Relazione
"Jetpack Joyride"

Lorenzo Annibalini
Lorenzo Bacchini
Mattia Burreli
Emanuele Sanchi

10 Giugno 2023

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il progetto si pone l'obiettivo di creare un videogioco single player d'azione a scorrimento orizzontale, ispirato a "Jetpack Joyride". Con gioco d'azione si intende un genere di videogioco il cui scopo è quello di sopravvivere il più possibile, evitando gli ostacoli della mappa.

Requisiti Funzionali

- Il gioco consiste nel controllare il personaggio principale e la sua interazione nella mappa con le varie entità, con lo scopo di sopravvivere il più a lungo possibile.
  All'interno della mappa il giocatore potrà:
  - Muoversi su e giù volando con un jetpack oppure correndo nel caso tocchi il terreno
  - Colpire degli scienziati, che corrono sul terreno, eliminandoli dalla partita corrente
  - Schivare eventuali ostacoli quali missili, raggi laser ed elettrodi
  - Collezionare monete da utilizzare nello shop
  - Raccogliere power-up
- Gli ostacoli sono di tre tipi:
  - **Elettrodi**: Sono gli ostacoli più comuni. Sono fissi sulla mappa e sono composti da due terminali collegati da una scarica elettrica. Possono essere disposti in verticale o orizzontale.

  - **Laser**: Compaiono casualmente nel tempo. Si dispongono orizzontalmente sulla schermata di gioco e la occupano interamente su un certo livello. Rimangono a schermo per un certo intervallo di tempo. Prima compaiono i terminali, per permettere all'utente di spostarsi, poi il laser congiunge i due terminali.
  - **Missili**: Prima dei missili compare un fumetto di avvertimento riguardo la loro posizione. Questo fumetto rimane fisso all'estremo della mappa fino a qualche istante prima della comparsa del missile che a quel punto attraversa lo schermo da destra a sinistra all'altezza dell'ultima posizione del fumetto.

- La mappa di gioco scorre dietro al personaggio fino alla sua morte, dove poi comparirà una schermata di fine partita con le statistiche della run corrente e la possibilità di uscire dal gioco o ritornare al menu principale.

- Nello shop si possono acquistare due tipi di oggetti:

  - **Skin**: aspetto del personaggio che dopo l'acquisto può essere attivato e disattivato a piacimento.
  - **Gadget**: elementi che modificano le caratteristiche del personaggio in modo permanente. Dopo l'acquisto, si può scegliere se equipaggiare il gadegt o meno.

Requisiti Non Funzionali

- Il programma dovrà reagire in modo efficace ed efficiente al variare delle situazioni di gioco, rimanendo sempre fluido e godibile.

- Per poter gestire lo shop, bisognerà prevedere il mantenimento del contatore delle monete tra le sessioni. Inoltre, dovranno essere salvate statistiche generali, come i metri massimi raggiunti o il numero di scienziati uccisi.

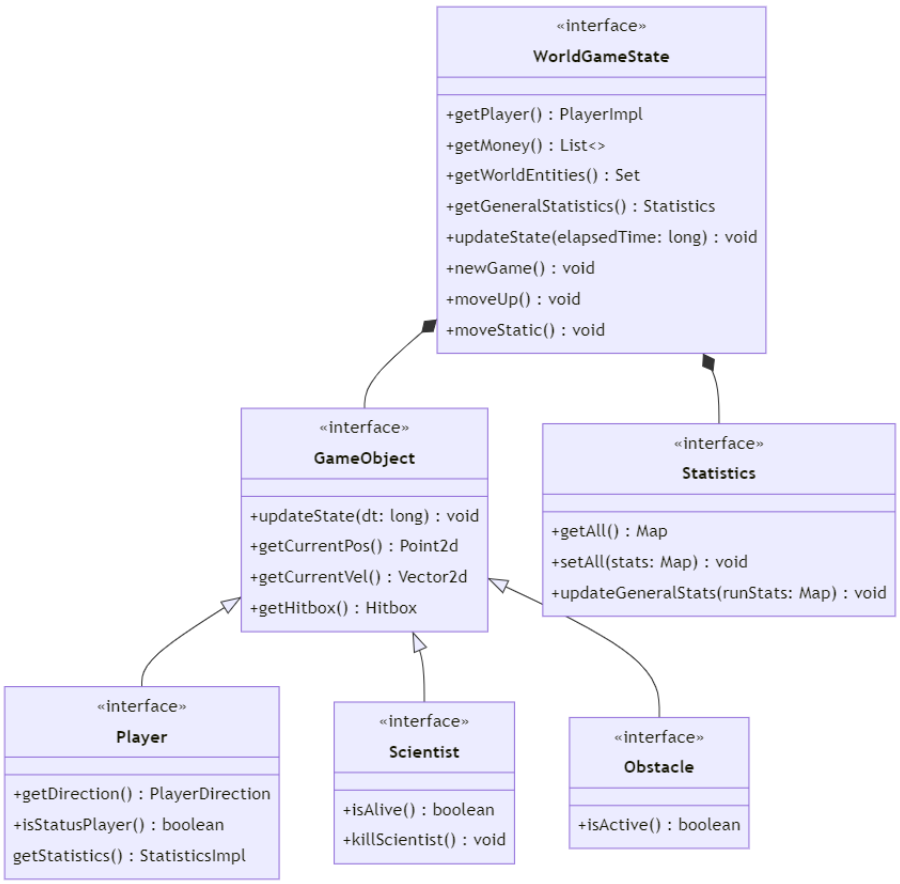
- Il gioco dovrà mantenere i salvataggi relativi all'acquisto di gadget e skin e alla loro attivazione in modo da non perdere gli acquisti effettuati e le modifiche apportate nelle partite precedenti.

- Il gioco dovrà gestire la progressione del mondo e le interazioni tra le varie entità presenti in modo efficace e ben organizzato in modo tale da permette un corretto utilizzo e avanzamento del mondo di gioco.

1.2 Analisi e modello del dominio

Jetpack Joyride permette all'utente di effettuare una partita single player e di interagire con le varie entità della mappa, la quale scorre dietro essi. Durante la partita l'utente può controllare solo il personaggio principale mentre le altre entità avranno un proprio movimento indipendente. Gli ostacoli compaiono durante il gioco man mano che si avanza nella mappa e l'utente può solo schivarli per evitare di perdere. Gli ostacoli possono essere di 3 diversi tipi: laser, elettrodi o missili. Inoltre vi è la possibilità di raccogliere power-up oppure delle monete. Le difficoltà del progetto saranno:

- la generazione delle monete secondo pattern specifici caricati da file.
- la gestione delle statistiche e le informazioni riguardanti skin e gadget che dovranno essere salvate su file opportuni, in modo da essere ricaricati ad ogni avvio del gioco.
- la gestione delle posizioni e dimensioni delle entitità di gioco , l'interazione tra entità e gli eventi che si vengono a generare a seguito di una collisione tra queste ultime.
- la gestione e l'aggiornamento dello stato di esecuzione del programma e gli eventi correlati a quest'ultimo.
- la generazione delle entità a seconda di quante sono presenti attualmente nel mondo di gioco, dello stato di gioco (generazione dei laser, generazione delle monete, generazione di ostacoli e power-up) e del tempo passato dall'ultima generazione.

