公共方法，用于管理整个游戏的进度并处理用户输入。这部分代码在后续更换了用户界面（例如，使用了其他图形库，而非终端的tui）后，需要进行重构。

* 1. **类 tui**
     1. **类的结构**

该类主要用于提供存储珠子的数据结构，并不承担很多实际的计算逻辑，其成员如下图所示：

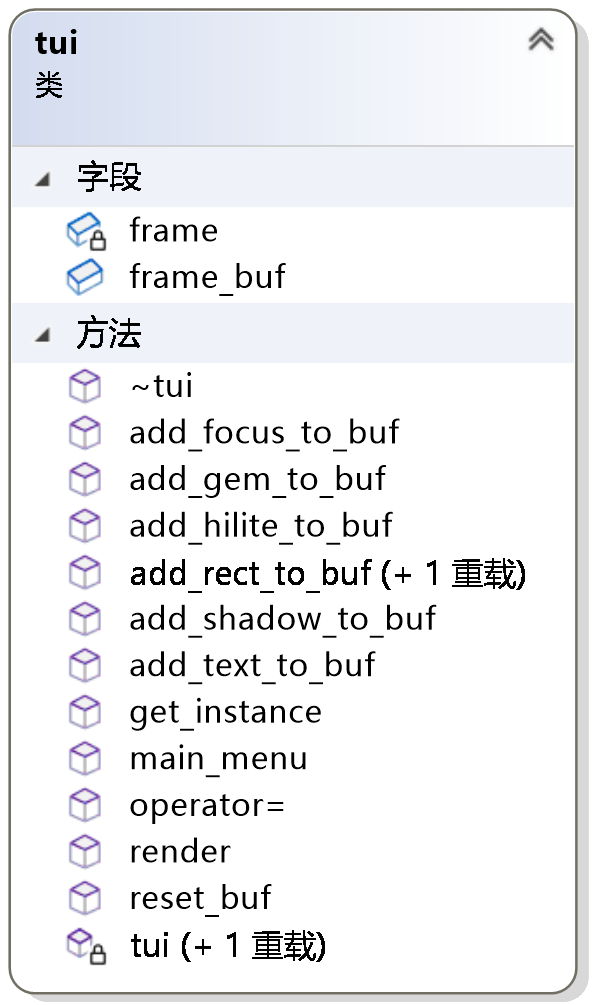


图 9：tui 的类图

* + 1. **设计细节描述**

由于游戏只有一个终端可以使用，故而笔者的 tui 类使用了单例模式。其意图是保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。与其他类不同，该类的构造函数是私有的，而该类的实例则是局部静态成员，在程序运行时就被私有的构造函数构造。获取该类的唯一实例则是通过公有的静态函数 static tui& get\_instance(); 来进行的。这样保证了全局只有一个 tui 类的实例存在。值得注意的是，使用局部静态成员实现的单例模式是 C++ 11 标准及之后的标准才支持的，在之前的标准中，局部静态成员的初始化是相对较为不确定的一个行为。这种方法也被称为Meyers' Singleton，是在《Effective C++》中提出的。

tui 类的主要功能是将其公共成员 frame\_buf 增量绘制到屏幕上。为了实现这个功能，私有成员frame 会在每次绘制时被更新，从而确保只有与 frame\_buf 不同的部分被绘制，这样就可以得到较高的帧率并且避免终端屏幕的闪烁。

绘制使用的方法是 render 方法，在 render 时，会比较当前 frame 和 frame\_buf 的不同之处，只将不同的部分进行绘制，并且更改 frame 使之与 frame\_buf 相同；在绘制完成后， render 方法会重置 frame\_buf ，从而等待下一轮添加的绘制元素。

除了上面的核心功能外，为了方便开发， tui 类中也加入了一些用于操作 frame\_buf 的函数，但这些函数本质上都可以通过直接操作 frame\_buf 来实现，故而不是说明的重点。

