GAETANOBROS

Alessandro Amella, Michele Garavani, Simone Ballo

2023

1 Introduzione e architettura del software

Abbiamo voluto sviluppare la nostra versione del gioco, denominata **GaetanoBros**, con l'obiettivo di creare un codice altamente modulare e ben strutturato, progettando la struttura del software seguendo i principi fondamentali del SOLID, un insieme di linee guida ampiamente riconosciute nell'ambito dello sviluppo software. Tra questi principi, due in particolare hanno guidato la nostra progettazione:

- **Principio di Singola Responsabilità**: a ogni classe corrisponde una e una sola responsabilità (una classe per il rendering, una per i nemici, una per il timer, ecc.).
- **Gerarchia di Classi**: abbiamo creato una gerarchia di classi in cui le sottoclassi rispettano i vincoli delle classi padri. Questo significa che abbiamo definito una classe base per le entità statiche, da cui derivano le entità dinamiche, e da queste ultime, a loro volta, derivano il giocatore, i nemici e i proiettili. Questa struttura gerarchica ci ha consentito di ereditare funzionalità comuni e mantenere una coerenza nel design.

Oltre a questi principi SOLID, abbiamo posto grande importanza sui concetti di **separation of concerns** ed **encapsulation**, separando chiaramente le responsabilità di ciascuna entità tramite le proprie classi o file header dedicati, e nascondendo i dettagli di implementazione delle classi mantenendoli privati, fornendo solo interfacce pubbliche per interagire con il resto del codice.
Questo approccio ha semplificato l'interazione tra i vari componenti del gruppo, poiché hanno potuto utilizzare le interfacce pubbliche delle classi senza dover conoscere i dettagli di implementazione sottostanti.

2 Implementazione

2.1 Codice condiviso e costanti di gioco

Per rendere il codice del nostro gioco più modulare e manutenibile, evitando di fare un *hard-coding* di valori specifici all'interno del codice, abbiamo separato le costanti e le funzioni condivise in file distinti nella cartella `src/shared`. Il file `settings.hpp` contiene svariate costanti e strutture dati che regolano i vari comportamenti del gioco, mentre il file `functions.hpp` contiene funzioni comuni usate ampiamente in diverse parti del codice, come la funzione `clamp()` o `collides()`.

2.2 Gestione centralizzata

Abbiamo subito pensato che, per unire tutti gli aspetti del gioco, avremmo avuto bisogno di una classe centralizzata che si occupasse di gestire il flusso complessivo del gioco, l'unica classe a conoscenza di tutte le altre classi. Abbiamo chiamato tale classe `GameManager`. Tale classe, infatti, gestisce l'inizializzazione del gioco, il ciclo di gioco principale, le interazioni dei giocatori, le collisioni, il rendering, le condizioni di game over, e in generale ogni situazione in cui viene richiesta la conoscenza di dati tra classi isolate tra loro (ad esempio, le collisioni tra il player e i nemici). Tutte le altre classi, direttamente o indirettamente, vengono implementate in `GameManager`.

2.3 Logica di tick

Il nostro gioco usa una **logica di tick** per aggiornare lo stato del gioco e gestire gli eventi in modo sincronizzato. Un tick è un'unità di tempo che rappresenta un intervallo fisso, definito dalla costante `TICK_INTERVAL` nel file `src/shared/settings.hpp`. Ad ogni tick, il gioco esegue una serie di operazioni che influenzano il comportamento degli oggetti e delle entità nel gioco.

Per implementare la logica di tick, abbiamo definito la classe `GameTimer`, che è una classe che misura il tempo trascorso a partire da un certo istante di tempo. Un oggetto `GameTimer` è naturalmente istanziato all'interno di `GameManager`, il quale ha anche una lista di puntatori alle classi che devono essere aggiornate ad ogni tick, come la classe `Player`, `Enemy`, `Projectile`, ecc.

