Relazione progetto per “Programmazione a Oggetti”

Mario Biavati
Michele Ravaioli
Manuel Tartagni
Francesco Valentini

25 aprile 2022

Indice

Capitolo 1

Analisi

Il progetto scelto dai membri del gruppo consiste nella realizzazione di un videogioco partendo dalla costruzione di un game engine. Il concept e la meccanica di questa applicazione sono ispirati al gioco per mobile "No Humanity", che abbiamo osservato e studiato per capire come strutturare dalle basi il progetto. Abbiamo ideato quindi un gioco stile "bullet-hell" dove il giocatore deve comandare un palloncino finché non scoppia. L'obiettivo è resistere più a lungo possibile mentre si è continuamente bombardati con oggetti pericolosi. L'applicazione è stata chiamata "Don't Pop" per far intendere immediatamente ai giocatori lo scopo del gioco.

1.1 Requisiti

Requisiti funzionali

- L'applicazione dovrà avere tre componenti principali, facilmente intercambiabili tra loro: il menù di gioco, la scena di gioco e la classifica dei punteggi. L'utente potrà interagire con ogni scena per eseguire azioni, come per esempio cambiare scena.
- Durante l'esecuzione dell'applicazione dovrà essere visibile solo una scena per volta, quindi o menù, o scena di gioco, o classifica.
- Il menù dovrà essere la scena principale, quella che si mostra all' avvio del videogioco. Dovrà permettere all' utente di poter cambiare il prorio nome utente e di avviare il gioco vero e proprio.
- La scena di gioco dovrà avere al suo interno l'oggetto che il giocatore manovra (in questo caso dovrà essere un palloncino) e l'indicatore del punteggio del giocatore. Col passare del tempo il punteggio aumenterà e compariranno i "nemici", ovvero gli oggetti che se toccati col palloncino lo faranno scoppiare. Il gioco deve terminare quando si tocca un nemico col palloncino.
- Ci saranno quattro tipi di nemici, divisi per comportamento: proiettili, laser, palle spinate ed esplosioni. I proiettili proseguono con moto

rettilineo a velocità costante lungo tutta la scena di gioco. I laser compaiono con posizione casuale nella scena e dopo aver atteso un tempo predefinito, si attivano e diventano pericolosi (quando viene creato non è nocivo, mostra solamente in che punto comparirà); Le palle spinate si muovono con moto parabolico e vengono lanciate dai lati della scena; Le esplosioni, come i laser, compaiono nella scena casualmente e detonano dopo un tempo predefinito, coprendo un' area maggiore dei laser.

- Durante il gioco compariranno dei potenziamenti che se toccati col palloncino daranno vari effetti al giocatore.
- La classifica si dovrà aprire quando il gioco termina e dovrà mostrare il punteggio appena ottenuto del giocatore corrente e la classifica dei giocatori appena aggiornata. Dovrà permettere all' utente di poter tornare al menù principale oppure di rigiocare.

Requisiti non funzionali

- Il giocatore dovrà essere in grado di controllare il palloncino con rapidità e precisione, così da poter schivare i nemici. La scena di gioco sarà quindi molto interattiva.

1.2 Analisi e modello del dominio

L'applicazione deve essere in grado di cambiare la scena attiva su comando dell'utente, e potrà cambiare solamente in tre scene diverse: menù, gioco e classifica. La parte più problematica da progettare è quella del gioco. Ogni videogioco per poter funzionare ha bisogno di un *game loop*, un ciclo ripetuto fino alla conclusione che permette di far progredire lo stato di gioco. Nel ciclo il game engine dovrà eseguire per ogni oggetto di gioco: l'aggiornamento dello stato dell'oggetto, a seconda del comportamento che l'oggetto stesso dovrà seguire; Un controllo delle collisioni col giocatore, e agire di conseguenza se ne rileva una; Aggiornamento della grafica dell'oggetto nella scena; Calcolo del punteggio totale.
La meccanica interna di gioco verrà delegata alla classe con il compito di eseguire un ciclo che aggiorna lo stato e la creazione dei nemici e del giocatore, ovvero la loro posizione e le eventuali collisioni col giocatore.
Quest'ultima avrà anche il compito di gestire le factory riguardanti nemici e powerup in maniera da essere sincronizzata con il tempo di gioco.

Capitolo 2

Design

Il design di questo progetto si propone di sfruttare differenti design pattern e i principali concetti di programmazione a oggetti.

