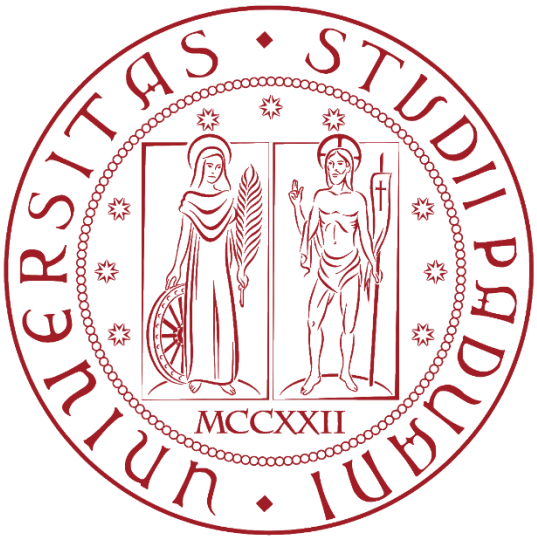

UNIVERSITÀ DEGLI STUDI DI PADOVA

RELAZIONE PROGRAMMAZIONE AD OGGETTI

A.A. 2020 / 2021

GRUPPO: FRANCESCO ROBERTO PASIN – URBANI SIMONE

RELAZIONE A CURA DI: URBANI SIMONE

MATRICOLA: 1193523

DATA: 14/02/2021

Introduzione

Il progetto realizzato è un gioco del tipo 'tower defense' dal nome 'CPP – Crush Poor Programmers'. In questi giochi l'obbiettivo è quello di piazzare delle torrette che, sparando ai nemici, devono eliminare quest'ultimi prima che arrivino al traguardo. I nemici seguono un percorso preciso e quando arrivano alla fine infliggono danno alla vita del giocatore. Se la vita del giocatore arriva a zero, perde. Nel nostro caso specifico, i nemici sono gli studenti che vogliono arrivare al 30 in programmazione ad oggetti, ma si imbattono nelle torrette, che rappresentano dei concetti di C++.

Descrizione gerarchia

La gerarchia alla base del modello è quella con radice `Turret`. Questa classe astratta rappresenta una torretta generale. Oltre ai parametri comuni ad ogni torretta implementa un metodo protetto `vector<SharedPtr<Enemy>> getEnemiesInRadius() const` che ritorna i nemici che si trovano nel raggio della torretta. Il metodo `virtual bool attack() = 0` invece è da ridefinire nelle classi figlie; è infatti il metodo che permette alla torretta di infliggere danno ai nemici che può attaccare. Definisce anche un metodo `virtual vector<SP<Enemy>> getTargetedEnemies() const` che si differenzia da getEnemiesInRadius perché tiene conto delle statistiche della torretta. Altri metodi sono semplici getter. Da `Turret` derivano `SlowTimeTurret`, `GranadeTurret` e `StandardAttackTurret`. Rispettivamente rappresentano una torretta che rallenta in nemici, una che colpisce i nemici decrementando la potenza e una torretta ad attacco standard (vedi Descrizione chiamate polimorfe). Al terzo livello si trovano `SplitTurret` e `ComboTurret`, entrambe derivate da `StandardAttackTurret`. Tali classi rappresentano una torretta ad attacco standard, ma con delle differenze nel metodo di attacco. `SplitTurret`, infatti, divide il danno per il numero di nemici nel raggio, mentre `ComboTurret` aumenta il danno di attacco in attacco, se il nemico bersagliato è sempre lo stesso.

Descrizione chiamate polimorfe

La chiamata polimorfa del modello è una, ma è il fulcro del funzionamento di `Game`. Grazie alla chiamata `Turret *s = new …(…); s->attack();` è possibile ottenere effetti sui nemici anche molto diversi tra loro.

