Dungeon report

112550015劉家琪

I. Implementation detailed

我的dungeon主要程式的運行都在Dungeon.cpp，會處理一開始遊戲的設定、玩家的選擇與遊戲邏輯的判斷

地圖：固定產生5個房間，startRoom, desert, forest, swamp與endRoom，進行隨機連接。desert, forest和swamp都是繼承room，他們各自有triggerEvent（壞事件）、handleEnvironment（環境造成每回合的數值變化）與findRestArea（好事件）不同的函數implementation。而好壞事件各有一個相對的bool來儲存是否發生過，同一房間有發生過就不會再發生了，沒發生過則每回合會random決定發生與否。每個房間的物件以及會遇到的事件我有全部先綁定好，順序是固定的但連接的上下左右會隨機：

GameStart→ startRoom ：無物件，玩家進入遊戲時currentRoom設定的地方

→ desert ：Desert的好事件(oasis)、Desert的壞事件(sandstorm)、Chest裡的cactus(Item))

→ forest：Forest的好事件(lake)、Forest的壞事件(wildlife animal, 打贏可獲得meat(Item))、**monster Dinosaur**

→ Swamp：Swamp的好事件(spring)、Swamp的壞事件(toxic gas, 有Poison)、NPC merchant（有milk, hamburger和water三種Item可購買）

→ endRoom：**monster Python（攻擊有Poison）** → victory

其中只有monster是一定要遇到且打贏才能繼續前進的

中毒：我讓swamp, monster和player都帶有Poison*的member variable，player的代表玩家中毒與否（nullptr沒中毒，有中毒便會指向中的poison），monster的則會在戰鬥時施加在player身上，swamp則是在其壞事件（遇到toxic gas）才會中毒，每回合duration會減少，並持續消耗player生命值，跟商人買到milk並equip就可以解毒。

遊戲結束：獲勝條件為打敗boss怪（endRoom的monster），死亡條件則是生命值歸零，每回合固定的數值減少（hunger, thirst與poison造成的）、沙塵暴、野外生物與monster的攻擊都有可能造成死亡。（如果在平常What do you want to do?時輸入'#'是方便測試用的suicide）。

戰鬥系統：寫在Monster::triggerEvent，當玩家在chooseAction時選擇Explore時，該房間的物件會出現，如果是monster就會觸發戰鬥，可以選擇撤退（返回previousRoom）或戰鬥，每次都可以選擇撤退或繼續戰鬥，而玩家攻擊後，會先檢查monster是否死亡，如果成功擊敗怪獸則會有數值的提升（但也會消耗體力），如果沒有死亡則monster也會攻擊玩家，生命值的減少量為（攻擊方攻擊值 - 防禦方防禦值*0.5），此時再檢查玩家是否死亡，沒有則繼續戰鬥直到一方死亡。