2.1 Architettura

Per avere un cambio di scena semplice e rapido, è stata progettata la classe GameApplication. Questa classe offrirà dei metodi pubblici per poter cambiare con facilità la scena corrente, senza però avere il controllo dello stato dell'applicazione. Il compito di cambiare scena non spetterà a questa classe, ma alle altre classi che gestiscono le singole scene, ovvero i gestori del menù, del gioco e della classifica, che modificano/gestiscono le scene a loro connesse. Facendo in questo modo, ogni gestore deciderà da sé quando passare alla prossima scena, inoltre risulta più semplice aggiungere scene nuove.
L'architettura della scena di gioco è costruita tenendo separata la parte di model dalla parte di grafica. L'unica interazione presente tra grafica e model si trova nel GameEngine, all'interno del *game loop*, il quale delega la rappresentazione degli oggetti ai componenti Renderer dell'oggetto stesso, utilizzando il *pattern Strategy* per implementare le diverse modalità di rappresentazione grafica. Lo stesso pattern è usato per il controllo delle collisioni con il player, usando l'interfaccia Collider.
All'interno del *game loop* dovranno essere eseguiti oltre all'aggiornamento degli oggetti di gioco presenti anche il ciclo di spawn dei nemici e il calcolo del punteggio, che vengono delegati a classi apposite.
Il ciclo di spawn verrà gestito dalla classe SpawnManager che delegherà la creazione dei singoli nemici e power up alle relative factory, seguendo il *pattern Builder*.