- `SlowTimeTurret`: setta il fattore di riduzione della velocità di un nemico, ha l'effetto di rallentarlo fino al prossimo tick.
- `StandardAttackTurret`: ottiene i nemici bersagliati dalla torretta, quindi infligge il danno della torretta.
- `GranadeTurret`: ottiene i nemici bersagliati dalla torretta, quindi infligge un danno decrementato in base alla posizione del nemico. (Esempio: primo nemico: 50 danni, secondo nemico: 40 danni, terzo nemico: 30 danni, …)
- `SplitTurret`: divide il proprio danno per il numero di nemici bersagliati, quindi attacca come StandardAttackTurret
- `GranadeTurret`: aumenta il proprio danno ad ogni attacco, se il nemico bersagliato è sempre lo stesso, quindi attacca come StandardAttackTurret. Se il bersaglio cambia, l'attacco si resetta.

Descrizione formati di I/O

Il salvataggio su file avviene cliccando il pulsante 'save' nella sezione 'set map'. Tale pulsante ha come effetto il salvataggio su file della mappa appena creata, con estensione personalizzata '.cppmap'. Quest'ultimo è effettivamente un file binario, il quale viene creato dapprima serializzando le posizioni del percorso e delle celle bloccate in un json, utilizzando le classi `QJsonObject` e `QJsonArray`, successivamente si utilizza la funzione `.toVariantHash()` per creare un hash dall'oggetto json, il quale viene poi salvato su un file in formato binario grazie alla classe `QDataStream` e al suo operatore di output `<<`. La creazione di un hash e il fatto che il file è binario, previene che agenti esterni possano modificare il file, andando a introdurre degli errori.

Il caricamento di una mappa da file avviene premendo il pulsante 'play map'. L'utente sceglie quindi un file .cppmap, e avviene il processo inverso del salvataggio sopra descritto. Grazie a `QDataStream` e al suo operatore in input `>>` viene estratto dal file un oggetto `QVariantHash`, il quale diventa parametro della funzione `.fromVariantHash()` di un oggetto `QJsonObject`. A questo punto si ha un oggetto rappresentante un json che contiene le posizioni delle celle bloccate e del percorso. Dopo aver opportunamente deserializzato il json, vengono utilizzate le posizioni in esso contenute per settare la mappa della classe game. Alcuni preset di mappe sono presenti nella cartella maps.

Manuale GUI

All'avvio dell'applicazione viene mostrata la schermata principale, dalla quale si possono eseguire le seguenti quattro azioni: avvio del gioco, avvio del gioco con una mappa custom, creazione di una nuova mappa e consultazione del tutorial. Premendo il pulsante 'start' si dà inizio al gioco. Se è la prima volta che si apre l'applicazione, verrà mostrato un tutorial esaustivo di tutte le informazioni necessarie per giocare. Tale tutorial è sempre disponibile dalla schermata iniziale premendo l'omonimo pulsante. Anche premendo il tasto 'play map' è possibile iniziare a giocare, ma solo dopo aver caricato una mappa da un file '.cppmap' (vedi Descrizione formati di I/O). Per creare una mappa custom è invece necessario cliccare sul pulsante 'set map': si viene indirizzati una pagina con griglia vuota. Cliccando su una cella appare quindi un pop-up da cui è possibile selezionare il tipo di cella voluta. È possibile piazzare una cella di inizio percorso solo nella prima colonna da sinistra, la cella di fine percorso solo nell'ultima colonna a destra, le celle di percorso intermedie in tutte le colonne a meno della prima e dell'ultima.

Istruzioni di compilazione

Per compilare ed eseguire automaticamente il progetto è sufficiente eseguire lo script `buildAndRun.sh`.

