Air-Hockey

Relazione per Programmazione ad Oggetti

Emanuele Borghini, Edoardo La Greca,
Francesca Lanzi, Pablo Sebastian Vargas Grateron

24 giugno 2021

Indice

Capitolo 1

Analisi

Air-Hockey è un videogioco che mira a riprodurre l'omonimo gioco americano inventato nel ventesimo secolo e successivamente diventato popolare nelle sale giochi di tutto il mondo.
Il gioco è composto da un dischetto e due piattini disposti su un tavolo costruito in modo tale da ridurre la frizione con il dischetto.
L'obiettivo del gioco è segnare dei gol nella porta dell'avversario, il primo giocatore a raggiungere il punteggio prestabilito vince.

1.1 Requisiti

Il software deve fornire la possibilità di giocare una partita dall'inizio alla fine mantenendo una esperienza di gioco simile alla realtà.

Requisiti funzionali

- Il gioco deve permettere all'utente di iniziare una nuova partita, riprendere una partita precedentemente memorizzata e di modificare le differenti opzioni del gioco tra quelle messe a disposizione.
- Per avere un'esperienza realistica, il programma deve essere molto veloce nel processare i dati di input dell'utente, elaborarli e aggiornare la fisica e logica di gioco.
- La fisica del gioco deve cercare di rappresentare al meglio la dinamica e le collisioni di ogni elemento presente nella partita.
- Il software deve fornire un'intelligenza artificiale, la quale deve avere diversi livelli di difficoltà, capace di affrontare il giocatore umano.

- Durante la partita, il giocatore deve avere la possibilità di pausare, riprendere o eventualmente, salvare e uscire dalla partita.
- L'utente deve avere la possibilità di personalizzare l'ambiente di gioco, cambiando il proprio soprannome, il colore degli elementi di gioco, il punteggio massimo di ogni partita, la difficoltà dell'intelligenza artificiale.
- Devono essere presenti alcuni obiettivi di gioco, ottenibili dal giocatore a seconda delle sue prestazioni durante la partita.

Requisiti non funzionali

- La dimensione della finestra del gioco deve essere scalata in modo tale da essere usufruibile indipendentemente dallo schermo del dispositivo.

1.2 Analisi e modello del dominio

Per gestire una partita di Air Hockey è necessario progettare singolarmente ogni elemento di gioco (un'*arena*, due *piattini* e un *dischetto*) e poi riuscire a farli interagire tra loro tramite collisioni limitate alla superficie interna dell'area di gioco.
Il giocatore dovrà essere in grado di controllare il proprio piattino nella sua metà campo, mentre il giocatore nemico verrà fornito di un'intelligenza artificiale che dovrà calcolare le proprie mosse durante lo svolgimento della partita.
Una volta definiti i modelli e le interazioni degli elementi di gioco sarà necessario stabilire le regole che definiscono la partita (quando finisce, il punteggio massimo raggiungibile, ...)
La difficoltà principale risiede nel riuscire a salvare tutte le informazioni riguardanti la partita in corso e permettere all'utente di riprendere lo stato precedente in qualsiasi momento.
Per personalizzare l'esperienza di ogni utente verranno introdotte delle impostazioni, modificabili a piacimento, in grado di alterare diversi fattori di gioco, quali difficoltà, durata della partita, nome del giocatore.
Infine per premiare l'utente una volta terminata la partita sarà necessario definire degli obbiettivi di gioco in base ai dati della partita stessa e mostrare quali sono stati completati e quali no.
Qui di seguito vengono mostrati degli schemi UML che descrivono la macrostruttura del dominio, si noti che gli schemi non descrivono la struttura del

software nel suo intero, ma vengono rappresentati solamente le componenti principali e le connessioni tra di essi.

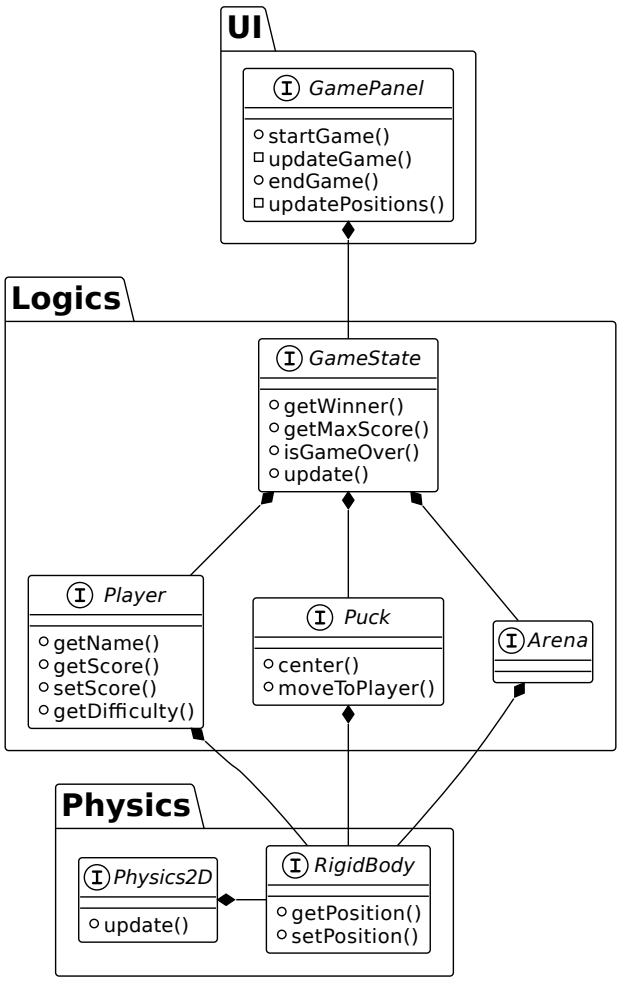

Figura 1.1: UML dell'analisi e dominio del gioco.

Capitolo 2

Design

2.1 Architettura

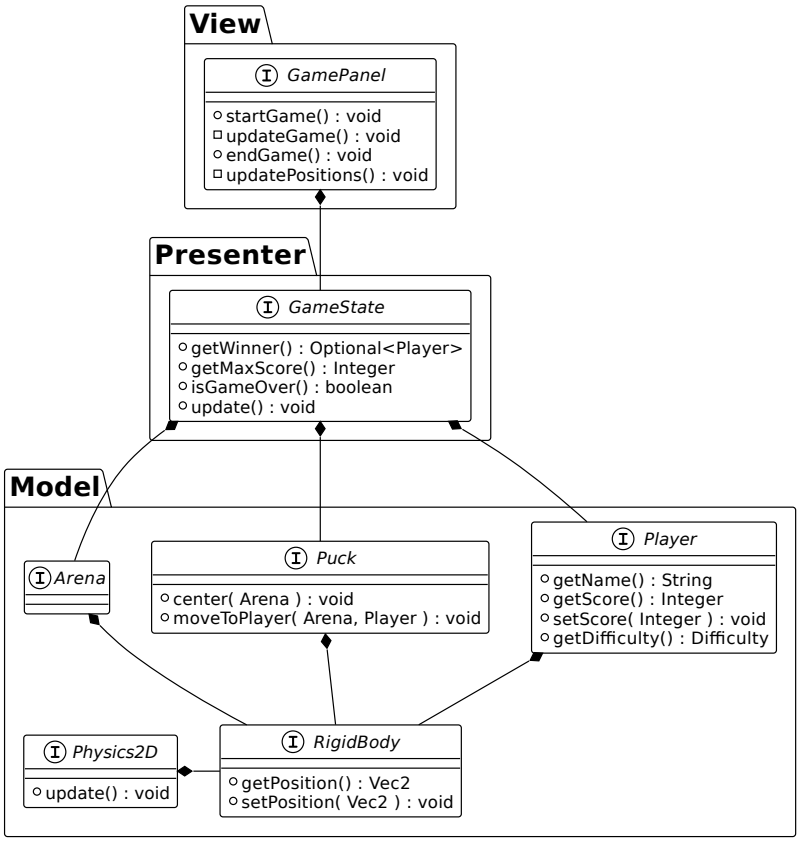

Figura 2.1: UML rappresentante le diverse componenti dell'architettura MVP personalizzata.

