Metal Shot

"Programmazione ad Oggetti"

Andrea Biagini
Filippo Gurioli
Matteo Susca
Tommaso Turci

25 giugno 2022

Indice

Capitolo 1

Analisi

Lo scopo del gruppo è la realizzazione di un gioco arcade shooter bidimensionale ispirato a Metal Slug (gioco della SNK Playmore, 1996).

1.1 Requisiti

Metal Slug è uno sparatutto a scorrimento orizzontale in cui il protagonista, munito di un'arma a raffica con colpi infiniti, deve eliminare tutte le forze nemiche che incontrerà lungo il percorso, fino ad arrivare a fine mappa dove lo attende il boss, un nemico dalle statistiche incrementate. Durante il suo tragitto potrà anche usufruire di nuove armi (a durata limitata), mezzi "Slug" e checkpoint.

Requisiti funzionali

- L'applicazione vuole emulare al meglio il gioco appena descritto ma riservandosi la possibilità di variare alcuni aspetti per questioni estetiche e di tempo
- Il giocatore avrà la possibilità di muoversi a suo piacimento nell'ambiente di gioco e di sparare ai nemici per avanzare
- Il giocatore avrà la possibilità di ripartire da vari checkpoint distribuiti lungo tutto il percorso
- Al termine della partita verrà registrato un punteggio associato al tempo rimasto, i nemici uccisi e le vite perse nella battaglia
- I nemici dovranno essere in grado di prendere decisioni basilari per uccidere il giocatore ed evitare ostacoli presenti sulla mappa

Requisiti non funzionali

- L'applicazione dovrà funzionare su qualsiasi Sistema Operativo
- L'applicazione dovrà essere ridimensionabile a seconda delle esigenze dell'utente

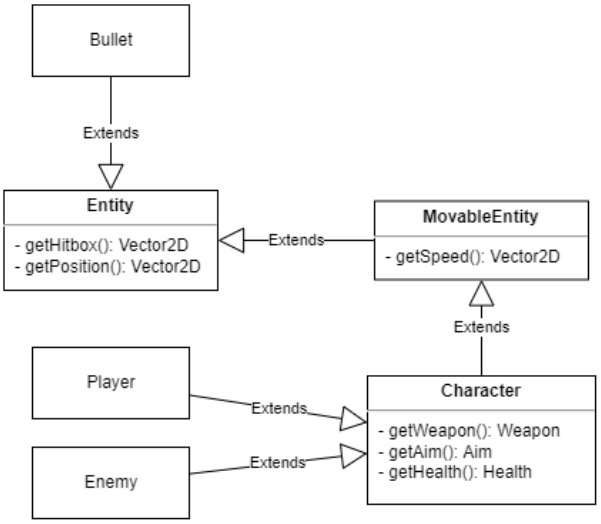
1.2 Analisi e modello del dominio

Metal Shot dovrà essere in grado di gestire il personaggio, la mappa, i nemici e tutte le possibili armi. Ognuno di questi aspetti presenta già in sé una sfida non indifferente che consiste nelle animazioni, dall'arma che spara al salto del personaggio alla morte del nemico fino allo scorrimento dello schermo. Un'altra difficoltà risiede nell'implementare una AI capace di gestire il nemico in modo tale da rendere il gioco fluido ma non banale. Metal Shot si impegna anche a gestire tutti i tipi di collisioni quali, per esempio, tra il player ed il terreno, tra il player ed i proiettili e tra tutte le entità con il terreno.

Alcune funzionalità presenti all'interno di Metal Slug non potranno essere implementate nel progetto in quanto le ore necessarie per lo sviluppo di queste porterebbero al superamento del monte ore previsto.

Le funzionalità in questione sono:

- Mezzi Slug
- Items e Buff
- Checkpoints e Respawn
- Boss finale



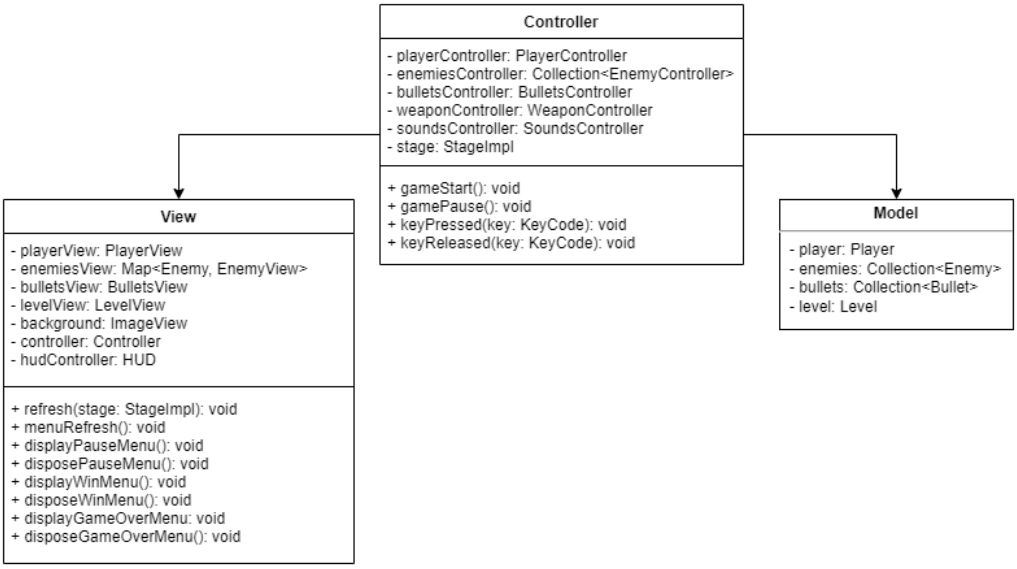
Capitolo 2

Design

Dalla descrizione precedentemente fatta si evincono 4 elementi costitutivi dell'applicativo: personaggio, nemico, mappa e armi. Questi comunicheranno tra di loro su diversi livelli: le armi potranno essere equipaggiati dal giocatore, il giocatore potrà a sua volta equipaggiare le varie armi ed eliminare i nemici, i nemici potranno ferire il giocatore e, infine, la mappa dovrà essere in grado di contenere tutte le entità.

2.1 Architettura

Il progetto segue il pattern architetturale MVC (Model-View-Control) secondo il quale ogni entità viene scorporata nella sua componente visibile all'utente (appartenente alla View) e nella componente logica (appartenente al Model), mentre il Controller ricopre il ruolo di intermediario tra le due macro-classi. Proprio il Controller avrà quindi i metodi che permetteranno l'interrogazione e la modifica delle informazioni contenute nel Model che verranno successivamente passate alla View per essere mostrate a schermo. Utilizzando questa architettura si rende possibile il completo rimpiazzo della View con un'altra qualsiasi che si adatti alle API del controller.

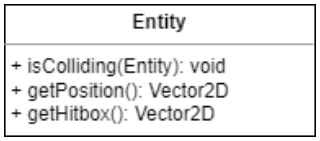

Controller
- playerController: PlayerController
- enemiesController: Collection<EnemyController>
- bulletsController: BulletsController
- weaponController: WeaponController
- soundsController: SoundsController
- stage: StageImpl
+ gameStart(): void
+ gamePause(): void
+ keyPressed(key: KeyCode): void
+ keyReleased(key: KeyCode): void
View
- playerView: PlayerView
- enemiesView: Map<Enemy, EnemyView>
- bulletsView: BulletsView
- levelView: LevelView
- background: ImageView
- controller: Controller
- hudController: HUD
+ refresh(stage: StageImpl): void
+ menuRefresh(): void
+ displayPauseMenu(): void
+ disposePauseMenu(): void
+ displayWinMenu(): void
+ disposeWinMenu(): void
+ displayGameOverMenu: void
+ disposeGameOverMenu(): void
Model
- player: Player
- enemies: Collection<Enemy>
- bullets: Collection<Bullet>
- level: Level

2.2 Design dettagliato

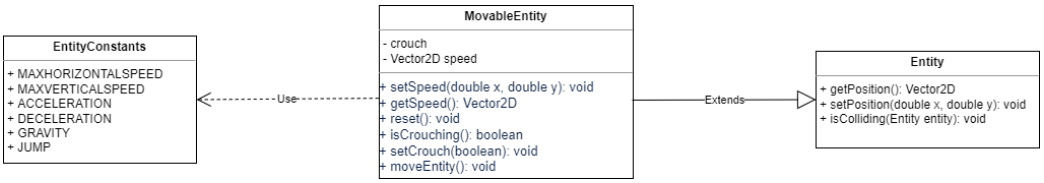
Filippo Gurioli

Entity - MovableEntity

La classe Entity si preoccupa di risolvere il problema semplice quanto importante di avere un oggetto che ogni entità all'interno del gioco potesse estendere per permettere una compatibilità più forte. Questa classe gestisce unicamente una posizione, una hitbox e le conseguenti collisioni tra le varie entità.



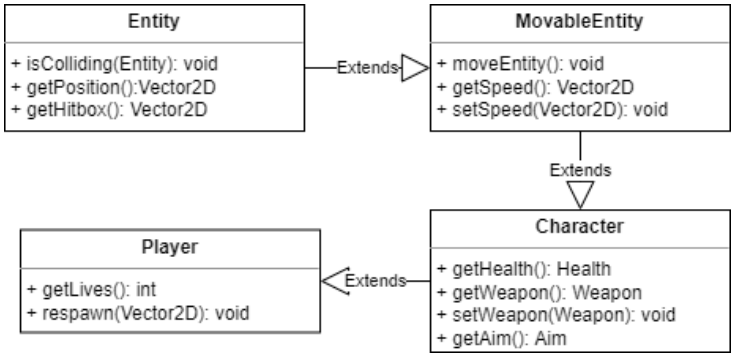
La sua estensione MovableEntity gestisce il movimento dell'entità. La sfida è ancora più interessante in quanto Metal Shot si sviluppa in una bidimensionalità dal lato, comportando così anche la gestione di un salto e della possibilità di crouchare(accucciarsi). Tutto ciò viene risolto usando una variabile velocità che viene sommata alla posizione per ottenere uno spostamento naturale, e delle costanti che limitano il valore della variabile per simulare una velocità massima. Queste costanti permettono anche la simulazione di un'accelerazione fisica in quanto variano la velocità nel tempo (definizione fisica di accelerazione). Per quanto riguarda il crouch invece è bastata l'aggiunta di un booleano che gestisse questa condizione.