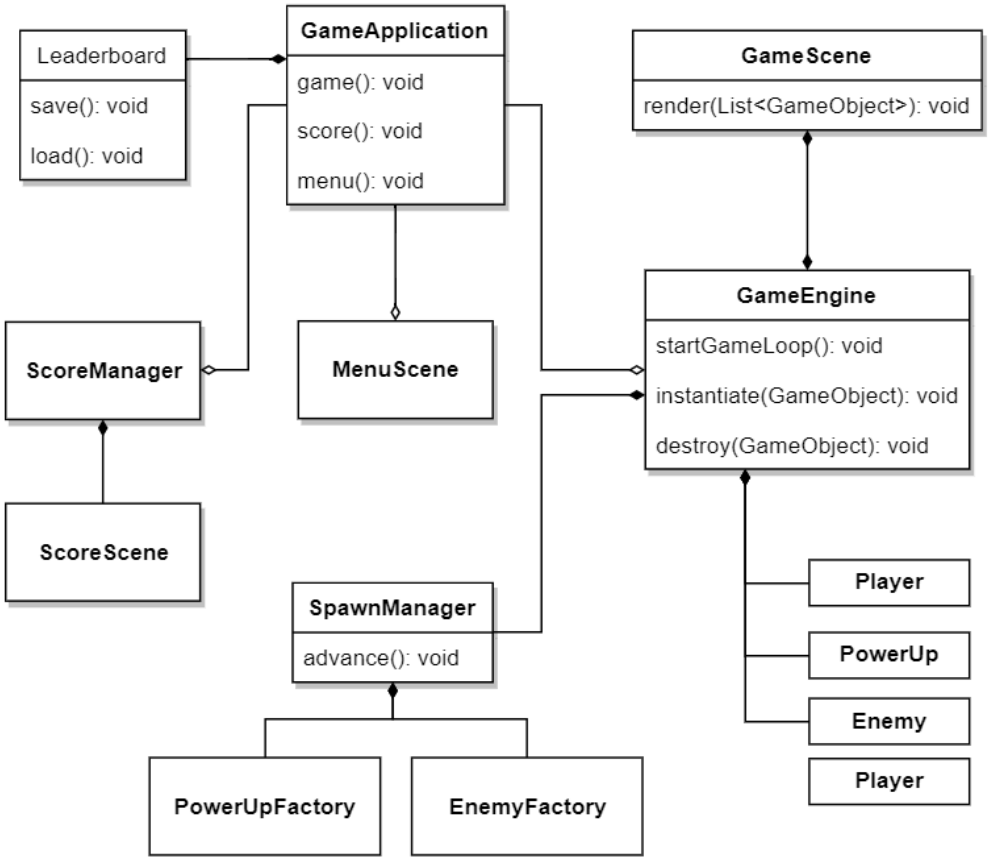

Fig. 2.1 : Schema generale dell'architettura usata

2.2 Design dettagliato

Diverse modalità di collisione

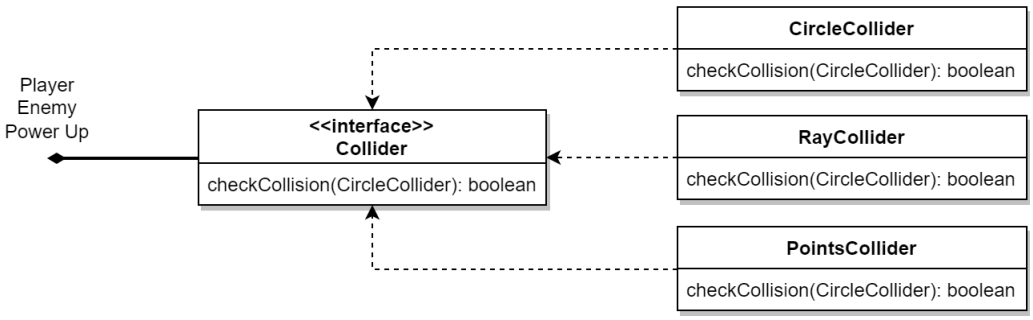

Fig. 2.2 : Piccolo schema che mostra la soluzione adottata riguardo alle collisioni.

Problema: Gli oggetti di gioco dovranno avere un modo di calcolare le proprie collisioni, tuttavia per molti oggetti queste modalità sono simili e quindi ripetute. Inoltre dovranno essere potenzialmente intercambiabili (se un nemico passa da quadrato a cerchio il metodo di collisione cambia).

Soluzione: Il sistema della gestione delle collisioni utilizza il *pattern Strategy*, come da figura (sopra). Il player, i nemici e i power up avranno associato un proprio Collider, che può essere implementato in più modi differenziandosi nel tipo di calcolo da eseguire per il controllo delle collisioni. Collider è implementato come CircleCollider (calcola come se fosse un cerchio), RayCollider (come se fosse una retta) e PointsCollider (come un insieme di punti nello spazio). Tutti i calcoli delle collisioni partono dal presupposto che il player sia composto da un CircleCollider.

Diverse modalità di renderizzazione degli oggetti

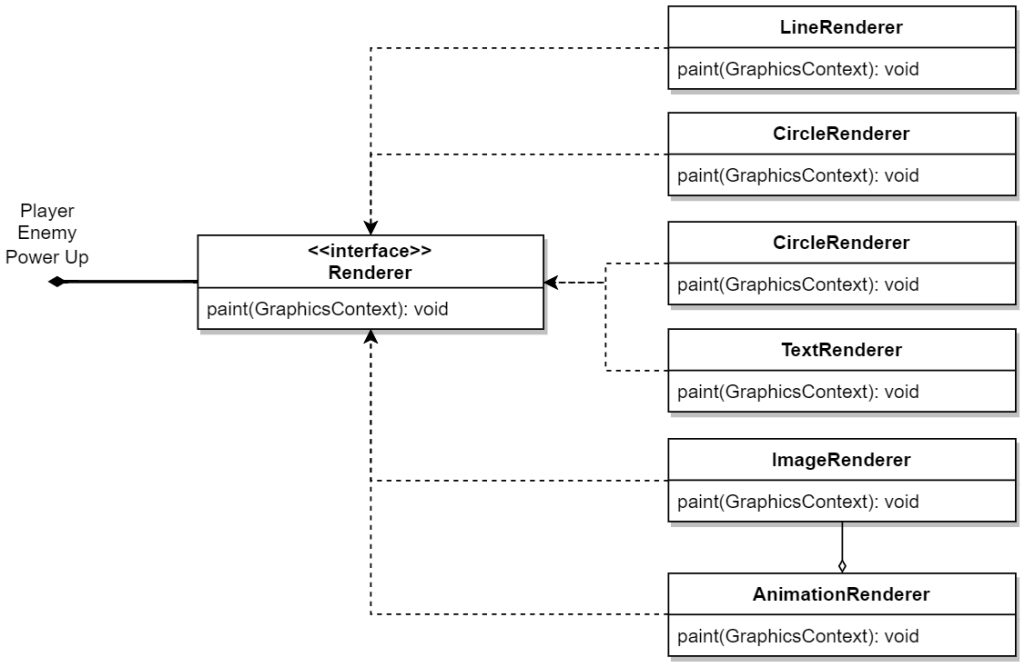

Fig. 2.3 : Piccolo schema che mostra la soluzione adottata riguardo alla renderizzazione degli oggetti di gioco.

