一、題目：

實作一個 AI 及 GameRunner 來遊玩/對戰海戰棋（BattleShipGame)，AI 負責計算船的移動及攻擊，而 GameRunner 負責檢測 AI 的指令是否合法，遊戲一開始，雙方分別有 4 艘船（3x3,3x3,5x5,7x7)，每回和，攻擊方可以攻擊己方活著的船數，並會得到是否擊中船，然後再移動一小格的船，被打中的船無法移動，擊中核心（中心點）的船為死掉。輪流攻擊直到某方無活著的船。

二、工作分配

（1）進度規劃：

5/31 ： 分組完成，並於當日下午進行初步攻擊戰術討論
6/18 ： 組員期末考結束，約定時間進行討論
6/20 ： 工作分配完畢，開發環境安裝完畢
6/21~22 ： 各自完成任務
6/23 ： 統整、統一編譯、debug
6/24 ： 進行 demo、結束後進行 report 製作

（2）分工：

蔡定芳 ： Bonus(圖形化介面)
王鈞 ： AI 進攻(callbackreporthit、queryWhereToHit)
黃淳侑 ： AI 移動(callbackreportenemy、queryHowToMove)
文逸雲 ： gamerunner

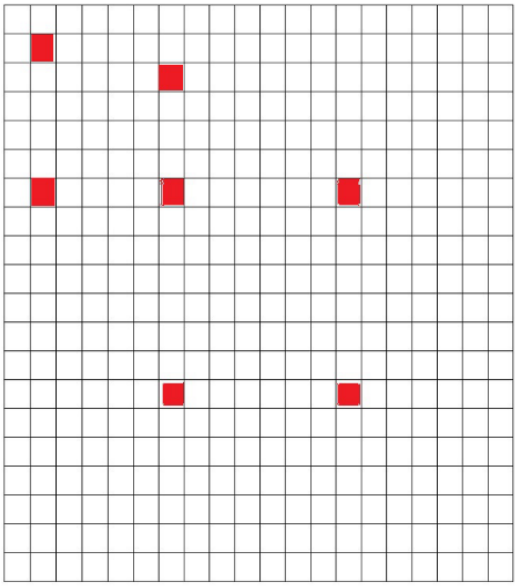
三、AI ：進行進攻及移動船

（1）進攻：

由於棋盤的大小是 20*20，所以在(6, 6)、(6, 13)、(13, 6)、(13, 13)這四個點之中，必定有一個點是屬於 7*7 那艘戰艦的，且任兩點不會屬於同一艘船，所以在最初的一波攻勢就先攻擊這四個點，另外，我們考慮到有些人可能會沒有修改初始的位置，所以第二波攻勢會攻擊初始的四艘船的核心，也就是(1, 1)、(2, 7)、(1, 6)、(13, 13)，不過(13, 13)已經攻擊過了就不再攻擊。

