ANALISI

Requisiti

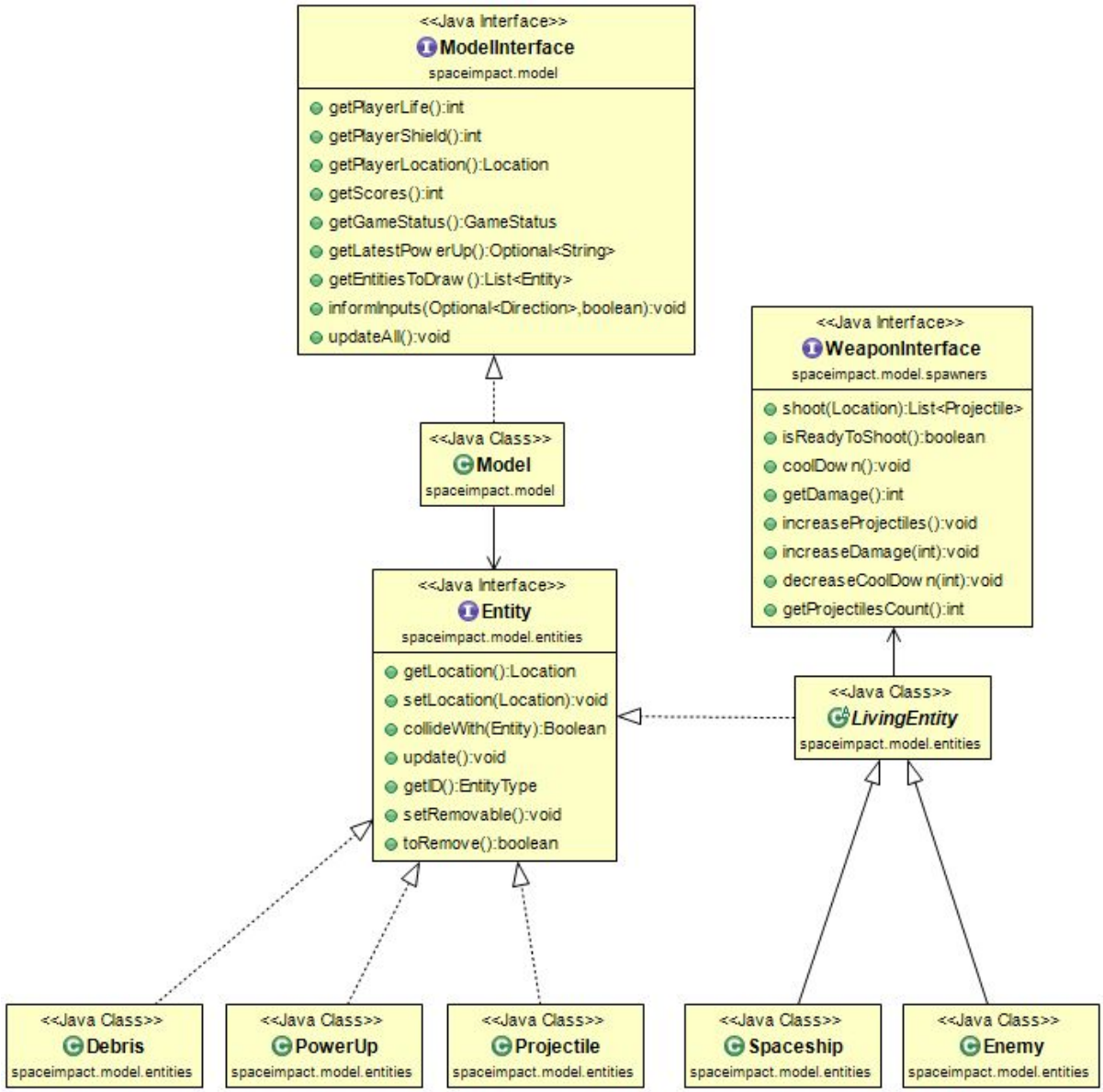
Il software mira alla realizzazione di un gioco simile a “Space Impact[1]” e a “Lost Viking[2]”. Il programma dovrà mostrare sullo schermo l'immagine di una astronave (i cui movimenti saranno controllati dall'utente) e le immagini di altre astronavi o ostacoli controllati dal computer.

- L'utente (o “giocatore”) dovrà essere in grado di muovere l'astronave orizzontalmente o verticalmente sullo schermo. Inoltre avrà a disposizione un tasto per sparare contro gli ostacoli e distruggerli.
- La collisione dell'astronave del giocatore contro le astronavi nemiche, ostacoli o proiettili causerà la riduzione di “scudi” e “vita” del giocatore. All'azzeramento di questi ultimi la partita sarà terminata.
- Per “partita” si intende una esecuzione del programma; una partita è composta da uno o più “livelli”, cioè scenari in cui il giocatore deve confrontarsi contro i nemici. I livelli dovranno essere generati dal programma in modo randomico, ma comunque in modo da fornire una difficoltà crescente.
- La collisione dell'astronave contro oggetti particolari, detti “powerups”, dovrà fornire al giocatore dei vantaggi, come il ripristino di parte degli scudi persi o una maggiore velocità di movimento.
- Al termine di una partita il programma dovrà fornire al giocatore la possibilità di salvare il proprio punteggio; i punteggi migliori (“high scores”) dovranno essere conservati anche dopo la chiusura del programma.