1. **应说明的其他事项**
   1. **项目特色**

本项目的特色点如下所述：

* 1. 友好的用户操作方式：
     1. 游戏主体为鼠标操作，退出等不常用操作为键盘操作，在方便用户游玩的同时防止误操作。
     2. 图形界面为终端伪彩色，且有选择框和相应的动画，并且会提示即将生成的三个珠子的颜色。
     3. 界面对色盲人士友好，即便在黑白的视野下也能通过珠子的编号分辨不同的珠子。
     4. 拥有基于蜂鸣器的红白机风格的游戏音效。
  2. 可扩展的项目结构：
     1. 类的设计尽可能解耦合，且提供后续预留的接口。
     2. 提供不必要但较为方便的成员函数。
     3. 提供易于修改的宏定义常量。
  3. 较成熟的面向对象编程思想：
     1. 合理的设计模式的使用（单例模式、类似正面模式等方式）。
     2. 组合优于继承原则在设计上的体现。
     3. 依赖倒置的应用即一些 SOLID 原则的体现。
  4. **整体设计的改进点**
     1. **图形界面绘制的结构化**

实际上，一些成熟的游戏引擎会使用树形数据结构来管理图形界面的绘制，即定义一个抽象类 ui\_node 提供 render 接口和儿子节点的数据结构，通过对该类的不同继承实现一个树形的数据结构，例如：根节点下拥有不同的窗口，窗口中拥有分数的字符串和珠子的表示。

调用 render 时，render 会调用其子节点的 render ，然后再进行自己需要对 frame\_buf 的操作，从而使得界面的可扩展性更好。需要添加新的图形元素时，只需继承 ui\_node 创建新的类即可，然后将该类加入到渲染树中，而无需对渲染的代码进行修改。

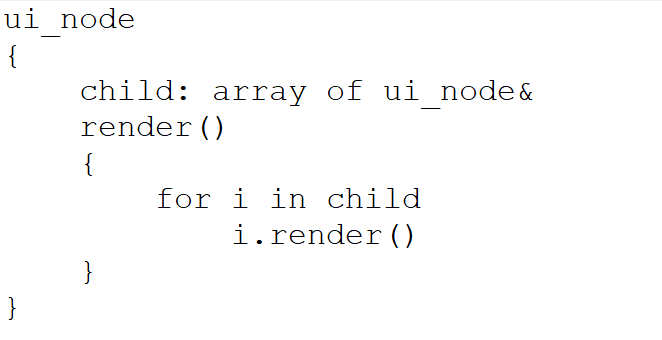


图 10：改进渲染方式的伪代码

* + 1. **用户输入的统一管理**

受限于项目的时间，笔者没有对用户的输入进行统一的处理，而是仅在 Game 类中添加了 session 方法用于管理游戏的流程。实际上，为了后续开发的方便起见，有必要将用户输入也使用一个类似tui的单例模式的类进行处理，从而减少项目各部分的耦合度。

1. **小结与心得体会**

在本项目中，笔者通过使用面向对象编程的范式，完成了项目的基本要求，并实现了项目的额外要求，例如：鼠标操作、寻路算法和音效等。

在项目的实现过程中，笔者；练习了常见的面向对象编程使用到的技术，除了基于 C++ 的一些常规写法外，笔者也有机会实现了一些与编程无关的面向对象的技巧，例如组合代替继承、单例模式、门面模式和依赖倒置等。将这些较为理论的技巧应用到实际的项目中去的过程，加深了笔者对于面向对象编程范式的理解，也让笔者体会到了面向对象编程范式相对于传统的结构化编程范式的优越性。

然而，在实现项目的一些功能时，笔者也遇到了一些问题，其中大部分问题从根本上来讲是面向对象编程范式的固有问题。例如，在一些使用了其他类对象的类的成员函数的实现中，需要时刻注意被使用的对象的状态变化。在实际的场景中，对于一些较为复杂的功能的实现，完全使用常值引用和常成员函数是很难进行的，故而对象的状态管理成为了面向对象编程的一个重要的问题。这个问题在单人的项目中较轻微，而在多人合作的项目中则十分明显，即便是有详细的文档，很多情况下开发者仍然很难找出在运行时对象状态变化的原因。

因而，在现代的一些编程实践中，一些项目开始逐渐只使用静态状态，即所谓的不可变状态（immutable）。像Rust等较为激进的编程语言，默认的对象就是不可变的，且可变对象的使用需要严格遵守控制权的管理，故而通过给开发者施加一些严格的限制来避免项目增长为一个无法调试、修改与管理的历史遗留。