並將有攻擊到東西的點記錄下來。

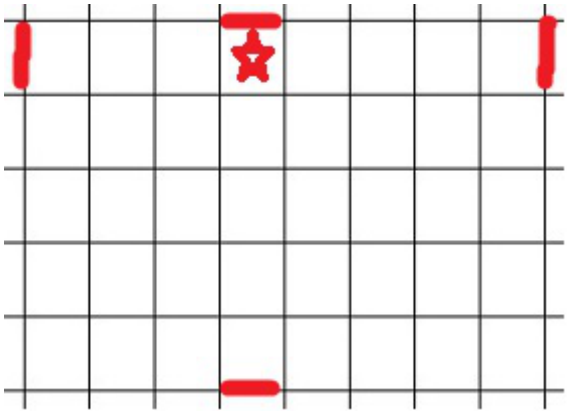
如圖所示：



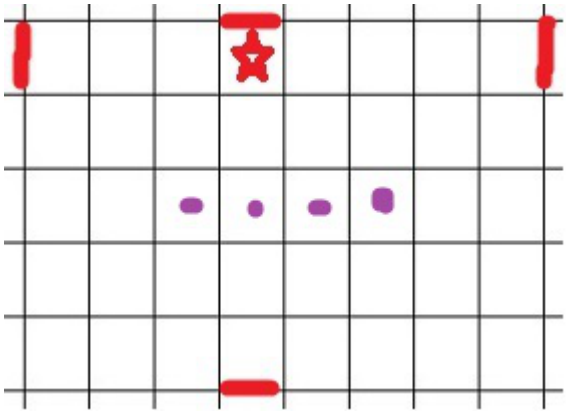
我們做了兩個版本的程式，前面的攻擊都是相同的。

第一版：

第一個版本的攻擊模式是首先把有找到東西的點上下左右邊界的判斷，如果最後為出來的四邊形是正方形，且邊長為 3、5、7，則直接攻擊他們核心的部分((左界+右界)/2, (上界+下界)/2)，並將整塊戰艦範圍自行記錄起來，之後不再攻擊(該戰艦已沉沒)；但如果是長方形，就有可能是兩艘戰艦有接觸，那就只攻擊其中一艘可能的核心位置，下面以找出的長方形為 5*8 的舉例：



以圖示狀況來說，星號為儲存到，有 Hit 到的點，我們往上下左右進行攻擊，求出邊界，但卻發現他是長方形，則我們只能確定他是一個 3*3 與一個 5*5 的有接觸，而我們不能確認確切怎麼排列，不過我們可以知道，星號一定屬於 5*5 的那艘船，則可以確定出 5*5 的位置可能，並攻擊那些可能性當中，所有的核心點，以上圖為例，就是紫色的那四個點。



攻擊結束以後，我們繼續看下一個有打到東西而被我們儲存的點，重複以上步驟

若沒有儲存點，則進行每 3 個一數的地毯式轟炸，確保沒有任何一艘 3*3 可能被遺漏或是經由移動來逃離，如果有發現任何 Hit，則進行以上步驟。

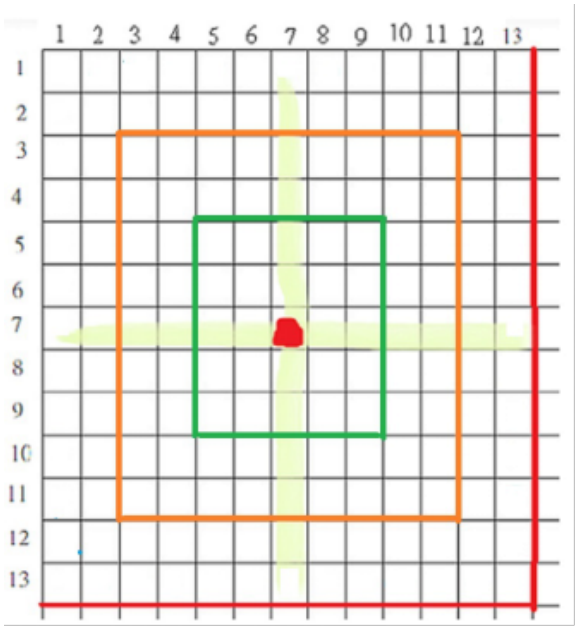
如果有遺漏的點，則原因會是因為以上的步驟中，我們即使找到了兩艘船，我們也只處理了一艘，所以產生遺漏，所以如果還是沒找到的船，直接採用一格一格的地毯式轟炸。