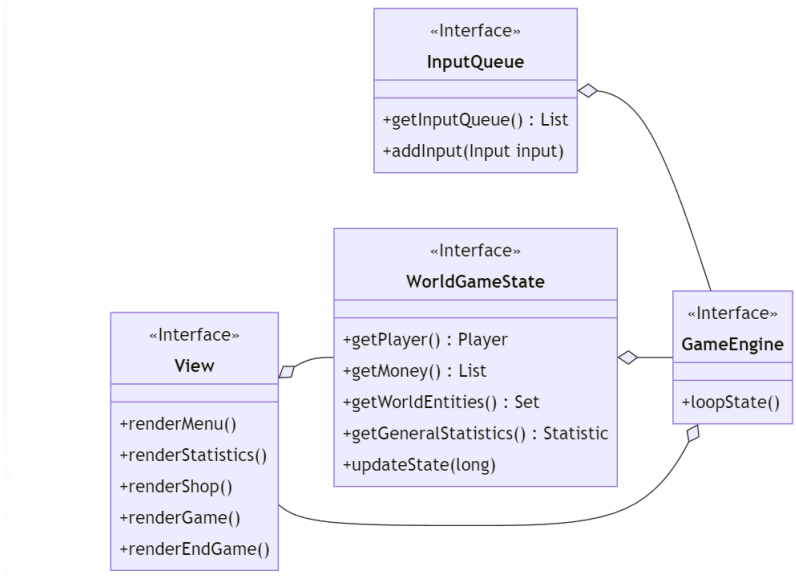

«interface»
WorldGameState
+getPlayer() : PlayerImpl
+getMoney() : List<>
+getWorldEntities() : Set
+getGeneralStatistics() : Statistics
+updateState(elapsedTime: long) : void
+newGame() : void
+moveUp() : void
+moveStatic() : void
«interface»
GameObject
+updateState(dt: long) : void
+getCurrentPos() : Point2d
+getCurrentVel() : Vector2d
+getHitbox() : Hitbox
«interface»
Statistics
+getAll() : Map
+setAll(stats: Map) : void
+updateGeneralStats(runStats: Map) : void
«interface»
Player
+getDirection() : PlayerDirection
+isStatusPlayer() : boolean
getStatistics() : StatisticsImpl
«interface»
Scientist
+isAlive() : boolean
+killScientist() : void
«interface»
Obstacle
+isActive() : boolean

Capitolo 2

Design

2.1 Architettura

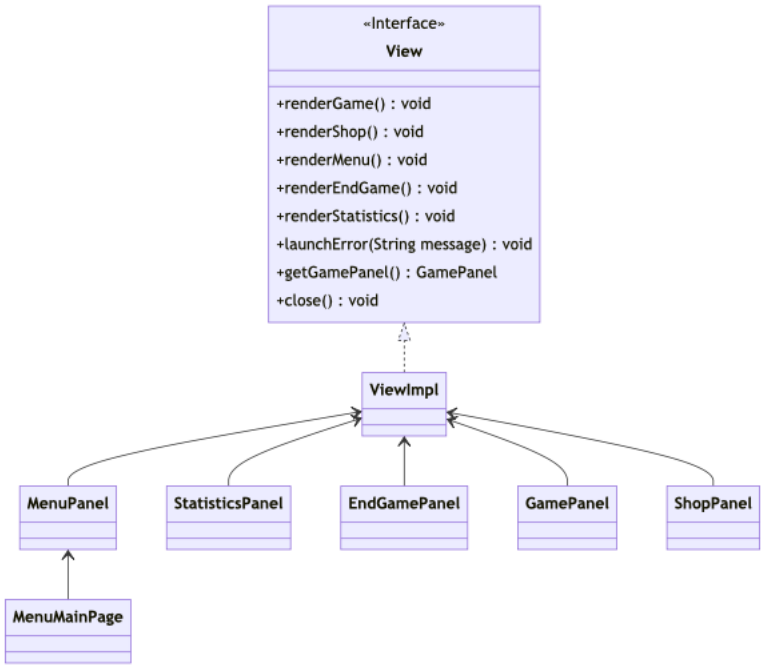
L'architettura di Jetpack-Joyride adotta il pattern architetturale MVC (Model-View-Controller). WorldGameState è l'interfaccia che si occupa della parte di model del nostro progetto. Gli elementi del model vengono poi resi disponibili a View e Controller tramite dei getter. Il WorldGameState si occupa anche di mantenere lo stato attuale del gioco e di aggiornarlo. Le informazioni riguardanti il model vengono prese dall'interfaccia View che si occupa poi di rappresentarle a video. Il tutto è controllato dal controller che nel nostro caso è rappresentato dal GameEngine che fa partite il loop di gioco. View e controlloer sono inoltre collegati tra loro tramite una InputQueue, infatti la view prende degli input che poi vengono processati dal GameEngine e permettono così di aggiornare il Model. Utilizzando questo pattern architetturale, le tre entità (Model,View e Controller) sono indipendenti tra loro e quindi cambiando la logica del GameEngine o il modo in cui vengono gestiti gli oggetti di Model, non ci saranno cambiamenti nelle altre parti.



2.2 Design dettagliato

Annibalini Lorenzo

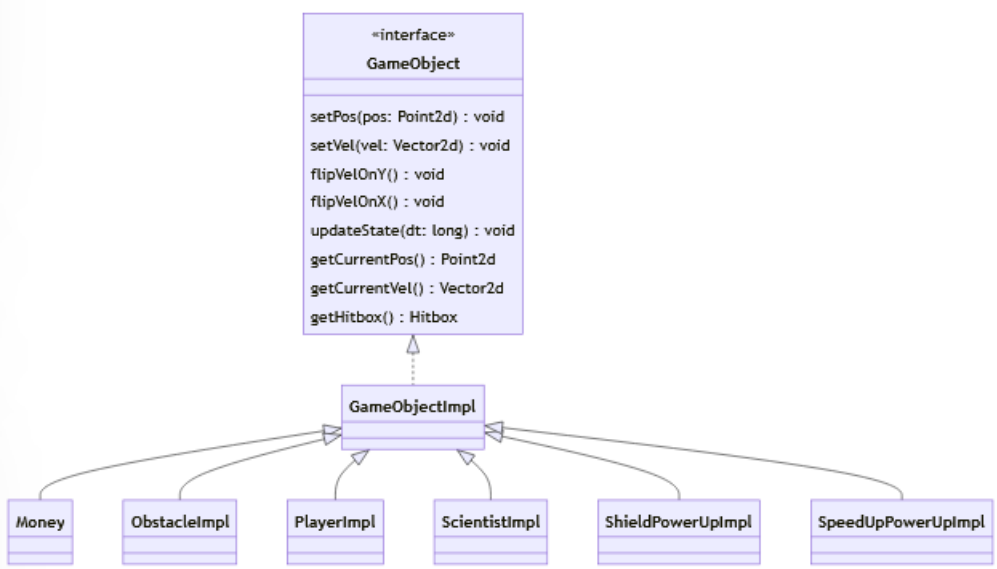
MENU



Problema
Il giocatore tramite il menu può scegliere di visualizzare diverse aree del gioco, durante l'analisi sono sorti diversi problemi su come rendere il più possibile scalabile il codice per facilitare l'aggiunta o rimozione delle sue componenti.
Soluzione
Dopo una dettagliata analisi e dopo diverse prove è stato deciso di creare la classe MenuPanel il quale compito è quello di poter caricare al suo interno dei menu generici con funzionalità diverse tra loro; ad esempio all'avvio del gioco viene caricato il menu di default che, inviando degli input all'inputHandler permette di avviare il gioco, aprire lo shop e vedere le statistiche generali. L'inputHandler una volta ricevuti gli input passati dal menu andrà a chiamare i setter della view per rendere visibili le varie finestre messe a disposizione, cosi facendo se più menu richiamano le stesse finestre dovremo soltanto agire sulla view.

