Relazione per "Programmazione ad Oggetti" Stubborn

Mario Ciccioni, Andrea Bianchi

6 novembre 2022

Indice

Capitolo 1

Analisi

Il software, un videogioco 2D bird-eye view survival, proporra' al videogiocatore un'esperienza di un classico videogame survival dove lo scopo e' quello di resistere il maggior tempo possibile cercando anche di accumulare punti durante la partita. Il videogiocatore avra' la possibilita' di muoversi nel mondo di gioco con il suo personaggio, sopravvivere ai nemici che incontrera' nella mappa, raccogliere oggetti e recuperare vita. Per bird-eye view si intende una visuale dall'alto del mondo di gioco, similare a quella di titoli già esistenti come i vecchi giochi di Legend Of Zelda per NES.

1.1 Requisiti

Requisiti funzionali

- Una volta avviato il software verra' presentato a schermo un menu' di gioco dove il videogiocatore potra' scegliere se iniziare una nuova partita, visualizzare la classifica dei punteggi o uscire dall'applicazione.
- La partita termina solo quando il personaggio perdera' tutte le vite a sua disposizione.

Requisiti non funzionali

- Stubborn dovra' garantire una buona gestione delle risorse, mantenere una buona prestazione per evitare di rovinare l'esperienza di gioco e una grafica che permetta il chiaro riconoscimento di oggetti di gioco e nemici sparsi per la mappa.

- Tutte le informazioni relative alla partita in corso saranno mostrate a schermo in modo intuitivo, senza essere troppo invasive ma allo stesso tempo facilmente controllabili anche durante la partita.

1.2 Analisi e modello del dominio

Il videogioco denominato Stubborn proporra' al giocatore delle semplici sfide, alcune di esse saranno implementate mentre altre potranno essere aggiunte in futuro. In particolare il giocatore avra' la possibilita' di raccogliere oggetti collezionabili che si trovano sulla mappa fin dall'inizio della partita ed al tempo stesso non deve prendere danno dai nemici. Il player ha a disposizione 3 vite, se verra' colpito da un nemico ne perdera' una. La fine del gioco avverra' solo quando il player finira' le vite a sua disposizione. Riguardo ai nemici, essi verranno spawnati nella mappa di gioco con una logica random, a inizio partita, mentre per quanto riguarda la loro Ai puo' essere random oppure focalizzata sull'inseguimento del player. Alcune funzionalita' come diverse implementazioni di Ai nemici, attacco player ¿ nemici, gameover ecc. verranno implementate in futuro e quindi il software sara' progettato in modo da appliare lo stesso in un futuro in modo semplice e veloce. Di seguito viene mostrato lo schema UML dell'anali del problema contenenete le entita' principali che costutuiscono il problema Figura 1.1.

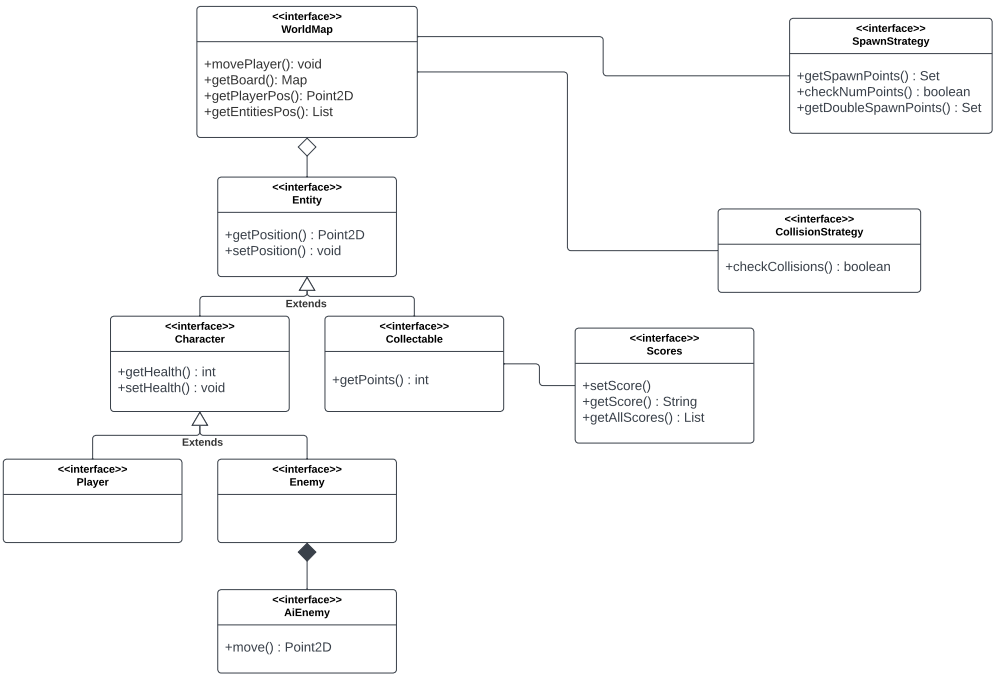

Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

Premessa: utilizzeremo il termine Controller al posto di Presenter (come si chiama effettivamente nel pattern MVP) perche' piu' esplicativo e vicino alla letteratura.

Il pattern architetturale scelto per la realizzazione del software e' MVP (Model View Presenter). In particolare abbiamo adottato la variante MVP Passive View, ossia la view e' un componente completamente passivo del sistema e non svolge nessuna operazione ma viene solo aggiornata dai Controller. L'entry point del model e' WorldMap, essa e' la main class del software che contiene la logica generale del videogioco. Per quanto riguarda i Controller abbiamo deciso di adottare una logica di divisione dello stesso. Per ogni macro compito del software abbiamo implementato un suo specifico Controller in grado di comunicare con la view e se necessario comunicare con/modificare lo stato del model. I principali vantagggi dell'adottare questa architettura sono i seguenti:

- Fornire al software una maggiore modularita', offrendo la possibilita' di rendere indipendenti gli elementi visivi, di model e di controller.
- E' possibile cambiare libreria grafica senza che essa impatti sul Controller, ne tantomeno sul Model (la logica del sistema non risentira' in alcun modo di questi cambiamenti). In quando e' possibile incapsulare la logica di view rendendo possibile l'integrazione di una nuova libreria grafica senza interferire con l'architettura del software.
- Offre una semplice gestione tri i vari Controller, permettendo una modularita' nella gestione degli stessi.

- Con l'aggiunta di nuove funzionalita', essere risultano molto piu' facili da testare in quanto la maggior parte del flusso di esecuzione e' gestito dai Controller.

Di seguito viene mostrato lo schema UML generale di inetrazione tra Controller e Model Figura 2.1.