此一方法最終由於戰艦位置的組合性太多，無法完全展開，而以失敗告終，不過這為下一個辦法提供了良好的思維模式與進攻基礎。

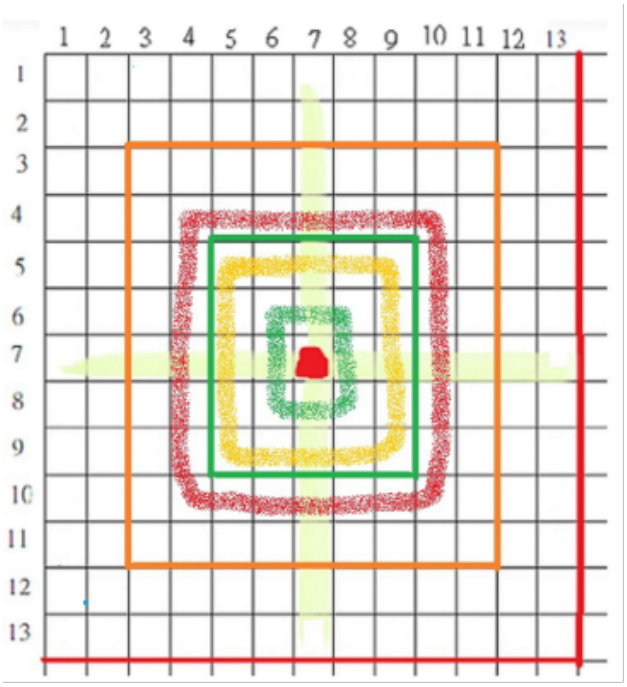
第二版：

第二個版本我們為了解決這個問題，採用了較暴力的方式，前面的前七點

都相同，儲存打過的點也相同，比較慢但是十分保險的攻擊法，如下圖所示：



此一方式的原理在於，如果紅點是被攻擊到的點，則 3*3 的戰艦最大範圍在綠色範圍內；5*5 的最大範圍在橘色範圍內；7*7 的最大範圍在紅色範圍內，知道了這件事以後，我們可以推出核心位置



我們可以知道 3*3 的核心可能落在綠色噴漆位置；5*5 的落在橘色噴漆位置；7*7 的落在紅色噴漆位置，因此，我們可以藉由轟炸這一個 7*7 範圍大小來保證會摧毀核心，並藉由 call_report_enemy 來推算對方剩餘的船數量，與前一次攻擊的數量是否相同，如果對方剩餘的船數量比上一次少，則我們的攻擊成功，另尋下一個點進行攻擊，找尋方式為 random。

效率分析：

當找到一個點為 Hit，且該戰艦並未與其他戰艦相連，第一個版本的 worst condition 為遇到 7*7，求上下界合計花 8 次進攻，左右界亦然(長度 7 - 已經擊中的 1 + 邊界外 1 點*2)，最後攻擊中心點，共計 17 次進攻時間可以摧毀一艘船；第二個版本的 worst condition 則是核心在我們寫的 for 迴圈中，會最後被判斷到的點，所以最糟狀況是 7*7-已經擊中的 1 = 48 次進攻，足足是第一個版本的 3 倍時間。

比較：

兩個方法優缺點十分明顯，第一版快，但是處理不了複雜的位置，程式也十分複雜；第二版慢，但是慢的很保險，只是第二版還有一個缺點，就是如果被攻擊過的點太多，random 一直選到同樣的點，就會重新選點，容易超時。

第一個版本最終因為只要遇到太複雜的形狀就會直接 crash 掉，只能執行地毯式轟炸，所以被我們放棄掉，重新弄出了第二個版本。

第二版也因為時間因素未能更加優化，例如判斷這個點打到的船是不是已經被轟炸過了，導致會做出滿多浪費時間的事情，倒是有些遺憾。

（2）移動船：

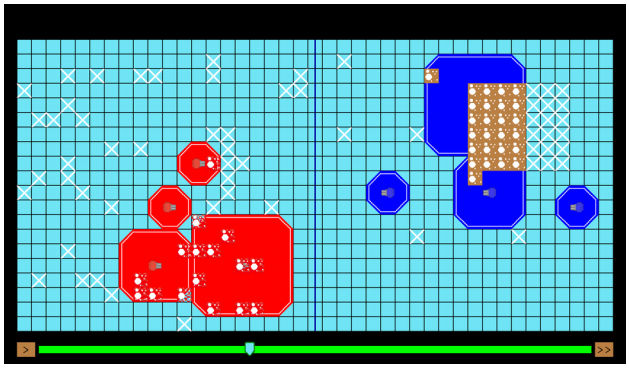
callbackReportEnemy ：

利用 Gamerunner 告訴我的敵方攻擊資訊，紀錄敵方剩餘存活的戰艦數目，以及我方已被攻擊過的區域，藉此避免移動時誤闖。

queryHowToMoveShip ：

利用 AI 中紀錄的地圖資訊，計算以九宮格劃分的我方地圖中，目前飛彈數量最多的區域，再讓戰艦往相反的方向逃離。移動前先判斷是否會出界、進入攻擊過的區域及與其他戰艦重疊。四、Bonus：GUI