Bacchini Lorenzo

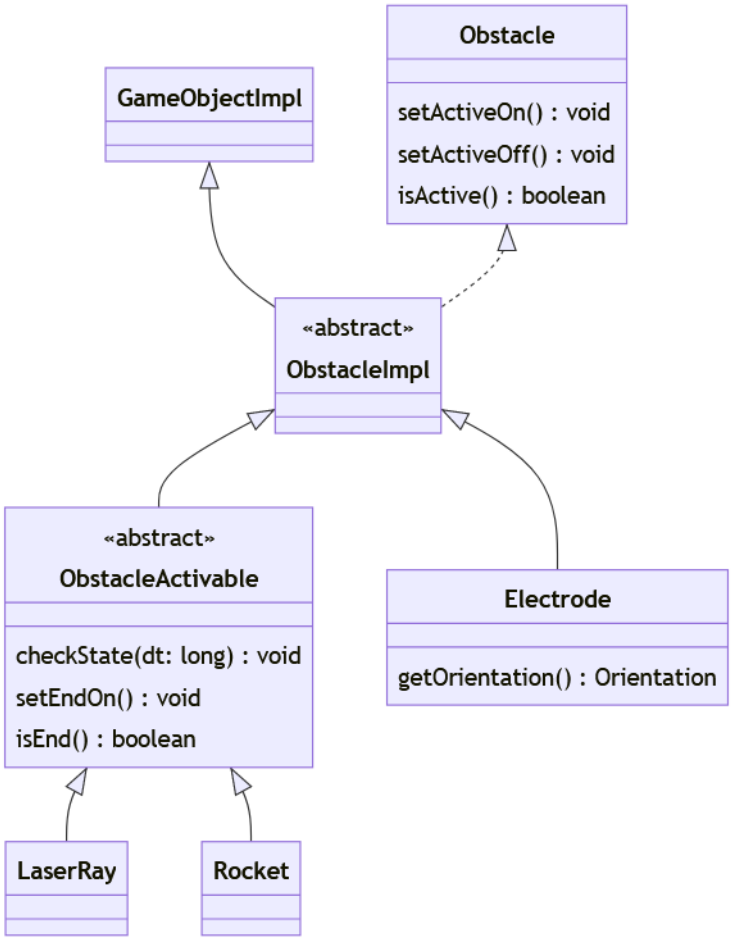
ENTITÀ



Problema
Uno dei problemi principali riguarda tutte quelle entità che fanno parte del modello di gioco e che quindi durante la partita dovranno aggiornare il proprio stato a seguito di eventi scatenati dal giocatore.
Soluzione
Dopo un analisi iniziale ci siamo resi conto che tali entità (powerUp, ostacoli,

monete, scienziati e il player stesso) avevano diverse parti in comune, soprattutto quelle riguardanti la posizione e la velocità che esse avrebbero avuto durante la partita, la soluzione proposta a portato quindi alla creazione di una superclasse (GameObject) la quale si occupa di incapsulare il comportamento generale di un qualsiasi elemento di gioco che durante la partita si muove verso il player, questo garantisce un riuso di codice considerevole limitanto le ripetizioni e fa si che in futuro si possano aggiungere nuove entità semplicemente estendendo la superclasse.

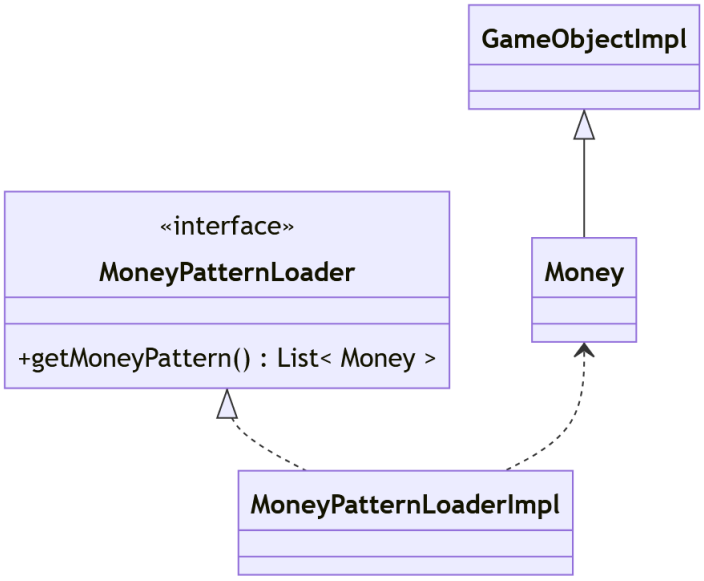
OSTACOLI



Problema
Tutte le entità riguardanti gli ostacoli devono essere modellate con metodi specifici per gestirne il comportamento durante la partita
Soluzione
Si è optato per l'utilizzo di due classi astratte, una più generica che racchiude

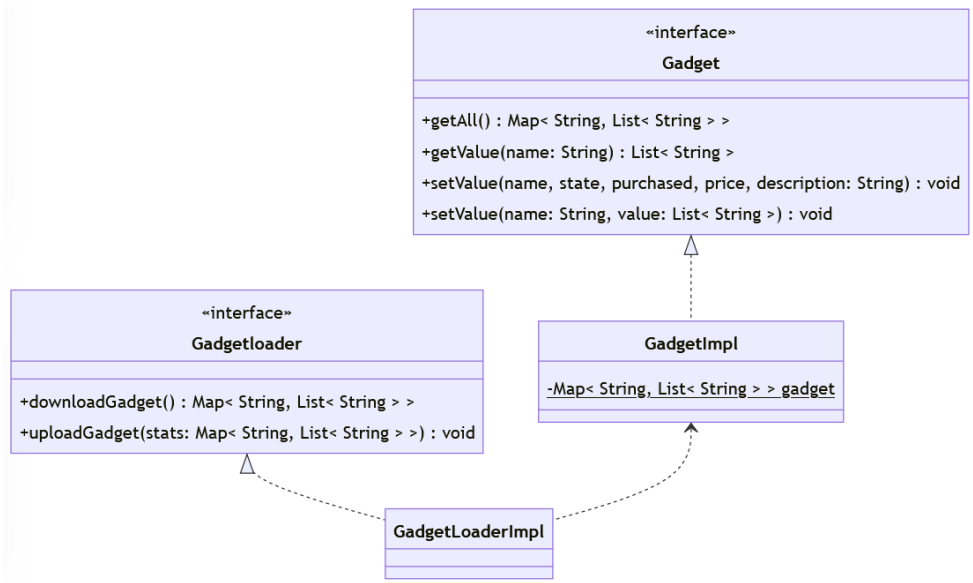
il concetto di ostacolo e implementa i metodi per gestirne l'attivazione e la disattivazione, un'altra più specifica, che eredita dalla prima e che modella tutti quegli ostacoli che dopo la loro creazione hanno un tempo di attesa prima che si attivino, questa soluzione permette di creare futuri ostacoli riusando gran parte del codice, dato che buona parte dell'implementazione risiede nelle classi astratte sopracitate.