Come mostrato in Figura 2.1, la struttura del progetto non segue il pattern architetturale *MVC*, ma una versione più simile (ma non direttamente comparabile) al pattern `MVP (Model-View-Presenter)` dove il *Presenter* fa sia da mediatore tra *View-Model* che aggiunge una piccola parte di Model complementare alla logica del Model già presente, l'architettura definisce un collegamento tra la classe di *View*: `GamePanel` e il *Presenter* definito dall'interfaccia di logica `GameState` che con dei metodi appositi permette di collegare le informazioni della parte di *Model* e gestire alcuni input della *View*.
Più in dettaglio, la *View* cattura gli eventi generati dall'utente (come movimenti del cursore o click del mouse) o generati periodicamente, questi vengono inviati alla parte di *Model* tramite il *Presenter* che gestisce i comandi ricevuti, dopo di ché una volta aggiornati i dati del *Model*, l'interfaccia grafica passando di nuovo attraverso il *Presenter* aggiorna il proprio stato per poi mostrarlo all'utente.
Il gioco è stato suddiviso in quattro `package`:
`gui` (interfaccia utente), `logics` (logica), `physics` e `utils`.
Ogni package è costituito da tutte le classi facenti parte di tale contesto. In `utils` sono state poste le cosiddette "utility classes", ovvero classi che forniscono metodi per le operazioni più comuni che altrimenti richiederebbero di scrivere codice ripetuto.
Tra i `package` menzionati, la **logica** è il componente più importante. Infatti, essa racchiude tutta la parte che descrive gli oggetti di gioco e definisce le regole che li collegano tra loro.
I quattro componenti erano nati all'inizio del progetto con l'obbiettivo di separare view, model e controller, ma con l'avanzare del progetto, una parte di *model* si è fusa con il *controller* creando il *presenter* finale.
I diversi package hanno i seguenti ruoli:

- Il package `gui` si occupa di tutta la parte grafica, ovvero l'interfaccia utente, gli eventi del mouse e le chiamate ai metodi della parte logica.
- Il package `logics` si occupa di tutto ciò che concerne la logica di gioco e definisce le regole del Model. Questo package fa ampiamente uso del package physics per modificare lo stato del Model.
- Il package `physics` svolge il ruolo di coordinare le interazioni fisiche tra gli oggetti.
- Il package `utils` racchiude al suo interno delle classi il cui unico scopo è fornire dei metodi utili ad evitare ripetizioni di codice.

2.2 Design dettagliato

2.2.1 Emanuele Borghini

Sviluppo dell'interfaccia grafica

Problema: Velocizzare lo sviluppo dell'interfaccia grafica
Serviva un modello che determinasse la struttura del codice riguardante l'interfaccia grafica, in modo da rendere l'aggiunta di interfacce più rapida e semplice.

Soluzione: Ho deciso di creare una classe astratta `AbstractGridBagLayoutJPanel`, che segue il pattern comportamentale **Template Method** (anche se semplificato a pochi metodi), permette a tutte le classi di interfaccia grafica di usare i metodi ereditati dal componente grafico della super-classe oltre a quelli definiti dal layout manager.
Ciò ha permesso uno sviluppo dell'interfaccia utente uniforme e senza ripetizioni di codice, inoltre verso fine sviluppo ha permesso anche di supportare la funzionalità **Look and Feel** di `java Swing`

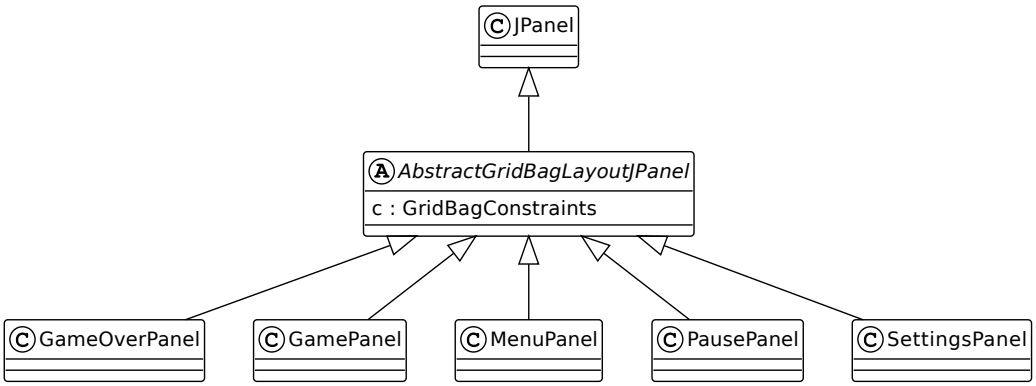

Figura 2.2: Classi di gui che fanno uso di `AbstractGridBagLayoutJPanel`

Problema: Scambiare efficientemente finestre
Una volta creata la finestra principale contenente il menù di gioco, era necessario trovare un metodo per riuscire ad istanziare una nuova schermata e rimuovere quella corrente in maniera rapida, senza creare problemi di prestazioni.

Soluzione: La soluzione concisa che ho creato risiede nell'utility class `JComponentLoader`, è una classe che fornisce dei metodi statici richiamabili da tut-

ta la UI che permettono di risalire al componente più in alto nella gerarchia e sostituirlo con un altro.

Logica di gioco

Problema: Creare l'astrazione di tutti gli elementi di gioco

Soluzione: L'idea originale era quella di creare un'interfaccia funzionale che permettesse a tutte le classi che la implementavano di aggiornare lo stato degli oggetti a ogni "*step*" di gioco, uno *step* non è altro che l'avanzamento dello stato del gioco in un determinato lasso di tempo. Ogni classe che implementa l'interfaccia `GameObject` deve definire il metodo `update()` che permette all'oggetto di modificare il proprio stato ad ogni step. Un ulteriore approfondimento di questa interfaccia è `GameObjectWithPhysics` che estendendo `GameObject` e `RigidBody` del package `Physics` definisce le regole per gli oggetti di gioco che rispettano le leggi della fisica.

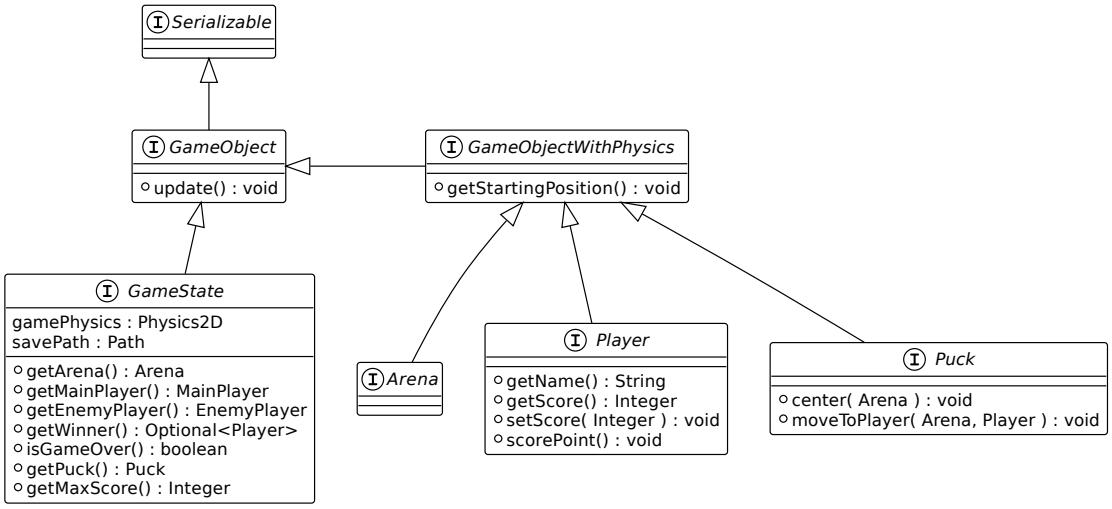

Figura 2.3: UML degli elementi di gioco

Problema: Serializzazione degli elementi di gioco
Salvare la partita è un elemento chiave della stragrande maggioranza dei videogiochi in mercato, quindi va implementata.

Soluzione: Ho pensato di usare la funzionalità di Java che permette di salvare le istanze degli oggetti su file chiamata Serializzazione.

L'implementazione inizialmente consisteva semplicemente nell'includere l'interfaccia `Serializable` tra le interfacce estese da `GameObject` per segnalare che tutti gli elementi di gioco debbano essere serializzabili.
Definite le prime implementazioni di alcuni `GameObject` funzionava tutto, il vero problema è sorto quando abbiamo collegato le parti sviluppate separatamente.
La parte sviluppata da *Pablo Sebastian Vargas Grateron* che riguarda la fisica andava in conflitto con la serializzazione in quanto le classi fornite non erano serializzabili, facendo decadere le regole della serializzazione. La soluzione a quest'ultimo problema è stata un po' complessa ed ha generato effetti collaterali che purtroppo a causa del monte ore non è stato possibile trovare una soluzione perfetta; essa consiste nel modificare il metodo predefinito per la "de-serializzazione" degli oggetti per ripristinare lo stato di tutti quelli statici o `transient`.
Gli effetti collaterali di questa soluzione hanno reso l'implementazione della classe `GameState` problematica, in quanto hanno limitato la creazione delle partite a una alla volta; non è veramente un problema per l'implementazione del nostro software ma previene eventuali riusi all'infuri di questo contesto. Mi sarebbe piaciuto riscrivere questo meccanismo, ma avrebbe richiesto troppo tempo.

