Relazione
"Devil and Angel"

Matilde D'Antino, Chiara Giangiulli,
Gaia Mazzoni, Emilia Pace

10 giugno 2023

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo si pone l'obiettivo di creare un platform multiplayer game suddiviso in 3 livelli a difficoltà crescente, per ognuno di questi l'accesso sarà vincolato al superamento del livello precedente. I personaggi in gioco saranno un angelo e un diavolo, caratterizzati rispettivamente dai colori azzurro e rosso. Lo scopo di ciascun personaggio sarà il raggiungimento della porta contraddistinta dal suo colore attraverso un percorso a piani, nei quali saranno inseriti oggetti che potranno aiutare o ostacolare il superamento del livello. Per poter superare il livello sarà necessaria la collaborazione dei due personaggi.

Requisiti funzionali

- Ogni livello conterrà diamanti che permetteranno di accumulare punti che si sommeranno ad un punteggio di livello.
- Gli ostacoli saranno:
  - Pozze avvelenate di tre tipi: pozze rosse il cui tocco da parte dell'angelo causerà game over, pozze azzurre il cui tocco da parte del diavolo causerà game over, e pozze viola che porteranno al game over se toccate da qualunque personaggio.
  - Piattaforme che limiteranno il passaggio.
- Gli oggetti che invece avranno lo scopo di aiutare i giocatori saranno:
  - Bottoni che, fintanto che sono premuti da un giocatore, faranno muovere una piattaforma in una determinata direzione, mentre

quando rilasciati riporteranno gradualmente la piattaforma alla sua posizione originaria.

  - Leve che, quando spinte dal giocatore, muoveranno una piattaforma in una direzione predeterminata e quando tirate la riporteranno alla posizione originaria.

- Il gioco metterà a disposizione un menù di pausa con l'opzione di restart del livello, utile nel caso in cui i personaggi si trovassero accidentalmente in condizione di non poter proseguire.

Requisiti funzionali opzionali

E' stato necessario lasciare alcune funzionalità opzionali, con l'intenzione di implementarle in un momento successivo, dal momento che non era possibile realizzarle all'interno del monte ore del progetto. Tali funzionalità sono:

- Aggiunta di ulteriori oggetti, quali la "corrente d'aria", che fa innalzare il giocatore e gli consente di raggiungere piattaforme situate in zone elevate della mappa.

- Timer utilizzato in aggiunta ai diamanti per calcolare il punteggio complessivo ottenuto in ogni livello.

- Menù dei livelli da cui è possibile selezionare e rigiocare livelli già sbloccati in precedenza.

Requisiti non funzionali

- Volendo elaborare un multiplayer game, D-n-A dovrà essere fluido per rendere l'esperienza di gioco più piacevole.

- D-n-A dovrà essere particolarmente efficiente nell'uso degli sprite per la visualizzazione grafica delle entità in gioco.

1.2 Analisi e modello del dominio

D-n-A mette a disposizione tre livelli all'interno dei quali sono presenti due personaggi che possono interagire con tutti gli oggetti presenti, ma non tra di loro. Nel momento in cui si verifica una collisione tra il personaggio con una qualsiasi altra entità, possono accadere diverse tipologie di eventi in base all'oggetto toccato:

- La collisione con un oggetto attivatore (leva o bottone) permetterà a quest'ultimo di attivarsi nelle modalità che predispone.
- La collisione con una porta potrebbe decretare il superamento del livello.
- La collisione con una pozza potrebbe produrre il gameover del livello.
- La collisione con un diamante incrementerà il punteggio.

Inoltre, il movimento del giocatore verrà limitato nello spazio dal modo in cui sarà costruita la mappa del livello (piattaforme fisse e mobili). Una delle challenge principali affrontate è la corretta gestione delle collisioni tra le entità e gli effetti da esse causati.

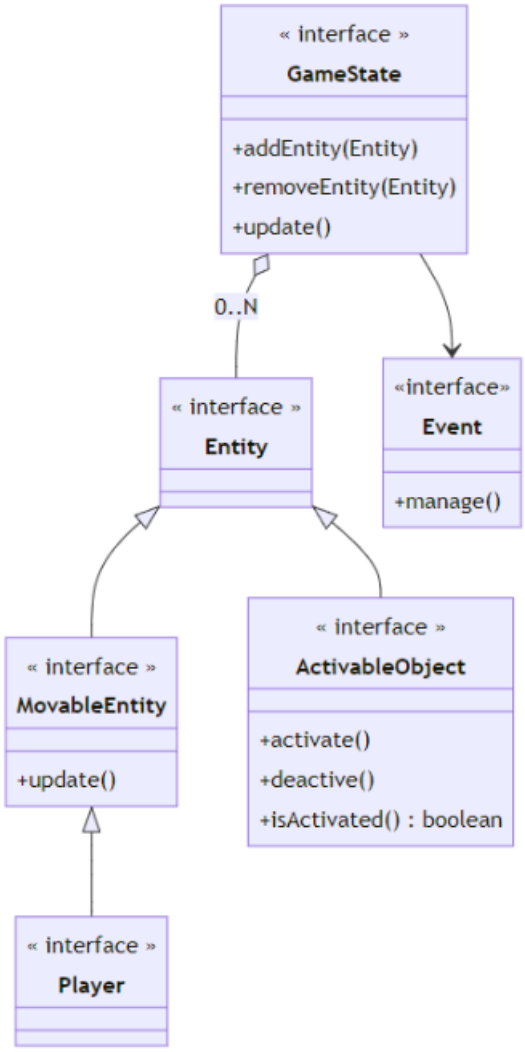

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro.