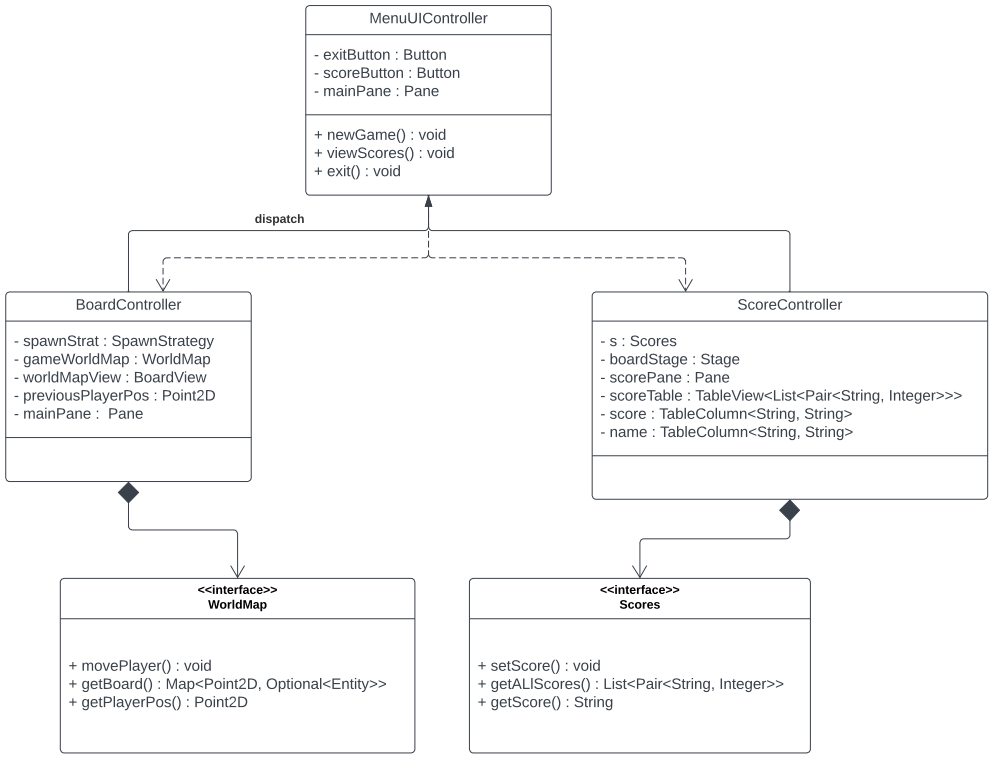

Figura 2.1: schema UML generale di interazione tra Controller e Model

2.2 Design dettagliato

ANDREA BIANCHI

Generazione/Posizionamento delle entità all'interno della mappa di gioco

- **Problema:** La corretta generazione del posizionamento del protagonista, dei nemici e degli oggetti raccoglibili (entità), che saranno collocati all'interno della mappa di gioco.

- **Soluzione:** Il sistema utilizza Pattern Strategy, rappresentato dall' interfaccia `SpawnStrategy` e implementata in questo caso dalla classe `RandomSpawnStrategy`: questo approccio ci permette di disaccoppiare l'effettiva mappa di gioco dalle diverse possibili implementazioni e variazioni su come verranno generati i punti dove saranno collocate le entità di gioco; si offre così una grande estendibilità utile ad ulteriori implementazioni e variazioni anche più complesse, permettendo la creazioni di sottoclassi ed eliminando complessità, dato che ogni comportamento viene implementato dalla propria classe (evitiamo l'uso di statement condizionali).

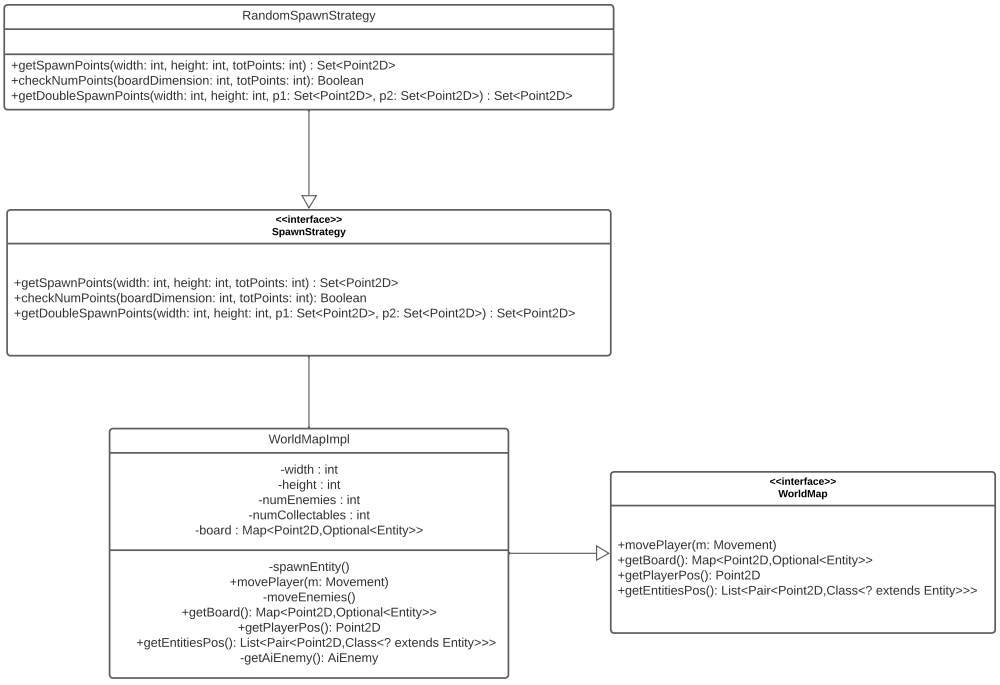

Figura 2.2: Schema UML a rappresentare la soluzione per lo spawn delle entità facendo uso di pattern Strategy con SpawnStrategy e RandomSpawnStrategy

Interazioni tra le varie entità e con i confini della mappa di gioco.

- **Problema:** Entità in grado di muoversi all'interno della mappa, quali il giocatore e il nemico, rischiano di riuscire a superare i confini della mappa o di raggiungere entrambi una stessa posizione, questo rende necessario lo sviluppo di un sistema di collisioni.

- **Soluzione:** Il sistema si avvale nuovamente del Pattern Strategy, tramite l'interfaccia `CollisionStrategy` e della sua classe implementativa `CollisionImpl`: questo sistema offre una forma di riuso di codice per gestire eventi di collisione comuni a tutte le entità, come nel caso delle interazioni di esse con i confini della mappa, permettendo anche l implementazione di estensioni personalizzate per alcune tipologie di entità, come ad esempio il fatto che due nemici o un nemico e il giocatore non potranno mai andare ad occupare la stessa cella, o che non vi sia alcun tipo di collisione tra Player e gli oggetti raccoglibili (il protagonista può andare nella cella dove si trova il collezionabile e "raccoglierlo").