[1] Pagina di Wikipedia relativa a Space Impact: https://en.wikipedia.org/wiki/Space_Impact; video di esempio: https://youtu.be/Ol9TsSs-9ok

[2] Video di esempio: https://www.youtube.com/v/0kXoKVXXqzM

Analisi e modello del dominio



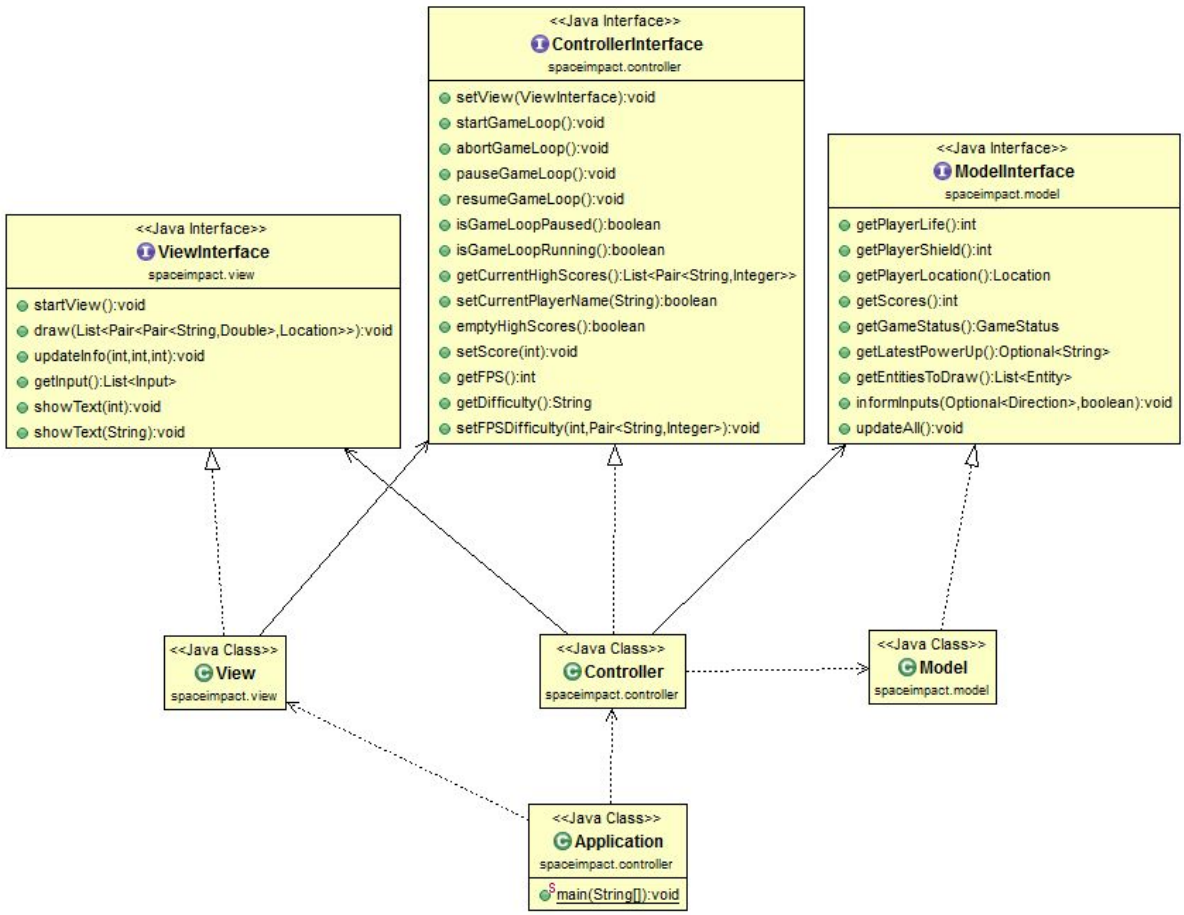
Il programma dovrà gestire una serie di livelli, al cui interno saranno presenti un certo numero di “entità” (Entity). Per entità si intende in generale un qualunque oggetto che verrà mostrato sullo schermo (sia esso un nemico, un ostacolo, un proiettile, un potenziamento o lo stesso giocatore). Queste entità potranno essere “viventi” e non: nel primo caso (LivingEntity) le entità avranno a loro disposizione un'arma (Weapon) per sparare. Tra le entità “viventi” individuiamo la nave del giocatore (Spaceship) e quelle nemiche (Enemy), tra quelle “non viventi” individuiamo gli ostacoli (Debris), i potenziamenti (PowerUp) e i proiettili (Projectile). Tutte le entità comunque avranno una posizione (Location), che ne identifica il punto preciso nel piano cartesiano che rappresenta astrattamente l’area di gioco nel model, e un’area (Area), cioè un riferimento allo spazio occupato dall'entità nello spazio di gioco, che verrà usato per capire quando due entità entrano in collisione. La difficoltà principale sarà gestire le varie entità del livello in modo fluido, ma al contempo mantenendo un costo computazionale basso. Nella prima versione del software fornito i nemici si sposteranno e

spareranno secondo schemi semplificati e senza fare valutazioni sulla posizione delle altre entità nel livello. Questo in quanto progettare nemici in grado di scegliere autonomamente le azioni da intraprendere è un compito molto complesso e non potrà essere effettuato all'interno del monte ore previsto, tale feature sarà oggetto di lavori futuri.