Capitolo 2

Design

2.1 Architettura

L'architettura di D-n-A segue il pattern architetturale MVC. Il ruolo di controller del sistema è affidato principalmente alla classe GameEngine che gestisce l'interazione tra lo stato attuale del gioco e la sua visualizzazione grafica. Lo stato del gioco (GameState) permette la gestione a livello model delle entità (Entity e sue estensioni), del loro comportamento e delle loro interazioni. La parte grafica (view) viene, invece, affidata alla classe Display che permette la visualizzazione a video dello stato del gioco sfruttando la classe ImageManager per l'utilizzo di immagini e l'interfaccia GameMenu per la visualizzazione dei menù. La gestione degli input è affidata alla classe InputControl, con il ruolo di controller, che permette l'esecuzione di comandi conseguenti alla pressione di specifici tasti. Come si può vedere dallo schema sotto-riportato la parte di model non contiene riferimenti a elementi di view, mentre quest'ultima interagisce con il model solo attraverso i controller. Pertanto ipotetiche modifiche alla view non impatteranno le classi di model.

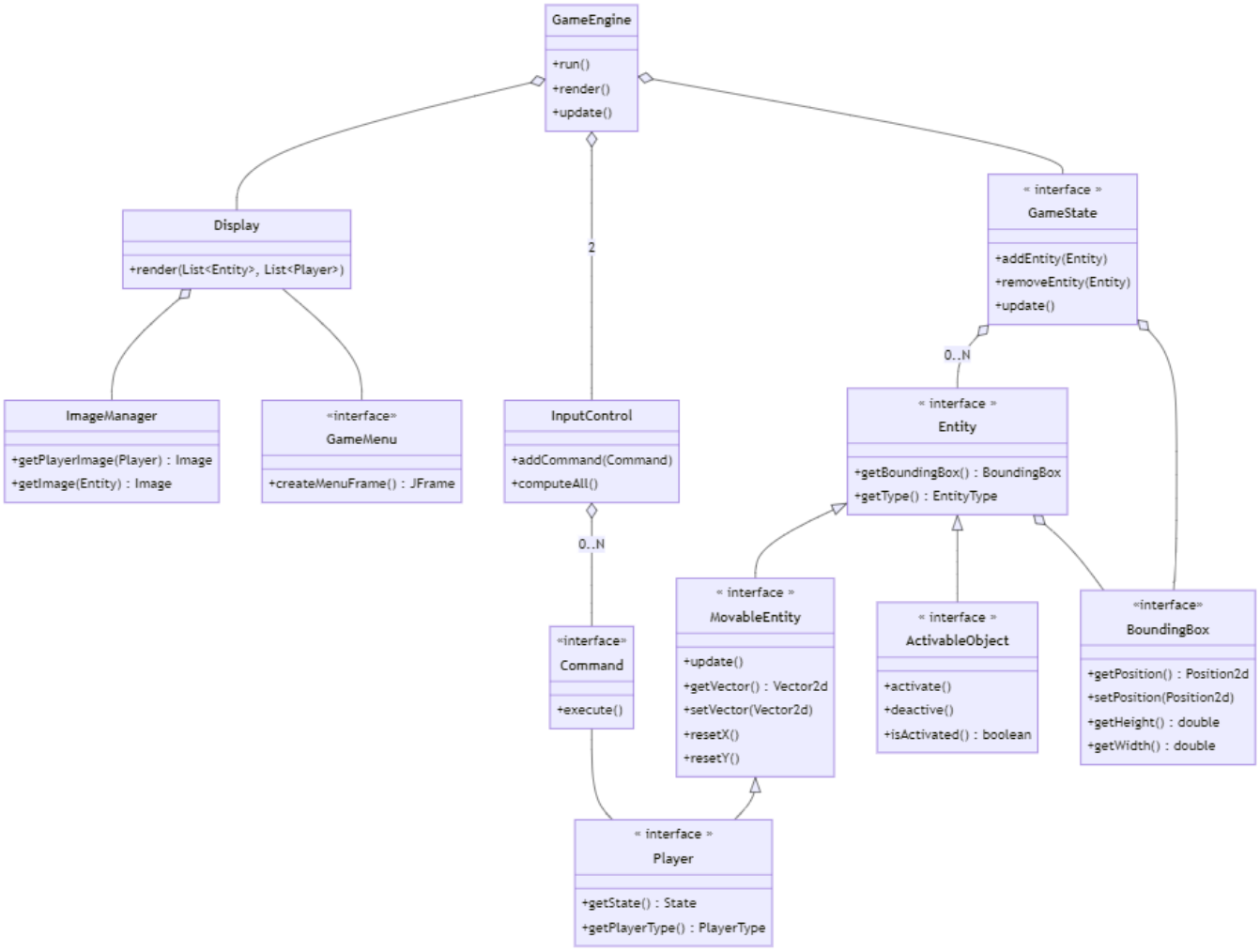

Figura 2.1: Schema UML dell'architettura.

2.2 Design dettagliato

Matilde D'Antino

Gestione Input

`Problema`

Il gioco deve gestire diversi input da tastiera per ogni personaggio. Inoltre ogni giocatore ha azioni predefinite che può compiere in base agli input rilevati.

`Soluzione`

L'interfaccia funzionale Command implementa un Command Pattern per descrivere una specifica azione che può essere eseguita. Per la definizione dei diversi comandi viene usato un Factory Method Pattern con l'interfaccia CommadFactory. La scelta di una factory è stata fatta anche per facilitare l'aggiunta di nuovi comandi.
La classe InputControl rappresenta una coda di Command, questi ultimi vengono eseguiti quando necessario dal metodo computeAll() che, di conseguenza, libera la coda. I comandi in coda vengono aggiunti dalla classe KeyboardHandler che riconosce il tasto coinvolto in un evento da tastiera e associa l'azione corrispondente.