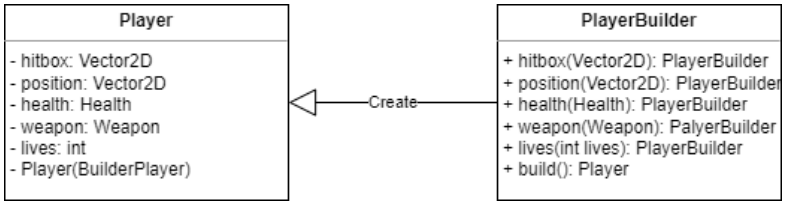
Character - Player

La necessità era quella di creare il personaggio principale che sarebbe poi stato comandato dall'utente. La cosa su cui mi sono concentrato è stata non rompere il single responsability principle per cui ho proceduto col creare una prima classe generica Entity che rappresentasse un oggetto qualsiasi nel mondo di gioco, per poi crearne un'estensione chiamata MovableEntity che

sia in grado anche di gestire un movimento (prevedendo così la creazione di oggetti quali, per esempio, i proiettili), poi ancora un'altra (Character) che nello specifico rappresentasse una qualsiasi persona del gioco per poi arrivare al Player che si sarebbe solo dovuto occupare della gestione delle vite multiple (unica caratteristica che lo differenzia dagli altri personaggi di MetalShot).

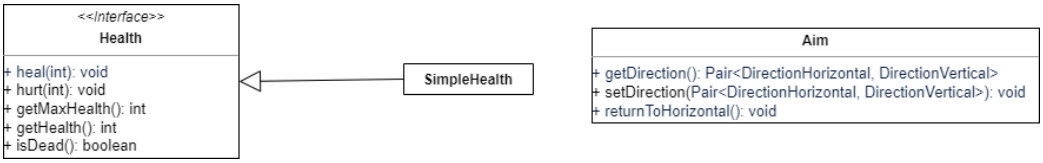


Data la grande complessità del Player in quanto a campi da inizializzare ho trovato opportuno applicare il builder pattern per rendere la costruzione più intuitiva e leggibile.



Health - Aim

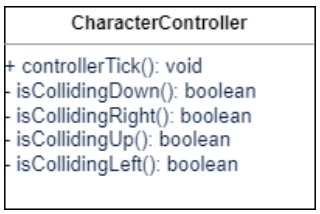
Queste due classi sono semplici classi di supporto al Character create unicamente per rispettare il SRP. La prima fornisce informazioni circa la gestione di una vita di gioco mentre l'ultima informazioni sulla mira che un personaggio di gioco può avere. Si vuole precisare che la classe Health rispetta lo Strategy pattern grazie al quale si rende pronta ad una qualsiasi espansione.



CharacterController - collisioni mappa

Per far sì che venissero gestite le collisioni tra characters e mappa non bastava la Entity.isColliding perché oltre all'informazione del fatto che stanno collidendo bisogna capire da che lato (sapere se il giocatore sta sbattendo la testa, le braccia o i piedi). Ho così sviluppato un metodo che controllasse ad ogni frame dove si sarebbe trovato il character al frame successivo, così facendo se la posizione fosse stata illegale (collisione con la mappa) bisognava solo spostare il soggetto alla posizione "legale" più vicina a quella effettiva. La conoscenza della direzione di collisione è necessaria per effettuare il riposizionamento del personaggio.

Oltre alle collisioni con la mappa il character controller doveva essere in grado di gestire tutta una serie di casi particolari che, essendo tali sono stati gestiti in modo singolare (e.g.:il personaggio che non può rialzarsi dal crouch in quanto c'è un tile sopra di lui che glielo impedisce).



Menù vari - MenuController

I menù, essendo parte di view, volevo che avessero la possibilità di essere ridimensionati a piacere. Questo mi è stato possibile applicando file css personalizzati che rendessero i vari menù sempre piacevoli all'occhio e adatti a qualsiasi risoluzione. Per lo sviluppo di questi menù mi sono appoggiato a dei file fxml che venivano scritti da SceneBuilder, uno strumento che tramite un'interfaccia utente ti permette di sviluppare una scena (javafx.scene.Scene) usando drag and drop di ogni componente che possa servire.

Ad ogni menù è quindi associato un menù controller che gestisce le operazioni da svolgersi alla pressione dei vari bottoni.