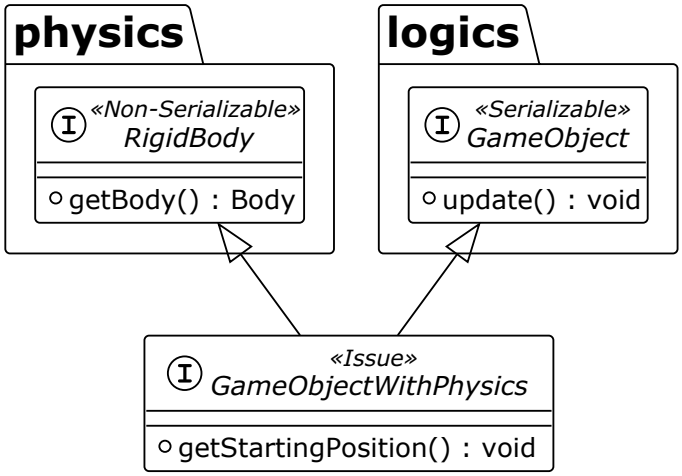

Figura 2.4: UML che descrive in modo semplificato il problema della serializzazione

Problema: Gestione degli input dell'utente per il movimento del proprio piattino

Soluzione: Seguendo l'architettura usata, ho deciso di implementare il movimento del giocatore in una nuova classe che estende l'astrazione di `AbstractPlayer` scritta da *Edoardo La Greca* aggiungendo metodi appositi per permettere di inviare input al piattino; questi metodi verranno chiamati dalla *View* passando attraverso `GameState` (*Presenter*)

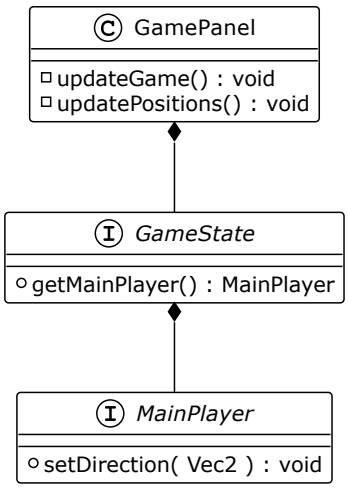

Figura 2.5: UML interazione *View-Model* per il controllo del piattino dell'utente

Classi di utility

Problema: Fornire metodi per cambiare colore a diverse risorse usate dall'applicazione

Soluzione: Ho creato la classe `ImageModifier` che oltre a colorare le immagini può anche ridimensionarle con due diversi algoritmi.
Questa classe è tornata molto utile durante lo sviluppo dell'interfaccia grafica e la creazione di diversi temi di colori applicabili a tutta l'interfaccia.

Problema: Metodo per convertire le coordinate della View (pixel su schermo) alle coordinate del Model (metri)

Soluzione: La classe `UnitConverter` per essere istanziata richiede due sistemi di coordinate, poi è in grado di convertire le coordinate di un punto da un sistema all'altro e viceversa.

Problema: Metodi pratici per serializzare diversi oggetti in file

Soluzione: Semplice classe con due metodi statici che fanno uso di generici per necessità che permettono di serializzare e de-serializzare oggetti in file. La classe risulta comoda perché gestisce la creazione di cartelle e file autonomamente.

2.2.2 Francesca Lanzi

Intelligenza artificiale nemica

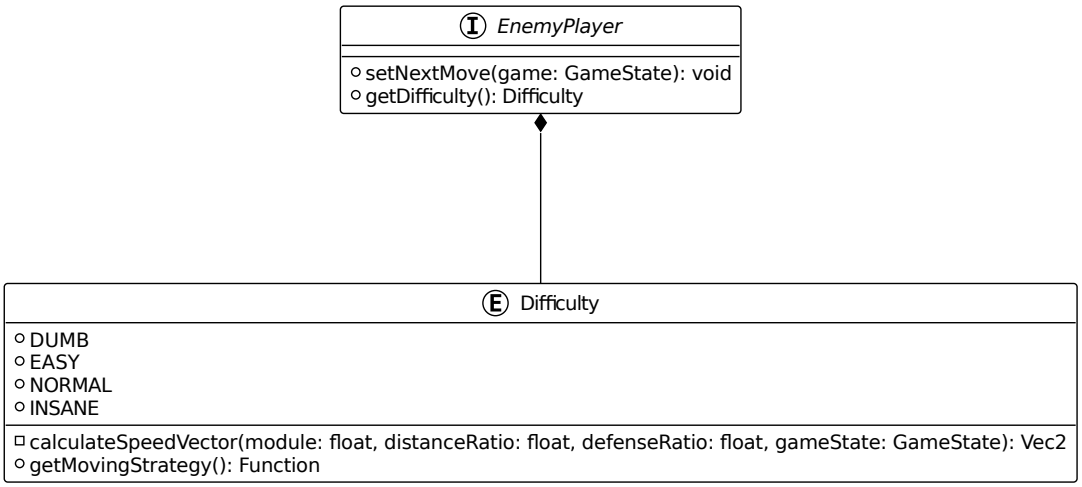

Figura 2.6: Rappresentazione UML della classe enum Difficulty

Problema Creazione delle difficoltà dell'intelligenza nemica

Soluzione Per le difficoltà dell'intelligenza artificiale ho scritto la classe enum `Difficulty` in Figura 2.6 ottenendo quindi diverse modalità di gioco per l'intelligenza nemica. Questa classe viene puoi usata nell'interfaccia `EnemyPlayer` scritta dal collega *Edoardo La Greca* per configurare l'intelligenza di gioco.

Obiettivi di gioco

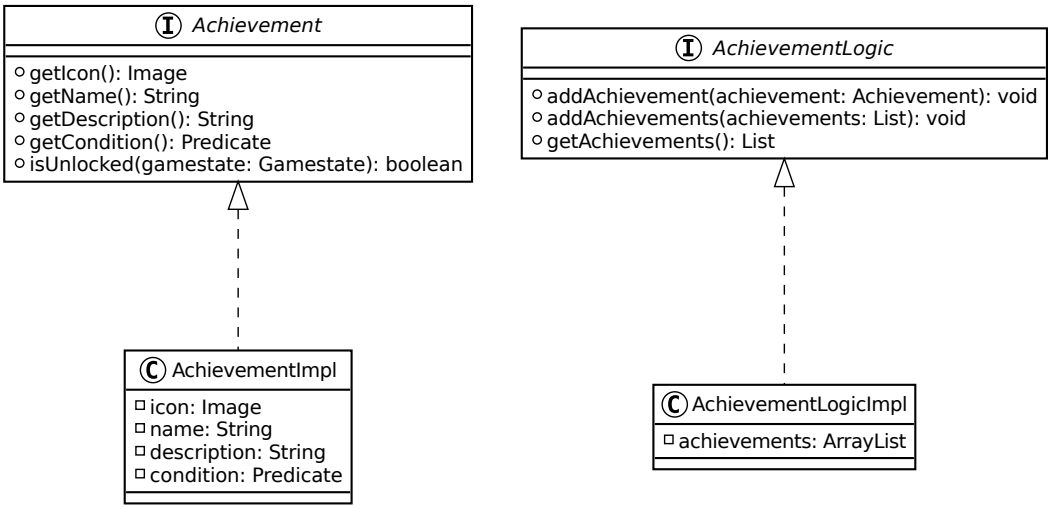

Figura 2.7: Rappresentazione UML della creazione degli obiettivi di gioco

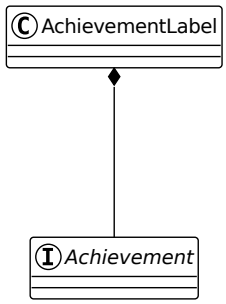

Figura 2.8: Rappresentazione UML dell'implementazione degli obiettivi nell'interfaccia grafica

Problema La creazione degli obiettivi di gioco in modo da non ripetere il codice.

Soluzione Come si vede in fig. 2.7 la classe `Achievement` ha il compito di definire un singolo obiettivo di gioco, con tutte le informazioni necessarie a questo, mentre `AchievementLogic` crea una lista che contiene tutti gli obiettivi. La classe `Achievement` viene quindi usata nell'interfaccia utente come si vede in Figura 2.8 per mostrare gli obiettivi di gioco a fine partita.

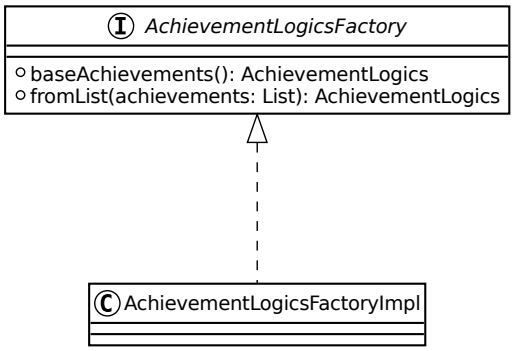

Figura 2.9: Rappresentazione UML della gestione degli obiettivi di gioco

Problema La gestione degli obiettivi di gioco doveva essere gestita in modo e semplice, non ostacolando l'aggiunta di obiettivi futuri.