MONETE



Problema
Dall'analisi del modello e del dominio risulta importante gestire la generazione delle monete secondo pattern specifici
Soluzione
Per risolvere il problema riguardante il caricamento di pattern specifici di monete si è ricorsi a dei file.txt, all'interno dei quali è contenuta la posizione delle immagini in formato (x,y), in questo modo attraverso una classe specifica sarà sufficiente richiamare il metodo che legge uno dei file di pattern e salvare le posizioni delle monete lette in una lista di monete

CARICAMENTO SALVATAGGI



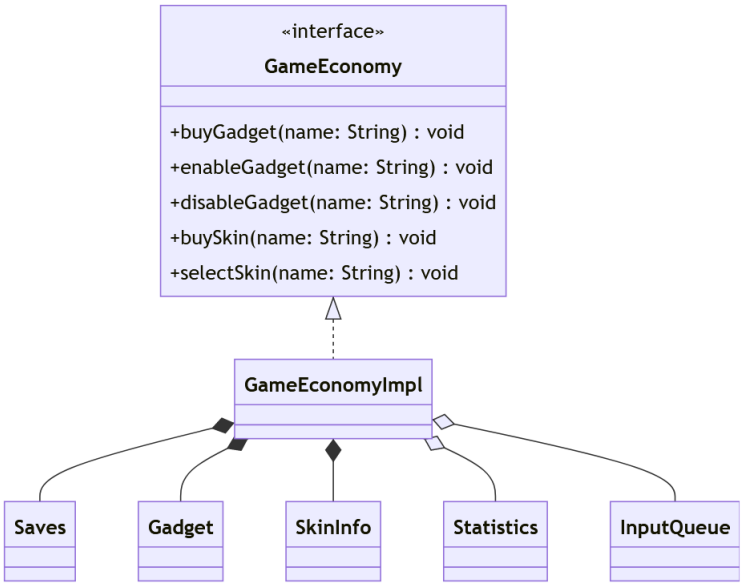
l'immagine mostra il solo uml relativo al salvataggio dei gadget per motivi di comprensione grafica, lo schema risulta identico anche per quanto riguarda le informazioni delle skin

Problema
Si rende necessario il mantenimento tra una partita e l'altra dello stato di skin e gadget, quindi delle informazioni riguardanti il loro acquisto e la loro attivazione

Soluzione
Dato che le informazioni di skin e gadget dovranno essere persistenti alla chiusura e riapertura del gioco (oltre che tra una partita e l'altra), sarà necessario salvare su supporti opportuni tali informazioni.
Per risolvere questo problema si è optato inizialmente per dei file csv, soluzione poi abbandonata a causa dell'impossibilità durante l'esecuzione del jar del gioco di accedere a file e risorse in modalità di scrittura/lettura mantenendo le informazioni necessarie.
Si è infine ricorsi all'utilizzo di una classe specifica che lavora con le java Preferences che si occupa di gestire download e upload per mantenere la persistenza dei dati e all'uso di una classe (una per i gadget e una per le skin) che contiene una mappa statica che è possibile inizializzare utilizzando i valori salvati; tale mappa sarà utilizzata da tutte le altre classi come punto di accesso per i dati e fornirà in un secondo momento le proprie informazioni per fare l'upload delle modifiche effettuate in modo da mantenerle tra una partita e l'altra e alla chiusura del gioco.

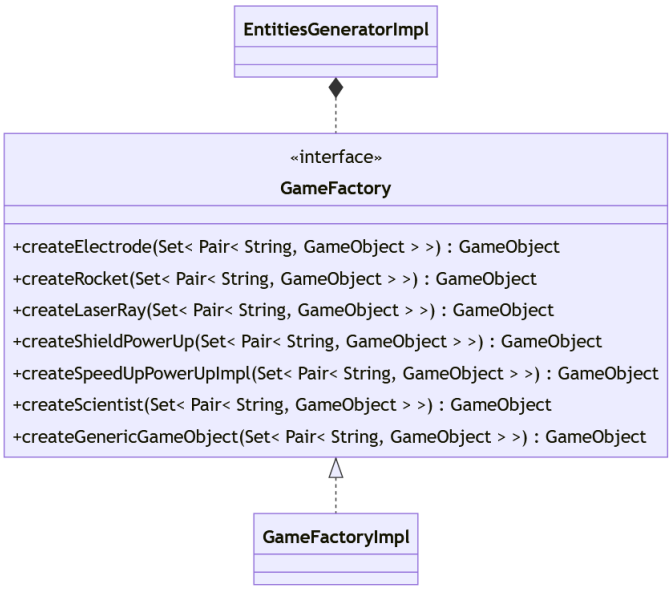
In questa soluzione si è fatto uso del pattern Caching, i dati letti attraverso le Preferences vengono aggiornati solo a seguito di modifiche sulla mappa statica di skin o dei gadget, tutte le volte che leggo solamente le loro informazioni non sto riscaricando tutto, semplicemente sto accedendo alla mappa contenente tali info.

ECONOMIA DI GIOCO



Problema
Data la presenza di uno shop bisogna gestire l'economia di gioco in modo coerente ed efficace.
Soluzione
Si è deciso di delegare la logica dell'economia di gioco ad una classe separata che al suo interno controlla la possibilità di compiere determinate azioni (acquisti/attivazioni/disattivazioni di gadget e skin) e di conseguenza modifica il model del gioco, che rifletterà le sue modifiche alla view. Per compiere queste azioni tale classe dovrà avere accesso a skin, gadget e salvataggi, oltre che alle statistiche generali che riceverà in fase di creazione assieme ad una coda degli input, che servirà per notificare eventuali problemi inerenti all'economia di gioco verso l'esetrno

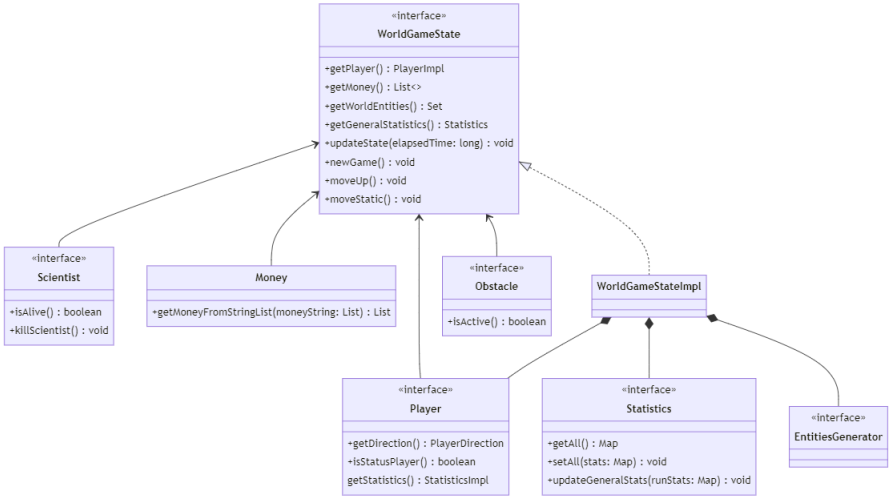
CREAZIONE ENTITÀ



Problema Durante la partita vengono generate una grande quantità di entità di tipo GameObject.
Soluzione Onde evitare problemi nella creazione di entita che estendono GameObject si è creata la classe GameFactory che si occupa di creare oggetti di tipo GameObject in base al metodo richiamato.
In questo modo si evitano creazioni di oggetti in diversi punti del codice che potrebbero risultare tra loro inconsistenti e si delega il tutto ad una classe specifica che istanzia ogni oggetto seguendo una logica ben precisa. In questa soluzione si è fatto uso del pattern Factory Method, è stata infatti definita l'interfaccia GameFactory (factory) e la sua implementazione GameFactoryImpl che al suo interno implementa i metodi factory a cui è delegata la creazione delle entità

Burreli Mattia

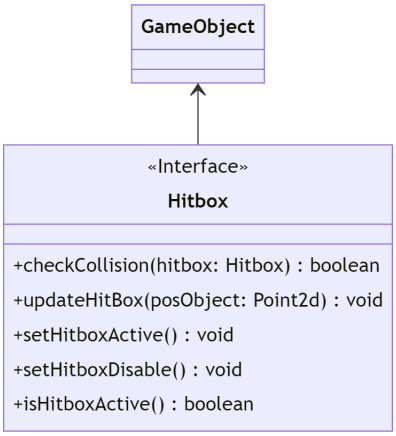
Gestione ed aggiornamento del mondo di gioco