Matteo Susca

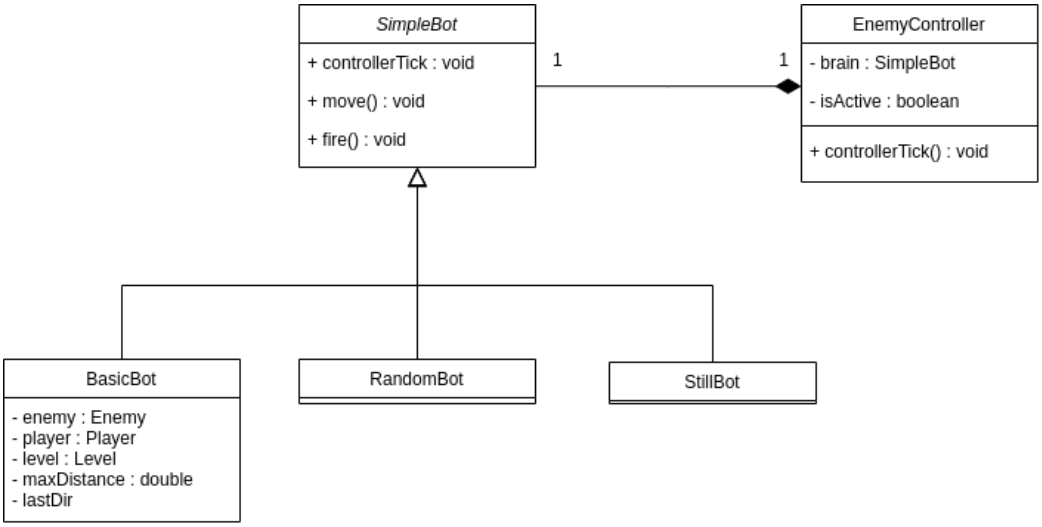
SimpleBot

Per fornire all'enemy un qualche tipo di "intelligenza" era necessario sviluppare un "cervello" in grado di prendere decisioni per quanto riguarda il movimento, l'inseguimento del player e l'attacco.

Per permettere una maggiore flessibilità e per facilitare sviluppi e migliorie futuri, ho deciso di adottare come design pattern il pattern Strategy. L'interfaccia utilizzata per modellare la struttura dei vari "Bot" è SimpleBot

Al suo interno ci sono metodi per il funzionamento di un Enemy basilare quali:

- controllerTick: viene chiamata per eseguire una serie di azioni e controlli.
- move: viene chiamata per eseguire azioni e controlli solo riguardanti il movimento.
- fire: viene chiamata per eseguire azioni e controlli solo riguardanti l'attacco nei confronti del player



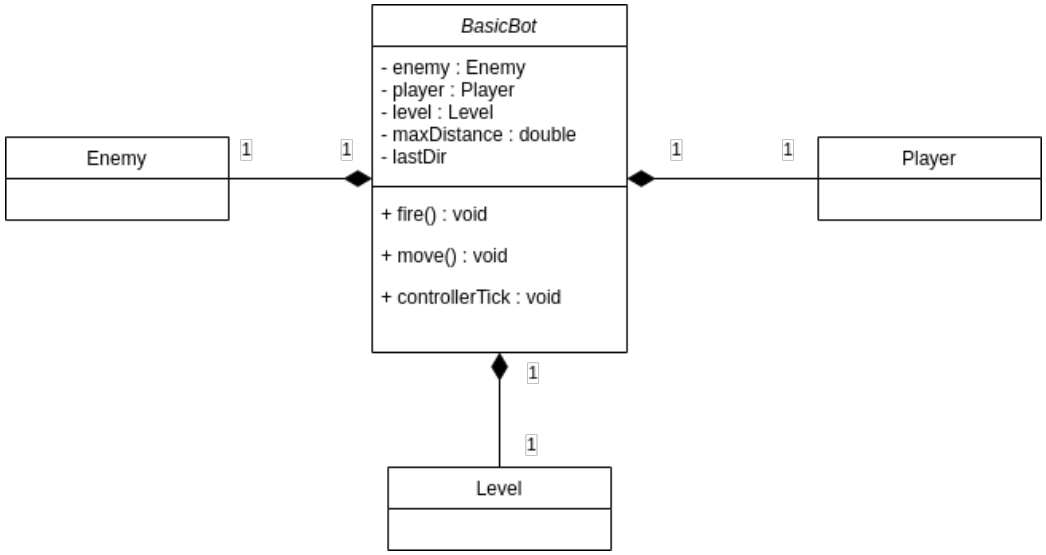
BasicBot Logic

BasicBot è il "cervello" utilizzato all'interno dell'attuale versione di Metal Shot. Il suo comportamento non è statico ma varia a seconda della sua vicinanza dal player. Il suo comportamento può essere diviso in due "Status" principali:

- Idle: in questo Status l'enemy si muove randomicamente avanti e indietro per la mappa cercando il player. Se il player si trova ad una distanza minore di maxDistance, calcolata tramite una costante in EntityConstants a cui viene aggiunto un valore randomico, allora lo Status viene cambiato in Active.

- Active: se in nemico si trova in questo Status significa che ha individuato il player e da quel momento in poi lo seguirà sempre mantenendo però una "distanza di sicurezza". Se il Player dovesse oltrepassare questo limite allora l'enemy si sposterà indietro mantenendo però la mira puntata verso il Player. Dopo aver eseguito il movimento, all'interno di controllerTick viene inoltre chiamato il metodo fire() il quale controllerà se il player si trovi ad una distanza ravvicinata; in tal caso modificherà, tramite Enemy.setFire() un booleano. Questo verrà controllato ad ogni frame dal Controller che, se la condizione avrà risultato positivo, chiamerà il metodo CharacterController.fire() sul controller dell'enemy associato.

L'Enemy è inoltre in grado di superare ostacoli semplici controllando se di fronte ai suoi "piedi" e direttamente davanti a lui ci sono dei Tile "Collidable". In tal caso salterà per cercare di superarli.



Stage

Lo scopo dello StageImpl è quello di raggruppare al suo interno tutte le parti di model ed essere in grado di fornirle alle classi che lo richiedono.

