Capitolo 1
Analisi

1.1 Descrizione e requisiti

RogueKong è una rivisitazione del famoso videogioco uscito nel 1981, chiamato Donkey Kong. Sviluppato come videogioco a piattaforme, dove il giocatore aveva come obiettivo raggiungere la cima dei vari livelli, superando ostacoli – anche mobili – come barili, nemici e saltando da una piattaforma all'altra.

RogueKong sarà una versione simile ma in chiave roguelike – ovvero dove i potenziamenti ottenuti saranno relativi solamente a quello specifico tentativo, e una volta vinto o perso, il giocatore comincerà senza di essi. Avremo più varietà di ostacoli e nemici, e a differenza dell'originale, RogueKong farà partire sempre dal primo livello in caso di sconfitta. Avremo anche la presenza, come accennato precedentemente, di vari aiuti come potenziamenti per aumentare le capacità del personaggio.

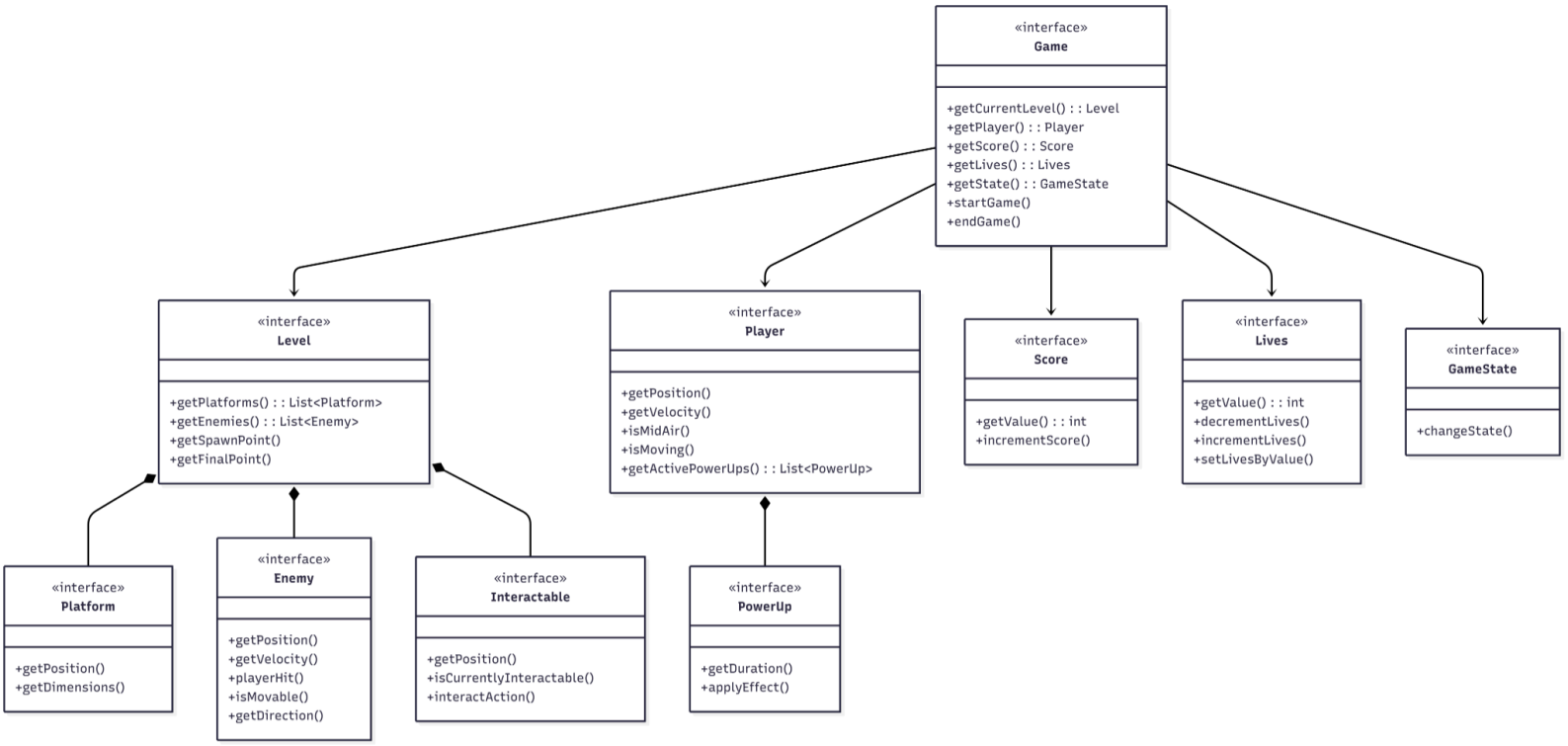
Requisiti funzionali

- Il giocatore dovrà essere in grado di poter muovere il personaggio orizzontalmente lungo le piattaforme, con la capacità di poter saltare.
- Il giocatore inizia la partita con 3 vite.
- Il gioco deve gestire le collisioni con muri oppure ostacoli/nemici (questi ultimi causeranno la perdita di una vita).
- Verranno gestiti punteggi qualora il giocatore superi ostacoli o raggiunga la fine.
- Quando il giocatore supera un livello che non sia l'ultimo, passa al livello successivo.
- Quando il giocatore supera l'ultimo livello, la partita sarà vinta, la classifica si aggiornerà con le statistiche del tentativo e posizionerà il punteggio nella relativa posizione in classifica.
- Nel caso di perdita di tutte le vite prima della fine dell'ultimo livello, la partita sarà persa e si aprirà un'interfaccia dove l'utente potrà scegliere se andare al menù iniziale o ricominciare.
- Tra un livello e l'altro, il giocatore potrà scegliere tra due potenziamenti, che lo aiuteranno a superare i livelli successivi.

Requisiti non funzionali

- Il numero di livelli verrà deciso in fase di progettazione e saranno diversi tra di loro, avendo caratteristiche uniche.