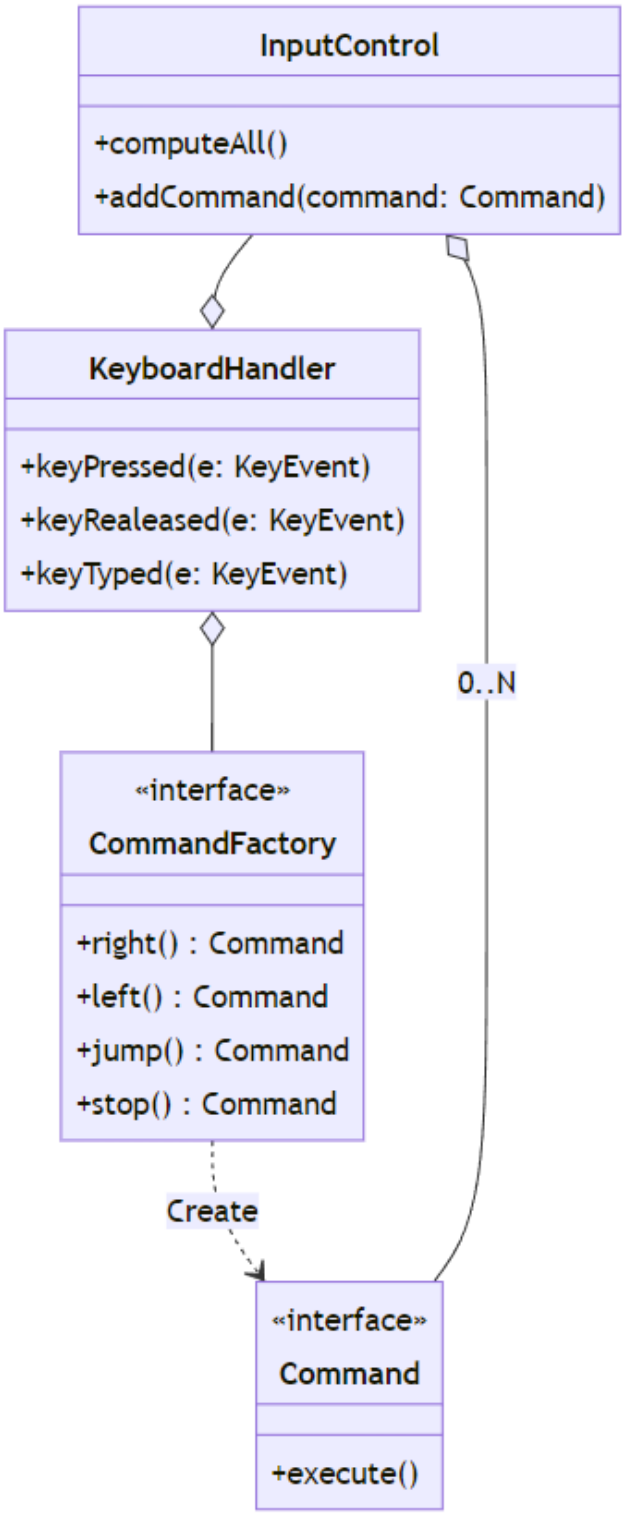

Figura 2.2: Schema UML della gestione input

Immagini del Player

`Problema`

Ad ogni tipo di personaggio (angelo e diavolo) verranno associate diverse immagini. Queste ultime dovranno cambiare in base al movimento del personaggio (camminare a destra o sinistra, saltare,...).

`Soluzione`

Ad ogni personaggio viene associato uno stato. La classe State contiene due campi di tipo StateEnum (enumerazione di stati possibili): il primo indica se il personaggio sta saltando o è a terra, mentre il secondo indica se è rivolto verso destra, sinistra o di fronte. La classe StateObserver associa ad ogni coppia di StateEnum e al tipo di personaggio (angelo o diavolo) un'immagine. Quest'ultima sarà poi utile a ImageManager che la restituirà correttamente al Display per la sua visualizzazione.

Per evitare di aggiornare, ad ogni giro di GameEngine, la corretta immagine del personaggio da visualizzare, le classi State e StateObsever implemenatno un Observer Pattern. La subject del pattern è la classe State che contiene una lista di observer, ogni volta che viene modificato il valore di uno dei due StateEnum vengono avvisati gli observer del cambiamento. Lo StateObserver, invece, appena riceve la notifica di un cambiamento dello State, ricerca l'immagine associata al nuovo stato per passarla correttamente alla classe ImageManager.

Per dare il senso di movimento quando il personaggio cammina verso destra o verso sinistra, la classe State ha un metodo update che notifica un cambiamento di immagine agli observer dopo un certo numero di frame.

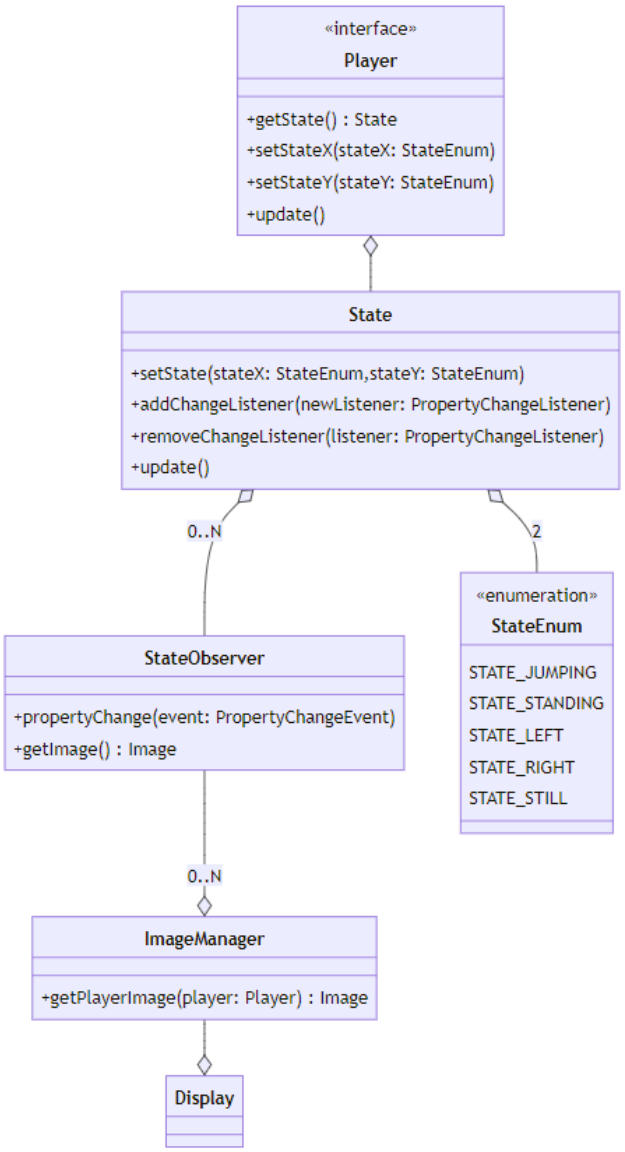

Figura 2.3: Schema UML della gestione delle immagini del player

Chiara Giangiulli

Creazione eventi di collisione

`Problema`

Durante lo svolgimento del gioco il personaggio si trova ad interagire con numerosi oggetti che lo ostacolano o gli permettono di superare il livello. Risulta necessario, per questo motivo, gestire separatamente i vari tipi di collisione, ognuna in base alle conseguenze che apporta al gioco.