Al suo interno troviamo 4 campi:

- player: fa riferimento al model del Player
- enemies: collection contenente tutti gli Enemy presenti nel Level
- collection contenente tutti i Bullets attualmente presenti nella scena
- level: fa riferimento al model di Level

Tommaso Turci

Level

Il livello è implementato attraverso una lista di Segment, passati in input al costruttore del Level stesso, ed è l'elemento più complesso che compone la mappa a livello di Model. Il Level è l'elemento tramite il quale le altre classi possono interfacciarsi con i Segment e rispettivi Tile, tramite l'utilizzo di più metodi interessati nello scorrimento della suddetta lista. Il metodo più frequentemente impiegato dalle altre classi è "getSegmentatPosition()"; grazie ad esso l'interazione con il livello è limitata a un singolo Segment per volta, gestendo più correttamente numerosi casi di accessi nulli (nonostante ciò, è comunque possibile far dialogare elementi presenti in due Segment diversi).

Segment

La mappa di gioco è percorribile dal giocatore da sinistra a destra, separata in "segmenti", aree di gioco popolate da nemici che devono essere sconfitti prima di poter avanzare. Un Segment è composto da un assortimento di "Tile" e "Marker Speciali" posizionati secondo come indicato nella TextMap di ciascun Segment: tale TextMap è un file di testo che contrassegna la posizione di ogni Tile e Marker Speciale all'interno di un Segment, e il tipo di ognuno, tramite un carattere alfanumerico. Questo sistema facilita la creazione di nuovi livelli, in quanto gli unici "step" da percorre per l'implementazione di nuovi livelli consistono nel creare i file di testo opportunamente e aggiungere le corrispondenti textmap alla lista contenente ogni segmento. Ogni Segment, per dialogare con altri elementi del gioco, come giocatori, proiettili o nemici, ha a sua disposizione più metodi in grado di restituire varie informazioni: ogni Segment è in grado di restituire le proprie dimensioni in numero di Tile, può restituire un Tile a una posizione specifica, ma anche operazioni più complesse come restituire tutti i Tile collisivi. Inoltre, un segment può anche contenere informazioni riguardo a certe componenti di gioco diverse

da Tile, i precedentemente menzionati "Marker Speciali": questi ultimi sono contrassegnati nella TextMap tramite caratteri speciali e sono di due tipi, "Player Spawn" (p) e "Enemy Spawn" (e). Essi servono a comunicare al gioco dove posizionare il giocatore a inizio partita con quanti nemici popolare il Segment. Le maggiori difficoltà emerse nella gestione dei Segment sono sorte dalla conversione delle posizioni espresse in Unità di Gioco (double) al sistema "a caselle" dei Segment; nonostante ciò, il divario di "granularità" dei due sistemi di riferimento può essere superato tramite una semplice formula matematica, implementata in ogni metodo che restituisce un tile.

Tile

Il Tile è l'elemento base che compone la mappa di gioco, conseguentemente la sua implementazione è relativamente semplice, seppur presenti qualche accorgimento su cui approfondire. Fondamentalmente, per l'interazione con il controller, ogni Tile presenta un solo metodo che ritorna un booleano, "isCollidable()"; come si può intuire dal nome, essa decreta se un Tile è collisivo, per esempio come il TileStone che compone maggior parte del livello, oppure no, come l'aria, contrassegnata nella TextMap da uno 0. I restanti metodi restituiscono valori concernenti l'aspetto visivo del livello e dei Tile stessi, e verranno esplorate più a fondo nella sezione successiva.

Autotile Manager

Il livello è una componente non particolarmente complessa a livello di model, ma a causa della complessità organizzativa dettata dalla separazione in Segment, il codice per visualizzare correttamente un livello risulta più complicato. Le principali difficoltà sono state causate dal cosiddetto "sistema di Autotiling", implementato tramite la classe AutotileManager. Ciascun Segment viene "percorso" dall'algoritmo a inizio livello, e le risorse vengono caricate. I Tile incontrati devono essere posizionati a schermo seguendo la "griglia" della TextMap del segmento, ciò è fatto tramite la creazione di gruppi di ImageView. Ognuno di questi gruppi sono "accorpamenti" di ImageView, ciascuna necessaria per visualizzare diverse porzioni del Tile stesso. Ciò è necessario perché ogni Tile possiede un aspetto unico e dinamico composto da 4 sezioni (angolo alto-destra, angolo alto-sinistra, angolo basso-destra, angolo basso-sinistra) che vengono alterate per permettere al Tile di "connettersi" esteticamente ad altri Tile adiacenti (tale feature è disabilitabile per ciascun Tile andando a cambiare il valore booleano restituito dal metodo "isTileable()"). Questo processo, oltre al suo peso computazionale, introduce una nuova problematica, ovvero quella del calo di performance dovuto

dal grande numero di elementi ImageView renderizzati a schermo. Questo problema si era già palesato nelle prime versioni dell'AutotileManager, ancor prima dell'implementazione dei segmenti, quando ancora il livello di gioco era trattato come un unico grande segmento. La suddivisione in Segment è stata introdotta proprio per questo, per poter "nascondere" le parti del livello non più a schermo, eliminando le rispettive ImageView. Durante l'esecuzione del metodo "refresh()", chiamato nel GameLoop dal Controller, la LevelView (classe incaricata di gestire l'occultamento e comparsa delle ImageView prima citate) controlla che il giocatore abbia avanzato o meno di Segment. Nel caso si abbiano sconfitto tutti i nemici e ci sia spostati nel Segment successivo, la LevelView esegue una transizione animata per raggiungere il segmento successivo, per poi "de-caricare" rimuovendo tutte le rispettive ImageView del segmento appena superato (non più utili, dato dal fatto che non è possibile ritornare al Segment precedente una volta avanzati). Infine, anche il segment successivo viene caricato, così da permettere una visualizzazione corretta della transizione nel caso il giocatore avanzi al Segment seguente.