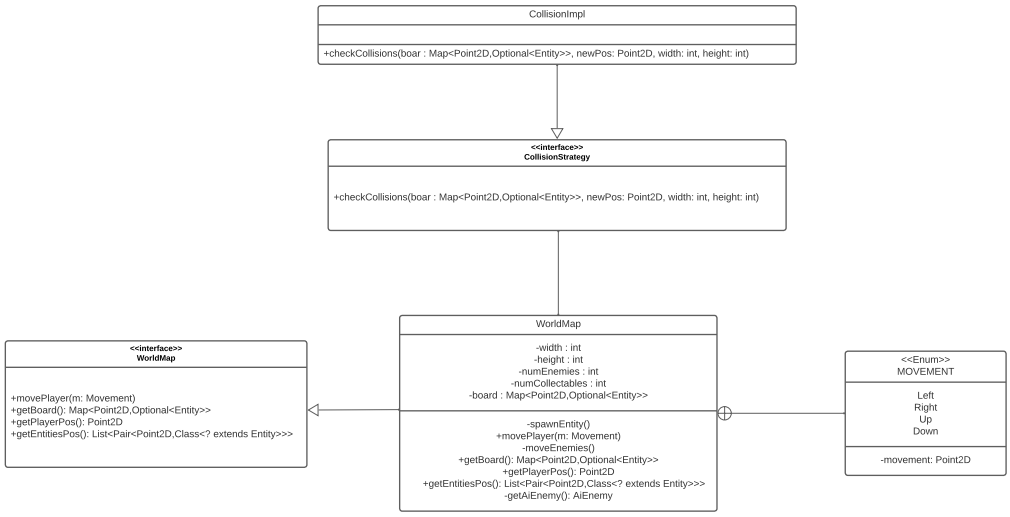

Figura 2.3: Schema UML a rappresentare la soluzione per le collisioni facendo uso di pattern Strategy con CollisionStrategy e CollisionImpl

MARIO CICCIONI

Creazione delle entita' Player e Enemy

- **Problema:** Fin dall'inizio, durante l'analisi del progetto da realizzare, ci siamo imbattuti nel problema delle varie tipologie di entita' presenti del videogioco. Infatti era necessario creare una gerarchia di Interfacce in grado di modellare al meglio la struttura delle entita' ottimizzando al massimo il riutilizzo e l'ampliamento futuro del progetto.

- **Soluzione:** Le entita' che compongono il videogico sono di 3 principali tipologie:

- `Player`
- `Enemy`
- `Collectable`

A questo punto e' stato necessario creare una interfaccia chiamata `Entity`, che contiene un contratto generale per tutte le tipologie di entita', a sua volta ho creato poi due interfacce che estendono da `Entity` chiamate `Character` e `Collectable`, ed infine le due interfacce `Player` e `Enemy`. Le interfacce che sono poi effettivamente implementate in classi sono `Collectable`, `Player` e `Enemy`. Questa implementazione fa uso del pattern Strategy, molto utile per tutti i vantaggi che offre dichiarati nel problema qui sopra. Di seguito lo schema UML rappresentante la gerarchia di interfacce/classi che modellano il problema. Figura 2.4.

- **Miglioramenti apportabili a questa soluzione:** La soluzione implementata e' sicuramente buona ma non ottima. Senza avere avuto le problematiche di tempo generate dall'abbandono di un membro del gruppo, con la conseguente ricaduta del suo lavoro nelle spalle dei due membri rimasti, avrei provato a programmare diversamente questo concetto di Entita' sviluppando una AbstractFactory (Design Pattern/Creational Pattern) in grado di modellare ancora meglio questo concetto generando un codice maggiormente modulabile, compatibile e modificabile.

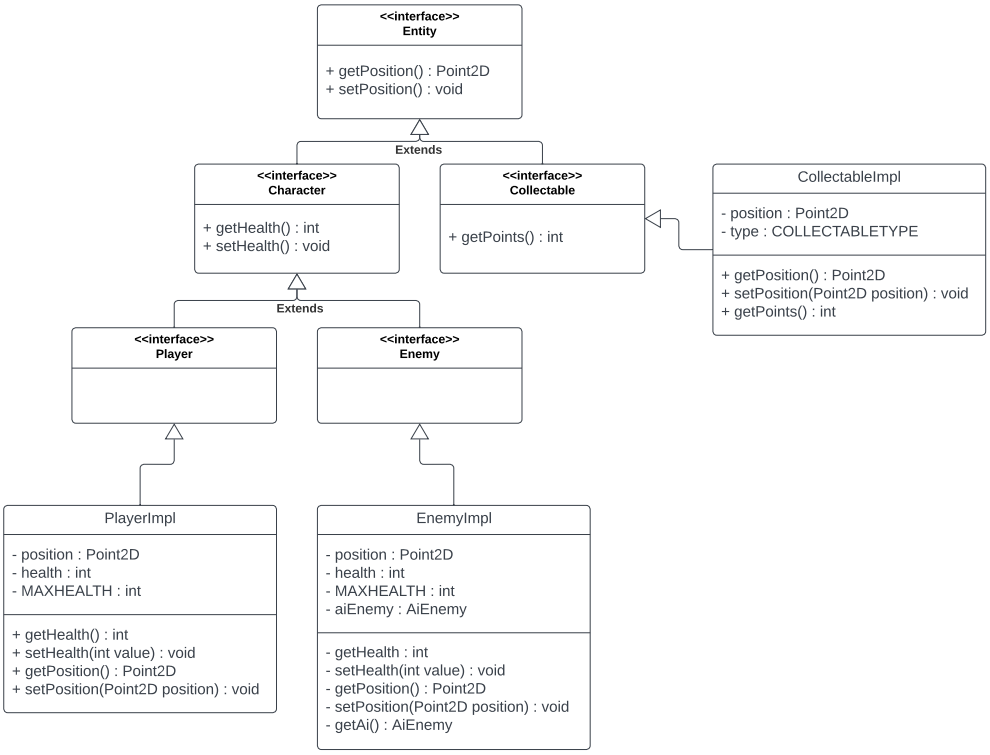

Figura 2.4: Schema UML rappresentante la gerarchia delle classi/interfacce di Entity.

Implementazione dell'intelligenza artificiale dei nemici e sue varianti

- **Problema:** A differenza degli oggetti collezionabili che si trovano nella mappa di gioco e sono elementi statici, i nemici devono necessariamente possedere un'intelligenza artificiale che gli permetta di muoversi. Quindi e' stato necessario implementare un sistema di Ai in grado di soddisfare questo problema.