`Soluzione`

Per la gestione dei vari tipi di collisione è stato utilizzato un Factory Method Pattern, che ha il compito di creare, tramite appositi metodi, gli eventi conseguenza di queste collisioni. Ognuno di essi viene creato, tramite uno specifico metodo dell'interfaccia EventFactory, nel momento in cui viene riconosciuta una collisione del personaggio con quel determinato tipo di entità. Tramite l'utilizzo di questa soluzione,inoltre, potrebbe risultare semplice l'aggiunta di eventi in futuro, magari legata all'introduzione di nuove entità di gioco, che può essere fatta con una minima modifica di codice, aggiungendo alla factory l'apposito metodo.

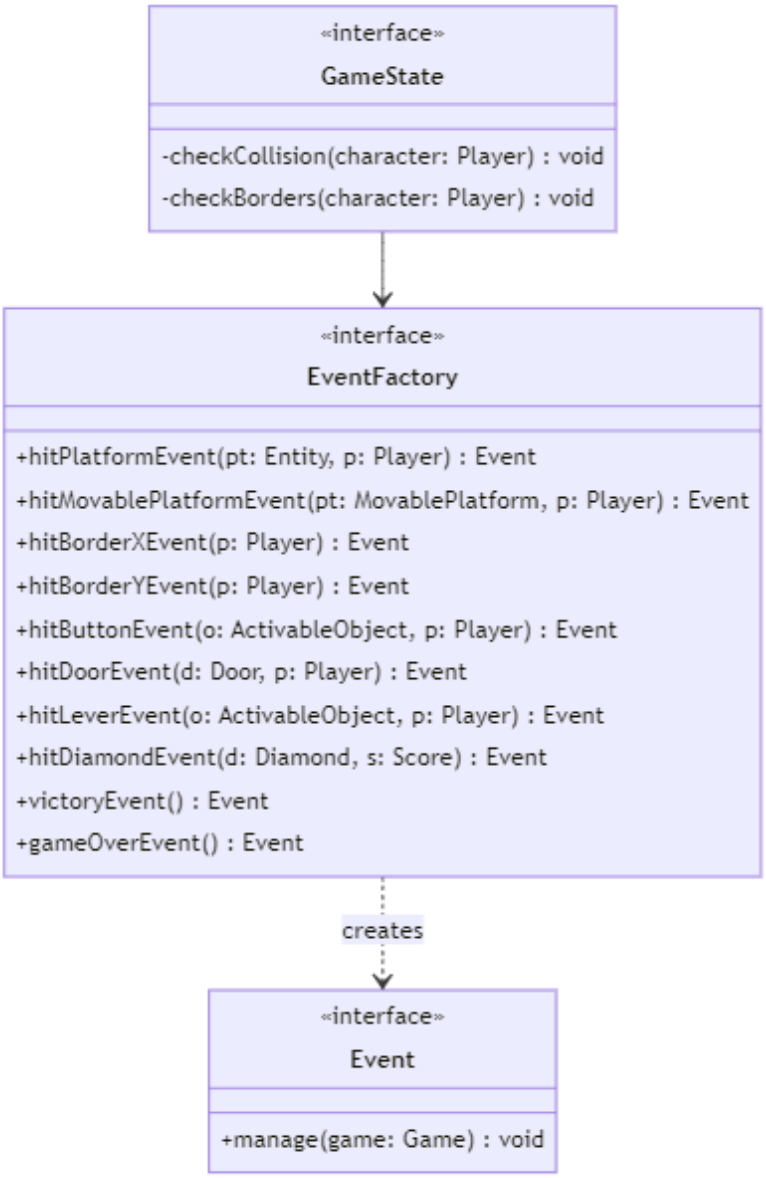

Figura 2.4: Schema UML della creazione degli eventi di collisione

Gestione eventi

`Problema`

La gestione degli eventi di collisione nel momento esatto in cui essi avvengono e sono riconosciuti causa modifiche concorrenti allo stato del gioco. Delle entità, ad esempio, potrebbero dover essere rimosse a seguito di una collisione con un personaggio, ma se la gestione del relativo evento avviene quando questo viene riconosciuto, l'insieme di entità viene modificato proprio durante il controllo delle collisioni sullo stesso.

`Soluzione`

Per risolvere questo problema è stata introdotta una coda di eventi, utilizzando un EventQueue pattern, che ha lo scopo di mantenere e conservare gli eventi mano a mano che avvengono, per poi gestirli successivamente secondo una politica First In, First Out, conservando l'ordine in sono avvenuti, ma ritardando la loro gestione in un momento più opportuno.