Soluzione Per risolvere tale problema ho usato, per quanto in maniera semplificata, il *pattern creazionale Factory* come in Figura 2.9.

2.2.3 Edoardo La Greca

Conversione di `AchievementsPanel` in `AchievementsScrollPane`

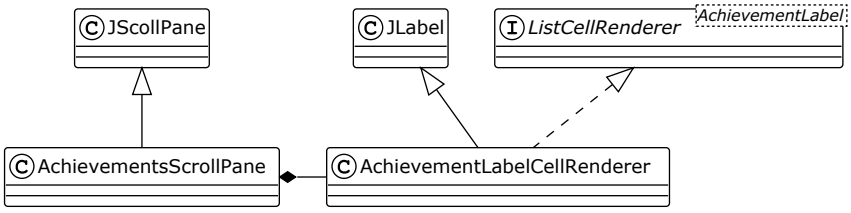

Figura 2.10: Schema UML di `AchievementsScrollPane` e della sua classe innestata: `AchievementLabelCellRenderer`.

Problema `AchievementsPanel` era una classe adibita all'esposizione degli obiettivi (sia sbloccati che non) nella schermata finale di gioco, con una immagine a fianco di ognuno.

Essa estendeva `JPanel` ma, durante lo sviluppo del gioco, tale classe è risultata inadatta a causa di successivi cambiamenti nell'interfaccia grafica che avevano prodotto svariati problemi. Eppure rimaneva necessario mostrare gli obiettivi a fine partita.

Soluzione Qui entra in gioco `AchievementsScrollPane`, ovvero un riadattamento di `AchievementsPanel` la cui super-classe non è più `JPanel` ma `JScrollPane`.

Questo cambiamento non è stato sufficiente, sebbene necessario, per riprodurre il risultato desiderato in quanto, normalmente, in un `JScrollPane` non è possibile mostrare le immagini. Quindi, la soluzione completa presenta la creazione di una classe innestata in `AchievementsScrollPane`, ovvero `AchievementLabelCellRenderer`, la quale estende `ListCellRenderer` e definisce il modo in cui gli obiettivi vengono rappresentati, cioè con un'immagine a fianco.

La classe `AbstractPlayer`

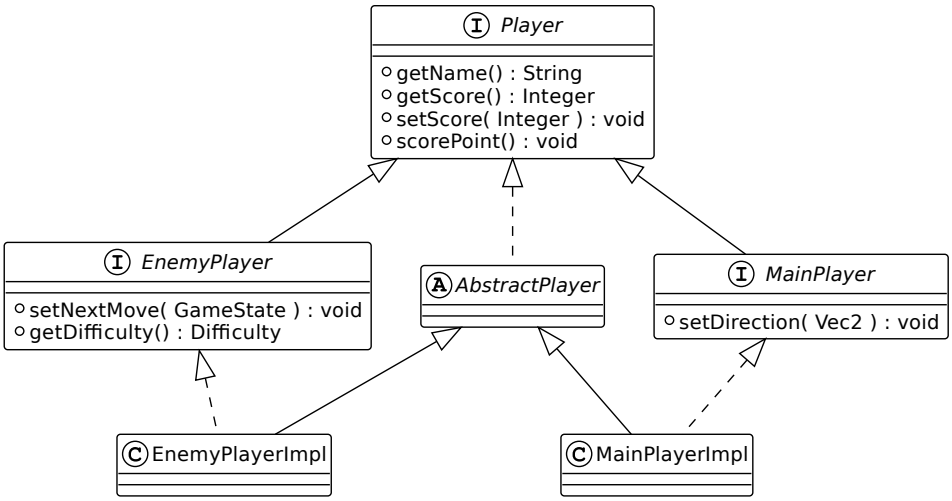

Figura 2.11: Schema UML di `AbstractPlayer` e tutte le classi/interfacce ad esso correlate. Come si può notare, `AbstractPlayer` riduce enormemente il codice ripetuto implementando `Player`.

Problema L'interfaccia `Player` viene ereditata in due sotto-interfacce, chiamate `MainPlayer` ed `EnemyPlayer`. Le classi che implementano queste ultime due interfacce, ovvero `MainPlayerImpl` ed `EnemyPlayerImpl` hanno gran parte del codice in comune siccome, sebbene vengano usate per scopi differenti, sono molto simili.

Soluzione Per ridurre il problema delle ripetizioni di codice è stata creata la classe astratta `AbstractPlayer`, la quale definisce le implementazioni comuni alle due classi sopra citate.

In questo modo, se verranno create altre classi che implementeranno delle sotto-interfacce di `Player`, basterà che estendano `AbstractPlayer` per poi modificare i metodi che differiscono.

La classe `GameStateBuilder`

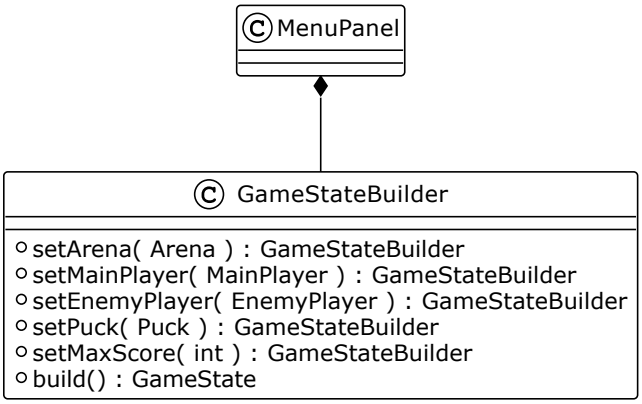

Figura 2.12: Schema UML di `GameStateBuilder`.

Problema `GameStateImpl` è la classe centrale per quanto riguarda la logica, quindi è naturalmente la più complessa di quel package. Questa sua complessità la porta ad avere molti costruttori, siccome ha molti campi e l'utilizzatore potrebbe voler impostare il valore *solo* di alcuni alla creazione mentre potrebbe volerne lasciare altri inizializzati ai loro valori di default.

Soluzione A fronte di questo problema, la soluzione più naturale è la creazione di una classe separata, la quale sfrutta il design pattern *Builder* per la costruzione di un oggetto di `GameStateImpl` mentre quest'ultima è lasciata con un unico costruttore, comprensivo di tutti i campi.
L'utilizzo è ampiamente semplificato rispetto a prima siccome ora l'utilizzatore di `GameStateImpl` dovrà soltanto chiamare il costruttore di `GameStateBuilder`, poi richiamare successivamente gli appositi metodi per l'impostazione dei valori che si vogliono personalizzare ed infine chiamare il metodo `build`, il quale restituisce l'oggetto desiderato.

2.2.4 Pablo Sebastian Vargas Grateron

Per sviluppare il modulo della fisica nel progetto, ho utilizzato una libreria di terze parti chiamata `JBox2D`.

JBox2D

`JBox2D` è una traduzione in linguaggio Java della conosciuta libreria C++ chiamata `Box2D`, la quale permette di ricreare i comportamenti di corpi rigidi in un ambiente virtuale realistico.

Creazione del mondo fisico

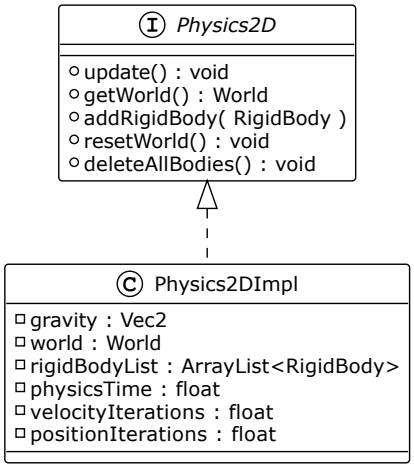

Figura 2.13: UML interfaccia e implementazione di `Physics2D`.

Problema Per gestire la fisica in `JBox2D` è necessario creare il "mondo" dove il motore fisico crea e gestisce i corpi, calcolando la velocità, posizione e collisioni di tutti i corpi nel tempo, ma contemporaneamente la fisica deve essere gestita in forma semplice e intuitiva, implementando solo i metodi necessari per il programma.

Soluzione Per semplificare la gestione della classe centrale della fisica, viene utilizzato il *pattern strutturale Facade*, implementando dei metodi che permettono di interagire con il motore fisico senza considerare tutte le costanti e variabili interne. L'interfaccia contiene dei metodi che permettono l'aggiornamento del mondo per ogni frame eseguito nel gioco, aggiungere corpi, cancellare quelli presenti nel mondo e reimpostarli nella loro posizione iniziale. L'implementazione dell'interfaccia dichiara dentro di sé le variabili necessarie e consigliate[1] per generare il mondo.

