UNIVERSITÀ DEGLI STUDI DI PADOVA

RELAZIONE PROGETTO PROGRAMMAZIONE AD OGGETTI A.A. 2020/2021

Francesco Roberto Pasin

1188642

Autori progetto:

Francesco Roberto Pasin, Simone Urbani

https://github.com/francescopasin/tower-defense

Presentazione progetto

Per il progetto di PAO dell'anno 2020/2021 abbiamo deciso di creare un gioco "tower-defense" a tema C++. Il gioco consiste in un percorso, attraverso il quale dei nemici cercano di raggiungere la fine, togliendo vita al giocatore. Quest'ultimo deve piazzare nelle celle della mappa delle torrette, con lo scopo di eliminare tutti i nemici, impedendo loro di arrivare alla fine del percorso.

Maggiori informazioni sulle regole del gioco sono presenti all'interno del gioco stesso, attraverso un tutorial che appare alla prima partita o premendo nel pulsante "tutorial" nel menù principale.

Compilazione ed esecuzione

Per compilare ed eseguire automaticamente il programma sarà sufficiente eseguire lo script `buildAndRun.sh` (assicurandosi che sia attivo il permesso di esecuzione).

In alternativa è possibile eseguire i seguenti comandi separatamente:

`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`

Organizzazione progetto ed architettura

Abbiamo scelto di utilizzare il pattern MVC per separare completamente il modello dalla vista (e dall'utilizzo delle librerie di Qt).

Il codice delle tre parti (Modello, Controller e Vista) si trova in cartelle diverse, ed ogni file appartiene ad uno specifico namespace (model, controller, view) per garantire una buona organizzazione e separazione del codice.

Sono presenti poi altre due cartelle: app, che contiene un enum per la navigazione e un file con degli shortcut e assets, che contiene immagini e font.

Tutta la GUI si basa sul Graphics View Framework di Qt (https://doc.qt.io/qt-5/graphicsview.html).
La finestra principale è una `QMainWindow`, che contiene una `QGraphicsView`.

La navigazione tra le diverse schermate è gestita cambiando la `QGraphicsScene` contenuta nella `QGraphicsView`.

Ogni schermata è collegata e controllata da un solo specifico controller, che ha il compito di costruirla e gestirne gli eventi.
Ogni controller deve derivare dalla classe astratta `Controller` che contiene il metodo per la navigazione e quello che ritorna alla finestra principale la scena da mostrare.

Questo sistema di “routing” garantisce una facile scalabilità dell’applicazione, rendendo facile l’aggiunta di nuove schermate.

La comunicazione tra modello, controller e vista avviene nel seguente modo:
l’utente interagisce con la vista, che manda eventi al controller, il quale chiama metodi del modello, modificandolo. Il controller può interagire con la vista chiamandone metodi. Quest’ultima può avere accesso al modello (solo in lettura) per ottenere direttamente i dati da esso senza passare per il controller.

Di seguito sono elencate le dipendenze che vengono a crearsi tra le tre parti (-> = dipende da):
Modello -> niente
Controller -> Modello e Vista
Vista -> Modello (opzionale)

Descrizione modello

L’intero flusso di gioco è gestito dalla classe `Game`. La classe `GameModel` fa da interfaccia tra `Game` ed il Controller, e si occupa della gestione dei livelli (mappa e nemici).

Il gioco si basa sul metodo `tick()` della classe `Game`. Ogni volta che questo viene chiamato viene calcolato il frame successivo: i nemici si muovono e le torrette sparano. La velocità del tick viene decisa dal controller (che utilizza un `QTimer` per richiamare periodicamente questo metodo ed aggiornare la vista di conseguenza). I tick del modello e della vista sono gestiti da due timer separati, in modo da poter velocizzare il gioco senza dover necessariamente aumentare il framerate (e gli update della GUI).

La gerarchia polimorfa è formata dalle torrette. La classe base `Turret` ha un metodo astratto `attack()`. Game contiene una lista di `Turret` e richiama ad ogni tick il metodo `attack()`, che esegue di conseguenza l’attack del tipo dinamico della torretta.

Le torrette con attacco “standard” derivano tutte (o sono) `StandardAttackTurret`. Questa classe concreta implementa il metodo di attacco “normale”, dove la torretta ogni `attackCooldown` tick attacca i nemici che ha in raggio (il numero di questi è deciso dal parametro `maxTargets` di `TurretStats`).

Le torrette che implementano un attacco in modo “speciale” (come, ad esempio, `GranadeTurret` che riduce la potenza di attacco per ogni altro nemico attaccato o `SlowTimeTurret` che rallenta i nemici) derivano invece direttamente da `Turret`, implementando il loro metodo `attack()`.

Si è scelto di inserire le statistiche base delle torrette in una struct a parte (`TurretStats`) e di creare una mappa di torrette, in modo che la vista possa accedere alle caratteristiche di queste anche se non sono ancora state costruite.

Note sull’uso di SharedPtr

Per poter spiegare e giustificare la scelta e l’utilizzo di `SharedPtr` è necessario prima analizzare la vista.

Descrizione vista

La view è separata in schermate: ogni schermata contiene una `QGraphicsScene` che contiene a sua volta una serie di `QGraphicsItem` (ognuno dei quali è una specifica classe). Gli elementi condivisi tra le schermate si trovano nella cartella hud.