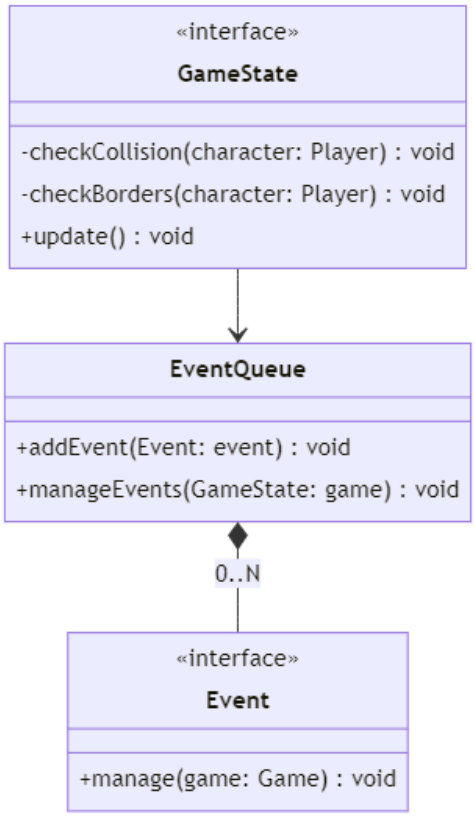

Figura 2.5: Schema UML della gestione degli eventi di collisione

Gaia Mazzoni

Creazione Entità

`Problema`

Quando viene inizializzata una partita, o al superamento del livello, è necessario inizializzare tutte le diverse entità presenti nel livello. Il codice cliente ha bisogno di sapere le classi da costruire, ed i relativi parametri da passare. In questo modo, il codice è complicato e difficile da mantenere.

`Soluzione`

Per minimizzare la ripetitività del codice, e rendere più semplice la creazione delle entità, si è creata una interfaccia EntityFactory che implementa il pattern Factory. Questa interfaccia è implementata concretamente nella classe EntityFactoryImpl, che contiene un solo metodo, createFactory(), al quale vengono passati alcuni parametri. Uno di questi parametri è una enum, definita nell'interfaccia Entity, che rappresenta di che tipo è l'entità che si vuole creare. In questo modo, per inizializzare le entità, il codice cliente necessita solo di sapere il tipo dell'entità che deve creare, senza preoccuparsi di quale sia la sua classe effettiva. Grazie a all'interfaccia EntityFactory, si riesce pertanto a ridurre la duplicazione di codice, ed a semplificare il codice di creazione delle entità. Inoltre, la conoscenza dei dettagli implementativi delle varie entità è delegata, dal codice cliente all'interfaccia. In questo modo, se dovessero esserci modifiche future alle classi concrete delle entità, non sarà necessario modificare il codice cliente,e una futura aggiunta di nuove entità potrà essere facilmente realizzata.

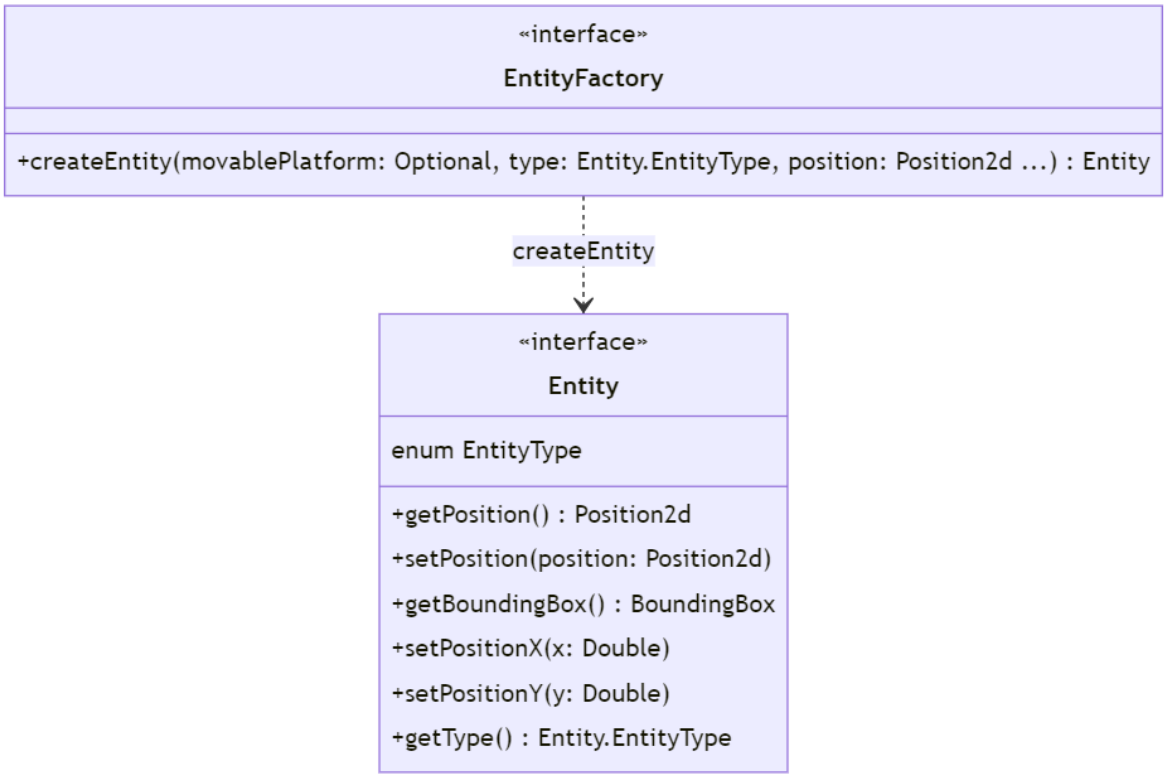

Figura 2.6: Schema UML della creazione delle entità.

Emilia Pace

Apertura Menù

`Problema`

Appena viene inizializzata una partita, al superamento di un livello, alla perdita di questo livello o quando si vuole mettere il gioco in pausa, è necessario aprire e gestire i diversi menù associati a queste azioni. Il codice cliente ha bisogno di saper costruire il JFrame con gli oggetti (bottoni, immagini o label), con le giuste azioni collegate a questi. In questo modo il codice si complica ed è difficile da mantenere.

`Soluzione`

Alla classe GameMenu si associa un'interfaccia GameMenuFactory che implementa il pattern Factory. L'interfaccia della Factory viene implementata effettivamente nella classe GameMenuFactoryImpl che ha diversi metodi in base al menu di cui si ha bisogno:

- startMenu: primo menù che appare appena si fa partire il gioco con bottoni start e guide.
- gameOverMenu: con un bottone di restart, per rigiocare il livello.
- victoryMenu: con un bottone NextLevel per giocare il livello successivo.
- lastVictoryMenu: che apre l'ultimo menu del gioco nel momento in cui si sono superati tutti i livelli disponibili nel gioco.
- pauseMenu: che è l'unico che non apre un JFrame ma in cambio apre un MessageDialog con un bottone Restart in caso si voglia riniziare il livello o un bottone "ok" per poter continuare il livello attuale.