[1]Gli sviluppatori di `JBox2D` consigliano delle variabili standard per evitare errori, instabilità del motore fisico e sovraccarico del processore.

Generazione dei corpi nel motore fisico

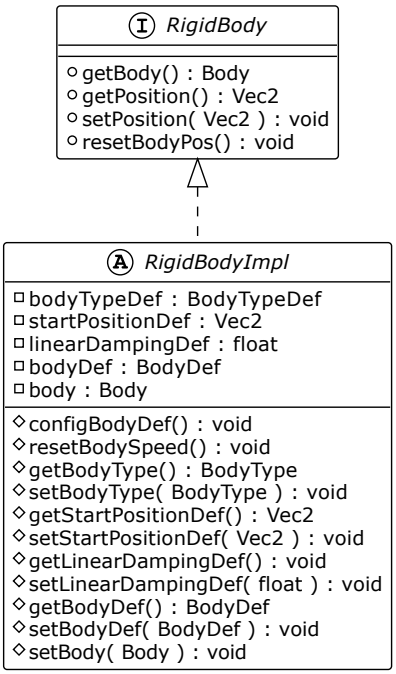

Figura 2.14: UML interfaccia e implementazione di `RigidBody`.

Problema Prima di generare un corpo nel motore fisico è necessario fornire una descrizione del corpo (`BodyDef`), la quale potrebbe risultare molto ripetitiva considerando che ci sono molti parametri richiesti da definire per ogni corpo generato. Il corpo deve anche fornire metodi che impostano e ottengono i dati sulla posizione del corpo.

Soluzione Seguendo il *pattern strutturale Facade*, è stata progettata l'interfaccia `RigidBody` che contiene dei metodi per la gestione delle coordinate del corpo nel mondo fisico. L'implementazione è una classe astratta che contiene tutte le variabili necessarie richieste dal motore fisico per generare un corpo e implementa dei *getter e setter* per modificare queste variabili. `RigidBodyImpl` è stato progettato per essere utilizzato come base per creare corpi personalizzati senza la necessita di ripetere tutte le variabili e codice in ogni classe (*Principio Don't Repeat Yourself*).

Definizione delle interfacce per ogni corpo del gioco

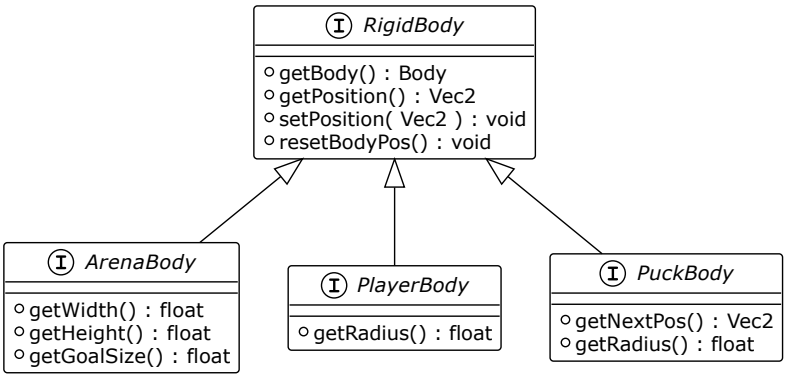

Figura 2.15: UML albero delle interfacce che estendono `RigidBody`.

Problema Il gioco è composto da 3 elementi diversi (arena, giocatore e dischetto) ed ognuno ha caratteristiche intrinseche che non possono essere definite per tutti i corpi fisici.

Soluzione Utilizzando l'interfaccia `RigidBody` che definisce i metodi per tutti i corpi del motore fisico, si progettano altre interfacce che contengono i metodi richiesti per ogni elemento del gioco. Tra queste interfacce c'è `ArenaBody` per l'arena, `PlayerBody` per il giocatore e `PuckBody` per il dischetto.

Cenni per le prossime implementazioni dei corpi nel gioco

L'implementazione dei corpi nel gioco necessitano della classe `Body` fornita da `JBox2D`, la quale contiene e registra i dati del corpo dentro il motore fisico in tempo reale. Per generare un `Body`, è necessaria la classe `World` contenuta dentro `Physics2DImpl`; infatti, ogni implementazione di ogni interfaccia che estende `RigidBody` (le interfacce contenute nella fig. 2.15) utilizza `Physics2D`.

Implementazione del corpo dell'arena

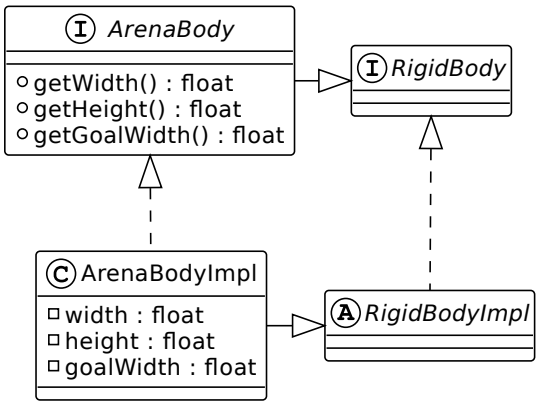

Figura 2.16: UML interfaccia e implementazione di `ArenaBody`.

Problema Per generare l'arena è necessario implementare una classe che con pochi parametri essenziali, riesce a generare tutto il corpo dell'arena utilizzando i metodi della libreria `JBox2D`.

Soluzione Seguendo il *pattern strutturale Facade*, è stata progettata un implementazione che richiede solo i parametri essenziali per la generazione dell'arena e contiene i metodi necessari per il funzionamento del gioco, lasciando tutta l'elaborazione del corpo dentro la classe. L'implementazione estende `RigidBodyImpl` e richiede anche la classe centrale della fisica (`Physics2D`), con i quali genera il corpo e tutti i muri che compongono l'arena. Nell'arena esistono due tipi di muri:

- **Muri normali**: i muri che delimitano l'arena, collidono con tutti gli oggetti.
- **Muri con mascheratura di bit**: questi muri collidono esclusivamente con `PlayerBody`.

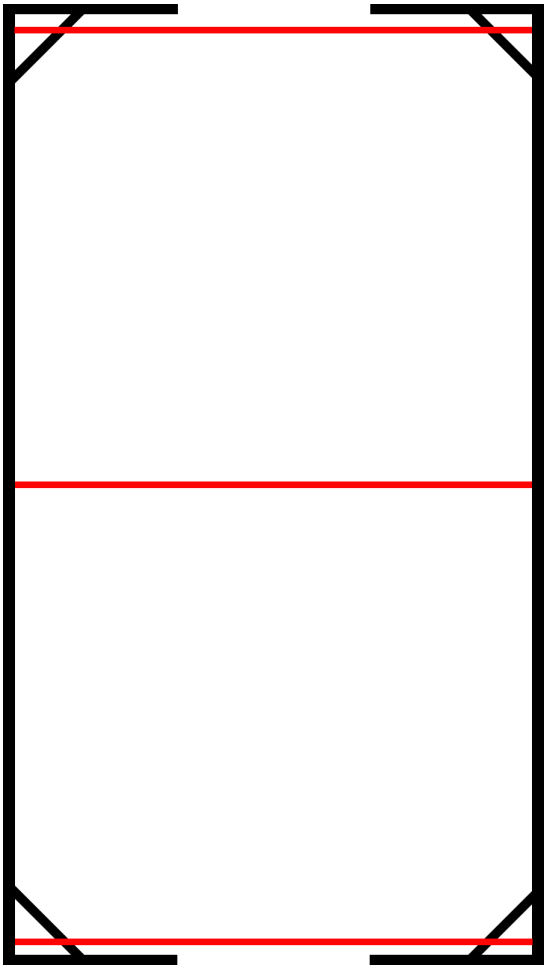

Figura 2.17: Disegno dell'arena creata da `ArenaBodyImpl` con rapporto delle dimensioni di 9:16. I muri neri rappresentano i *muri normali*, mentre i muri rossi rappresentano i *muri con mascheratura di bit*.

Implementazione del corpo del giocatore

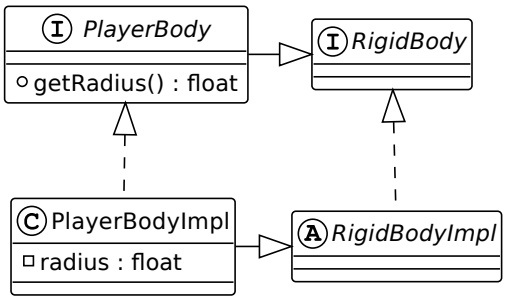

Figura 2.18: UML interfaccia e implementazione di `PlayerBody`.

Problema Per generare il corpo del giocatore, è necessario creare una classe che tramite il raggio, riesce a generare tutto il corpo del giocatore.