Problema: Tutti gli oggetti di gioco dovranno essere renderizzati sulla scena di gioco senza ripetizioni di codice e in maniera diversa tra loro. Inoltre dovranno poter cambiare renderizzazione durante l'esecuzione del gioco.

Soluzione: Come per le collisioni, si è utilizzato il *pattern Strategy*. Si è creata quindi un'interfaccia Renderer che in base all'implementazione scelta dall'oggetto di gioco possa mostrare immagini, linee, cerchi oppure testi.

Ripetizione di codice in player, nemici e potenziamenti

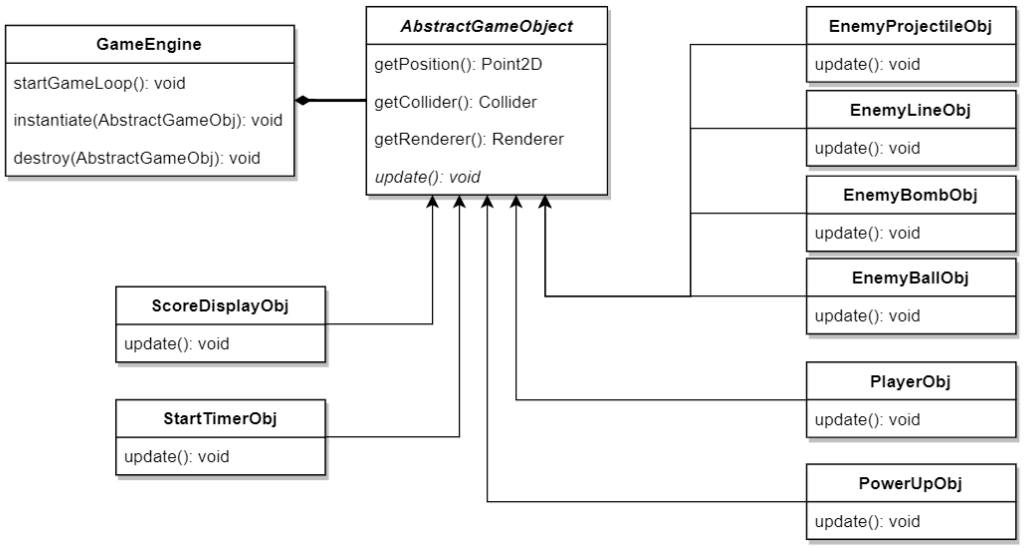

Fig. 2.4 : Piccolo schema che mostra la soluzione adottata riguardo agli oggetti di gioco.

Problema: Sia player, sia nemici sia potenziamenti presentano caratteristiche comuni, infatti tutti e tre hanno una posizione nello spazio, un Collider, un Renderer e un metodo update() per aggiornare il proprio stato di gioco.

Soluzione: Per evitare la ripetizione del codice si è utilizzata una classe astratta AbstractGameObject che contiene tutte le caratteristiche comuni degli oggetti di gioco (contiene i metodi getter e setter). AbstractGameObject ha un metodo astratto update() da implementare nelle classi che la estendono, così da poter avere oggetti con comportamenti differenti. Il GameEngine userà quindi non le singole implementazioni dei nemici, player

e power up, ma degli AbstractGameObject i quali verranno aggiornati, controllati e renderizzati dentro il *game loop*.

Game Engine come god-class

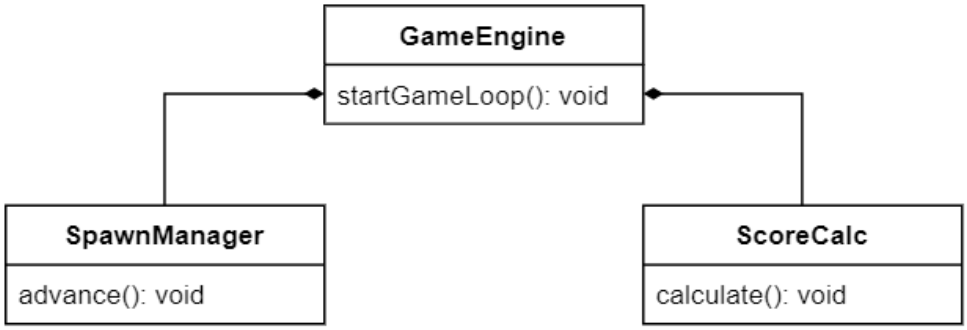

Fig. 2.5 : Piccolo schema che mostra le classi addette al gameplay.