- Il giocatore potrà muovere il personaggio unicamente tramite comandi da tastiera.
- Il gioco potrà essere girato sui principali OS (Windows, Linux, macOS).
- Framerate variabile.

1.2 Modello del Dominio
Il gioco terrà conto del proprio stato (schermata iniziale, gioco in esecuzione, gioco in pausa), il punteggio, i punti vita e il livello attuale.
Verranno implementate interfacce per garantire una struttura gerarchica, come ad esempio l'interfaccia per dinamici e statici.
Per quanto riguarda il giocatore si dovranno considerare la sua posizione, velocità, stato (se sta saltando o è a terra), e potenziamenti attivi.
Per i nemici verranno considerati posizione e velocità. Mentre le piattaforme e altri oggetti statici di gioco, dovranno avere posizione e dimensione.
Il livello elencherà tramite liste le piattaforme e i nemici che contiene, il punto di inizio e punto di arrivo. Ogni potenziamento avrà durata e una funzione di applicazione e rimozione potenziamento.



Capitolo 2
Design

2.1 Architettura
L'architettura del videogioco RogueKong è basata sul modello MVC.
La sezione di Model comprende una serie di interfacce, queste ultime suddivise in entità di gioco, funzionalità e stati di gioco, e valori di gioco.

Entità di gioco
Giocatore, la cui implementazione registra e aggiorna posizione, vita, velocità, numero di salti massimi, etc…

All’interno di ogni livello sono presenti dei nemici, in grado di cambiare posizione e di danneggiare il giocatore.
PowerUp, suddivisi in sottoclassi diversi a seconda del tipo e degli effetti che ha sul giocatore (i PowerUp modificano direttamente i parametri del giocatore).

Funzionalità e stati del gioco

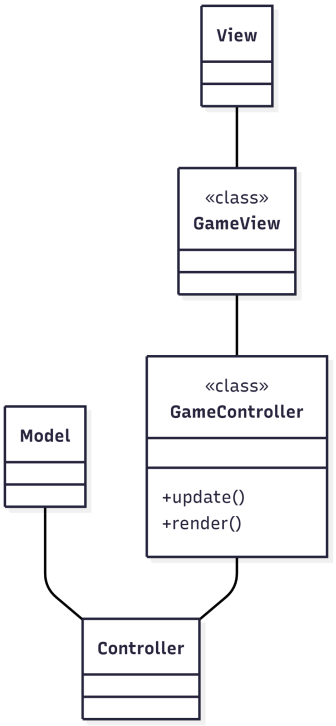
Stati del gioco, rappresentati da un’enumeration e gestiti dalla classe **GameStateImpl**, che modifica e memorizza gli stati attuali.
Piattaforme di gioco, la cui implementazione gestisce i parametri delle ‘tiles’ (blocchi con cui il giocatore interagisce tramite collisioni o ricevendo danno).
Livelli, le cui implementazioni gestiscono il punto di inizio e di fine, stato di completamento e la mappa che verrà visualizzata.