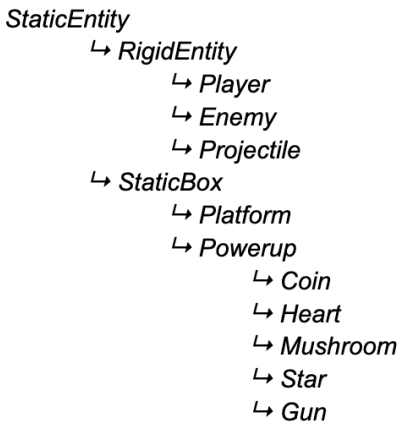
`GameManager` esegue un loop infinito in cui controlla se è il momento di fare un tick, chiamando il metodo `should_tick()` di `GameTimer`. Questo metodo restituisce `true` se il tempo trascorso dall'ultimo tick è maggiore o uguale a `TICK_INTERVAL`: in tal caso, `GameManager` chiama a sua volta il metodo `tick()` su tutte le classi di cui è a conoscenza sul quale tale metodo è definito, le quali chiameranno, a loro volta, il medesimo metodo sulle classi di cui esse sono a conoscenza. In questo modo, la logica di tick si propaga da una classe all'altra, creando una catena di chiamate che coinvolge tutte le classi che devono essere aggiornate col procedere del tempo.

2.4 List

Per avere una struttura dati dinamica che ci permette di rappresentare una sequenza di oggetti, abbiamo creato la classe `List`, decidendo di implementarla come lista concatenata semplice, sfruttando la **programmazione generica** offerta da C++: la classe viene definita come *classe template*, permettendoci di utilizzarla estensivamente nel codice con qualunque tipo di dato (`int`, `char`, `Platform`, `Room*`, ...). Abbiamo naturalmente definito ed esposto metodi pubblici per l'accesso e la modifica ai dati (`push`, `pop`, `at`, `length`, `contains`, `clear`, ...).

2.5 Gestione input

Abbiamo deciso di raggruppare la gestione degli input da tastiera dell'utente nella classe `InputManager`. Ciò che succede se i tasti vengono premuti viene definito nelle classi specifiche. Ad esempio, le azioni dei comandi del giocatore sono definite nella classe `Player`. I comandi dello shop sono gestiti nella classe `Shop`. Il tasto '`q`', che permette di uscire dal gioco, viene gestito nella classe `GameManager`.



Uno schema con la gerarchia di classi partendo da `StaticEntity`.

2.6 Entità

Per implementare le varie entità del progetto abbiamo pensato di usufruire dell'ereditarietà tra classi. Abbiamo iniziato con la classe base `StaticEntity`, usata come classe astratta per rappresentare una generica entità la cui posizione resta costante nel tempo, alla base di molte altre classi.

Per creare i potenziamenti e la valuta, in quanto entità, abbiamo usufruito dell'ereditarietà. Abbiamo prima di tutto creato la classe `StaticBox`, composta di due punti, che possono formare un rettangolo rappresentato dalle coordinate in basso a destra e in alto a sinistra. Abbiamo poi creato la sottoclasse `Powerup`, da cui vengono create le classi con i vari potenziamenti singoli. Le abilità dei potenziamenti vengono come al solito gestite da `GameManager` (ad esempio, essendo in carico delle collisioni, farà sì che il player non subisca danno quando esso possiede il powerup *stella* e viene colpito da un nemico).

Per implementare le "rigid entities", ovvero le entità dotate di velocità, abbiamo aggiunto attributi importanti come ad esempio:

- Le velocità orizzontali e verticali dell'entità.
- Un riferimento alle regioni di inizio e di fine, alle piattaforme, all'altezza del pavimento e del soffitto della stanza attuale.
- `starting_position` e `last_position` che rappresentano rispettivamente la posizione iniziale e la posizione precedente all'attuale (usata poi per cancellare dallo schermo la vecchia posizione).

Inoltre, abbiamo creato diversi metodi utili, per esempio, a simulare la gravità e l'attrito. Abbiamo poi implementato la classe-figlia `Player`, con altri aspetti importanti, come le capacità di salto, i tasti per muoversi, e le interazioni con i powerups (tra cui la gestione di monete e salute). Similmente, vengono create anche le classi figlie che gestiscono nemici e proiettili.