Problema
Uno dei principali problemi emersi durante la fase di analisi, riguarda il fatto di riuscire a gestire le varie dipendenze nel progetto. Infatti, abbiamo cercato un modo per far interagire ed unire tutte le varie entità del gioco (ostacoli, powerup) e il player, in modo tale da premettere l'unicità delle entità e l'aggiornamento dello stato di gioco in modo corretto.
Soluzione
La soluzione adottata è quella di creare una classe WorldGameState basata sul pattern Mediator. Infatti all'interno di questa classe sono contenute tutte le entità presenti attualmente nel gioco, il player e le monete. In questo modo riusciamo a coordinare la gestione e l'interazione delle varie entità tra di loro in modo efficace. Il mondo viene poi aggiornato attraverso il suo metodo updateState, il quale segue il pattern "Update Method". Questo tipo di pattern permette di aggiornare ogni oggetto in modo separato e simultaneamente a tutti gli altri presenti nel mondo di gioco. Questa funzione quindi aggiorna le posizioni delle entità, verifica che queste ultime non siano uscite dalla mappa, genera nuove entità attraverso la classe EntitiesGenerator e verifica se ci siano collisioni tra il player e le entità del gioco. Inoltre all'interno della classe WorldGameState vengono salvate le statistiche generali del giocatore,

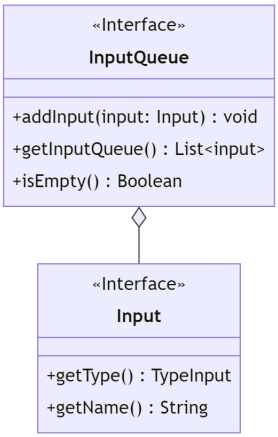
permettendo quindi il loro aggiornamento in modo corretto ed efficace. In questa maniera riusciamo a mantenere tutte le entità univoche e permettere l'interazione tra di esse all'interno del mondo di gioco.

Hitbox e gestione delle collisioni



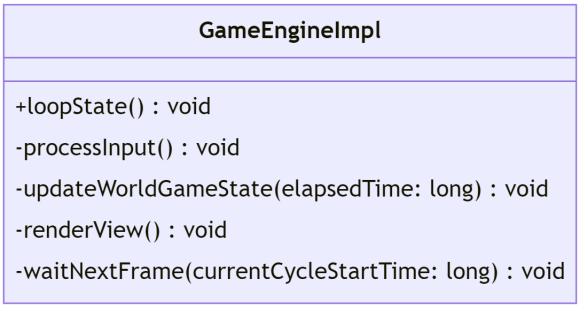
Problema
Uno dei problemi principali che abbiamo riscontrato durante lo sviluppo del gioco è la gestione delle collisioni tra le entità e il player. Abbiamo bisogno di un sistema che ci permetta di verificare in modo corretto se due oggetti sono in collisione tra loro oppure no.
Soluzione
La soluzione scelta per questo problema è quella di creare una classe Hitbox. La hitbox verrà quindi creata in base alla posizione dell'entità nel mondo di gioco e alla sua dimensione. Inoltre attraverso il metodo checkCollision possiamo verificare se due hitbox collidono oppure no; in questa maniera riusciamo ad associare ad ogni oggetto una propria hitbox modellata su se stessa e dunque facilitare la gestione delle collisioni tra oggetti.

Gestione degli eventi di gioco



Problema
Bisogna immagazzinare le richieste di eventi da parte di altre classi in modo efficace e univoco, quindi tutte le richieste devono essere centralizzate ed eseguite successivamente in un unico punto.
Soluzione
Per risolvere questo problema è stato scelto l'utilizzo del pattern Command, nel quale le richieste di eventi vengono incapsulate in un oggetto e inserite all'interno di una coda. Per fare ciò sono state utilizzate le classi Input e InputQueue. Le richieste vengono quindi incapsulate sotto forma di oggetto Input nel quale viene specificata l'azione da compiere scelta dalla enum TypeInput della classe Input. Successivamente l'Input viene inserito all'interno della InputQueue dove finirà in una coda di Input. In questo modo riusciamo a gestire tutti gli input provenienti dalle varie classi del progetto in modo unico e ordinato.

Progressione del gioco

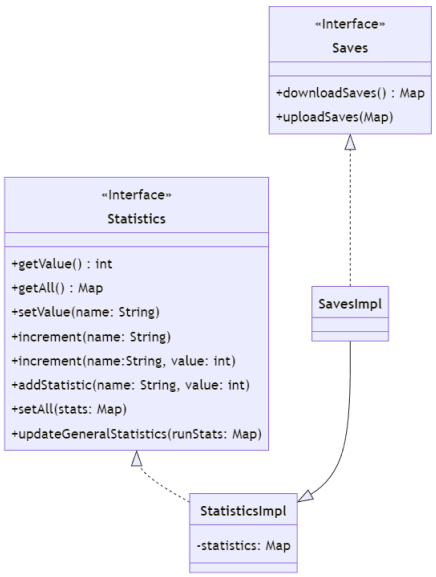


Problema
Il programma necessita di aggiornare il proprio stato di esecuzione, la logica e la grafica in modo indipendente e automatico a seconda del calcolatore su cui viene eseguito, garantendo stabilità ed efficienza su ogni dispositivo in cui verrà eseguito.
Soluzione
Per risolvere questo problema è stato utilizzato il pattern "Game Loop", il quale permette attraverso un loop di eseguire una serie di aggiornamenti che rendono lo scorrimento del programma regolare e controllato. Attraverso il metodo loopState vengono eseguiti i seguenti metodi:

-**processInput**: Si occupa di processare una coda di input presa dall'oggetto InputQueue "inputHandler" e, a seconda del tipo di input, esegue una determinata azione. Gli input vengono eseguiti per ordine di arrivo, quindi dal meno recente al più recente, fino a quando la coda non è vuota.
-**updateWorldGameState**: durante la partita, il mondo viene aggiornato a seconda di un tempo dato dalla variabile "elapsedTime" in millisecondi, la quale rappresenta il tempo passato dall'ultimo ciclo di istruzioni del loopState.
-**renderView**: Attraverso questo metodo viene aggiornata la grafica del programma, che varia a seconda dello stato del GameEngine, osservando le informazioni ottenute dal model.
-**waitNextFrame**: Dopo aver eseguito i precedenti metodi, nel caso non sia passato abbastanza tempo dall'ultimo ciclo di istruzioni, viene fatto aspettare il tempo necessario. In questo modo viene limitato l'utilizzo delle risorse del calcolatore migliorando le prestazioni.