Inoltre nella GameMenuFactoryImpl ci sono i getter di ogni bottone utile ai menu creati con i parametri necessari per definire l'ActionListener di ognuno.

Grazie a questa factory il codice si semplifica molto unendo la creazione di questi frame e bottoni in una sola classe che li va a posizionare e inizializzare non appena si chiama l'apertura del menù di cui si ha bisogno.

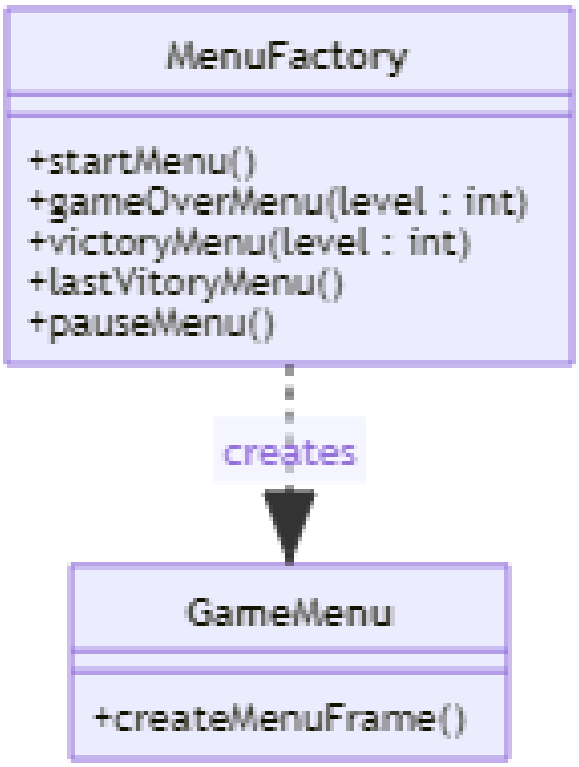

Figura 2.7: Schema UML dell'apertura dei Menu.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per effettuare i test si è scelto di utilizzare JUnit 5. I test implementati sono:

- MovementTest che si occupa di testare il movimento dei personaggi (modificando posizione e vettore associati) e del loro cambio di stato corrispondente alle azioni eseguite.
- CollisionTest che si occupa di testare la collisione dei personaggi con i bordi di gioco e con altre entità, utilizzando per il controllo bounding box rettangolari.
- ObjectTest che si occupa di testare la creazione delle entità e i il corretto funzionamento dei loro metodi, quali apertura/chiusura di porte, attivazione di leve e bottoni, spostamento di piattaforme nella corretta direzione.

3.2 Metodologia di lavoro

Prima di iniziare lo sviluppo vero e proprio del progetto ci siamo incontrati per individuare le interfacce principali. La suddivisione del lavoro è stata pensata in modo da poter lavorare individualmente per la maggior parte dello sviluppo e incontrarci solo per unire le diverse parti e implementare le classi condivise, riguardanti per lo più la grafica del gioco o l'interazione tra entità. Abbiamo deciso di utilizzare il DVCS con l'approccio spiegato in aula. Abbiamo creato il repository git che è stato clonato da ognuno di noi per lavorare individualmente, condividendo le aggiunte mediante i comandi pull/push.

Matilde D'Antino

- Gestione dell'Input (packages it.unibo.dna.view.input, it.unibo.dna.controller.inputcontrol,it.unibo.dna.model.command)
- Implementazione AbstractEntity (nel package it.unibo.dna.model.object.entity.impl)
- Implementazione AbstractMovableEntity (nel package it.unibo.dna.model.object.movableentity.impl)
- Implementazione Player (package it.unibo.dna.model.object.player)
- Impletazione StateObserver (new package it.unibo.dna.view.image)
- Implementazione SoundManager (package it.unibo.dna.view.sound)

Chiara Giangiulli

- Gestione hitbox (package it.unibo.dna.model.box)
- Gestione collisioni e stato di gioco (package it.unibo.dna.model.game.gamestate)
- Gestione factory di eventi e coda di eventi (package it.unibo.dna.model.game.events)
- Implementazione Diamond (nel package it.unibo.dna.model.object.stillentity.impl)

Gaia Mazzoni

- Gestione entità e sua factory (package it.unibo.dna.model.object.entity)
- Gestione entità mobili (package it.unibo.dna.model.object.movableentity)
- Gestione entità fisse (package it.unibo.dna.model.object.stillentity)
- Implementazione ImageManager (package it.unibo.dna.view.image)

Emilia Pace

- Implementazione dei menù (package it.unibo.dna.view.menu.impl)
- Implementazione della classe Level (package it.unibo.dna.model.game.level)
- Implementazione GameLoop (package it.unibo.dna.controller.core)

In condivisione

- Classi con funzionalità di base (package it.unibo.dna.model.common)
- Classe relativa alla grafica (classe Display nel package it.unibo.dna.view)
- Implementazione della factory di eventi (classe it.unibo.dna.model.events.impl.EventFactoryImpl)

3.3 Note di sviluppo

Matilde D'Antino

Feature avanzate di linguaggio utilizzate:

- Interfaccia funzionale:
  - Intefaccia Command: `https://github.com/emilia-pace/OOP22-D-n-A/blob/ef74a3aee435f4edc7a9cc30ac050f743fefaab1/src/main/java/it/unibo/dna/model/command/api/Command.java#L6`
- Lambda expressions:
  - Nella coda di comandi della classe InputControlImpl: `https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/controller/inputcontrol/impl/InputControlImpl.java#L23`
  - Per ogni metodo di CommandFactoryImpl: `https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/command/impl/CommandFactoryImpl.java#LL34C9-L37C11`

https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/command/impl/CommandFactoryImpl.java#LL45C9-L48C11
https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/command/impl/CommandFactoryImpl.java#LL56C9-L67C11
https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/command/impl/CommandFactoryImpl.java#LL75C9-L78C11