In alternativa eseguire i seguenti comandi:

`sudo apt-get install -y qt5-default` per installare le librerie di Qt (mancanti dalla vm).
`mkdir build` e `cd build` per creare una cartella in cui verranno creati tutti gli output.
`qmake ../TowerDefense.pro` e `make` per creare il Makefile ed eseguirlo.
Dopodiché sarà possibile eseguire l'applicazione con il comando `./TowerDefense`

Ore di lavoro

Il tempo complessivo per la realizzazione del progetto è di 60 ore a persona, per un ammontare di 120 ore complessive. Tale numero è approssimativo, in quanto non si è tenuto conto in maniera precisa del tempo effettivo. È possibile comunque farne una stima: avendo incominciato ad inizio anno e finto il 14 febbraio, per un totale di 6 settimane (di cui 2 dedicate alla preparazione di altri esami) e tenuto conto di una media di 15 ore di lavoro a settimana, le ore totali risultano: 15 x 4 = 60. Le mie dieci ore in eccesso sono dovute allo studio del framework Qt nonché alla familiarizzazione con l'ambiente di sviluppo Qt Creator, i quali hanno necessitato di uno studio molto più approfondito rispetto a quanto visto a lezione.

Suddivisione del lavoro

Tendendo conto della maggiore conoscenza di Qt da parte di Pasin Francesco, il lavoro è stato in questo modo suddiviso:

- Urbani Simone:
  - Container `MyList`
  - Smart Pointer `SharedPtr`
  - `Game` / `GameModel`
  - Controller e vista 'Set Map', con gestione I/O
- Pasin Francesco:
  - Gerarchia `Turret`
  - `Enemy`
  - Astrazione di controller e vista
  - Controller e vista di 'game', 'tutorial', 'initial'

Un lavoro di revisione è stato fatto da entrambi alla fine.

Si è scelto di usare git come software di controllo di versione distribuito, e github per ospitare il codice condiviso. Ad ogni modifica corrisponde un nuovo branch, con relativa pull request. Solo quando l'altro componente del gruppo approva le modifiche di una pull request, allora è possibile farne il merge. Questo assicura che il branch main, che non è direttamente modificabile, resti sempre stabile e approvato da entrambi.

Motivazione di SharedPtr

Al contrario di quanto scritto sulle specifiche del progetto, si è scelto, in accordo con il professore, di realizzare una versione semplificata di `std::shared_ptr`, ovvero `SharedPtr`. Questa classe rappresenta un puntatore condiviso: molti `SharedPtr` possono puntare allo stesso oggetto, che viene deallocato se e solo se non c'è nessun altro `SharedPtr` che punta all'oggetto. Questo offre numerosi vantaggi nella nostra applicazione. Quando il gioco viene avviato tramite il pulsante 'start', viene avviato un timer, il quale ha il compito di chiamare la funzione di update della scena ad ogni intervallo di tempo, in modo da aggiornare la rappresentazione corrente dell'oggetto game (quindi nemici, torrette, ecc.). Per aggiornare tutti gli oggetti della scena che dipendono dal modello, se non avessimo `SharedPtr`, bisognerebbe ad ogni tick del timer andare ad estrarre i dati passando da `GameModel`, e di conseguenza `Game`, che devono essere nuovamente gestiti dalla scena, complicando e rallentando il processo. Grazie al `SharedPtr` invece, ogni item della scena che dipende dal modello ha un puntatore all'oggetto stesso contenuto nel modello, dato che è permesso avere due `SharedPtr` che puntano allo stesso oggetto. Ad ogni tick del timer quindi ogni item della scena utilizza i dati ottenuti direttamente all'oggetto del modello collegato per ridisegnarsi, evitando di passare attraverso `GameModel` e `Game`, con un grande risparmi in termini di efficienza.

Ambiente di sviluppo

Sistema operativo: Windows 10 Education N su macchina virtuale (sistema operativo della macchina reale: macOS Big Sur).

Versione Qt: Qt 5.14.1 MSVC2015 64bit

Versione compilatore: MSVC2015 64 bit (Microsoft Visual C++ Compiler 16.8.30804.86 (x86_amd64))