- **Soluzione:** Per risolvere in modo pulito e per sfruttare al meglio il riutilizzo di codice ho adottato l'uso del pattern Strategy, il quale e' ottimo per affrontare e risolvere al meglio questo problema. Con l'interfaccia `AiEnemy` viene dichiarato il contratto che deve essere rispettato da tutte le classi che estendono da essa. Piu' in particolare ho implementato due diverse tipologie di intelligenza: `RandomAiEnemy` e `FocusAiEnemy`. `RandomAiEnemy` e' un'intelligenza che fa muovere il nemico in modo completamente casuale all'interno della mappa di gioco.

`FocusAiEnemy` invece, e' un'intelligenza che offre al nemico la capacita' di inseguire il Player.

Di seguito viene mostrato lo schema UML del modello di questo specifico sotto-sistema. Figura 2.5.

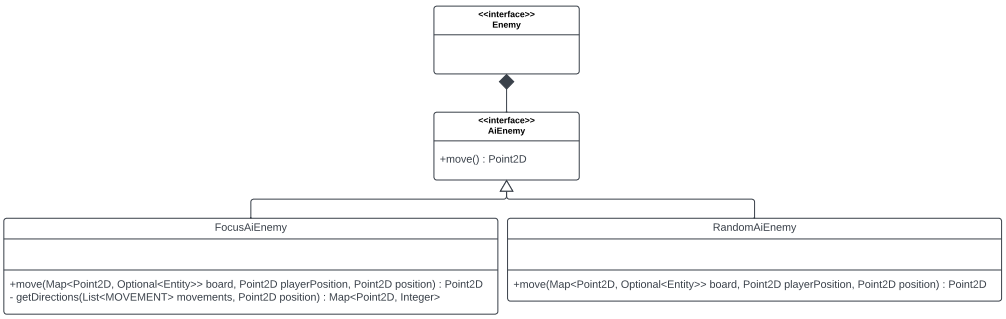

Figura 2.5: Schema UML rappresentante la gerarchia delle classi/interfacce che modellano l'intelligenza artificiale

Gestione punteggi giocatore

- **Problema:** Per tracciare i punteggi ottenuti nelle varie partite di gioco e' stato necessario implementare un sistema di salvataggio punteggi. Con la possibilita', cosi', di essere visualizzati dall'utente a fine o inizio di una partita.

- **Soluzione:** Per fare cio' e' stato necessario implementare una logica di salvataggio punteggi su file e visualizzazione sull'interfaccia grafica in una sezione appositamente creata per questo scopo (voce di menu Score con visualizzazione a tabella Nome : Punteggio). Dal punto di vista progettuale ho utilizzato anche per risolvere questo problema il pattern strategy creando un'interfaccia `Scores` e la sua implementazione `ScoresImpl` offrendo cosi' facile estensione o cambiamento del codice.

Di seguito viene mostrato lo schema UML del modello di questo specifico sotto-sistema. Figura 2.6.

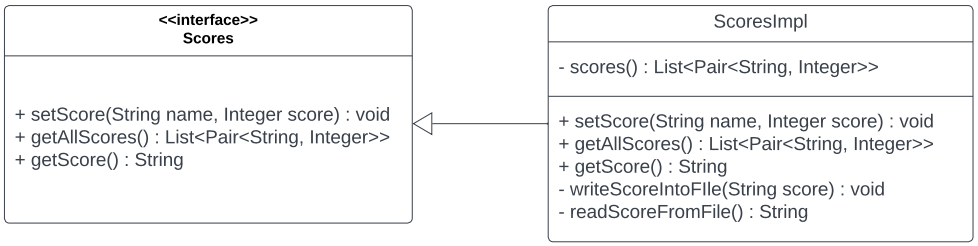

Figura 2.6: Schema UML rappresentante la gerarchia delle classi/interfacce che modellano il sottosistema che gestisce i punteggi partite.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Numerosi sono i compiti e i dettagli che durante lo sviluppo del progetto si sono dovuti in particolare modo controllare e testare affinchè l'applicazione funzionasse correttamente. Per elaborare ciascuno dei test necessari abbiamo sfruttato i metodi di Assertions (quali `assertTrue, assertEquals...`) contenuti all'interno della libreria di `JUnit 5`.Ognuno di questi sarà qui brevemente descritto in una sua sottosezione, indicando i punti principali di cui assicurarsi che il testing avesse successo. Per quanto riguarda, invece, i test manuali, visto che tutti abbiamo sviluppato su sistema operativo Windows 10, abbiamo proceduto con il testing del jar anche su linux (distribuzione lubuntu 22.04), Windows 11 e 8 e MacOs 13 e 12. Senza riscontrare problemi in nessuno di essi.

ANDREA BIANCHI

WorldMap e SpawnStrategy: La prima casistica da verificare era controllare se la mappa di gioco logica venisse creata e riempita con successo con il numero di entità richieste. Bisognava perciò dapprima controllare che l'implementazione dell'interfaccia `SpawnStrategy` funzionasse perfettamente. Ho perciò creato in un package separato denominato `worldMapTest` una classe con lo stesso nome puramente dedicata agli scopi precedentemente elencati. Ho quindi istanziato delle variabili iniziali contenti le dimensioni (sufficientemente grandi) della mappa, il numero di entità da inserire, una `randomSpawnStrategy` e la `WorldMap` stessa. Il primo test creato `testRandomSpawnStrategy()` comincia generando due set di `Point2D` creati tramite l'utilizzo del metodo `getSpawnPoints(...)` della `SpawnStrategy`.

All'interno di entrambi i set così realizzati si aggiunge poi uno stesso elemento `Point2D`. Si verifica da qui tramite l'utilizzo di `assertEquals` che la size di entrambi i set così creati sia uguale al valore `EXPECTED_SIZE`, corrispondente al numero di punti richiesti da creare più uno (il duplicato che era stato aggiunto in seguito). Una volta verificato ciò, si è generato un nuovo set utilizzando il metodo `getDoubleSpawnPoints(...)`, passando come dati di input i due set precedentemente creati. Utilizzando `assertEquals` si è prima di tutto verificato che il size fosse corrispondente alla somma dei size dei due set antecedenti, per poi continuare con una serie di `assertTrue` per ispezionarne i suoi contenuti e confermare che questo contenesse sia tutti i valori del primo set sia quelli del secondo senza duplicati al suo interno. Il secondo test definito come `testWorldMapCreation()` prende la mappa creata all'inizio e conta il numero di Entities che sono contenute all'interno della mappa. Tramite `assertEquals` si verifica che le dimensioni della mappa corrispondano a quelle indicate ad inizio file e che il numero totale di entità salvate sia esatto a quanto stabilito. Si termina questo test con un `assertTrue` per verificare che al centro della mappa vi sia istanziata l'unica entità `Player`. L'ultimo test creato `testMovePlayer()` parte facendo compiere al giocatore due spostamenti all'interno della mappa, e utilizza una serie di `assertTrue` e `assertEquals` per verificare che la precedente posizione sia vuota e che il giocatore abbia raggiunto la destinazione.