Problema: Poiché deve eseguire aggiornamenti, controlli collisioni, rendering, calcolo punteggio e spawn dei nemici, il codice del GameEngine risulta lungo e poco leggibile.

Soluzione: Il problema della creazione viene delegato allo SpawnManager per la creazione randomica dei potenziamenti e temporizzata per i nemici secondo i timer associati ad ogni categoria di nemico. Anche il calcolo del punteggio viene assegnato a un'altra classe ScoreCalc, la quale calcola basandosi sul tempo di gioco trascorso ed effettuando eventuali operazioni sul punteggio dovute ai power up.

Creazione dei nemici

Problema: I costruttori dei nemici sono complessi e ricchi di parametri, pertanto il codice della gestione dello spawn risulta poco leggibile e molto complesso.

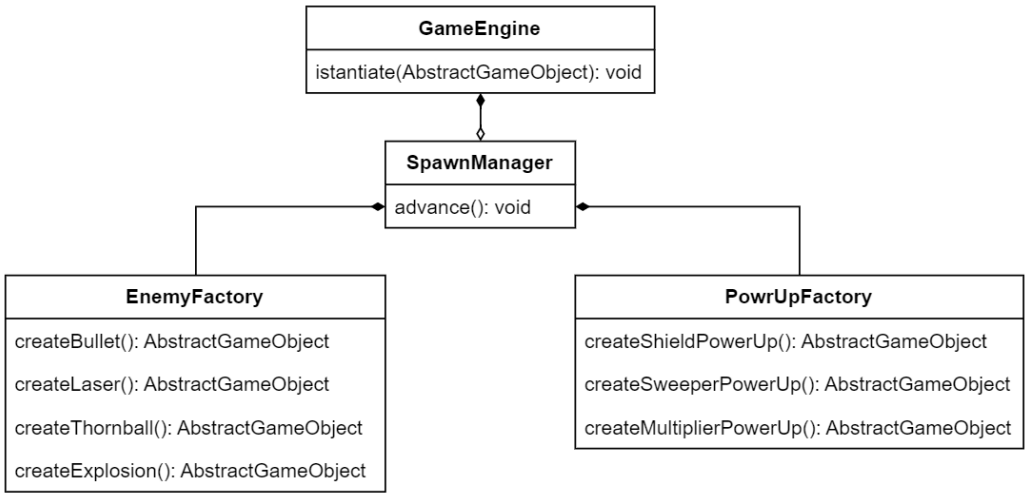

Fig. 2.6 : Piccolo schema che mostra i builder utilizzati.

Soluzione: Seguendo il *pattern builder* lo SpawnManager crea tramite le due factory (EnemyFactory e PowerUpFactory), come nella figura (sopra). I potenziamenti vengono creati in maniera randomica mentre i nemici seguiranno dei tempi stabiliti dai rispettivi timer segnati nello SpawnManager, differenti per ogni tipologia di nemici. Lo SpawnManager dovrà solamente istanziare nel GameEngine gli oggetti creati con le factory secondo i timer.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il test delle varie funzionalità è stato eseguito utilizzando la libreria JUnit5 di Java, in modo tale da avere un test automatico delle classi. Seppur siano test minimali e semplici, sono importanti per capire il funzionamento dei vari componenti dell'applicazione.

Sono stati eseguiti test per le collisioni, dove si provano le varie implementazioni di Collider, così da vedere se la collision detection è calcolata correttamente sia per i cerchi sia per le rette.
Test anche per la creazione, tramite enemy factory di nemici, controllando che la tipologia di nemico creata fosse effettivamente quella desiderata.

3.2 Metodologia di lavoro

Si è cercato di suddividere il carico di lavoro tra gli sviluppatori del progetto in maniera bilanciata, così da non avere membri del gruppo che hanno programmato più degli altri. Anche la parte grafica del progetto è stata divisa equamente. Il lavoro è stato suddiviso nella seguente maniera:

Michele Ravaioli - gli è stato assegnato il compito di programmare i Collider e le varie implementazioni delle collision detection (package game.collider); I Renderer e le rispettive implementazioni (package game.renderer); la classe AbstractGameObject (package game.model); La GameScene come parte di grafica (package game.ui). Nel package game.util ha creato la classe Point2D e ha aiutato nello sviluppo delle classi GameEngine, AudioManager, SpawnManager (game.engine), Leaderboard e ScoreCalc(game.util).