2.7 Livelli

Un altro aspetto molto importante del gioco è la suddivisione in livelli. Abbiamo pensato di implementare una struttura a stanze (classe `Room`). Il player può andare

avanti e indietro tra le stanze entrando nelle **regioni di inizio e fine** (`StaticBox`). Una stanza è definita con vari parametri, tra cui una lista delle sue piattaforme, le sue dimensioni, l'altezza del pavimento e del soffitto, regioni di inizio e fine: questi parametri sono ciò che rendono le stanze diverse tra loro. I layouts delle varie stanze, ciascuno definito all'interno di un file `layout<i>.cpp`, sono tutti accessibili attraverso la classe `Maps`.

La classe `LevelManager`, la quale contiene una lista di stanze istanziabili (che punta alle mappe salvate all'interno di `Maps`), si occupa di gestire il cambiamento di stanza. Essa possiede:

- Una lista di oggetti `RoomState` che associano un puntatore a una determinata stanza (con tanto di nemici, powerups, ecc.) alla difficoltà di tale stanza (`visited_rooms`).
- Un indice interno della stanza alla quale si trova attualmente il player (`cur_visited_room_index`).
- Un metodo pubblico che restituisce il puntatore alla stanza attuale (`get_cur_room()`) che restituisce la stanza nella lista `RoomState` all'indice `cur_visited_room_index`. Viene usato ampiamente su `GameManager` per avere un riferimento alla stanza attuale.

Quando quest'ultimo si posiziona alla regione di inizio di una stanza e se essa non è la prima stanza, allora l'indice viene decrementato, mentre inversamente se il player si trova alla regione di fine, allora questo viene incrementato. Se l'indice è già a `visited_rooms.length() - 1`, allora `LevelManager` provvede a selezionare casualmente una stanza dalla lista di stanze "base" e di istanziarlo all'interno della propria lista di `RoomState`, con tanto di nuova difficoltà. Per ultimo, vengono generate casualmente le posizioni di nemici e potenziamenti (incluse le monete). Quando il player muore e riavvia il gioco, l'indice viene riportato a 0 e gli elementi della lista di stanze visitate svuotata, ma la difficoltà, come da consegna, viene mantenuta.

Abbiamo inoltre creato la pagina dello shop, con la sua interfaccia grafica e le sue funzionalità (comprare potenziamenti). Abbiamo poi aggiunto l'istanza dello shop in `GameManager`, facendo in modo che lo shop compaia all'inizio del gioco e **ogni tre stanze**.

2.8 Difficoltà

Un ultimo aspetto centrale del gioco è l'aumentare di difficoltà da una stanza all'altra. La classe `LevelManager` si occupa anche di cambiare le specifiche della stanza per renderle proporzionali alla velocità. Tali cambiamenti includono:

- Un numero maggiore di nemici in base ad un aumento di difficoltà del livello. I nemici diventano anche in grado di sparare proiettili nel caso la difficoltà sia abbastanza alta.
- Il numero di powerups presenti in un livello. La funzione `get_powerup_number()` si occupa di ottenere il numero di powerups da generare nella stanza in base alla difficoltà. Invece di utilizzare una sequenza lineare come per i nemici, i powerups seguono un pattern esponenzialmente decrescente.

3 Conclusioni

Possiamo affermare che il progetto "GaetanoBros" è stato un'esperienza formativa e stimolante per il nostro gruppo. Abbiamo mantenuto un'architettura software consistente, modulare e facilmente scalabile, sfruttando fortemente la programmazione orientata agli oggetti per definire una gerarchia di classi coerente e flessibile. Il risultato è un gioco di piattaforme in stile retrò, ispirato al celebre Super Mario Bros, che offre una vera sfida e divertimento al giocatore.

A Contatti

Alessandro Amella - alessandro.amella@studio.unibo.it - 0001070569
Michele Garavani - michele.garavani@studio.unibo.it - 0001079937
Simone Ballo - simone.ballo@studio.unibo.it - 0001069408