Collisioni: Gli obiettivi più importanti in questo test erano quelli di accertare la correttezza dei metodi costruiti in `CollisionImpl`. All'interno di un package separato contenente solo la classe `CollisionTest`, ho scelto di creare tre semplici metodi di test, ognuno contenente una diversa mappa di gioco di stesse dimensioni 3x3 ma con un numero diverso di entità. Nel primo caso detto `testEnemyCollision()` ho riempito completamente la mappa di entità `Enemy` (con la quale il giocatore può scontrarsi) lasciando solo la casella al centro per il `Player`: da qui ho tentato di far muovere il giocatore e ho verificato tramite `assertEquals` che il player fosse rimasto bloccato al centro della mappa. Nel secondo test `testBorderCollision()`, la mappa conteneva esclusivamente il personaggio giocante, senza alcuna altra entità. Ho richiamato più volte la funzione interna alla mappa `movePlayer(...)` per spostare continuamente il giocatore in una stessa direzione, fino a raggiungere e (teoricamente) superare il bordo della mappa. Con un ulteriore `assertEquals` ho invece dimostrato che il player fosse rimasto bloccato lungo i bordi della mappa. L'ultimo test `testCollectableCollision()` contiene una mappa di soli oggetti raccoglibili dal player. Ho così fatto muovere il player in una direzione qualunque e testato prima con un `assertTrue` che il movimento fosse avvenuto con successo, senza alcuna collisione; successi-

vamente, ho raccolto il numero di entità presenti all'interno della mappa e tramite l'uso di `assertEquals` ho certificato che il numero di entità presenti al suo interno fosse diminuita di 1, ad indicare che il giocatore ha "raccolto" l'oggetto.

Collectable: Il più semplice tra i test ma sicuramente non meno importante, anche questo è stato creato all'interno di un package `entitiesTest`, contente non solo la classe di verifica dei `Collectable`, ma anche per le altre entità esistenti nel programma. Data l'interfaccia decisamente meno complessa di `Collectable` e la sua classe di implementazione, l'unico obiettivo di questo esperimento era quello di controllare il corretto assegnamento del "tipo" ai diversi collezionabili tramite l'enum `COLLECTABLETYPE`. Ho perciò creato un singolo test creando una `List<Collectable>` che ho riempito con una quantità a mia scelta di oggetti. Ho poi chiamato il metodo `removeIf` della lista per rimuovere tutti i collezionabili il cui valore non superava una certa soglia. Utilizzando un iteratore, ho infine terminato con un semplice `assertTrue` che ciascun elemento avesse un valore superiore a quanto precedentemente specificato.

MARIO CICCIONI

EnemyTest, Player Test e intelligenza artificiale: Dopo aver implementato correttamente le due classi `EnemyImpl` e `PlayerImpl` e' risultato necessario programmare dei test automatici per verificare il corretto funzionamento delle due classi appena citate. Cosi', nel package `entitiesTest` (contenuto in una cartella dedicata a tutti i test del software) ho creato due classi: `EnemyTest` e `PlayerTest`. Per `EnemyTest` e' stato necessario istanziare la WorldMap e dei nemici per popolare la mappa, poi ho eseguito dei classici test per verificare la loro presenza nella mappa di gioco e il loro effettivo spostamento, casuale oppure focus verso il player in base all'intelligenza fornita a quello specifico nemico. Invece per `PlayerTest` ho creato una classe di test separata per testare il corretto spawn del player e il suo corretto spostamento allinterno della mappa di gioco.

Score test: Per quanto riguarda il testing del sistema che tiene traccia dei punteggi (`Scores` e `ScoresImpl`) ho creato un package dedicato di nome `scoresTest` contenente la classe `ScoresTest`. Al suo interno ho testato la corretta scrittura e lettura su/da file verificandone il corretto funzionamento.

3.2 Metodologia di lavoro

Il primo passo che è stato fatto a seguito della scelta di un'architettura soddisfacente, si è iniziati cercando quali possibili librerie, framework o altri strumenti si sarebbero potuti utilizzare per meglio facilitare lo sviluppo dell'applicazione. Si è deciso così di utilizzare **Gradle**, uno strumento open-source di *Build automation* noto per la sua incredibile flessibilità e capacità di costruire i più svariati tipi di software. I principali vantaggi che questo è in grado di offrire sono:

1. **Performance elevate**, dato che questo fa partire solo i task necessari, generalmente quelli a cui sono stati modificati input-output;
2. **Grande estendibilità** per poter facilmente costruire i propri modelli e task;
3. **supporto a numerosi IDEE**, tra cui anche Eclipse (quello che è stato utilizzato per questo progetto);
4. **Scansioni accurate** per comprendere al meglio i problemi e gli errori che si sono compiuti durante lo sviluppo del codice.

Si è preferito inoltre utilizzare un framework specifico notosi come **JavaFX**, al contrario dell'estremamente comune Swing. Rispetto a quest'ultimo infatti, JavaFX presenta notevoli benefici e offre una qualità nettamente superiore. Alcune tra le differenze maggiori presenti sono:

- **Eventi ad Alto livello**, leggermente più difficili da comprendere e da ben realizzare a prima vista, ma in grado di offrire una consistenza che gli event in Swing non possono minimamente raggiungere;
- **Uso di Proprietà**, delle specie di variabili determinabili che posso "bindare" tra loro per gestire diverse meccaniche all'interno del programma (event handling di questo tipo non è supportato da Swing);
- **Vasta gamma di controller utilizzabili**, tra cui BorderPane,Anchor Pane,TableView,Menu . . . Mentre Swing è solo in grado di fornire semplici forme di controllo quali bottoni, checkbox e simili;
- **Animazioni fluide**, cosa non impossibile da eseguire in Swing, ma molto più complesso e prolisso.