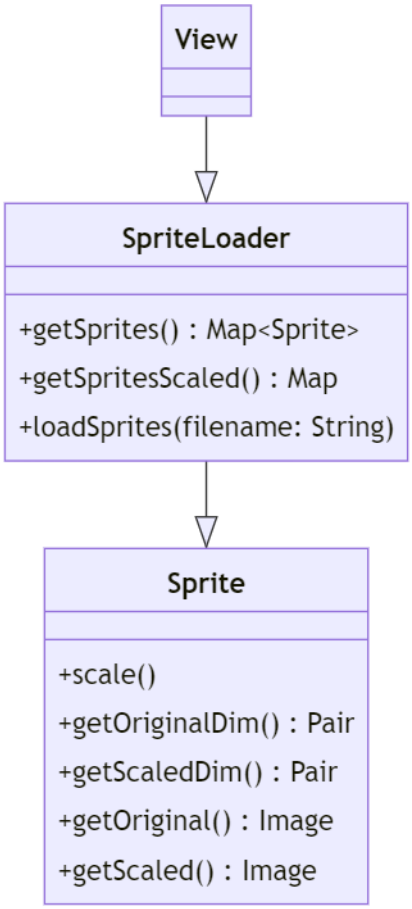
Sanchi Emanuele

IL SALVATAGGIO DELLE STATISTICHE



Problema
Nel gioco uno dei requisiti è quello di mantenere le statistiche sia durante una partita che tra una partita e l'altra fornendo quindi delle statistiche generali.
Soluzione
Una prima soluzione pensata è stata quella di modellare ogni statistica come un oggetto a se stante con una superclasse che ne definisse gli aspetti comuni. Alla fine si è optato per una classe che definisce al proprio interno una mappa di statistiche alla quale è possibile accedere in diversi modi in base all'utilizzo che se ne vuole fare. In questo modo sia chi gestisce la partita sia chi gestisce le statistiche generali, può creare un oggetto inserendo all'interno le statistiche che meglio crede (per esempio nelle statistiche della partita corrente non sono interessato ad avere la statistica riguardante la partita con percorso migliore). Inoltre bisogna salvare le statistiche sul sistema per renderle persistenti tra una partita e l'altra. Per fare ciò è stato creata un'entità specifica che permette di fare download e upload di statistiche. Da notare che ciò non obbliga il salvataggio delle sole statistiche generali, bensì di qualsiasi venga passata e quindi, ipoteticamente, anche di statistiche della partita corrente.

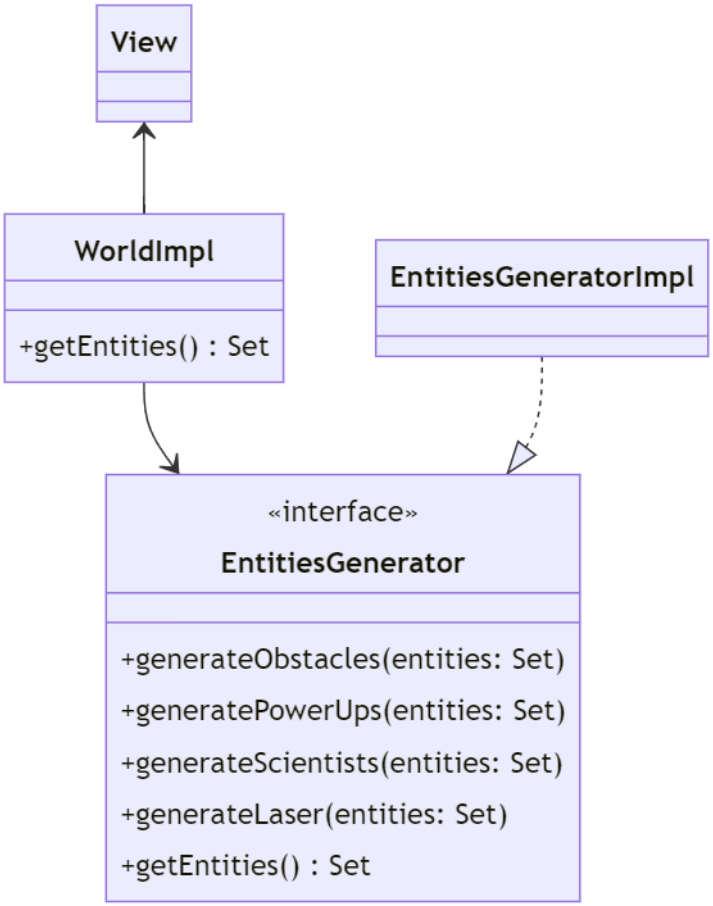
IL CARICAMENTO DELLE RISORSE PER LA VIEW



Problema
Nella view ci sono diversi pannelli che hanno bisogno di risorse caricate da memoria, in modo particolare gli sprite del gioco.
Soluzione
Inizialmente si era pensato di far caricare le risorse necessarie al bisogno, in realtà poi ci si è resi conto che servisse un'entità prestabilita che caricasse le risorse e le tenesse in memoria in modo che fossero accedibili da tutti. Questa tecnica fa uso del pattern Caching, infatti gli sprite vengono caricati dalla memoria e inseriti in una mappa statica in modo tale che ogni entità che necessita di utilizzare gli sprite possa poi farne da riferimento, senza dover caricare ogni volta degli elementi da memoria. Il metodo di load prende in input il nome di un filejson, perciò se si volesse dare la possibilità al giocatore di scegliere tra diversi pattern di grafiche, si potrebbe fare semplicemente cambiando il nome del file da caricare. È anche una sorta di pattern Proxy,

infatti non si può fare riferimento direttamente allo sprite, ma solo tramite al loader e i relativi getter.

LA GENERAZIONE DELLE ENTITÀ IN GIOCO



Problema
Ogni entità presente nel gioco è stata modellata tramite oggetti specifici, ognuna con le proprie caratteristiche. Essendo però un gioco con schermata in scorrimento, le entità compaiono in modo randomico e bisognava prevedere un modo per farlo.
Soluzione
Una prima idea è stata quella di stabilire dei pattern fissi di generazione, ma poi il gioco sarebbe stato monotono. Si era anche pensato di delegare la generazione al world, ma poi sarebbe stato un carico eccessivo da gestire e quindi, anche per favorire il Single-responsibility principle, si è deciso di predisporre una oggetto dedicato alla sola generazione, che esponesse dei me-

todi ben specifici in base alla classe di entità che si voleva generare (Powerup, Ostacoli, Npc, ecc). I metodi forniti dal generatore non creano direttamente le varie entità, ma richiamano a loro volta una classe di factory che ne generalizza la creazione e che utilizza il pattern Factory Method. Ogni metodo prende come parametro di input un set di entità al quale poi appende quella appena generata. Questo set non viene poi restituito, almeno finchè non viene chiamato il getter. Questo perchè le entità che vengono generate potrebbero non essere subito da visualizzare e quindi da restituire.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

- InputPanelTest: viene testato l'input da tastiera e l'invio di quest'ultimo all'InputQueue. Viene creato un JFrame e viene inserito un InputPanel al suo interno per permettere la cattura degli eventi da tastiera; in questo caso l'evento da tastiera viene creato nel test.

- InputTest: viene testata la corretta creazione di vari Input e il loro inserimento all'interno di una InputQueue. Successivamente si verifica se è possibile ottenere la coda dell'InputQueue, la quale poi avrà dimensione 0, e se la dimensione della coda ottenuta risulta uguale al numero di Input inseriti precedentemente.

- HitboxImplTest: viene testata la corretta collisione tra il player e un oggetto da ogni angolatura frontale. Il player rimane sempre in una posizione fissa mentre l'oggetto si muove verso il player. Partendo dal basso, ogni volta che le due entità collidono, l'oggetto viene posizionato più in alto rispetto al punto di inizio precedente e dunque viene nuovamente testata la collisione. Questo test va avanti fino a quando non vengono testate tutte le possibili angolazioni frontali.

- WorldGameStateTest: viene testata la creazione del mondo di gioco e l'avanzamento dello stato di gioco. Successivamente viene controllato se le varie entità, le statistiche e il player sono presenti nel mondo di gioco in modo corretto.

- SavesTest: viene testata la corretta lettura e scrittura delle statistiche generali e la loro persistenza tra una partita e l'altra simulando letture e scritture. Prima di lanciare il test, le statistiche attuali vengono salvate

in una mappa temporanea e poi vengono risalvate a fine test, in modo da non intaccare le statistiche del giocatore con i test.

- GenerateEntitiesTest: viene testata la corretta generazione delle entità controllando che venga generata effettivamente l'entità richiesta e che nel set delle entità ci siano tante entità quante sono quelle richieste.