Soluzione Seguendo il *pattern strutturale Facade*, l'implementazione di `PlayerBody` genera il corpo e costruisce la forma del giocatore utilizzando il raggio del giocatore, la posizione nella quale il giocatore è generato e la classe centrale della fisica (`Physics2D`). La forma del giocatore viene generata con un parametro definito in `JBox2D` chiamato *mascheratura di bit* che permette di impostare filtri per le collisioni con altri oggetti, infatti in questo caso la mascheratura viene utilizzata per fare collidere il corpo con i *muri con mascheratura di bit* dell'arena.

Implementazione del corpo del dischetto

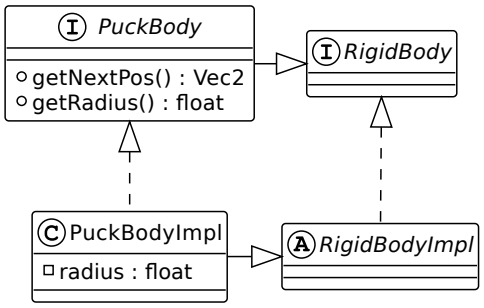

Figura 2.19: UML interfaccia e implementazione di `PuckBody`.

Problema Per generare il corpo del dischetto, è necessario creare una classe che tramite il raggio, riesce a generare tutto il corpo del dischetto. Contemporaneamente, il corpo deve calcolare quando richiesto[2], la futura posizione nel tempo.

Soluzione Analogamente all'implementazione del giocatore seguendo il *pattern strutturale Facade*, l'implementazione di `PuckBody` genera il corpo del dischetto utilizzando come parametri di *input* il raggio del dischetto, la posizione nella quale verrà generato il dischetto e la classe centrale della fisica (`Physics2D`). Per processare il calcolo della posizione futura del dischetto si utilizza la velocità istantanea e la posizione attuale fornite in tempo reale del corpo (Classe `Body`).

Implementazione dello spostamento del giocatore

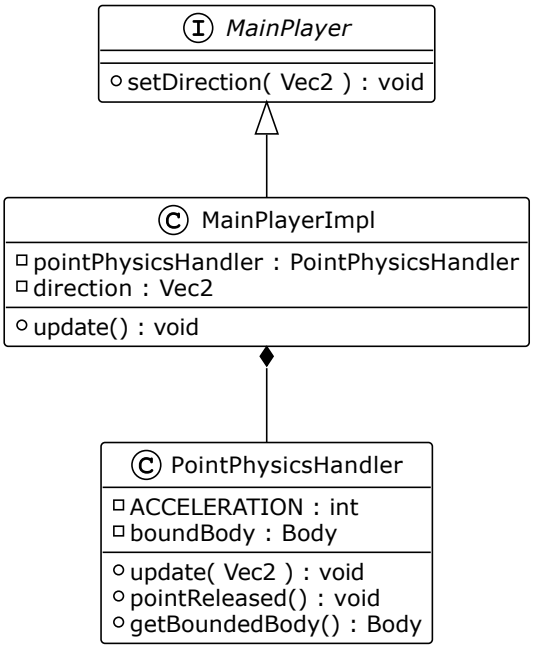

Figura 2.20: UML implementazione della classe `PointPhysicsHandler` contenuta in `MainPlayerImpl`.

Problema Il programma deve comunicare e interagire con il motore fisico per spostare il giocatore verso la posizione indicata.

[2]Il calcolo viene utilizzato per programmare la IA del gioco.

Soluzione Seguendo l'implementazione della fig. 2.5 fatta da *Emanuele Borghini*, è stata progettata la classe `PointPhysicsHandler`. Il funzionamento della classe consiste nel generare una forza agente sul corpo del giocatore verso le coordinate ricevute in `input`. Per bilanciare il spostamento del giocatore e la velocità del mouse, vengono utilizzati dei calcoli vettoriali e funzioni trigonometriche con `java.lang.Math`.

Finestra delle impostazioni del gioco `SettingsPanel` e la classe `Settings`

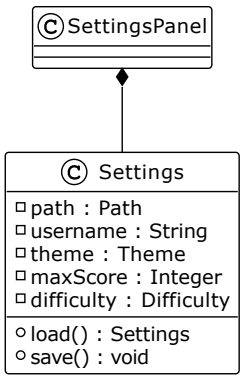

Figura 2.21: UML classe `SettingsPanel` che contiene `Settings`.

Problema Il gioco deve permettere all'utente di modificare le opzioni e memorizzarle. Analogamente, il programma può estrarre i dati memorizzati per impostare la partita.

Soluzione L'interfaccia grafica permette all'utente di visualizzare i dati memorizzati e reimpostare i dati ed è sviluppata estendendo la classe astratta `AbstractGridBadLayoutJPanel` come mostrato nella fig. 2.8.
Per memorizzare i dati viene progettata la classe `Settings`, la quale contiene i parametri che l'utente può modificare e contiene i metodi che sono utilizzati per memorizzare o caricare i parametri. Il processo di caricamento e memorizzazione dei dati viene eseguito con la libreria esterna `Jackson`, che permette di gestire i file `json`. Infatti, i dati del gioco sono memorizzati o caricati (se il file esiste) sul file `config.json`.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Anche se le strutture software che compongono Air Hockey sono semplici separatamente, una volta unite possono dar vita a una logica complessa, per questo il gruppo ha deciso di creare qualche test che permette di verificare il buon funzionamento del software parallelamente al suo sviluppo. Oltre alla logica di gioco, i test sono stati scritti per verificare il corretto funzionamento di alcune classi di utility, a volte sperimentali. Per il testing automatizzato sono state usate le librerie `JUnit4.13.2` e `JUnit5`

3.1.1 Emanuele Borghini

- `GameTest`:
  In questo test viene controllato se in una partita accadono problemi di compenetrazione con l'arena e gli elementi di gioco, in quanto è capitato durante del test manuale che gli oggetti mobili finissero fuori dall'arena, ovvero in una zona non accessibile all'utente. Il lungo tempo di esecuzione del test è normale in quanto vengono effettuati calcoli equivalenti a circa 166 secondi di gioco. Inoltre viene controllato il corretto funzionamento del metodo `equals` della classe `GameState.java`

- `ImageModifierTest`

- `ObjectSerializerTest`

- **Test manuale**:
  Durante dei test manuali mi è capitato di incappare in un bug nella GUI su sistema *nix che causava comportamenti non prevedibili a causa di un `NullPointerException`, per risolvere il problema è risultato

necessario l'uso di una macchina virtuale e tool di debug che purtroppo non si sono rivelati utili in quanto il programma non seguiva l'ordine logico del codice. Ultimamente si è deciso di cambiare la classe padre di `AchievementLabel.java`, risolvendo temporaneamente il problema. A causa del limite di tempo previsto e la tardiva scoperta del baco non è stato possibile trovare una soluzione migliore.

3.1.2 Francesca Lanzi

- `AchievementTest`:
  Nel test viene controllato che gli obiettivi di gioco (sia creati appositamente per eseguire il test, quelli presenti dalla lista fatta per il gioco) vengono ottenuti o meno a seconda delle condizioni di gioco che si verificano.

3.1.3 Edoardo La Greca

- `SerializationTest`:
  In questo test viene verificato che i vari oggetti del gioco possano essere serializzati, e successivamente de-serializzati, in modo corretto. La differenza rispetto a `ObjectSerializerTest` è che quest'ultima classe verifica la serializzazione in modo generale mentre `SerializationTest` esegue dei controlli mirati sugli oggetti che verranno effettivamente serializzati. Ciò è utile siccome non tutto in Java può essere serializzato, quindi è necessario testare anche il corretto funzionamento dell'apposita classe negli oggetti con cui verrà usata.

3.1.4 Pablo Sebastian Vargas Grateron

- `PhysicsTest`:
  Questo test genera un mondo fisico e 2 corpi preimpostati dal package `Physics2D` (`PlayerBody` e `PuckBody`), nei quali si controlla la corretta funzione della libreria *JBox2D* e i metodi implementati nel codice. Nel test, vengono controllate le funzioni base della libreria e anche dei metodi implementati nelle classi, come il movimento, applicazione di forze e controlli per la generazione dei corpi.

- **Test manuale**:
  Le collisioni sono state controllate attraverso uno strumento di *JBox2D* per visualizzare e interagire con il motore fisico (`JBox2D-Testbed`). Dentro questo strumento sono stati utilizzati copie di tutti gli oggetti

del gioco (`PlayerBody, PuckBody e ArenaBody`) e per provare le collisioni, gli oggetti sono stati spinti tra di loro e poi sono stati visualizzati i dati erogati dal motore fisico.