Valori di gioco

Posizione e velocità, necessarie a gestire il movimento del giocatore e dei nemici, e il loro rendering sulla scena.
Vite, che registrano e alterano lo stato delle vite (o punti vita) del giocatore.

La sezione di View gestisce principalmente i cambi di scene della pagina. Ogni scena, infatti rappresenta uno stato di gioco. Più scene possono essere rappresentate dallo stesso stato di gioco.
View si occupa anche di registrare e reagire ad eventi, nel caso l’utente interagisca con l’interfaccia tramite i bottoni proposti, o alla pressione di un tasto della tastiera.

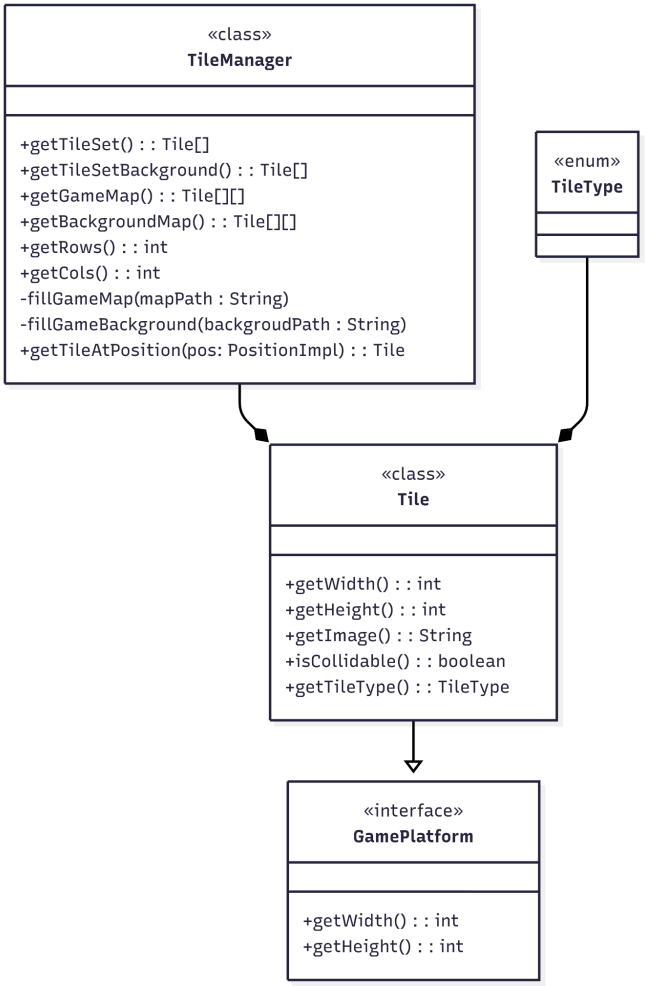
La sezione Controller è responsabile della gestione del flusso di gioco e di come l’utente interagisce con esso. Le componenti del Controller gestiscono quindi l’evoluzione della partita tra cui la progressione dei livelli, sincronizzazione tra aggiornamento e logica di rendering, salvare il punteggio del giocatore e gestire i suoni in base alle azioni dell’utente e dello stato di gioco.



2.2 Design dettagliato

Manuel Menghetti

Una delle parti fondamentali del progetto era la creazione dei vari livelli di gioco, i quali comprendevano delle mappe tile-based.



Problema:

Ogni livello presenta una mappa tile-based completamente diversa dalle altre, composte da tile con comportamenti diversi. Ad ogni livello quindi è necessario gestire la diversità tramite il caricamento di file, per rappresentare la mappa con relativo background.
Inoltre, sapere quali tipologie di tile contiene il livello, che comportamento ha la tile o sapere su che tipo di tile si trova il player è utile per gestire le collisioni con tile “cattive” e “buone”.