CameraManager

Il CameraManager gestisce il ridimensionamento di tutti gli elementi a schermo, per fare si che l'applicazione sia utilizzabile su qualsiasi tipo di schermo. Ciò è ottenuto alterano la proprietà ScaleX e ScaleY di una Camera. Questo ridimensionamento è eseguito in modo intelligente, non permettendo al giocatore di vedere oltre alla fine del Segment corrente, ma al tempo stesso evitando "stiramenti" delle immagini, e assicurando sempre che il gioco riempia l'interità dello schermo. Il CameraManager agisce anche sullo scorrimento della schermata lungo i Segment, sia nella fase di "transizione", sia nel caso il Segment non sia interamente inquadrabile a schermo (questo può dipendere dalla dimensione del Segment stesso ma anche dalla risoluzione), consentendo alla telecamera di "scorrere" (Questo effetto è ottenuto non spostando la telecamera, bensì traslando le ImageView presenti sulla root) dinamicamente.

Andrea Biagini

Per lo sviluppo del gioco è stato congiuntamente deciso di utilizzare un gameloop, che nella sua semplicità simula l'esecuzione contemporanea di più parti di codice, processandole l'una dopo altra in sequenza.

Controller per gestione delle armi

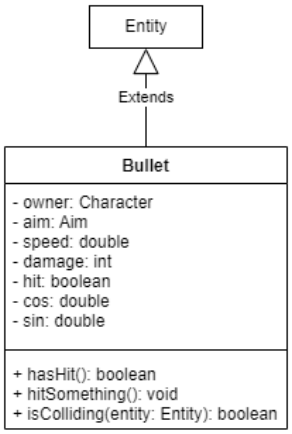
Una delle principali difficoltà riscontrate nell'implementazione delle armi è stata la gestione del tempo tra uno sparo e l'altro e della durata della ricarica di un'arma; per risolvere questi problemi è stata introdotta la meccanica del cooldown. Nel controller che gestisce le armi sono mantenuti dei Cooldown associati ai singoli Character, che a ogni tick vengono fatti avanzare; quando un Character spara, viene controllato l'ultimo Cooldown ad esso associato: se non è ancora terminato allora non può sparare, mentre se è terminato può sparare; in quest'ultimo caso, il Character in questione spara, e gli viene associato un nuovo Cooldown azzerato.

Inizialmente era stata presa in considerazione la possibilità di creare thread separati, ma la proposta è stata messa da parte, in quanto generava problematiche ritenute più difficili da gestire, come la sincronizzazione con il gameloop e l'appesantimento del gioco, e non si voleva considerare così presto l'utilizzo di più thread, quindi ci si è sforzati di trovare una soluzione alternativa.

La soluzione attuata conferisce comunque un certo grado di pesantezza all'esecuzione, dovuta all'incremento di tutti gli oggetti Cooldown e il controllo degli stessi a ogni game tick. In futuro, per ottimizzare l'esecuzione dell'applicazione, si potrebbe attuare una strategia multithreading, ponendo particolare attenzione alla sincronizzazione con il resto del gioco.

Controller per la gestione proiettili

A ogni game tick si controlla se ciascun proiettile collide con un Character (diverso da quello che l'ha sparato), Player incluso, o un tile della mappa. Quando un proiettile collide, esso viene eliminato dalla collezione di proiettili presenti in gioco. Inoltre, questa classe rimuove i proiettili che sono fuori dal Segment visualizzato a video (quello in cui è il Player) e quelli che si avvicinano ai bordi del Level, che quindi hanno una delle due coordinate uguale a zero.



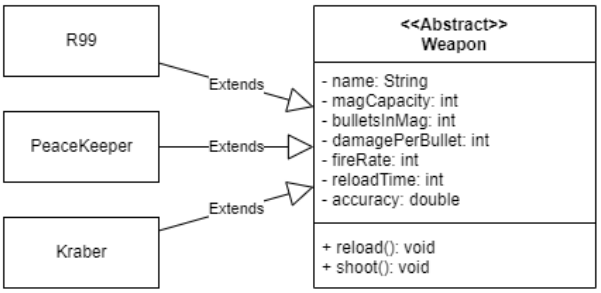
Aspetti View dei proiettili

Per quanto riguarda la parte grafica dei proiettili, si è dovuto fare i conti con le prestazioni: inizialmente venivano rimossi ImageView dalla lista di oggetti a video, ne venivano reistanziati di nuovi con le posizioni aggiornate, e questi venivano mostrati; effettuare questa operazione a ogni game tick si è rivelato essere molto impattante nell'esperienza di gioco.