Si è partiti perciò dal template in Gradle, apportando minuscole modifiche al gradlew per far correttamente funzionare il tutto sui nostri dispositivi. Sistemato ciò, abbiamo subito iniziato a suddividerci i compiti per il progetto,

cercando di assegnare a ciascun membro sia almeno un aspetto riguardante i Models, sia almeno un aspetto grafico di View e Controller. E' stata quindi creata una repository contenente una serie di branch ognuna con il nome dello studente a cui è stata assegnata e una branch master dove poter unire i lavori svolti dai diversi partecipanti. Abbiamo scelto di utilizzare solo branch separate personali invece di un sistema di branching piu' complesso in quanto le funzionalità da svolgere erano già state correttamente divise tra i membri. Mano a mano che un affiliato completava i propri compiti, questi venivano pushati sulla branch personale da mostrare e discutere con gli altri per accertarsi che il lavoro fosse stato completato con successo e ricevere possibili consigli per miglioramenti e ottimizzazioni. Vi si è prima concentrati sulla realizzazione dei modelli, le fondamenta di tutto il programma: questi sono stati studiati, elaborati, testati, ottimizzati e trattati per gran parte dello sviluppo per conferirne un alto livello di performance ed estendibilità. Ciascuno di questi viene meglio descritto nelle sottosezioni qui sotto definite da ciascun socio. Si è poi passati ai controller, creandone 3 diversi tipi: **MenuController**, contenente 3 diversi pulsanti, uno per far partire il gioco effettivo e iniziare nuove partite, uno per vedere gli scores fatti dai giocatori e uno per chiudere l'applicazione; **boardController**, che gestisce le interazioni tra l'utente e la mappa di gioco e le mostra a schermo nella finestra, contiene al suo interno un metodo iniziale per la creazione della view che conterrà gli strumenti visivi corrispondenti alle diverse entità, suddivisi in colori appositi a rappresentarne il tipo (giallo per gli oggetti raccoglibili, grigio per i nemici, verde per il giocatore); contiene inoltre i metodi di update della mappa a seguito di un movimento o della "raccolta" di un oggetto, insieme ad un metodo che avvisa quando il giocatore ha preso danno e uno che viene richiamato quando il giocatore muore, facendo terminare la partita e tornare al menù iniziale; ultimo controller utilizzato è **ScoreController**, che mostra tramite una classifica di dati (non ordinata) di tutti i giocatori e il punteggio accumulato durante la partita. Dopo aver concluso il lavoro del controller, si è incapsulato i dettagli grafici (Canvas, TableView, Scene...) all'interno di View specifiche, quali la `BoardView`, interfaccia contenente i contratti per lo sviluppo di una qualsiasi View ideata per la mappa di gioco (implementata in questo caso da `GameBoardView`).

ANDREA BIANCHI

Data la grande quantità di lavoro da svolgere, numerosi sono stati i compiti che ho dovuto completare. Le classi, interfaccie ed enum a cui ho lavorato sono stati:

- la mappa di gioco `WorldMap` e `WorldMapImpl`;

- la strategia per lo spawn delle entità `SpawnStrategy` e `RandomSpawnStrategy`;
- i possibili movimenti all'interno della mappa con `MOVEMENT`;
- un sistema di collisioni per le entità con `CollisionStrategy` e `CollisionImpl`;
- la creazione di `Collectable` con `CollectableImpl` e le tipologie con `COLLECTABLETYPE`;
- lo sviluppo del `BoardController` presenter e della view della mappa `BoardView` con `BoardViewJavaFX`.

Alcuni bug noti da segnalare all'interno di questi file sono:

- nonostante sia un caso raro, è possibile che uno dei nemici possa andare ad occupare la stessa posizione di un oggetto raccoglibile, andando così ad eliminarlo dalla mappa di gioco "mangiandolo";
- anche se gli elementi non risultano mai uscire dalla mappa logica nè da quella grafica, può capitare che un movimento che faccia uscire il giocatore dalla mappa restituisca un erore di tipo "NullPointerException" (non vi è un'appropriata cattura delle exception).

Questi problemi sono derivati dal fatto che tutta la gestione delle collisioni doveva essere originariamente completata da un altro ragazzo che ha a seguito abbandonato il gruppo, lasciando file con codice mal curato e che anzi andava a danneggiare il lavoro altrui. Sono state perciò necessarie delle lavorazioni più rapide e meno pulite per completarne le funzioni principali limitandone il numero di bug possibili.

MARIO CICCIONI

Nel seguente software mi sono occupato dell'implementazione/gestione dei seguenti sotto-sistemi oltre ad altri particolari meno rilavanti:

- Creazione dei Nemici e del Player.
- Implementazione dell'intelligenza artificiale da assegnare ai nemici (`RandomAiEnemy` e `FocusAiEnemy`).
- Implementazione e gestione degli Scores (`Scores` e `ScoresImpl`).
- Implementazione di `MenuUIController` e `ScoreController`.

3.3 Note di Sviluppo

MARIO CICCIONI

Nel codice da me personalemte sviluppato non ho fatto molto uso di features avanzate del linguaggio e dell'ecosistema Java (fattore parzialmente determinato dall'uscita del gruppo di un membro). Nonostante cio' sono riuscito a lavorare, in modo molto basic, con i senguenti strumenti avanzati:

- Uso di `Optional` per quanto riguarda la gestione e identificazione dei Nemici o del Player, rendendo il codice piu' pulito e comprensibile.
- Uso della libreria grafica `JavaFX` per la creazione dell'interfaccia grafica, in particolare per la creazione del Menu di gioco e visualizzazione Punteggi.
- Uso di `lambda` nei Controller per le chiamate, per esempio, di inizializzazione view.

In generale ho utilizzato internet per documentarmi sulla *javadoc* e sulla documentazione di *JavaFX*. Ci sono, pero', delle piccole parti di codice che ho sviluppato prendendo spunto da forum online (come *stackoverflow* e simili). In particolare per lo sviluppo e creazione dell'interfaccia grafica e per i relativi Controller (personalmente le parti piu' ostiche che ho dovuto affrontare).

ANDREA BIANCHI

Per evitare di scrivere del codice verboso e ridondante, ho fatto uso di svariate tipologie di features avanzate offerte dall'ambiente Java e da librerie esterne. Quelle che ho in particolar modo sfruttato durante le mie ore lavorative sono state:

- gli `Stream` e in particolar modo gli `IntStream` per la creazione della mappa di gioco inizialmente vuota;
- la classe `Optional` per identificare le diverse entità collocate nei punti della mappa di gioco e le zone libere dove poter compiere movimenti.
- la libreria grafica `JavaFX` per poter facilmente tradurre la mappa logica presente nei modelli in una interfaccia grafica con cui l'utente può facilmente interagirci.
- le `lambda` all'interno di numerosi metodi in svariate classi, quali l'inizializzazione con stream della mappa, la chiamata all'inizializzazione di

view nei controller o all'interno di foreach per elementi creati all'interno della view stessa.