3.2 Metodologia di lavoro

La suddivisione del lavoro è stata pensata per dare le giuste porzioni di lavoro in base alle abilità dei componenti del gruppo, mantenendo una soglia minima per dare la possibilità di lavorare a tutti.
Anche se da un punto di vista esterno del progetto può sembrare che le parti non siano eque, all'interno del gruppo ci siamo assicurati che ogni componente fosse al corrente di tutta la struttura del software, per garantire che ognuno abbia imparato le nozioni generali necessarie per il completamento del progetto.
Come DVCS abbiamo usato Git, anche se, osservando l'andamento di lavoro dei nostri colleghi, abbiamo optato per un workflow più semplice, dato il piccolo numero di componenti del gruppo e la complessità del codice relativamente non troppo elevata è stato deciso di lavorare su un unico *branch* e non usare *fork* o *pull request*.
Per rattoppare la mancanza delle funzionalità sopra elencate abbiamo fatto uso di *rebase*, *squashing* e un *tag* per la versione finale.
Questo workflow accoppiato alla ampia comunicazione tra i membri del gruppo ci ha permesso di velocizzare la produzione iniziale del software.

3.2.1 Emanuele Borghini

- `Interfacce` *`GameObject`* e *`GameObjectWithPhysics`*:
  Astrazione principale di tutti gli elementi di gioco
- Interfacce di logica e le loro implementazioni:
  `GameState`, `Arena`, `MainPlayer` e `Puck`.
- Sistema di movimento del giocatore (Interazione View e Model)
- *`Serializzazione`* delle classi di logica:
  Questa funzionalità è stata forse la più complicata da implementare correttamente. A causa della struttura delle dipendenze create dalla parte di fisica è stato necessario riscrivere la procedura di serializzazione e de-serializzazione di numerose classi e ha causato degli effetti collaterali, non gravi, ma non risolti a causa dei limiti di tempo.
- Entry point del programma e di GUI (*Main.java* e *GUI.java*)

- **Astrazione** della parte di GUI
  in una classe *AbstractGridBagLayoutJPanel* comune a molte parti di interfaccia
- La maggior parte dei componenti grafici:
  tra cui: `MenuPanel`, `CreditsPanel`, `GamePanel`, `GameOverPanel`, `IconButton`
- Ed infine classi di utility, ovvero:
  `ImageModifier`, `JComponentLoader`, `ObjectSerializer`, `UnitConverter`

3.2.2 Francesca Lanzi

- **Interfacce Achievement** e `AchievementLogics`
- **Interfaccia AchievementLogicsFactory** e la sua implementazione
- **Difficulty** per le difficoltà dell'intelligenza nemica

3.2.3 Edoardo La Greca

- Conversione del precedente `AchievementsPanel` nell'attuale `AchievementsScrollPane`.
- Parti dell'interfaccia grafica come `ExceptionPanel` e `PausePanel`.
- Diverse classi/interfacce della logica di gioco, quali: `AbstractPlayer`, `EnemyPlayer`, `Player` e `GameStateBuilder`.
- L'utility class `ResourceLoader` per il caricamento delle risorse direttamente dall'archivio JAR.

3.2.4 Pablo Sebastian Vargas Grateron

- Interfacce e implementazione della fisica di gioco come `Physics2D`, `RigidBody` e le sue interfacce estese `PlayerBody`, `PuckBody` e `ArenaBody`.
- L'interfaccia grafica `SettingsPanel` e la classe `Settings` per la memorizzazione e caricamento delle impostazioni.
- Complemento al sistema di movimento del giocatore per il motore fisico con la classe `PointPhysicsHandler`.

3.3 Note di sviluppo

La parte di interfaccia grafica, sviluppata da tutti i membri del gruppo, fa uso di `lambda expressions` per i gestori dei click del mouse, i quali vengono implementati nei `ActionListener`.

3.3.1 Emanuele Borghini

- `Lambda Expressions`
- `Optional`
- `Generics`
- Aspetti avanzati della `Serializzazione`
- `javax.swing.Timer`

3.3.2 Francesca Lanzi

- `Lambda Expressions`
- Algoritmo creato ad hoc per l'intelligenza artificiale
- `Stream`

3.3.3 Edoardo La Greca

- `Lambda Expressions`
- Il build-system `Maven` per gestire le dipendenze e automatizzare le parti di compilazione ed impacchettamento in un archivio `JAR` a sé stante.

3.3.4 Pablo Sebastian Vargas Grateron

- `Lambda Expressions`
- Utilizzo di librerie esterne:
  - `JBox2D` per i calcoli della fisica.
  - `Jackson` per il salvataggio e caricamento di file `json`.

Si è preso spunto dalla `testbed` fornita nella repository di GitHub di `JBox2D` per capire l'utilizzo ottimo della libreria.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Emanuele Borghini

Secondo me il codice che ho scritto è ordinato e funzionale, nulla da ridire se non che l'idea per il progetto purtroppo non offriva molte opportunità di sfruttare i pattern di design offerti dal mondo della programmazione ad oggetti, tanto meno alcune funzionalità caratteristiche del linguaggio Java. Purtroppo il codice finale ha problemi di riusabilità all'infuori della possibilità di eseguire il gioco indipendentemente dell'interfaccia grafica (che può essere totalmente assente). Com'è strutturato il codice ora può essere eseguita solo una partita alla volta per istanza di processo. La serializzazione ha causato troppi difetti per i miei gusti e se in futuro continuerò a lavorare sul gioco farò in modo di sostituire la limitata serializzazione di Java con altre strutture dati come quella fornita da Json. Il mio ruolo nel gruppo era quello di coordinare il lavoro di tutti, in quanto secondo me posseggo più conoscenze della programmazione in generale e su quella ad oggetti, ruolo che è stato molto difficile svolgere, mi sono ritrovato spesso a dover controllare il codice degli altri colleghi a causa di difetti di programmazione o design, inoltre le sezioni da svolgere indipendentemente non sono sempre state chiarite bene fin dall'inizio. Tutto sommato ho migliorato le mie abilità con Git e fatto ulteriore pratica con concetti già conosciuti della programmazione ad oggetti.

4.1.2 Francesca Lanzi

Questo progetto mi ha dato la possibilità di provare per la prima volta come si lavora in gruppo e come gestire il carico di lavoro tra diverse persone. Sono felice di aver contribuito a progettare un'applicazione che ha richiesto un

grande impegno da parte di tutti i componenti del gruppo e di aver potuto migliorare le mie capacità di programmatrice.
Personalmente ho trovato molto difficile all'inizio gestire il lavoro con altri colleghi, visto che in generale sono abituata a lavorare da sola, ma pazientando e continuando l'esperienza è stato mano a mano più semplice.
Sono soddisfatta che il codice sia venuto molto pulito e senza eccessive ripetizioni, per quanto sarebbe stato meglio utilizzare più pattern di Java. Inoltre questo progetto mi ha aiutato a comprendere meglio il funzionamento di Git (che ci era stato spiegato durate il corso), anche se ogni tanto si sono verificati dei piccoli problemi di *"merge conflict"* (che sono stati prontamente risolti) visto che abbiamo usato un singolo *"branch"*; a parere mio la scelta di usarne solo uno è stata comunque logica, visto che i componenti del gruppo sono solo quattro. Il mio ruolo nel progetto non è stato particolarmente grande: mi sono concentrata sulla parte logica e su quella grafica relativa agli obiettivi di gioco e la parte logica delle difficoltà dell'intelligenza nemica, in cui ho scritto l'algoritmo.

4.1.3 Edoardo La Greca

Punti di forza

- Il codice che ho scritto è semplice e risolve il proprio problema senza troppi giri o "workarounds".
- Il codice che ho scritto è facilmente comprensibile, anche per merito delle strutture di alto livello fornite da Java.
- Il codice che ho scritto è correttamente suddiviso in classi in base all'ambito a cui appartiene.

Punti di debolezza

- A causa dell'abitudine ad una programmazione più di basso livello, a volte viene spontaneamente da considerare, o peggio *usare*, concetti di tale ambito durante la scrittura di codice Java.
- Il codice che ho scritto scritto non sempre usa i pattern di programmazione ad oggetti, anche quando sarebbero utili.

Ruolo all'interno del gruppo

Il progetto svolto mi ha consentito di capire il significato di *lavoro di gruppo*. Grazie a questo progetto ho potuto combinare le conoscenze passate con le

competenze acquisite durante il corso e, confrontandomi con i miei colleghi, sviluppare un progetto degno di essere chiamato tale. Anche se la programmazione ad oggetti non è il mio forte, e nemmeno il mio paradigma preferito, ho cercato di fare del mio meglio per seguire le regole e le buone pratiche di tale paradigma e collaborare con i miei compagni. Non è stato facile coordinarsi con gli altri membri del gruppo: a volte, a causa di opinioni discordanti, sono iniziate discussioni, le quali, però, sono sempre state risolte in modo civile. In generale sono soddisfatto di ciò che è stato prodotto, poteva venire molto peggio.