La GUI contiene 2 gerarchie:
`Modal` (hud) è la classe base di tutti i modali, che definisce un metodo astratto `paintContent()` per disegnare l'effettivo contenuto del modale.

`Projectile` (screens/gameScreen/spawnable) che è la classe base per i proiettili, da cui derivano `Bullet` e `Granade`. Questa classe gestisce il movimento dei proiettili ma non il contenuto grafico. Difatti i 2 metodi astratti di questa classe sono (derivati da `QGraphicsItem`) sono `boundingRect()` e `paint()` , ovvero i metodi che gestiscono l'aspetto del graphics item.

SharedPtr

Si è scelto di utilizzare uno smart pointer con il comportamento di `std::shared_ptr` al posto di `std::unique_ptr` (indicato nelle specifiche) per permettere una connessione diretta tra gli elementi grafici e i rispettivi oggetti del modello.

Nello specifico questa scelta è nata dall'implementazione di `EnemyItem` e `TurretItem` (gameScreen): ogni nemico e torretta nella vista è gestito da un'istanza della rispettiva classe. Ad ogni tick di gioco è necessario che questi vengano aggiornati in base ai corrispettivi oggetti nel modello (`Enemy` e `Turret`).

L'utilizzo di uno unique pointer avrebbe reso poco pratica ed elegante la condivisione dei dati tra modello e vista. Per aggiornare nemici e torrette sarebbe stato necessario ritornarne l'elenco ad ogni tick.
Dopodiché per aggiornare gli elementi grafici ci sarebbero state 2 strade:

- eliminare quelli esistenti e ricrearli (scelta poco efficace e poco pratica);
- individuare l'elemento grafico corrispondente all'oggetto del modello ed aggiornarlo. Per farlo sarebbe stato necessario confrontare, ad esempio, le posizioni degli oggetti oppure aggiungere alle classi un campo dati id, per identificarle.

In entrambi i casi secondo noi il codice sarebbe stato poco pratico ed efficace.
La nostra scelta è stata quella di inserire il puntatore all'oggetto (`Enemy` e `Turret`) del modello direttamente dentro il corrispettivo oggetto (`EnemyItem` e `TurretItem`) della vista. Così facendo ogni elemento grafico è strettamente collegato ai propri "dati" e, ad ogni tick, può aggiornarsi con facilità.

Di conseguenza lo `SharedPtr` è necessario per garantire un controllo della condivisione della memoria, necessaria quando diversi oggetti accedono allo stesso puntatore. Ad esempio, nel caso dei nemici, il modello elimina il puntatore ad un `Enemy` quando questo muore, ma la vista ha necessità comunque di accedervi anche dopo, per controllare se è morto (vedere metodo `isDead()` di `EnemyItem`). Se non fosse stato usato un puntatore condiviso questo non sarebbe stato possibile poiché la vista non avrebbe potuto più accedere ai dati dopo che il modello li avesse eliminati.

La nostra implementazione di `SharedPtr` è usata anche in altri frangenti: ovunque sia necessario condividere puntatori tra diversi oggetti (es.: il modello dentro i controller e le viste).

IO

L'input ed output su file viene utilizzato da un semplice editor di mappe, dov'è possibile creare la propria mappa di gioco. I vettori contenenti le informazioni sulla mappa vengono convertiti in Json e salvate su un file grazie all'utilizzo della classe di Qt `QJsonObject`. I file non vengono salvati in formato .json ma vengono convertiti in binario da Qt e salvati con un'estensione personalizzata (.cppmap). Questa conversione riduce il peso del file e rende più difficili letture e scritture esterne.

Suddivisione ore di lavoro

La collaborazione è stata gestita utilizzando GitHub (repo: https://github.com/francescopasin/tower-defense). Per garantire un controllo completo del codice ogni modifica deve obbligatoriamente essere controllata ed approvata da entrambi i membri per poter essere inserita.

Il lavoro è stato approssimativamente suddiviso nel seguente modo:

- Urbani Simone:
  - Container `MyList`
  - Smart Pointer `SharedPtr`
  - `Game` / `GameModel`
  - Controller e vista "Set Map", con gestione I/O
- Pasin Francesco:
  - Gerarchia `Turret`
  - Classe `Enemy`
  - Astrazione di controller e vista
  - Controller e vista di "gameScreen", "tutorialScreen", "initialScreen"

Molti file sono stati modificati e progettati da entrambi ed una revisione finale totale è stata effettuata insieme.

Di seguito le mie (Francesco Roberto Pasin) ore di lavoro approssimative:

- Analisi e progettazione ~5h
- Sviluppo modello ~10h
- Organizzazione e sviluppo architettura MVC ~10h
- Sviluppo vista ~35h
- Debugging e testing finale ~2h

Lo studio di Qt non rientra nelle ore poiché era già stato studiato per il progetto dell'anno scorso (non consegnato).

Le ore sono state leggermente sforate (di ~10h) per i seguenti motivi:

- Refactoring completo del sistema di navigazione e dei controller
- Sistemazioni e abbellimento GUI
- Disegno grafiche e sprite (alcuni disegni e animazioni sono state fatte da una persona esterna che ci ha aiutato)

Ambiente di sviluppo

Il progetto è stato sviluppato su Windows 10 utilizzando Qt 5.15.2 e Mingw 7.3.0 (64 bit) come compilatore.