  - Nella lista di observer della classe State:
    https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/object/player/impl/State.java#LL108C9-L108C113

  - Nella mappa delle immagini dello StateObserver:
    https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/view/image/StateObserver.java#L83

Chiara Giangiulli

Feature avanzate di linguaggio utilizzate:

- Interfaccia funzionale:
  - intefaccia Event:
    https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/events/api/Event.java#LL8C1-L15C2
- Lambda expressions:
  - nell'implementazione del metodo manage() dell'interfaccia Event in tutti i metodi di EventFactoryImpl (riportati alcuni esempi):
    https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/events/impl/EventFactoryImpl.java#LL24C9-L24C25

https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/events/impl/EventFactoryImpl.java#LL43C9-L43C23
https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/events/impl/EventFactoryImpl.java#LL66C9-L66C24

  - nello switch case nel metodo freeObject():
    https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/game/gamestate/impl/GameStateImpl.java#LL145C21-L163C22
  - nello switch case del metodo checkCollision():
    https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/game/gamestate/impl/GameStateImpl.java#LL191C21-L217C22

- Stream:
  - per il controllo dei personaggi nell'update() del GameStateImpl:
    https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/game/gamestate/impl/GameStateImpl.java#LL59C9-L63C12
  - per il controllo sul rilascio di oggetti in freeObject():
    https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/game/gamestate/impl/GameStateImpl.java#LL141C9-L143C33
  - per il controllo delle collisioni in GameStateImpl:
    https://github.com/emilia-pace/OOP22-D-n-A/blob/650d9181d304cc8d98a266a046a71e74a4851058/src/main/java/it/unibo/dna/model/game/gamestate/impl/GameStateImpl.java#LL189C9-L190C33

Gaia Mazzoni

Feature avanzate di linguaggio utilizzate:

- Lambda expressions:

  - Nello switch-case in:
    https://github.com/emilia-pace/OOP22-D-n-A/blob/693e60af8c60d6f202e87baeea5054a0019c89c9/src/main/java/it/unibo/dna/model/object/entity/impl/EntityFactoryImpl.java#LL30C5-L58C6

- Optionals:
  - Come parametro del metodo createEntity():
    https://github.com/emilia-pace/OOP22-D-n-A/blob/693e60af8c60d6f202e87baeea5054a0019c89c9/src/main/java/it/unibo/dna/model/object/entity/api/EntityFactory.java#L77
  - Come campo in ActivableObjectImpl:
    https://github.com/emilia-pace/OOP22-D-n-A/blob/693e60af8c60d6f202e87baeea5054a0019c89c9/src/main/java/it/unibo/dna/model/object/stillentity/impl/ActivableObjectImpl.java#L20
  - Come campo in Door:
    https://github.com/emilia-pace/OOP22-D-n-A/blob/693e60af8c60d6f202e87baeea5054a0019c89c9/src/main/java/it/unibo/dna/model/object/stillentity/impl/Door.java#L29

- Varargs:
  - Come parametro del metodo createEntity():
    https://github.com/emilia-pace/OOP22-D-n-A/blob/693e60af8c60d6f202e87baeea5054a0019c89c9/src/main/java/it/unibo/dna/model/object/entity/api/EntityFactory.java#L77

- Stream:
  - Per fare l'update delle MovablePlatform:
    https://github.com/emilia-pace/OOP22-D-n-A/blob/693e60af8c60d6f202e87baeea5054a0019c89c9/src/main/java/it/unibo/dna/model/game/gamestate/impl/GameStateImpl.java#LL69C1-L72C64
  - Per controllare le Door aperte:
    https://github.com/emilia-pace/OOP22-D-n-A/blob/693e60af8c60d6f202e87baeea5054a0019c89c9/src/main/java/it/unibo/dna/model/game/gamestate/impl/GameStateImpl.java#LL171C1-L177C6

Emilia Pace

- Interfaccia funzionale:
  - Intefaccia GameMenu:
    `https://github.com/emilia-pace/OOP22-D-n-A/blob/ccf861d5228bca4e605e2052735b12e078c11098/src/main/java/it/unibo/dna/view/menu/api/GameMenu.java#LL8C1-L15C2`
- Optionals:
  - Come parametro del metodo per i bottoni e le leve:
    `https://github.com/emilia-pace/OOP22-D-n-A/blob/ef74a3aee435f4edc7a9cc30ac050f743fefaab1/src/main/java/it/unibo/dna/model/game/level/Level.java#LL87C1-L93C27https://github.com/emilia-pace/OOP22-D-n-A/blob/ef74a3aee435f4edc7a9cc30ac050f743fefaab1/src/main/java/it/unibo/dna/model/game/level/Level.java#LL87C1-L93C27`
- Stream:
  - Per filtrare gli oggetti MovablePlatform dalla lista di entità e ottenere l'ultimo elemento.:
    `https://github.com/emilia-pace/OOP22-D-n-A/blob/ef74a3aee435f4edc7a9cc30ac050f743fefaab1/src/main/java/it/unibo/dna/model/game/level/Level.java#LL87C1-L93C27`

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Matilde D'Antino

Sono soddisfatta dell'esperienza intrapresa grazie a questo progetto. Mi ha dato la possibilità di completare delle conoscenze apprese durante lo studio di teoria. E' stata sicuramente una sfida lavorare in gruppo, avendo una personalità che tende ad essere molto individualista nelle decisioni. Mi sono, però, fidata ciecamente del lavoro dei miei compagni e sono molto contenta del risultato che siamo riusciti ad ottenere insieme. Se potessi tornare indietro con le conoscenze ottenute durante questi mesi di programmazione probabilmente avrei impostato il mio lavoro iniziale in modo differente, magari utilizzando design pattern diversi, per rendere la sfida più complicata, ma sicuramente meglio organizzata. In futuro ho intenzione di riprendere questo lavoro per revisionare le parti di codice che sono certamente migliorabili come la visualizzazione delle immagini del personaggio. Inoltre mi piacerebbe aggiungere ostacoli per riprendere in modo fedele il gioco originale "FireBoy & WaterGirl" dal quale il nostro progetto è stato fortemente ispirato.