Per risolvere questo problema, ogni proiettile è stato ridotto (a livello di View) a una posizione, tralasciandone la direzione; a ogni game tick, la View ottiene dal Model una collezione di posizioni: se la cardinalità di questa è uguale al numero di ImageView attualmente a video, allora le vecchie posizioni dei proiettili sono sostituite da quelle aggiornate. Se la cardinalità è maggiore, oltre all'aggiornamento delle posizioni, vengono istanziati gli ImageView che servono per mostrare a schermo i proiettili aggiunti. Se invece la cardinalità è minore, gli ImageView non utilizzati dopo l'aggiornamento vengono semplicemente rimossi.

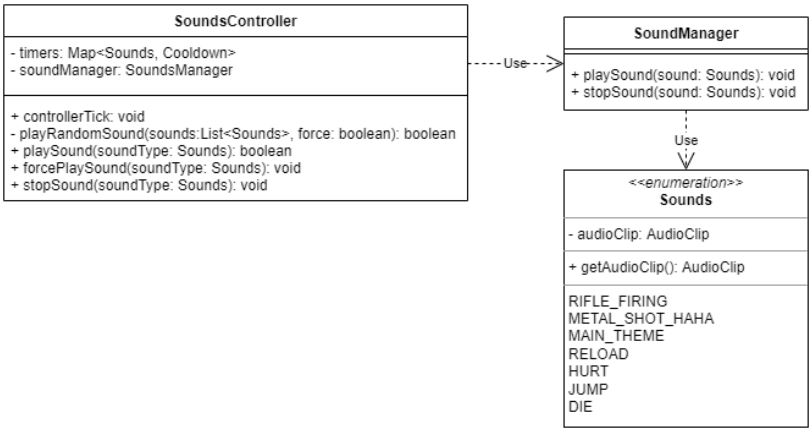
Implementazione Model delle armi

Nel Model, le armi sono implementate utilizzando una classe astratta Weapon, contenente tutti i campi e i metodi necessari alla caratterizzazione di un'arma, e creando classi che estendono Weapon, il cui compito è solo quello di impostare i valori nella superclasse. In questo modo, è stato costituito uno Strategy Pattern, in cui la classe astratta Weapon funge da interfaccia, e le armi definiscono i valori da utilizzare; Character ha infatti al suo interno un campo Weapon.



Suoni

I suoni sono riprodotti utilizzando un elemento di JavaFX, che permette di riprodurre con facilità più suoni contemporaneamente. Anche qui è stata utilizzata la classe Cooldown, per gestire intervalli tra la riproduzione di due suoni uguali o dello stesso tipo (HURT, JUMP e DIE); per esempio, si vuole evitare che un Character ripetutamente colpito da un gran numero di proiettili in un breve lasso di tempo causi la riproduzione dello stesso tipo di suono molte volte temporalmente vicine tra loro.



Capitolo 3

Sviluppo

3.1 Testing automatizzato

Si è deciso di testare tutte le componenti di model che girano intorno al Player oltre che il player stesso in quanto questo caratterizza forse la componente più importante di ogni gioco arcade. Nello specifico sono state testate: Aim, Health, MovableEntity e PlayerBuilder.

3.2 Metodologia di lavoro

Filippo Gurioli

Il mio ruolo nel progetto era la creazione di tutta quella parte di model che gira intorno al Player, Player compreso.

Nel particolare ho sviluppato le classi Entity, MovableEntity, Character, Player, Aim, Health, una parte di CharacterController e i vari menu.

Matteo Susca

Sviluppo dello Stage e model di Enemy per quanto riguarda la parte di model.
Sviluppo di EnemyController e delle diverse AI per il movimento e attacco automatizzati del nemico.
Logica dietro fire() all'interno di CharacterController FXML e Controller di HUD
Gestione dei file di tutte le classi per la corretta generazione del jar e il suo funzionamento in sistemi operativi differenti.

Tommaso Turci

Realizzazione della mappa in tutti i suoi aspetti (Level, Segment, Tile) a livello di Model e realizzazione delle classi per una corretta visualizzazione di essa (LevelView, AutotileManager);

Realizzazione della classe Animation, classe di utility volta a implementare un sistema in grado di gestire animazioni (Utilizzata da CharacterView);

Andrea Biagini

Implementazione delle armi e gestione del rateo di fuoco e del tempo di ricarica

Implementazione dei proiettili, gestione delle collisioni con tile e con Character, visualizzazione a schermo

Implementazione dei suoni

3.3 Note di sviluppo

Filippo Gurioli

Durante lo sviluppo del software mi sono trovato ad usare:

- File fxml e css (sviluppo di menù)
- Utilizzo della libreria Javafx

Matteo Susca

All'interno di Controller e GameView vengono utilizzati spesso:

- Stream
- Lamba Expression

Entrambe sono utili per lo scorrimento della Collection di EnemyController ed Enemy o della mappa ¡Enemy,EnemyView¿.