Ho fatto abbastanza uso di internet per documentarmi sulla *javadoc* e sui componenti offerti da *JavaFX*. Tra i siti che ho sicuramente più visitato per risolvere dubbi che mi sono posto durante le ore di stesura del codice vi è sicuramente *stackoverflow*, dove ho potuto osservare problemi similari a quelli con cui mi sono scontrato e che ho a seguito usato come spunto per risolvere i miei casi (come quando si è presentato un problema nel "catturare" la pressione di uno specifico tasto da parte dell'utente). La classe `Point2D` che abbiamo fortemente utilizzato non è nient'altro che un'estensione della classe `Pair` creata dal professor Viroli e presente in numerose esercitazioni fatte in laboratorio. Dato il poco tempo e le numerose difficoltà che ci si sono poste durante lo sviluppo, vi sono alcune modifiche e dettagli che si sarebbero potuti introdurre per ottimizzare ancora meglio il codice e renderlo più ricco, quali:

- la possibilità di creare nuovi tipi di SpawnStrategy specializzate, per esempio una dove tutti le entità fossero collocate in una maniera tale che due entità non fossero inizialmente collocate una vicino all'altra (creando un pattern similare a quello di una scacchiera);
- lo sviluppo di un pattern Combinator più esteso,utilizzabile per poter creare le posizioni per tutte le entità e combinarne tutti i punti evitando la presenza di duplicati: la struttura così realizzata al momento all'interno di `SpawnStrategy` applica una versione "similare" ad un 2-Way Combinator al suo interno, ma non risulta trattarsi dell'utilizzo di un vero e proprio pattern Combinator;
- la concretizzazione di altri tipi di collisioni, ciascuna specializzata per il tipo di entità a cui deve essere applicata (un po' come una Factory);
- sostituire l'utilizzo dei colori nella `BoardViewJavaFX` con degli sprite effettivi di gioco, con animazioni.

Capitolo 4

Commenti Finali

4.1 Autovalutazione e lavori futuri

ANDREA BIANCHI

Ritengo che questa esperienza sia stata molto difficile, impegnativa (soprattutto dovuto ai tempi stretti e ai problemi incontrati lungo il percorso) ma anche da un certo punto di vista appagante: vedere che sono riuscito a creare un' applicazione funzionante con anche una buona struttura mi ha dato un certo livello di soddisfazione, soprattutto considerando il fatto che è da sempre un mio personale sogno sviluppare un videogioco. Spero in futuro di poter continuare in questo campo e rilasciare dei progetti decisamente più raffinati e curati che dimostrino quanto sia cresciuto in questo campo.

MARIO CICCIONI

Al completamento del monte ore indicato per il progetto sono rimasto un po' insoddisfatto del mio lavoro finale perche' avrei potuto sicuramente fare meglio. E' anche vero che ho dovuto affrontare molte difficolta', alcune di esse non derivanti da me (abbandono da parte di un nostro compagno di progetto) ma nonostante cio' e' stato di grande aiuto lavorare duramente su un progetto abbastanza corposo come questo, che ha sicuramente aumentando le mie skill in questo campo. Mi piacerebbe apportare alcune migliorie al progetto, si sotto l'aspetto implementativo di alcune sotto-parti del progetto, ma soprattutto a livello grafico in quanto siamo riusciti solo a sviluppare un'interfaccia grafica molto molto minimale che sicuramente sminuisce un po' il lavoro (abbastanza buona) che abbiamo fatto sotto il punto di vista progettuale di modello.

4.2 Difficolta' incontrate e commenti per i docenti

ANDREA BIANCHI

Le principali difficoltà che sono state incontrate durante il percorso erano principalmente lo sviluppo della archiettura e degli aspetti grafici tramite l'utilizzo di JavaFX: non che non sia un ottimo strumento per la graficazione di dati in Java, ma le poche risorse che sono state trovate per risolvere alcuni dei dubbi che ci siamo posti lungo il cammino (come ad esempio il ridimensionamento corretto della mappa di gioco da un aspetto logico a quello grafico). Sicuramente il più grosso ostacolo che ci siamo trovati davanti è stato la tempistica molto stretta (di circa 2 mesi di puro lavoro non-stop), senza considerare che originariamente il progetto doveva essere sviluppato da 3 persone e siamo invece stati costretti a recuperare e sistemare velocemente tutti gli errori lasciati dal ragazzo che si è ritirato durante lo sviluppo dell'applicazione. Spero vivamente che casi simili non ricapitino a futuri studenti di questo corso: vedere un compagno che stimi e che ti abbandona lasciandoti tutto il suo carico di lavoro è veramente stressante. Consiglierei ai prof di mantenere maggiori contatti con gli studenti quando elaborano un progetto per poter meglio comprendere le dinamiche all'interno di ciascun team e valutare i lavori di questi in base sia al lavoro di gruppo sia quello individuale.

MARIO CICCIONI

Durante i mesi di lavoro al seguente progetto mi sono reso conto delle difficolta' che porta lo sviluppare un software complesso come un videogioco. Anche se Stubborn e' un videogioco con funzionalita' molto basilari, sono stati molti i problemi incontrati duranti lo sviluppo. I principali problemi sono stati ruguardanti gli aspetti di interfaccia grafica con l'uso di *JavaFX* e i vari concetti di Controller e gestione di essi. Arrivato alla fine del percorso non sono molto suddisfatto del mio lavoro perche' avrei potuto fare di piu' anche se molte cose le avrei potute fare meglio se un membro del gruppo non ci avesse abbandonato in corso d'opera, senza contribuire per niente al lavoro a lui inizialmente assegnatogli, appesantendo conseguentemente il lavoro di me e Andrea che per forza di cose abbiamo duvuto prenderci carico del suo lavoro penalizzando il nostro. E' stata comunque un'esperienza molto utile e costruttiva, insegandomi a come si lavora in gruppo, e che mi ha formato e fatto capire i miei punti di forza e i miei punti deboli.

Appendice A

Guida utente

Avviata l'applicazione ci si trova davanti al menu di gioco con i tre bottoni: **Play**, **Scores** e **Exit**.