環境事件

- **triggerEvent**

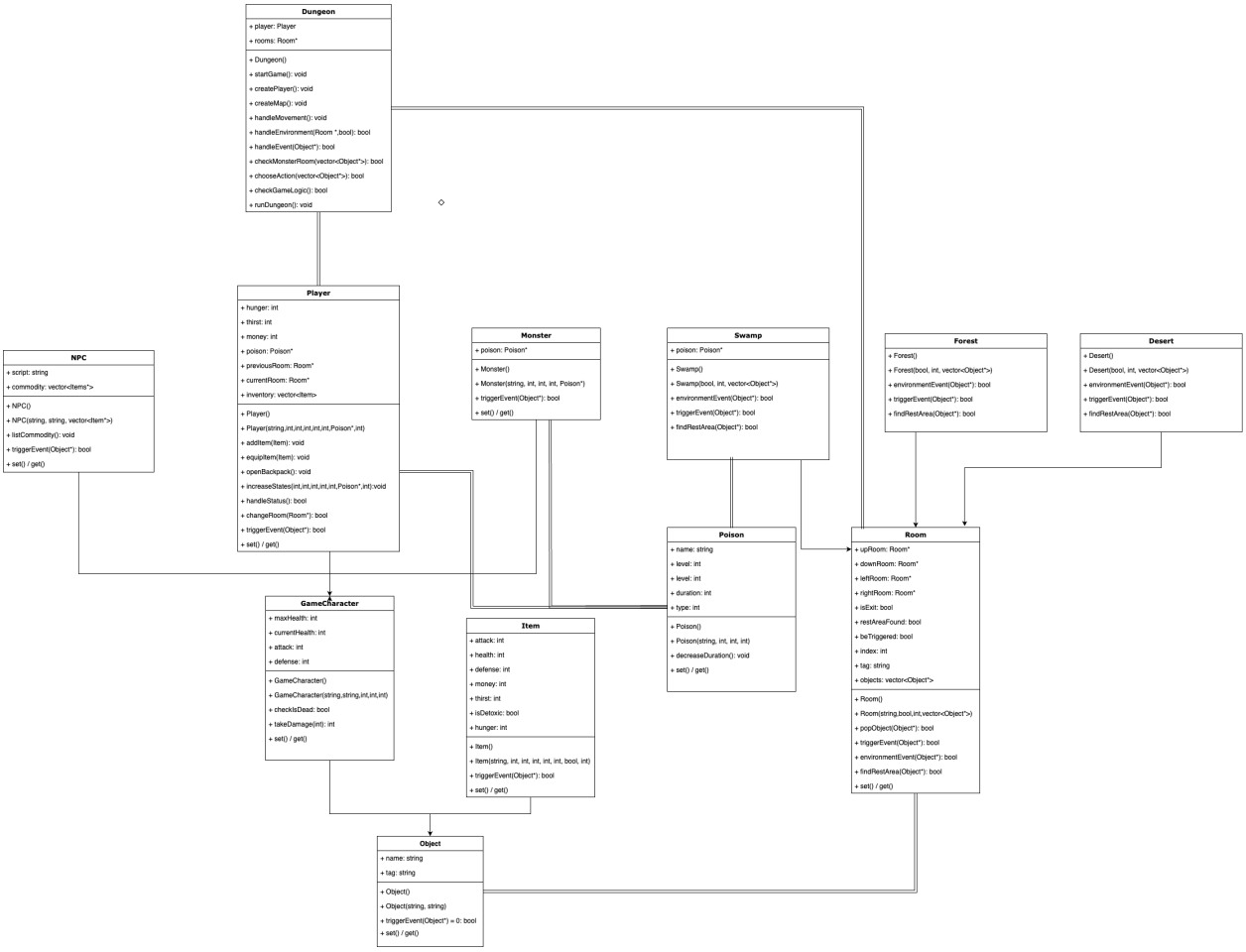
  Desert：sandStorm，將會造成hunger, thirst, attack, defense的減少，player只能選擇躲藏或繼續前進，但繼續前進會直接死亡，要躲藏直到sandStorm結束（1/2的機率）才能回到一般的行動介面。

  Forest：wildlife animal，player可以選擇留下戰鬥或逃跑，留下戰鬥可能死亡或戰勝，戰勝會獲得meat，逃跑則直接回到原先的行動介面，都會消耗hunger與thirst。

  Swamp：toxic gas，玩家中毒。

- **findRestArea :** 皆為thirst數值增加。

II. UML design



III. Result

程式開始運行後，會先創建一個Dungeon的物件，然後執行runDungeon( ) @main, runDungeon會負責呼叫遊戲開始的設定以及遊戲遊玩的部分@Dungon。

遊戲開始 startGame( )

- 創建地圖createMap( )：建立五個房間以及房間中的物件，並將房間隨機連接 connectRooms( )
- 創建玩家createPlayer( )：提示玩家選擇職業後，創建對應的player並設定其數值與開始的房間（如果選擇玩家的時候輸入‘#’會創建一個超級帳號）

遊戲運行

- 確認遊戲是否結束checkGameLogic( )：每回合都會進行確認玩家是否勝利（抵達出口房間並打贏該房間的monster）或死亡（生命值歸零），如果是的話就印出相關結果