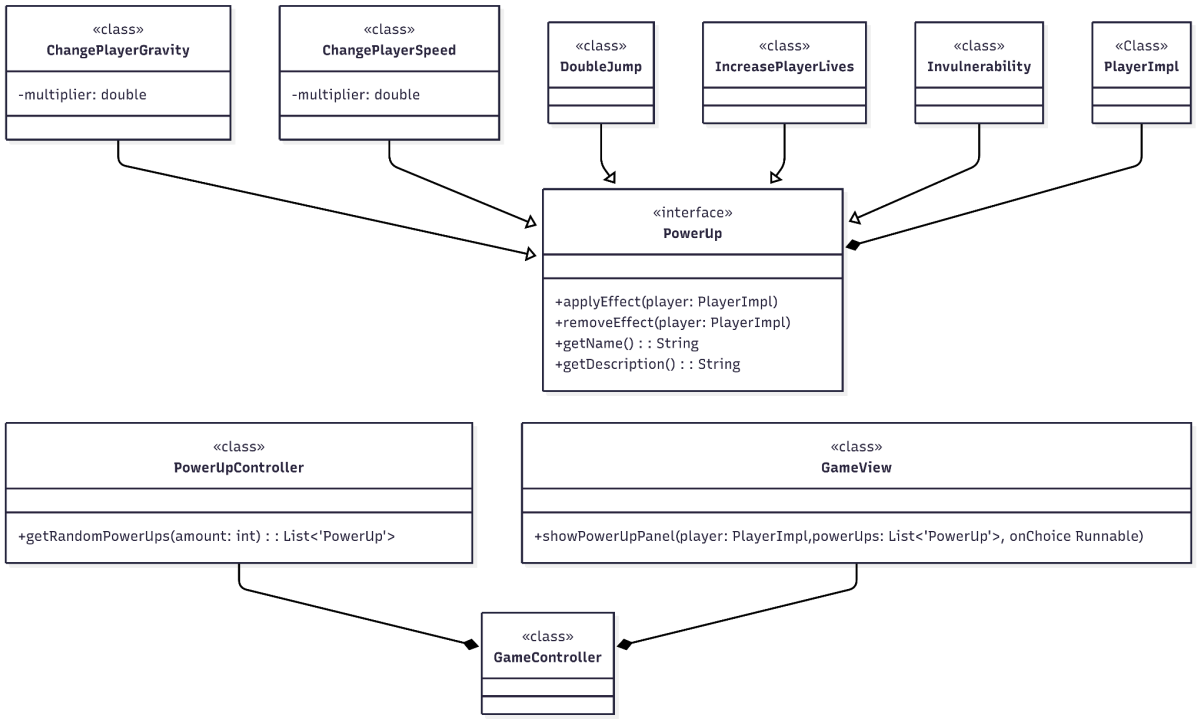
Soluzione:

Ad ogni livello creato, è stato dato un **TileManager** differente. Il suo scopo è quello di salvare, all’interno di array bidimensionali, la mappa di gioco e la mappa di background del relativo livello, inserendo i valori letti da file tramite le funzioni fillGameMap e fillBackgroundMap. I valori rappresentano le tile che andranno caricate.
I Tile sono stati modellati per poter contenere al loro interno, il relativo comportamento, come le tile con collisioni attive o con la possibilità di poter causare danno.

All’interno di TileManager vengono creati tutti i tile possibili e gestiti tramite array, cosi da poter facilitare nella parte di render, la visualizzazione dell’immagine.
Inoltre per garantire una gerarchia tra tile, viene usata la enum **TileType,** la quale elenca le tipologie di piattaforme che il il livello potrebbe contenere.

Fabian Calangiu

Una caratteristica distintiva del nostro gioco è l’applicazione di PowerUp e moltiplicatori sul giocatore ad ogni cambio di livello, in modo da rendere l’esperienza più variegata, meno ripetitiva e il gioco più rigiocabile.