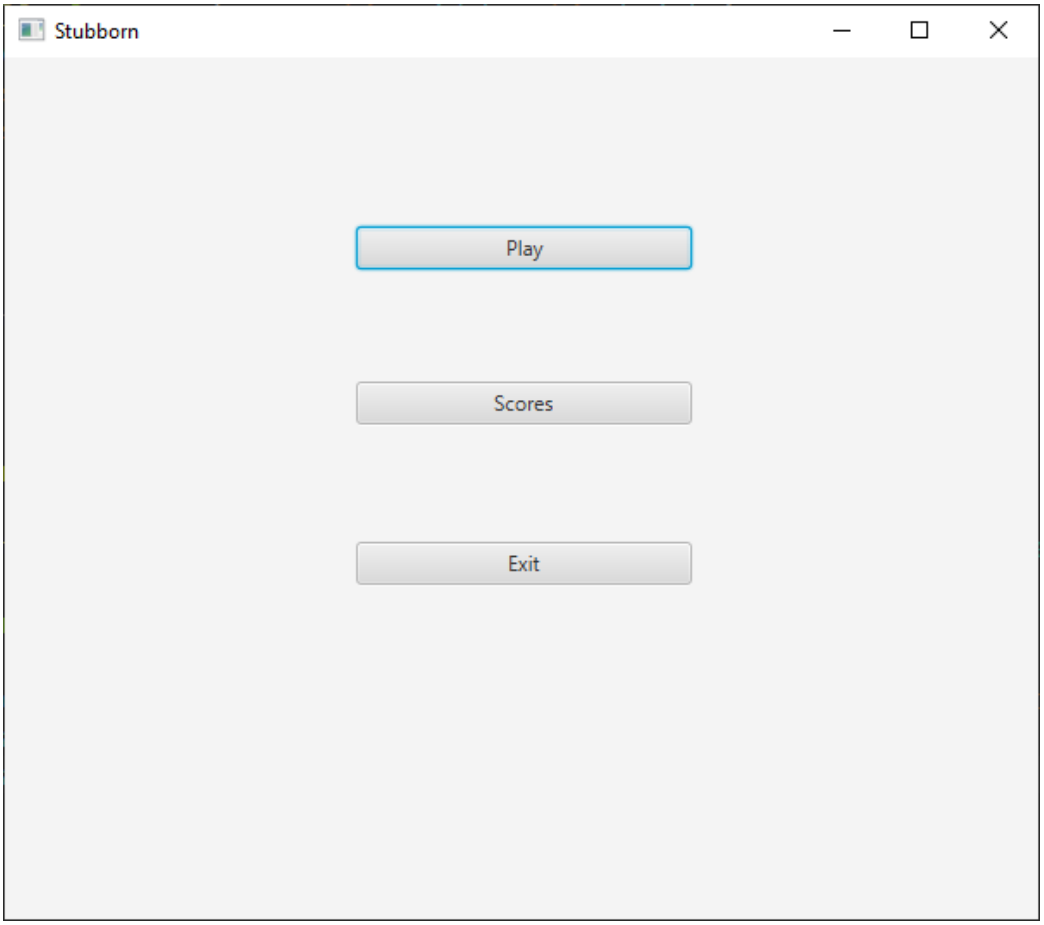

Figura A.1: Schermata di menu

Se si preme su **Play** la partita inizia immediatamente, come si puo' notare dallo screenshot sottostante il quadratino verde indica il **player**, mentre i quadratini gialli e grigi sono rispettivamente gli **oggetti collezionabili** e i **nemici**. Si hanno a disposizione 3 vite, quando si arriva a 0 la partita termina e si ritorna al menu di gioco, quindi si ha la possibilita' di iniziarne una nuova.

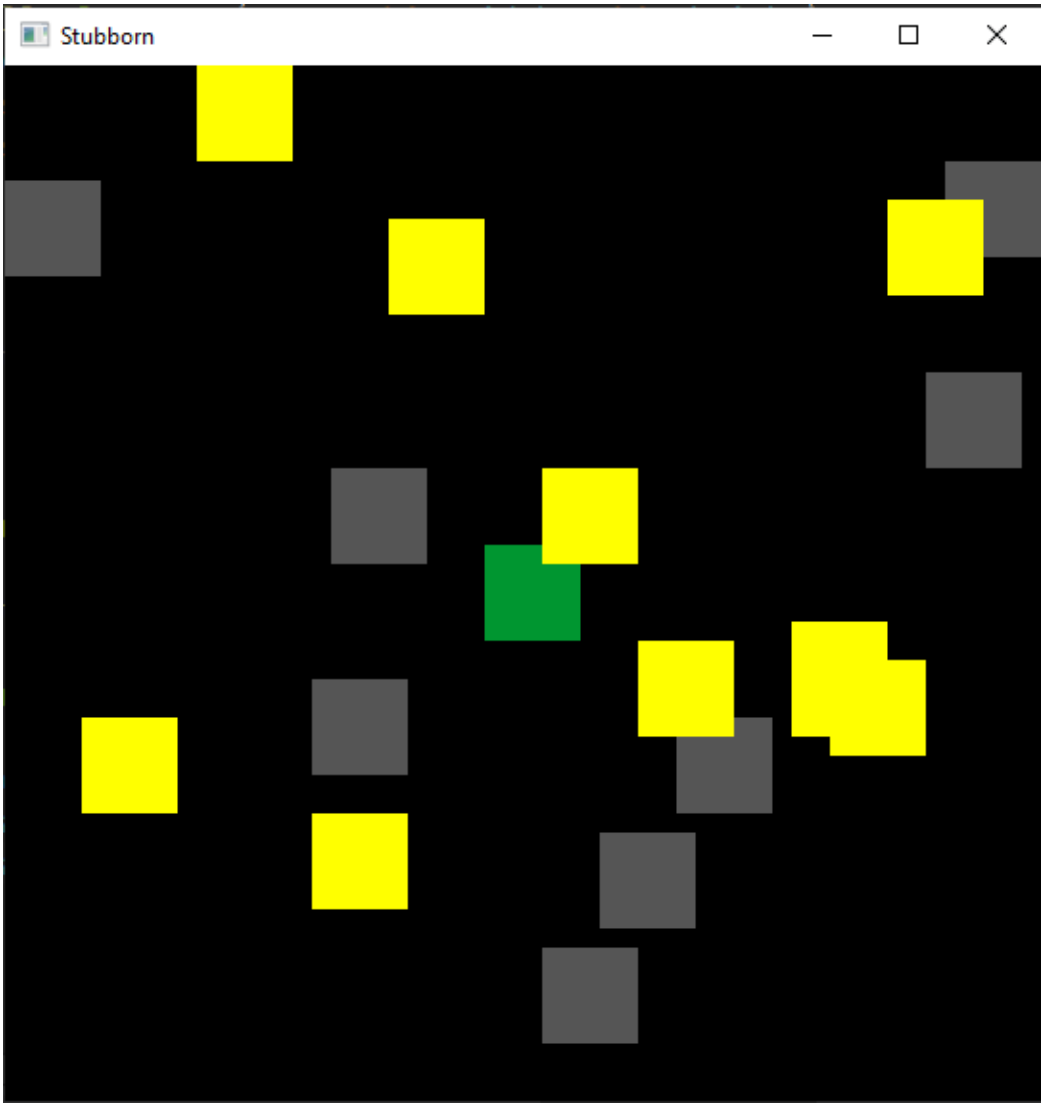

Figura A.2: Schermata di gioco

Bibliografia

- MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java (document)
- Design Patterns doc e tutorial `https://refactoring.guru/design-patterns`
- JavaFx documentation `https://openjfx.io/`
- JavaFx repository `https://github.com/openjdk/jfx`