- SpriteLoaderTest: viene testato l'effettivo funzionamento del loader richiamando il metodo per caricare gli sprite da un file json e successivamente controllando che il numero di sprite caricati sia giusto

- DownloadUploadSkinGadgetTest: viene testato il corretto funzionamento delle classi GadgetLoaderImpl e SkinInfoLoaderImpl, per quanto riguarda i gadget, vengono caricati alcuni gadget in una mappa e ne viene fatto l'upload, poi per verificarne la buona riuscita vengono riscaricati e confrontati con la mappa usata per l'upload. il testing avviene analogamente anche per le skin

- MoneyPatternTest: viene testata la corretta generazione delle monete, le quali saranno sempre caricate seguendo pattern specifici da file, in questo test infatti viene controllato che i valori letti siano quelli previsti

- PlayerImplTest: viene testato il corretto funzionamento dei gadget all'interno del Player modificando durante il test lo stato dei gadget e verificando che le caratteristiche del player cambino di conseguenza.

3.2 Metodologia di lavoro

Dopo un iniziale fase di analisi in cui è stato definito il dominio e le principali interazioni che le classi avrebbero avuto tra di loro, il gruppo è riuscito bene a lavorare separatamente ai propri compiti all'inizio dello sviluppo, nella sua fase centrale (in cui il model era già quasi completo) ci sono stati alcuni problemi dovuti alla fusione di moduli di un partecipante con un altro ma dopo alcuni aggiustamenti siamo riusciti a proseguire senza intoppi. In alcuni casi ci siamo trovati a modificare alcune scelte fatte in sede di analisi per poter accorpare al meglio il lavoro di tutti.
Per quanto riguarda il DVCS abbiamo adoperato uno schema di lavoro abbastanza semplice che consisteva nel creare un branch per ogni funzionalità da sviluppare e nel effettuare un merge su master solo quando la proprie modifiche fossero concluse e soprattutto funzionanti, in modo da tenere il master sempre al sicuro da eventuali problematiche
Seguono le classi realizzate da ogni componente:

Annibalini Lorenzo:

- Scientist
- ScientistImpl
- ShieldPowerUpImpl
- SpeedUpPowerUpImpl
- Direction
- MenuPanel
- MenuMainPage
- View
- ViewImpl
- MenuSettingsPage(Cancellata)
- GameSettings(Cancellata)
- GameSettingsImpl(Cancellata)

Bacchini Lorenzo:

- Gadget
- GameObject
- Obstacle
- Orientation
- SkinInfo
- Electrode
- GadgetImpl
- GameObjectImpl
- LaserRay
- Money

- ObstacleActivable
- ObstacleImpl
- Rocket
- SkinInfoImpl
- ShopPanel
- GadgetLoaderImpl
- GameEconomyImpl
- GameFactoryImpl
- MoneyPatternLoaderImpl
- SkinInfoLoaderImpl
- GadgetInfoPositions
- GadgetLoader
- GameEconomy
- GameFactory
- MoneyPatternLoader
- SkinInfoLoader
- SkinInfoPositions
- Pair
- Point2d
- Vector2d

Burreli Mattia:

- GameEngine
- GameEngineImpl
- InputPanel
- Input
- InputQueue
- InputImpl
- InputQueueImpl
- Hitbox
- Player
- WorldGameState
- HitboxImpl
- PlayerImpl
- WorldGameStateImpl

Sanchi Emanuele:

- Saves
- SavesImpl
- Slider
- SliderImpl
- GamePanel
- StatisticsPanel
- Sprite
- SpriteLoader
- EntitiesGenerator

- EntitiesGeneratorImpl
- Statistics
- StatisticsImpl

3.3 Note di sviluppo

Annibalini Lorenzo

Utilizzo di Stream

Permalink: https://github.com/LorenzoBacchini/OOP22-JetpackJoyride/blob/a39989881ff7d8695603aa949e398d8b0cf8eaf9/src/main/java/it/unibo/jetpackjoyride/graphics/impl/MenuMainPage.java#L87-L103

Bacchini Lorenzo

Utilizzo di Stream

Permalink: https://github.com/LorenzoBacchini/OOP22-JetpackJoyride/blob/a39989881ff7d8695603aa949e398d8b0cf8eaf9/src/main/java/it/unibo/jetpackjoyride/core/impl/GameEconomyImpl.java#L130-L135

Utilizzo di java.util.prefs.Preferences

Permalink: https://github.com/LorenzoBacchini/OOP22-JetpackJoyride/blob/a39989881ff7d8695603aa949e398d8b0cf8eaf9/src/main/java/it/unibo/jetpackjoyride/core/impl/GadgetLoaderImpl.java#L86-L114

Burreli Mattia

Utilizzo di Stream e Lambda expressions

Permalink: https://github.com/LorenzoBacchini/OOP22-JetpackJoyride/blob/298271256c5fff89984d8a7e17b1afb3aa9e1fd7/src/main/java/it/unibo/jetpackjoyride/model/impl/WorldGameStateImpl.java#L167-L173

Utilizzo di Optional

Permalink: https://github.com/LorenzoBacchini/OOP22-JetpackJoyride/blob/298271256c5fff89984d8a7e17b1afb3aa9e1fd7/src/main/java/it/unibo/jetpackjoyride/input/impl/InputImpl.java#L24-L27

Sanchi Emanuele

Utilizzo di Stream e Lambda expressions

Permalink: https://github.com/LorenzoBacchini/OOP22-JetpackJoyride/blob/3c66da3efc7650001daba4eda173051546eea97b/src/main/java/it/unibo/jetpackjoyride/graphics/impl/GamePanel.java#LL298C9-L300C27

Utilizzo della libreria JSON-Simple

Permalink: https://github.com/LorenzoBacchini/OOP22-JetpackJoyride/blob/3c66da3efc7650001daba4eda173051546eea97b/src/main/java/it/unibo/jetpackjoyride/graphics/impl/SpriteLoader.java#LL56C5-L90C14

Utilizzo di java.util.prefs.Preferences

Permalink: https://github.com/LorenzoBacchini/OOP22-JetpackJoyride/blob/a39989881ff7d8695603aa949e398d8b0cf8eaf9/src/main/java/it/unibo/jetpackjoyride/core/impl/SavesImpl.java#LL40C5-L50C6

Menzioni:

durante tutto lo sviluppo del progetto ci siamo basati sui concetti presentati nella repository oop-game-prog-patterns-2022 di aricci303 soprattutto per quanto riguarda alcune classi quali GameObject, Vector2d e Point2d e per la comunicazione tra le varie componenti dell'MVC. Inoltre per la realizzazione dello scheletro della classe GameEngineImpl è stato preso spunto dalla classe step-05-events-observer/src/rollball/core/GameEngine.java e per la realizzazione della logica nella classe HitboxImpl dalla classe step-04-collisions/src/rollball/model/RectBoundingBox.java.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Annibalini Lorenzo

Realizzare un gioco da zero è stata un'esperienza molto stimolante, una volta superato lo scoglio iniziale dovuto alla paura di non essere all'altezza posso affermare a progetto finito che nel complessivo sono stato molto soddisfatto sia del team e sia del lavoro svolto. Non mi ritengo ancora all'altezza di riusce ad auto criticare il lavoro svolto in quanto mi ritengo ancora alle prime armi con delle esperienze di questa portata anche se penso di aver sempre dato il massimo di me stesso soprattutto nelle ultime settimane dove mi sono ritrovato a dover far combaciare studio e lavoro; per questo mi sento in dovere di chiedere scusa e di ringraziare i mie compagni di progetto per avermi sopportato e supportato. Complessivamente penso che il nostro gruppo sia riuscito a superare eccellentemente tutti gli ostacoli incontrati durante le varie fasi del progetto, merito anche della comunicazione semplice, precisa e sincera che si è instaurata fra di noi sin dal primo giorno. Credo fortemente che un progetto di questo genere sia molto importante per il nostro futuro in quanto l'autogestione del lavoro assegnato, il dover rispettare le varie scadenze ed il doversi confrontare e scontrare tra più teste sia un aspetto essenziale a prescindere dal tipo di lavoro andremo a fare finita l'università.