Manuel Tartagni - gli è stato assegnato il compito di implementare le classi delle factory dei nemici (EnemyFactory , package: game.engine) e dei potenziamenti (PowerUpFactory , package: game.engine) e le relative implementazioni. Sviluppo della classe WhereToSpawn pì (game.util) e RandomInt (game.util) utilizzata per la scelta del lato spawn. Nel Package game.engine ho implementato lo SpawnManager e il GameApplication (funge da contoller). Classe grafica HowToPlay e HowToPlayController (entrambe nel package game.ui).

Francesco Valentini - Classi assegnate:

- GameEngine (game.engine)
  Questa classe è il *core* del videogioco: gestisce il coordinamento tra i vari componenti, impartendo a ciascun oggetto il compito da svolgere. In particolare, ordina alla GameScene di fare il *refresh* della scena di gioco 60 volte al secondo (60Hz).

- ScoreManager (game.engine)
  Gestisce il caricamento dei dati nella ScoreScene: invoca la

Leaderboard, fa il *fetch* dei dati, li salva in un array e infine li mette al posto giusto dentro la tabella della ScoreScene.

- MenuScene (game.ui) + controller, fxml, css
  Mostra il menù principale del gioco, permettendo al giocatore di iniziare il gioco, visualizzare la classifica corrente, o vedere le regole di gioco in una schermata apposita.

- ScoreScene (game.ui) + controller, fxml, css
  Mostra la classifica dei 50 giocatori migliori in una tabella, scartando quelli con i punteggi più bassi. Se questa classe viene invocata dopo il gameover, mostra anche il punteggio del *player* corrente.

- Pair (game.util)
  Utility class che memorizza coppie di valori <*String, Integer*>.

- RankItem (game.util)
  Memorizza tutti i dati relativi a un giocatore: posizione in classifica, nome e punteggio ottenuto.
  Il controller della ScoreScene converte i dati ottenuti dalla Leaderboard da un array di Pair a un array di RankItem: mentre scorre la classifica della Leaderboard, si salva la sua posizione nel campo apposito di RankItem. In questo modo, la PropertyValueFactory (nel file menuscene.fxml) deve semplicemente fare la *fetch* dei dati contenuti in questo array, invece di scorrere la lista nella Leaderboard ad ogni iterazione, diminuendo la complessità computazionale da $O(n^2)$ a $O(n)$.

- ScoreCalc (game.util)
  Calcola e memorizza il punteggio corrente, tenendo conto di eventuali moltiplicatori.

Mario Biavati - Gli è stata assegnata l'implementazione delle classi del model, quindi nemici (EnemyProjectileObj, EnemyLineObj, EnemyBallObj, EnemyBombObj), powerup (PowerUpObj), grafica del timer (StartTimer), grafica del punteggio (ScoreDisplay) e la classe del giocatore (PlayerObj). Ha contribuito anche ad aggiungere effetti sonori per eventi come ottenere powerup o essere colpiti, e ha contribuito alla classe menuScene.

Utilizzando il DVCS Git, abbiamo lavorato sul progetto contemporaneamente, evitando di sovrapporci. Ognuno lavorava sui package assegnati, senza creare conflitti con altri file.

Il progetto è stato sviluppato usando l'IDE Eclipse, come progetto di JavaFX. Per poter esportare l'applicazione in un jar cross-platform, siamo infine passati a un progetto Maven, così da usare il plugin per il shadow jar.

3.3 Note di sviluppo

Per il progetto è stata utilizzata la libreria JavaFX per la grafica, scelta per motivi di performance e praticità.

Michele Ravaioli:

- Uso di lambda expressions nei cicli for each del GameEngine;
- Uso di stream per il ciclo di controllo delle collisioni nel GameEngine;
- Uso di funzioni di libreria di JavaFX per le implementazioni del metodo render(GraphicsContext) dell'interfaccia Renderer;
- Uso di lambda expressions Runnable per l'esecuzione degli eventi nella classe ScoreCalc (onMultiplierStart e onMultiplierEnd);

Manuel Tartagni:

- Uso di funzioni di libreria di JavaFX per le implementazioni delle classi HowToPlay e del rispettivo controller HowToPlayController.
- Utilizzo di JavaFX Scene Builder per la creazione della scena grafica e per la creazione del file .fxml
- Utilizzo di file Fxml per creare la scena e associare al bottone presente l'azione corrispondente.