4.1.4 Pablo Sebastian Vargas Grateron

Questo progetto mi ha permesso di mettermi alla prova come programmatore, sia personalmente che in squadra. L'utilizzo di un solo branch per la programmazione del progetto mi ha permesso di imparare come gestire molti errori su `Git`. Riguardo al progetto, mi ritengo molto soddisfatto dell'implementazione del motore fisico nel progetto. Infatti, dopo tanti ridisegni e re-fattorizzazione del `package physics`, sono riuscito a renderlo indipendente dal gioco. Analogamente, ho potuto capire tutta una libreria di `JBox2D` utilizzando una documentazione scritta in C++ (la documentazione in `Java` non c'era).
Inoltre, dovuta alla mia poca esperienza in grandi progetti e nel linguaggio, ci sono state certe progettazioni e certi elementi che potevano essere implementati meglio e più avanzava il tempo, scoprivo sempre più funzionalità in `Java` che mi sarebbero piaciute integrare.
Per concludere, mi ritengo soddisfatto della mia parte nel progetto, e apprezzo molto la esperienza e abilità per lo sviluppo personale tanto come programmatore, come nel linguaggio `Java`.

Appendice A

Guida utente

Avvio del gioco e menu principale

Per avviare il gioco è sufficiente eseguire l'archivio `jar` che lo contiene. Nell'avvio del gioco compare il menu principale con i seguenti bottoni:

- **New game**: Inizia una nuova partita.
- **Continue**: Riprende una partita precedentemente memorizzata.
- **Settings**: Apre il pannello delle impostazioni del gioco.
- **Credits**: Apre il pannello degli sviluppatori del gioco.
- **Exit**: Esce dal gioco.

Figura A.1: Menu principale.

Le impostazioni

Il pannello delle impostazioni permette all'utente di:

- Modificare il nome dell'utente.
- Modificare il tema del gioco.
- Modificare il punteggio massimo per partita.
- Scegliere la difficoltà del gioco.
- Salvare le impostazioni.
- Ripristinare le impostazioni a quelle predefinite.

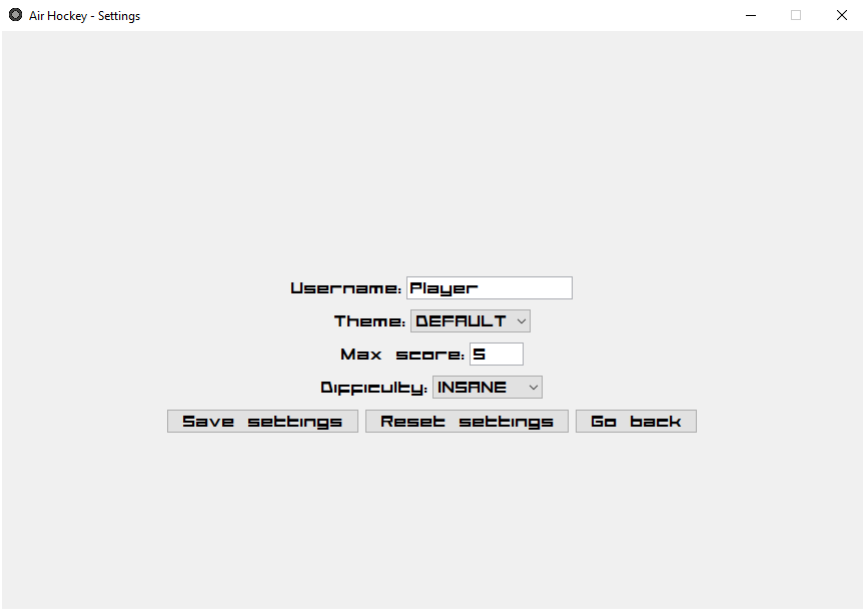

Figura A.2: Pannello impostazioni.

Il gioco

La partita inizia con due giocatori: uno viene controllato dall'utente tramite la pressione ed il trascinamento del *mouse* mentre l'altro viene coordinato dal computer. In mezzo all'arena di gioco è presente un dischetto.
L'obiettivo del gioco è di far entrare il dischetto nella porta nemica. Vince il primo giocatore che raggiunge il punteggio massimo impostato nel gioco.
L'utente durante la partita può vedere il punteggio dei giocatori e può pausare la partita. Il menu di pausa permette all'utente di uscire dal gioco, con o senza memorizzazione della partita, e di riprendere la partita.

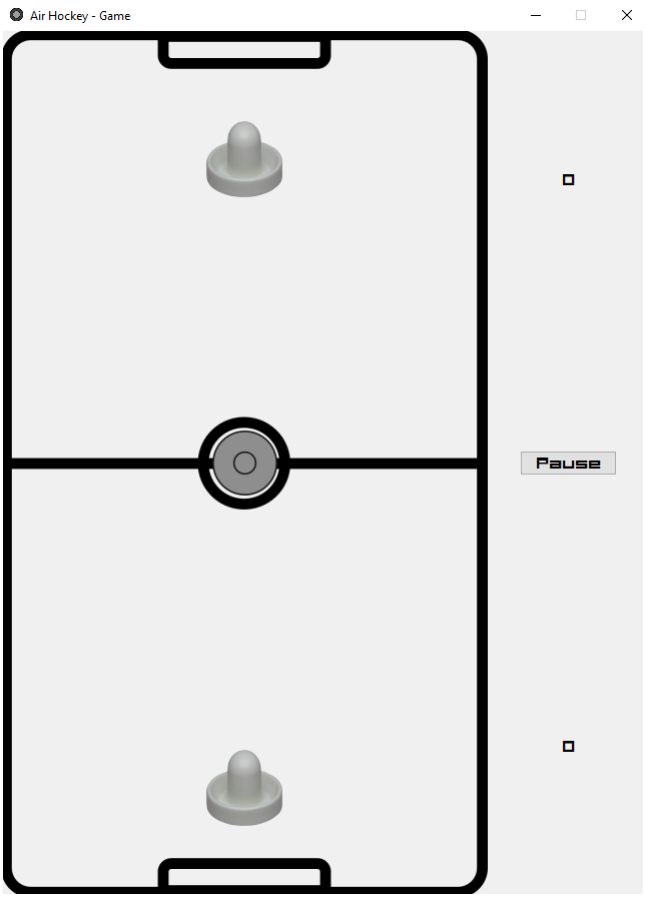

Figura A.3: Partita appena avviata.

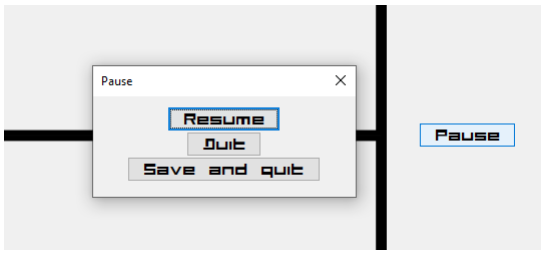

Figura A.4: Pannello di pausa.

Obiettivi del gioco

Alla fine di ogni partita, l'utente potrà osservare gli obiettivi di gioco, i quali vengono mostrati con un *trofeo dorato* se sono stati sbloccati, altrimenti

trofeo grigio. Infine, l'utente potrà scegliere se tornare al menu principale o uscire dal gioco.

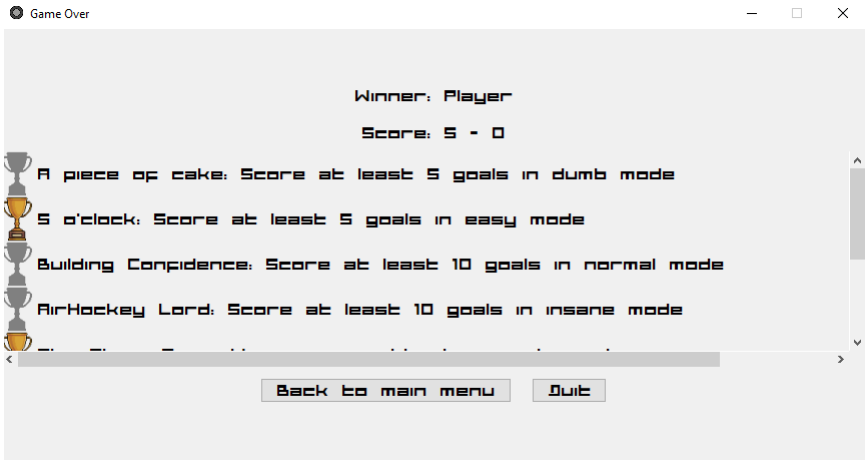

Figura A.5: Schermata finale di gioco contenente il vincitore, il punteggio e gli obiettivi.

Appendice B

Esercitazioni di laboratorio

B.0.1 Emanuele Borghini

- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=89272#p136889`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p138581`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=91128#p140021`

B.0.2 Francesca Lanzi

- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=90125#p138586`