Problema:

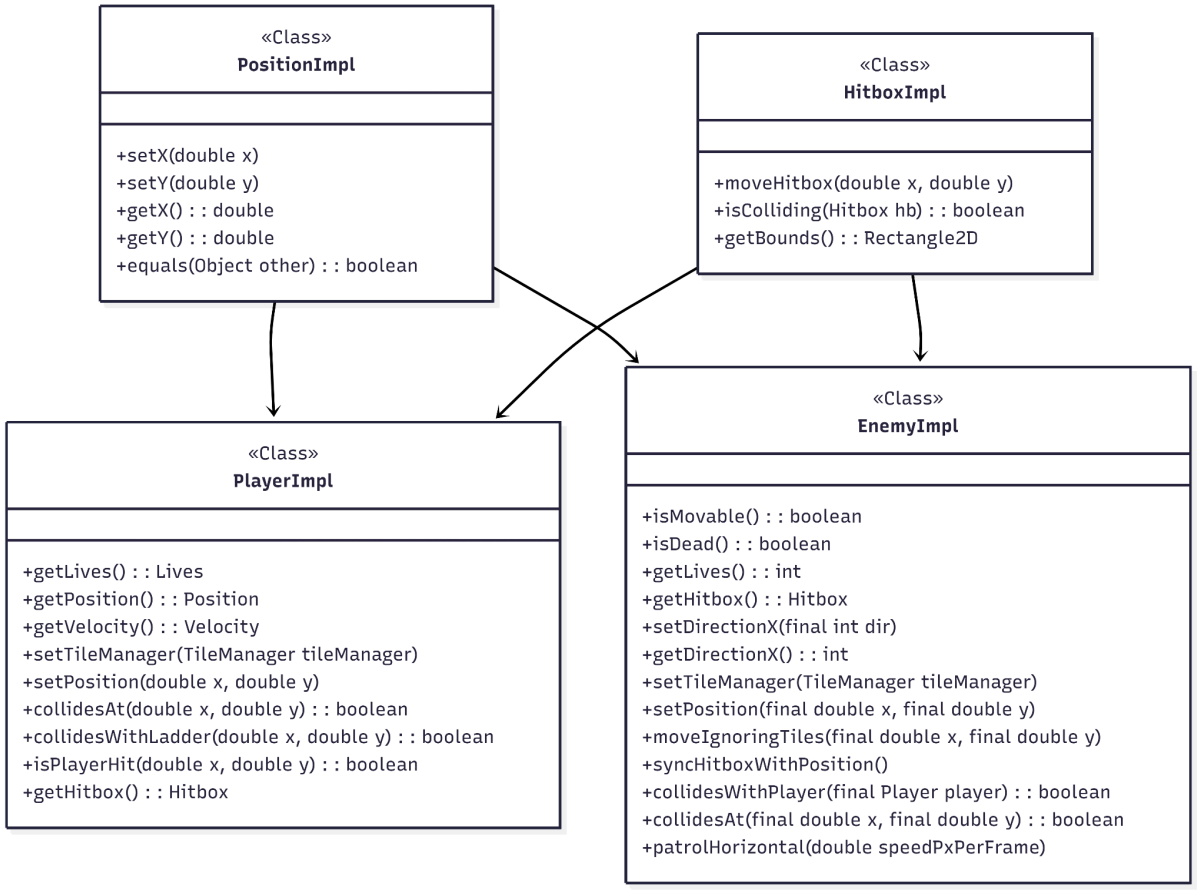
Ogni volta che il giocatore completa il livello, il manager dei PowerUp deve generare un numero casuale di PowerUp, far sì che il giocatore li visualizzi e li scelga tramite l’interfaccia, applichi l’effetto scelto e rimuova i PowerUp al completamento della partita.

Soluzione:

L’implementazione consiste in una lista di Power Up definita dal controller stesso che ritorna due potenziamenti casuali per ogni livello completato tra cui il giocatore può scegliere, e che quindi viene applicato allo stato attuale del Player.
Il giocatore visualizza le scelte attraverso un pannello sovrapponendosi all’interfaccia di gioco e mettendo temporaneamente in pausa il ciclo di gioco.
Il Power Up viene applicato immediatamente al player dopo la scelta del giocatore, modificandone i parametri come il numero di salti massimi, la gravità, la velocità, la vita e uno stato di invulnerabilità momentaneo.
Nel caso il gioco venga interrotto, sia nella sconfitta che nella vittoria, i Power Up azzerano i moltiplicatori sul giocatore.
Questo introduce più variabilità al gioco, concedendo al player di sperimentare con diverse combinazioni ad ogni partita.