Francesco Valentini:

- Utilizzo di lambda e stream per effettuare iterazioni sulle liste del GameEngine
- Utilizzo di Scene Builder per velocizzare la creazione di una scena grafica in FXML
- Utilizzo di file CSS per definire l'aspetto della GUI in modo dettagliato

Mario Biavati:

- Uso di funzioni di libreria di JavaFX per renderer e MenuScene
- Utilizzo di SceneBuilder per MenuScene
- Utilizzo di lambda expressions in ScoreDisplayObj

Capitolo 4

Commenti finali

Grazie al tempo speso nella stesura dello schema finale del progetto, comprendente tutte le varie classi, abbiamo decisamente semplificato la programmazione successiva, dato che avevamo già una traccia da seguire. Per tutti e quattro, la parte decisamente più complessa è stato imparare a costruire l'interfaccia grafica, in quanto eravamo principianti nell'uso della libreria JavaFX.

4.1 Autovalutazione

Michele Ravaioli: Ritengo di aver eseguito un buon lavoro, soprattutto nella parte di analisi e di design. Mi sono impegnato molto nel cercare la migliore progettazione possibile per il game engine e per la meccanica di gioco, facendo molte prove per conto mio. Complessivamente sono soddisfatto del lavoro svolto, infatti molte parti del progetto possono essere riusate per progetti futuri (es: il game engine, collisioni, renderer, cambio scene etc...). Inoltre ho imparato a usare bene la libreria di JavaFX, che utilizzerò molto d'ora in poi.

Manuel Tartagni: Ritengo di aver lavorato bene e sono soddisfatto del lavoro svolto in quanto ha coperto non solo la parte di model ma anche quella grafica. Ritengo utile la parte grafica in quanto mi ha permesso di utilizzare stumenti come il java fx Scene Builder e i file di formato xml.

Francesco Valentini: Sono decisamente contento del risultato che abbiamo ottenuto: nonostante tutte le varie difficoltà che abbiamo incontrato, tipiche di ogni progetto di questo tipo, siamo riusciti a realizzare un videogioco a mio parere carino e molto giocabile. Vedere la nostra idea diventare realtà mi ha dato un grande senso di soddisfazione.
Personalmente, ritengo di aver svolto bene il mio lavoro, mettendoci della dedicazione, soprattutto nella parte di realizzazione dell'interfaccia grafica con JavaFX. È stato lì che ho imparato a usare lo Scene Builder per velocizzare

la scrittura di codice FXML, che ho poi sempre revisionato, e a integrare dei file CSS per definire nei dettagli la GUI.
La scrittura del GameEngine è stata molto impegnativa, soprattutto perché è stata la prima volta in cui mi sono cimentato nella creazione di un videogioco, ma sono sicuro di aver imparato molte abilità che mi potranno servire in futuro.

Mario Biavati - Mi ritengo soddisfatto del lavoro svolto sul model, e delle soluzioni che ho implementato per i diversi tipi di nemici, per il giocatore e per gli elementi di grafica in gioco. Ho imparato molto dal progetto: come usare JavaFx, come usare SceneBuilder, come usare e modificare i file FXML e CSS, e come usare Git e GitHub in modo spedito ed efficace. L'unico rimpianto è che la parte a me assegnata si è rivelata meno impegnativa del previsto. Ho quindi cercato di assistere in altri ambiti al meglio delle mie abilità, ma comunque avrei voluto contribuire di più al progetto.

Appendice A

Per avviare l'applicazione è necessario lanciare il comando:

```
java -jar download/dont-pop.jar
```

Per aprire il progetto su Eclipse è necessario:

1. Clonare il repository;
2. Importare il progetto facendo *Import -> Existing Project into Workspace.*

Per avviare il progetto su eclipse è necessario:

1. Creare una nuova Maven Build configuration;
2. Inserire nel campo base directory `${workspace_loc:/Dont-Pop}`
3. Inserire nel campo goals `clean javafx:run`
4. Cliccare su Run.

Per avviare i test automatici è necessario:

5. Creare una nuova Maven Build configuration;
6. Inserire nel campo base directory `${workspace_loc:/Dont-Pop}`
7. Inserire nel campo goals `clean javafx:run`
8. Cliccare su Run.