Chiara Giangiulli

Ritengo che questo progetto sia stata una vera e propria sfida per me. Per quanto io sia consapevole del fatto che il mio lavoro sia da migliorare, sono molto soddisfatta del risultato finale e ritengo che sia stato molto utile per comprendere e saper applicare meglio quanto imparato durante il corso. Se mi trovassi a dover ricominciare da zero questo progetto ad oggi probabilmente mi troverei davanti molti meno ostacoli, le tempistiche di sviluppo sarebbero decisamente migliori, così come la fase di analisi più sicura e spedita. Il mio obiettivo sarebbe sicuramente quello di migliorare il codice e

l'utilizzo di pattern, per questo mi piacerebbe sicuramente tornare a lavorarci in un futuro, magari dopo aver acquisito maggiore esperienza e con più tempo a disposizione. Nonostante ciò sono comunque fiera della gestione del nostro lavoro, del dialogo, della determinazione e della perseveranza dimostrata da tutti i componenti del gruppo perché, per quanto questo progetto fosse una realtà quasi del tutto nuova per noi, siamo comunque riusciti passo per passo a gestirla al meglio delle nostre capacità.

Gaia Mazzoni

Nonostante le difficoltà incontrate nella realizzazione di questo progetto, mi ritengo soddisfatta dei risultati ottenuti. Certamente il mio codice non è perfetto, sarebbe potuto essere più efficace, meno ripetitivo, tuttavia sono contenta di quello che ho fatto. Ho imparato a progettare meglio le classi, a prevedere possibili cambiamenti futuri, e a scrivere codice più conciso, ed ho potuto applicare molti degli argomenti visti a lezione. Se potessi tornare indietro, e dare un consiglio alla me del passato, le direi di dare maggiore importanza alla fase di pianificazione, di abbozzare subito molte idee diverse, e di valutare la loro implementazione e pattern utilizzabili, in modo da velocizzare le, inevitabili, sessioni di refactoring e ristrutturazione delle classi. Sono rimasta molto soddisfatta dalla nostra capacità di lavorare in gruppo, di organizzare il lavoro, e di rispettare le scadenze da noi stesse prefissate. Sebbene il lavoro non sia stato perfetto, considerando che era il mio primo lavoro di programmazione in Java su questa scala, mi ritengo soddisfatta di me stessa. In futuro mi piacerebbe rimettere mano a questo progetto, per aggiungere nuovi oggetti e funzionalità di gioco, che non è stato possibile implementare all'interno del monte ore, ma soprattutto per mettere in pratica ulteriori pattern.

Emilia Pace

Questo progetto è stato una vera e propria sfida personale per me. Lavorare in un progetto di queste portate e complessità senza avere nessuna esperienza al riguardo è stato molto impegnativo. A prescindere dalle difficoltà e dai problemi, tanto riguardanti il progetto come problemi personali che hanno interferito con il mio lavoro, sono abbastanza soddisfatto del risultato finale, che, anche se può essere migliorato in tanti modi è frutto di un lavoro in cui abbiamo messo tantissimo impegno.
Ho imparato molte cose su come si progettano le classi, su come si costruisce un progetto in più persone in cui si deve far in modo che tutti abbiano accesso alle parti dei codici altrui di cui hanno bisogno. Inoltre ho imparato a scrivere un codice più formale, meno ripetitivo e più comprensibile. Tutte queste cose ancora migliorabili.

Al me del passato direi di gestirsi meglio le scadenze e costruire meglio un'idea iniziale, che si può si cambiare ma non stravolgere. Di informarsi meglio su quello che può o non può fare prima di pianificare un lavoro che poi scoprirà inutile o irrealizzabile.
Non posso dire niente di negativo sul nostro lavoro di gruppo, ognuno ha fatto la propria parte rispettando scadenze e aiutando nel possibile gli altri. Le mie compagne sono state pazienti e utili quando mi trovavo in difficoltà nel lavoro che stavo svolgendo. A prescindere dal risultato finale sono molto soddisfatto della dinamica di questo gruppo e sicuramente mi hanno aiutato ad ampliare le mie conoscenze.
In futuro mi piacerebbe rimettere mano al codice per migliorare con conoscenze aquisite il codice gia esistente aggiungendo anche utilizzo di pattern o features più avanzate. Inoltre credo che questo progetto abbia grandi potenzialità per aggiungere funzioni, livelli o ogetti nuovi.

Capitolo 5

Guida utente

All'inizio del gioco viene visualizzata una finestra con il logo del gioco e due pulsanti:

- Start per far partire il gioco.
- Guide per visualizzare le istruzioni di gioco e i tasti da utilizzare. (l'angelo si muove con le freccette della tastiera, il diavolo con i tasti "A" "W" "D")

Durante lo svolgimento del gioco è possibile inoltre interromperlo grazie al pulsante di pausa che si trova in cima alla finestra. Alla pressione del pulsante di pausa è possibile scegliere di uscire dal gioco (X in cima alla finestra), continuare (Ok) o ricominciare il livello corrente (Restart). Superato un livello è possibile passare al successivo (quando esiste, tramite Next) o uscire dal gioco. In caso di game over è possibile ricominciare il livello corrente (Restart) o uscire dal gioco. Infine, in caso di vittoria finale (superamento di tutti e tre i livelli) il gioco è terminato.

Capitolo 6

Bibliografia

Le classi Position2d e Vector2d sono state prese e modificate a partire dalle classi P2d e V2d viste con il Professor Ricci, a partire da un codice fornitoci su Virtuale al link:

`https://github.com/pslab-unibo/oop-game-prog-patterns-2022/tree/master/step-02-first-modelling/src/rollball/common`

La classe Pair è stata presa da un appello d'esame presente su Virtuale al link:

`https://bitbucket.org/mviroli/oop2022-esami/src/master/a01a/e1/Pair.java`

Immagini e suoni sono a produzione personale ad esclusione dello sfondo di gioco, link:

`https://pixabay.com/it/illustrations/mattone-mattoni-disegno-disegno-4956032/`