Enrico Vampa

Una delle parti principali di mia competenza era la gestione delle Hitbox (parti di collisione) del giocatore e dei nemici.



Problema:
Il player e i nemici hanno bisogno di un controllo per la gestione delle collisioni con i Tile**.** Non era possibile implementare una classe Hitbox per ogni Tile della mappa, poiché era necessario creare ed aggiornare la posizione delle hitbox in modo corrispondente alla posizione delle tile che ne necessitano e soprattutto perché una gestione delle hitbox sulle tile avrebbe richiesto molta memoria per un controllo su ogni Tile ad ogni ciclo di gioco.

Soluzione:
Vado a definire dentro le due classi citate il set di tile create per ogni livello, dopodiché faccio un controllo predittivo sulla posizione del player o dei nemici e vado a controllare nella direzione in cui si stanno muovendo se ci sarà una tile “solida”. Se il controllo ha esito positivo blocco completamente lo spostamento del player o del nemico in quella direzione, se invece è negativo, permetto lo spostamento come previsto, Inoltre per quanto riguarda i nemici aggiungo una funzione “setDirectionX()” che mi permette di invertire la direzione di spostamento automatico del nemico se incontra un ostacolo.
Infine sfrutto la classe Hitbox per capire se player e nemici collidono tra di loro, così da poter diminuire le vite al giocatore in modo corrispondente alla logica di gioco.

Capitolo 3
Sviluppo

3.1 Testing automatizzato

I test automatizzati sono stati implementati tramite l'uso della libreria jUnit. I test sono stati applicati sulle implementazioni fatte per verificare la correttezza dei metodi e dei cambi di stato e dei loro campi.

- Test su stati del gioco: verifica se **GameStateImpl** applica correttamente il cambio di stato tramite i metodi.
- Test sul controller dei livelli: verifica se le varie implementazioni dei livelli contenuti all'interno di una lista venissero applicate correttamente al superamento di ogni livello.
- Test sui livelli di gioco: verifica se la logica interna del livello corrispondesse allo stato reale, come la posizione del player alla partenza.
- Test sui power up del gioco: verifica se i moltiplicatori venissero applicati correttamente ai campi del giocatore, e l'eventuale totale rimozione di essi.
- Test sui suoni del gioco: verifica l'esistenza di una clip di suono, e la relativa logica.

3.2 Note di sviluppo

Fabian Calangiu

Utilizzo della libreria JavaFX, per la creazione del menù principale, menù classifica, finestra di gioco principale, game over.
Permalink:
https://github.com/FabianCalangiu/PSS25-RogueKong/blob/2717266aa3b71b6a454671d5662665ceac7a3876/src/main/java/it/unibo/roguekong/view/impl/ScoreView.java#L17C1-L17C50

Utilizzo di Lambda Expression e Runnables per bottoni event-driven.
Permalink:
https://github.com/FabianCalangiu/PSS25-RogueKong/blob/0fe58bddd6c950b79d6d195eab1317f7b86c9267/src/main/java/it/unibo/roguekong/Main.java#L111-L114

Utilizzo di I/O tramite package util, nio(Paths, Files, StandardOpenOption) per lettura e scrittura su file.
Permalink:
https://github.com/FabianCalangiu/PSS25-RogueKong/blob/0fe58bddd6c950b79d6d195eab1317f7b86c9267/src/main/java/it/unibo/roguekong/controller/ScoreManager.java#L17-L109

Manuel Menghetti

Utilizzo della libreria JavaFX per la creazione della finestra di regole e comandi
Permalink:
https://github.com/FabianCalangiu/PSS25-RogueKong/blob/2717266aa3b71b6a454671d5662665ceac7a3876/src/main/java/it/unibo/roguekong/view/impl/RulesView.java#L11

Utilizzo di InputStreamReader, BufferedReader per lettura di file .txt. In seguito un esempio
Permalink:
https://github.com/FabianCalangiu/PSS25-RogueKong/blob/2717266aa3b71b6a454671d5662665ceac7a3876/src/main/java/it/unibo/roguekong/model/game/impl/TileManager.java#L107