我們的 GUI 是使用 Allegro 遊戲引擎進行底層的繪圖，我們是將 Mini Project 2 的 Template 和 Final Project 的 GameRunner 合併，並額外添加開始畫面（StartScene）和比賽的畫面（PlayScene）



（1）Engine ：

所有的物件（Scene、Image、ImageButton、Slider...）都繼承至 Object，並放進 Group 才能繪畫，每個 Object 都有一個函式叫 Update(float deltaTime)，輸入值為和上一次 Update 的時間差，單位為秒。

（2）比賽的畫面（PlayScene）：

我們比賽的畫面分成 3 層，由下而上分別為 Ocean(水藍色+黑格子+中間

深藍色分隔）、Ships(紅色與藍色船）、Hitted（被打過的點，包含 X 和汙點，分別為 State::Hit 和 State::Empty ）。

我們的海面、船、被攻擊的圖片都是一張張正方形，大小為海的水藍色＋黑框

Ocean ：只在初始化時加入 Group

Ships ：我們會把上一個船的位子存起來，若和上一次不同位子，就清空 Group 並從新加入船

Hitted ；每次更新（在 Update 中，每 1 秒會更新）時都會清空 Group 並從新加入被攻擊的圖片，其實可以只加入當次被攻擊的地方就可以，但是我們的 GameRunner 可以觀看之前的紀錄（下方的 Slider 可以往左右拉）

遇到問題：

在開發中時，我們試著將 Mini Project 2 的 Template 放在一個資料夾（Engine)中，但在編譯時，linker 會找不到定義（應該是實作的 object code 吧），試著改 Makefile 但沒成功，最後是和 main.cpp 放在一起。

另一個問題是如何將 GameRunner 和 GUI 合併，一開始，我們是將 PlayScene 和 BattleShipGame 合併，但會發生 Double Free ，可能是 Engine::GameEngine 會自動釋放資料吧，我們的解決辦法是將 BattleShipGame 當指標傳入 PlayScene。

五、問題與討論

（1） 請解釋下編譯指令，每一個參數代表的意涵：

```
g++-8 -std=c++17 -O2 -Wall -Wextra -fPIC -I./ -shared AITemplate/Porting.cpp -o ./build/a1.so
```

1. g++-8 ：指定要執行 g++-8(第 8 版的 g++)
2. -std=c++17 ：編譯器要支援 C++ 17
3. -O2 ：等級 2 的優化選項
4. -Wall ： 顯示所有的警告訊息
5. -Wextra ： 開啟更多不包含於-Wall 的 Warning
6. -fPIC ： 若系統支持，則編譯出與記憶體地址無關的機器碼，使得同一個程式庫能夠被載入到不同應用程式的
7. -I./ ： 指定編譯器在尋找#include <...>的資料夾
8. -shared ：編譯出 Shared Library
9. AITemplate/Porting.cpp ：要編譯的檔案
10. -o ： 編譯後輸出的檔名

（2） 請解釋 Game.h 裡面 “call” 函數的功能

call 函數的功用：使用 call 函數來呼叫 AIInterface 的函數的話，就可以保證他的運行時間不超過 runtime limit，也就是一秒。

（3） 請解釋什麼是 Shared library，為何需要 Shared library ，在 Windows 系統上有類似的東西嗎？

Shared Library（動態連結）是可以用來執行中才載入程式的一個方式。一般而言，編譯時若沒指定編譯為動態連結（-shared），則為靜態連結，linker 在連結時，會把所有用得到的 object code 合併，如果 5 個地方需要，就會複製 5 次，浪費許多空間。Shared Library 的好處是減少記憶體空間的