Tommaso Turci

- Lambda Expression (Segment)
- Stream (Segment)

- Optional (Segment, Level)
- Reflection (Segment)
- JavaFX

Andrea Biagini

- Lambda expressions: utilizzate con le collezioni, spesso nei metodi Collection.forEach()
- Optional: utilizzati nel rilevamento delle collisioni
- JavaFX: nella parte View dei proiettili e per la riproduzione di suoni con la classe AudioClip.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Filippo Gurioli

Mi sono sentito molto coinvolto dal progetto in quanto l'idea di creare un software che anche solo si avvicinasse a Metal Slug è stata mia. Dato che la mia parte riguardava principalmente sezioni di model non ho avuto troppe difficoltà nelle loro implementazioni. Riconosco di aver puntato troppo in alto volendo raggiungere un risultato che, con il tempo a disposizione, non sarebbe mai stato fattibile.

Mi piacerebbe portare avanti il progetto a livello personale proprio per raggiungere quello che con il monte ore non sono riuscito a raggiungere.

Matteo Susca

Personalmente il progetto mi ha coinvolto molto e il creare qualcosa che mi piacesse come un gioco sicuramente ha alleggerito il carico di lavoro percepito. Mi rendo però conto di aver gestito male le tempistiche e di aver lavorato troppo lentamente nella prima metà dello sviluppo. Nonostante le difficoltà sono contento del risultato finale ottenuto che, andando oltre le mie capacità ad inizio progetto, mi ha spinto durante la realizzazione ad imparare tecniche di sviluppo differenti.

Sarebbe interessante continuare a sviluppare ed ottimizzare il progetto in futuro in modo da renderlo ancor più simile a Metal Slug.

Tommaso Turci

La realizzazione del mio incarico mi ha portato a approfondire la mia comprensione del "modo di pensare" OOP, nello specifico una delle sfide affrontate è stata la necessità di rispettare il pattern architetturale MVC. Quest'ultimo ha introdotto delle complicazioni nella fase di pianificazione del codice, ma in retrospettiva ha permesso la realizzazione di un codice molto facilmente comprensibile ed espandibile, consentendo una collaborazione "seamless" fra I miei colleghi (per esempio, la separazione completa fra View e Model mi ha permesso di lavorare sull'ultima a pari passo con un mio compagno di progetto, impegnato sulla prima). Scendendo più nello specifico, la difficoltà maggiore è nata dalla gestione della libreria JavaFX, la quale introduceva numerosi nuovi oggetti e concetti avanzati, possibilmente al di là delle mie capacità correnti e del livello al quale la libreria era stata affrontata durante il mio studio. Ciò ha reso piuttosto ostica l'implementazione di certi elementi come la Camera, ma che tramite testing estensivo sono stati realizzati con successo.

Andrea Biagini

Mi ritengo complessivamente soddisfatto del pregetto svolto, anche se personalmente ho percepito una crescente impressione di sbagliare, di scrivere codice troppo C-like, o di sbagliare qualche operazione svolta in Git. Questo mi ha portato a vivere lo sviluppo di questo progetto un livello di ansia particolarmente alto, probabile motivo di alcuni errori da me commessi. Lo sviluppo di un eventuale futuro progetto mi risulterà sicuramente più "tranquillo", avendo già vissuto una esperienza (seppure piuttosto basilare) della programmazione di un gioco, della collaborazione con altre persone, dell'utilizzo di software come Git, e della consegna entro una deadline.

4.2 Difficoltà incontrate e commenti per i docenti

Andrea Biagini

All'inizio dello svolgimento di questo progetto ho notato che la difficoltà principale era riunire e "incastrare" le idee di tutti i membri del gruppo, principalmente per fare sì che ognuno mettesse del proprio; con il progredire del progetto, i problemi sono diventati sempre più pratici, a mio avviso imprevedibili, essendo per me il primo progetto "grande". Alcune difficoltà

pratiche che ho incontrato sono state con l'IDE Eclipse: mi è spesso capitato di trovare errori o warnings che andavano via solamente dopo una qualsiasi modifica al file e il suo salvataggio; inoltre, l'interazione con Git potrebbe essere leggermente migliore: molte volte mi sono trovato costretto a dover eseguire (più volte consecutive) il refresh o la reimportazione del progetto.

Matteo Susca

Personalmente ho riscontrato varie difficoltà nell'utilizzo dell'IDE Eclipse in quanto spesso dava errori non presenti risolvibili solamente con la ri-importazione del progetto.

A livello di codice invece ho trovato complesse molte parti di JavaFX soprattutto per quanto riguarda il movimento di Node all'interno di una Scene. Mi sono inoltre occupato in prima persona di tutta la parte relativa alla gestione della repository di Git e della risoluzione delle problematiche incontrate e man mano mi son reso conto sempre meglio di quanto in realtà sia in grado di utilizzare questo strumento ad un livello molto basilare.

Appendice A

Guida utente

Il gioco parte da un menù intuitivo che, dopo aver seguito dei semplici passaggi, manderà l'utente direttamente in gioco. Una volta iniziata la partita i comandi da utilizzare saranno:

- W: per mirare in alto
- A: muoversi e mirare a sinistra
- D: muoversi e mirare a destra
- S: accovacciarsi se a terra o mirare in basso se si è in volo
- J: sparare
- SPAZIO: saltare
- ESC: menù di pausa
- 1-2-3: cambiare le armi

Appendice B

Esercitazioni di laboratorio

B.0.1 Filippo Gurioli

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=87881#p138558
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p137688
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p137714
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p138888
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p138557
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p141187