Utilizzo di AudioInputStream per la gestione di file .wav. Usata insieme alla libreria javax.sound.sampled .
Permalink:
https://github.com/FabianCalangiu/PSS25-RogueKong/blob/0fe58bddd6c950b79d6d195eab1317f7b86c9267/src/main/java/it/unibo/roguekong/controller/SoundManager.java#L37

Utilizzo lambda expression per la creazione di runnable, usate per la RulesView e OnVictory
Permalink:
https://github.com/FabianCalangiu/PSS25-RogueKong/blame/0fe58bddd6c950b79d6d195eab1317f7b86c9267/src/main/java/it/unibo/roguekong/Main.java#L147

Enrico Vampa:
Utilizzo della libreria javafx.geometry.Rectangle2D per la gestione delle hitbox tra giocatore e nemici

Permalink:
https://github.com/FabianCalangiu/PSS25-RogueKong/blob/2717266aa3b71b6a454671d5662665ceac7a3876/src/main/java/it/unibo/roguekong/model/game/impl/HitboxImpl.java

Capitolo 18
Commenti finali

4.1 Autovalutazione e lavori futuri
Fabian Calangiu
La scelta di creare un gioco come primo vero progetto software orientato ad oggetti ha dimostrato quanto fosse formativo e difficile gestire la complessità di un programma in continua evoluzione. Alla fine ho ottenuto una serie di competenze nella creazione di interfacce con cui l'utente potesse interagire, il rendering in un ciclo di gioco completo, l'utilizzo di funzioni anonime e la gestione di I/O, sia nel caso di scrittura di file che per input dell'utente. Il mio compito principale all'inizio è stato lavorare nella sezione di visualizzazione dei dati ma in seguito sono riuscito a cimentarmi anche nei modelli del nostro programma e nella gestione di controllo. Questo mi ha reso soddisfatto del mio contributo.
Tra le difficoltà iniziali trovate vi era la questione di dipendere dai progressi altrui, ma anche rivedere alcune sezioni della libreria grafica, oltre l'implementazione di funzioni anonime all'interno di Runnables.

Manuel Menghetti
Questo progetto è stato parecchio impegnativo per me, perchè ho iniziato da solo un anno con la programmazione e ritrovarmi a dover fare un videogioco con tutte queste funzionalità, soprattutto in gruppo, non mi era mai successo. Ho apprezzato molto il fatto che fosse in OOP, poiché si è riuscito a dare un'ottima gerarchia e a dividere coerentemente le funzionalità in classi. Credo di aver appreso le competenze per poter lavorare su un progetto in gruppo e ci siamo integrati perfettamente poiché i compiti da svolgere sono stati divisi egregiamente. Ho anche potuto apprendere le funzionalità di un linguaggio storico ma fortemente utilizzato come Java. Le principali difficoltà le ho trovate all'inizio, poiché non sapevo bene come poter partire, ma mentre scrivevo codice e si vedevano i risultati, ottenevo sempre più fiducia. Tutto sommato credo di aver dato un buon contributo all'interno del gruppo, cercando sempre di utilizzare codice pulito e riusabile, soprattutto nella creazione dei livelli con le loro relative mappe.

Enrico Vampa
Nonostante io avessi già lavorato con Java in passato, non avevo ancora gestito un progetto di grandi dimensioni come un videogioco. Ho imparato a lavorare con le altre persone del gruppo per arrivare ad un'implementazione che potesse essere il più coerente possibile con il codice fatto da altri e quando non era possibile ho imparato a fare del reverse engineering per comprendere a pieno il codice che poteva risultare utile per le parti di mia competenza. In generale credo che tutto il gruppo abbia gestito il tempo e le conoscenze nel modo più efficiente possibile per arrivare all'obiettivo finale nei tempi concordati, tuttavia per quanto mi riguarda, penso che avrei dovuto spendere più tempo nella ricerca di classi già fatte da altri che hanno riscontrato i problemi che ho dovuto affrontare nel corso della programmazione.

4.2 Difficoltà incontrate e commenti per i docenti
Manuel Menghetti
Una delle principali difficoltà che ho appreso durante il corso che reputo valga la pena da sottolineare è la poca praticità che è stata fatta durante i laboratori per JavaFX. Infatti all'inizio abbiamo avuto qualche problema durante l'avvio. Per il resto il corso è stato ottimo.