- 選擇行動chooseAction( )：每回合詢問玩家想做的事，可以移動到其他房間、確認狀態、打開背包和探索房間
    - 移動房間handleMovement( )：更改currentRoom以及previousRoom
    - 確認狀態：列出player的所有數值、中毒狀況與背包中的Item @Player :: triggerEvent( )
    - 打開背包：列出背包中的Item，並處理裝備Item（數值提升）@Player :: openBackpack( )
    - 探索房間：可能遇到各種事件，如：發現寶箱、怪獸或商人，會呼叫對應的triggerEvent
        - 發現寶箱Item :: triggerEvent：詢問玩家是否要打開寶箱和撿起Item
        - 發現怪獸Monster :: triggerEvent：處理玩家與怪獸的戰鬥或撤退
        - 發現商人NPC :: triggerEvent：處理玩家與商人的交易
        - 沒有發現東西（該房間的物件已被探索完畢）
- 體能消耗handleEnvironment( )與Player :: handleStatus( )：部分行動（移動、探索）會進行玩家部分數值(hunger, thirst)的減少，並處理環境的影響（如在沙漠中thirst減少特別快），以及因為部分狀態造成的生命值減少（hunger, thirst歸零或中毒造成health減少）
- 環境偶發狀況：分為好事件與壞事件，在不同房間(forest, swamp, desert)會有不同的結果
    - 好事件findRestArea( )：會造成數值提升或是解毒
    - 壞事件triggerEvent( )：可能遇到沙塵暴、野外動物或毒氣，玩家將可能有不同選擇造成數值影響或是得到物件

IV. Discussion

1. triggerEvent：起初，我對於如何操作triggerEvent毫無概念，即使知道要用物件的不同實作來運作整個系統，依然會不知道很多不同物件交替作用的程式到底要寫在哪一邊，也仍會受C語言sequential programming的想法桎梏，無法完全以物件導向來思考。但後來隨著一些基本的setup完成後，開始實作各種implementation，就比較能體悟到Dungeon是由物件與物件的交互作用來運行的含義了，寫不同的implementation也就像把東西丟到分門別類的資料夾一樣，做習慣之後就變快許多了。

2. ChatGPT協作：前期在處理各種getter和setter與後期在加入一些ASCII art時，我都十分依賴ChatGPT幫我做code的修改，不過還是需要人工的檢查與微調，如thirst的值不會繼續歸零後不會扣到負數故setter就會有變化，以及cout出來的如果有變數的話加框會變得很困難（可以靠count字串長度處理，但很繁瑣所以我最後沒有選擇實作，而是取巧讓checkStatus時框只有四個角，其他時候只有固定的文本才會有框），總體上只有省到一點時間。然而，若能精準identify ChatGPT可以做得比人類好的事情還是蠻方便的，如我在貼上網路上的ASCII art時，就會直接貼給ChatGPT告訴它前面加`cout`後面加`endl`，就能馬上獲得可執行的code。
3. 繼承class：在實作swamp, forest與desert時必須繼承room，我在room加了三個virtual function，並在三個子物件中放不同的實作來處理不同環境的事件，這裡的virtual function**不能做成pure virtual**，因為我的startRoom與endRoom都是普通的room而已，因此room還是會有被initialize出來的時候，我就只是把room中環境事件的部分都寫成單純的`return false;`這樣我依然可以每回合都跑random決定要不要運行那些function，在普通房間一樣會觸發的到但不會發生事情。
4. 遊戲測試：最初運行的時候有bug，因此debug的時候就使用了下面這種語法，用`#define`一次處理所有test相關的輸出以及`__FUNCTION__`來代表所在函數的位置，我認為比我一個一個去寫和註解掉快上許多。

```
#define TEST
#ifdef TEST
    cout<<"Test:"<<"run "<<__FUNCTION__<<" successfully"<<endl;
#endif
```

V. Conclusion

我覺得Dungeon背後的程式邏輯不難，但十分繁瑣所以處理起來很費心力，常常要加一個feature就要花很多的時間，而且測試會花上不少的時間。不過，經過這次的實作Dungeon我也明白到了一開始做好UML design以及規劃好遊戲流程的重要性，可以有效避免掉更改code所花的時間，且常常code沒有全部都修改到就會造成難以察覺的error。

如果要重做一次的話，我會花更多時間在構思階段，並盡量讓程式被重複利用、避免冗餘的if-else、設定好function call 的return value的含義與建立更精確的分類法則（哪個function應該給誰call要有一致的標準，否則常常搞不清楚而重複執行），讓讀code跟寫code的複雜度都降低。

我認為我現在的code為了符合作業的requirement把很多東西先寫死了，但我希望之後能夠做到讓新加入物件或feature都是`O(1)`的處理時間而非現在的接近`O(n)`。不過總體來說這份作業還是學到了非常多，平常聽教授講都覺得概念有理解了，但實作一個多個檔案的project才能了解到許多實作細節以及對物件導向有更深刻的認識。