Bacchini Lorenzo

Sviluppare un progetto come questo mi ha permesso di imparare molto soprattutto sotto il punto di vista del metodo di lavoro, era infatti la prima volta che mi trovavo a sviluppare un gioco (o un qualsiasi applicativo) di queste dimensioni e soprattutto da zero, sento quindi di aver colto le principali

tematiche riguardanti il gameProgramming e di averle fatte mie, anche se in alcuni ambiti tendo ancora a preferire soluzioni meno eleganti (fatico ancora a riconoscere pattern e uso di rado stream e lambda) sono sicuramente cresciuto a livello di analisi del problema.
In generale mi sento molto soddisfatto del progetto realizzato e spero in futuro di potervi apportare alcune altre modifiche quali: migliorie grafiche, difficoltà crescente e aggiunta di nuove skin e gadget.
Oltre al punto di vista pratico mi rendo conto di aver imparato molto anche a livello di lavoro di squadra, era la prima volta che lavoravo in team ad un progetto con una data di scadenza prefissata e mi è piaciuto confrontarmi con i miei compagni per risolvere dubbi e problemi.
Credo infine che un progetto di questo tipo sia molto importante per il futuro, in quanto permette di affacciarsi ad un contesto lavorativo nel quale personalmente non ho ancora alcuna esperienza.

Burreli Mattia

Lo sviluppo di questo progetto è risultato una sfida interessante e stimolante per diversi motivi. Nonostante qualche lacuna riguardo l'utilizzo di pattern architetturali e analisi di design, mi sento di aver realizzato un buon prodotto e di aver dato un contributo significativo al mio gruppo. Grazie a questo progetto mi sento più sicuro e maturato sulla programmazione ad oggetti e sull'utilizzo di Java, in particolare su alcuni dubbi che avevo riscontrato durante il corso. Penso però ancora di avere qualche difficoltà sull'utilizzo delle librerie grafiche come Swing, a causa del poco contributo che ho dato alla grafica del progetto. Spero in futuro di migliorare sotto questo punto di vista.
Per quanto riguarda il gruppo, penso che abbiamo lavorato bene e soprattutto di squadra. Davanti alle varie difficoltà e problematiche ho riscontrato un aiuto da parte di tutti i componenti del gruppo, una partecipazione attiva al progetto e una buona comunicazione. Ogni volta che qualcuno aveva bisogno di una mano cercavo di essere disponibile a risolvere qualsiasi problema. Inoltre questa esperienza mi ha permesso di migliorare a livello di team working.

In conclusione posso dire che questo progetto mi ha maturato molto sotto diversi punti di vista e tornerà molto utile in futuro nella creazione e gestione di nuovi progetti, anche più grandi e complessi. Inoltre spero di colmare le mie lacune precedentemente citate il prima possibile.

Sanchi Emanuele

La realizzazione di questo progetto è stata un'occasione per testare le mie conoscenze e le mie abilità nello sviluppo di programmi. Se ripenso a questi mesi di lavoro, penso che siano stati davvero tosti e impegnativi, ma non scorderò mai la prima volta che abbiamo visto muoversi barry nello schermo o quando lo sfondo ha iniziato a scorrere. Devo ammettere che ci sono stati anche momenti difficili anche perchè lavorando in gruppo è ovvio che ci siano stati degli scontri dovuti a diversità di vedute o approcci diversi a certi problemi riscontrati. Riguardo a ciò penso di essere sempre stato disponibile per i miei compagni ogni volta mi venisse chiesta una mano o un consiglio; penso però di essere stato troppo pesante per loro, per via dell'ansia di non finire in tempo il progetto.
Infine penso di potermi ritenere soddisfatto per come è stato svolto questo progetto e spero che l'esperienza accumulata mi possa aiutare nella creazione e sviluppo del prossimi.

4.2 Difficoltà incontrate e commenti per i docenti

Nonostante la grande utilità di un progetto di questo tipo, sia a livello pratico (scrittura di codice), sia a livello personale (lavoro di squadra) all'inizio ci siamo sentiti abbastanza spaesati in quanto non sapevamo bene come muoverci dovendo partire da zero e dovendo realizzare un applicativo di un certo tipo.
Durante il corso, in laboratorio abbiamo sempre dovuto completare codici già parzialmente scritti o comunque scrivere brevi codici per dimostrare di aver compreso le tematiche di programmazione ad oggetti, sarebbe forse stata utile una parte più sostanziosa di introduzione al gamePrograming anche solo di qualche ora, questo perchè abbiamo tratto un grande aiuto da parte della repository del professor Ricci e penso che se avessimo approfondito meglio l'argomento ci saremmo trovati più a nostro agio durante tutto lo sviluppo del progetto. Infine ci teniamo anche a dire che forse lo sviluppo di un progetto di queste dimensioni oltre all'esame scritto (a fronte dei 12 crediti assegnati) potrebbe portare via tempo allo studio di altri corsi.

A Guida Utente

All'inizio del gioco l'utente si trova davanti ad un menu in cui può:

- Iniziare una nuova partita premendo il bottone "New Game"
- Controllare le proprie statistiche premendo il bottone "Statistics"
- Comprare nuovi gadget o skin premendo il bottone "Shop"
- Uscire dal gioco con il bottone "Exit"

All'interno di ogni schermata ci sono altri due bottoni che permettono di tornare al menu o di uscire dal gioco. All'interno dello shop ci sono dei bottoni specifici per ogni gadget o skin per comprarli o indossarli. Se non si hanno abbastanza monete per effettuare l'acquisto ci si ritroverà davanti una finestra d'errore. Se si decide di iniziare una nuova partita, tutto ciò che va fatto è cercare di evitare gli ostacoli andando in alto tramite la pressione del tasto **Spazio**. Se si arriva al soffitto si verrà bloccati e ugualmente se si arriva sul pavimento. Ogni ostacolo uccide il player, a meno che non abbia addosso uno scudo. Gli scienziati che corrono sul pavimento possono essere uccisi andandoci addosso. Una volta terminata la partita, ci si ritroverà davanti una schermata per vedere le statistiche della partita appena conclusa e due bottoni per uscire dal gioco o tornare al menu.

B Esercitazioni di laboratorio

B.0.1

lorenzo.annibalini@studio.unibo.it

- Laboratorio 5: `https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p170019`

B.0.2

lorenzo.bacchini4@studio.unibo.it

- Laboratorio 4: `https://virtuale.unibo.it/mod/forum/discuss.php?d=113869#p169180`
- Laboratorio 5: `https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169932`
- Laboratorio 6: `https://virtuale.unibo.it/mod/forum/discuss.php?d=115548#p171636`
- Laboratorio 7: `https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p173026`
- Laboratorio 8: `https://virtuale.unibo.it/mod/forum/discuss.php?d=117852#p174202`
- Laboratorio 9: `https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p174741`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p175888`

B.0.3

mattia.burreli@studio.unibo.it

- Laboratorio 5: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p170104

B.0.4

emanuele.sanchi@tudio.unibo.it

- Laboratorio 5: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169929
- Laboratorio 7: https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p173096
- Laboratorio 9: https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p173096