使用量（多出地方使用只需一份在記憶體裡）、方便更新（若有功能需要更新，只需要更換那部份的程式庫，大部份都不用更動），而 Windows 也有類似 Shared Library 的東西，副檔名通常是 dll,ocx(ActiveX),drv(舊式的驅動程式)

六、心得

（1）蔡定芳　　：

這次是我第一次和其他人一起團體寫 Project，之前有過想參加一些專案，但是通常是看不太懂其他人的 Code，每個人都有自己的風格，我也要試著不要管太多，因為我之前覺得自己寫才能把它做好，我也比較好控管整個專案的進度與品質（有點自我感覺良好，自己生出的 Bug 其實會更多且奇怪），但是 Final Project 後，第一次體驗到團體合作的效率與品質蠻高的，大家都要互相相信他人的 Code 的完善性。雖然第一次我們分配的工作量似乎不是那麼平均，但我們仍可以在 4 天內生出一個完整的專案。

我是負責 GameRunner's GUI 的部份，在整個專案裡，算是比較簡單的部份，大部份的 Code 是 Mini Project 2 （Engine 的部份）助教的 Template，不過藉由這次 Final Project，我是更了解 Template 在做什麼，上次只把需完成的部份複製、補上而已。

再最後一天再合併程式碼的時候，遇到不少問題，像是程式碼丟失（刪錯檔案，這是我的錯：（，因為專案放在 D 槽，Linux 的 ntfs 支援無法移至回收桶，所以都直接刪掉），有時候會不知道誰的程式碼是最新的，有點混亂。

（2）王鈞　　：

本次的 final project 中，我負責的部分是 AI 的進攻，一開始我感到興致勃勃，很開心地自己畫了圖，寫了進攻路線，結果實際在製作的時候，我發現有很大量的東西要注意，遠比自己腦袋思考還要多非常多，所以一開始製作的時候非常不順利。

經過了一次次的修正，我做出了第一個版本，不過在一次次的 debug 中，我發現了製作上的問題——船艦互相接觸的狀況，令我非常挫折，在最後衝刺的時候，由於有非常奇怪的 queue 的清空問題以及邏輯不適用於過多船艦接觸，導致程式當掉，在最後的時間重新製作了第二版本，用了比較笨、比較慢但是相對保險的辦法，但也因為時間因素，無法做非常好的優化。

這一次的團隊合作打 code 是我第一次接觸的，我們所有人都必須完成自己的東西，我們 AI 組必須信任 gamerunner 組，他們也同樣要信任我們，只有兩邊的 code 都正確無誤，我們才能交出一份完整的 project，這個經驗是我從未經歷過的，也在這之中學習到如何跟組員正確溝通、準確交換資訊的方式。

（3）黃淳侑　　：

移動戰艦比想像中還要複雜，除了要思考策略以外，麻煩的是不合法移動的判斷，戰艦與戰艦之間的重疊只要一個不注意就會發生。也要謝謝我的其他組員，大家負責的部分看起來都很不簡單，特別是 GUI，能做得那麼精緻真的讓我很佩服。C++和物件導向的概念真的不簡單，要不是助教幫我們寫好了這麼多檔案，我真的做不出這麼好的遊戲，看來還有很多東西需要學習。

（4）文逸雲　　：

這次的作業是我第一次和別人一起共同開發一個專案，有很多新的經驗，也遇到了很多問題，在寫 gamerunner 的時候遇到最大的困難大概就是搞懂 call 函數了，這個部分搞懂之後其他就是照著 ppt 所述的規則寫而已。而在跟其他人

的 AI 程式對接的時候也遇到一些問題，像是一開始寫移動的函數的人誤以為船進去的 ship 陣列就是只有可移動的船，就被我的 gamerunner 判輸了。解決方法就是回去改了。這次的作業讓我學到了很多東西，是一次很寶貴的經驗。