Design

Architettura

Per fare interagire le componenti principali del software abbiamo adottato una architettura MVC (ossia una suddivisione in tre parti distinte Model, View e Controller). Il seguente diagramma UML rappresenta l'architettura del software ad alto livello.



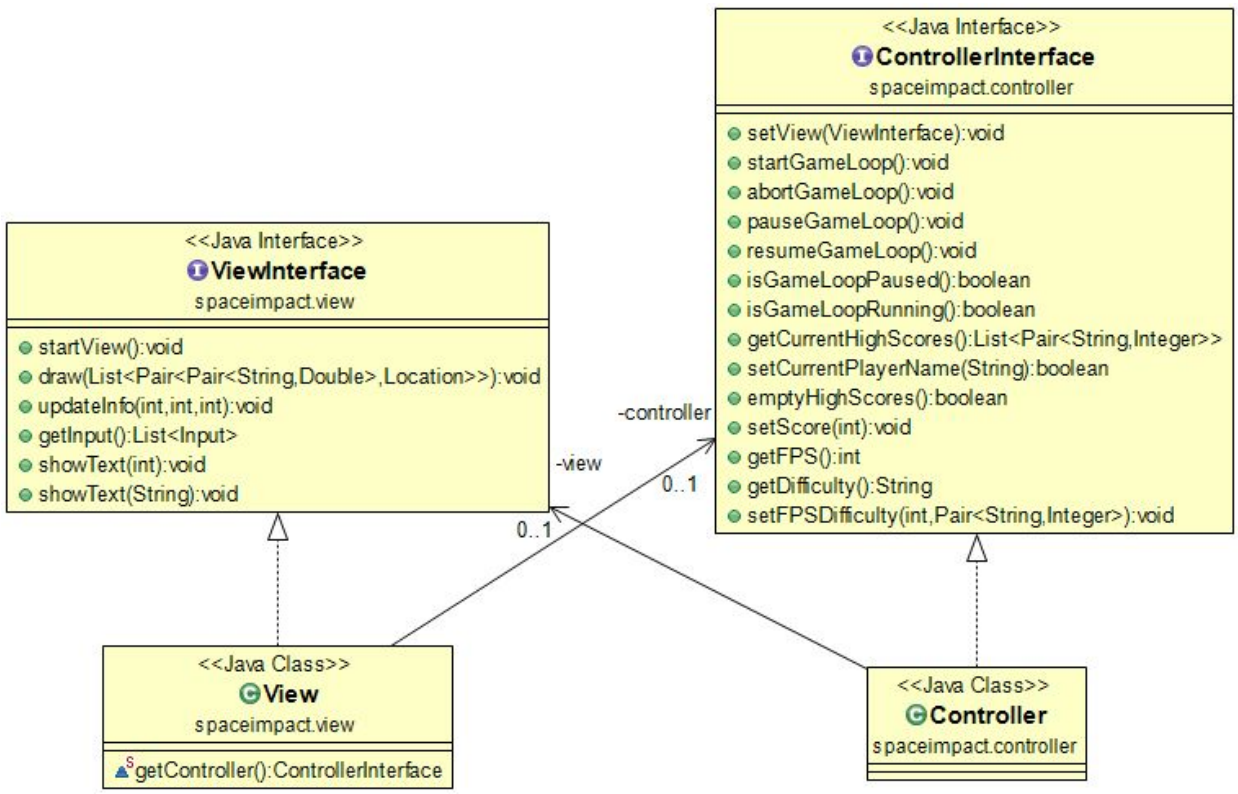
Model, View e Controller si compongono di altre classi che per motivi di chiarezza non abbiamo incluso in questo primo diagramma.
All'avvio dell'applicazione viene creata una View e un Controller e quest'ultimo si occupa di creare il Model quando necessario. Il Controller usa il Model in modo indiretto (attraverso di una delle classi non visibili).
La View si occupa di fornire una rappresentazione grafica del model, ed è studiata in modo da poter essere sostituibile interamente senza intaccare le altre due parti. Infatti sarebbe sufficiente creare una nuova classe che implementi ViewInterface e modificare la sola classe Application perché venga usata questa nuova classe al posto della attuale.

Le componenti M V C interagiscono tra di loro unicamente attraverso i metodi pubblici delle interfacce (implementati dalle rispettive classi).

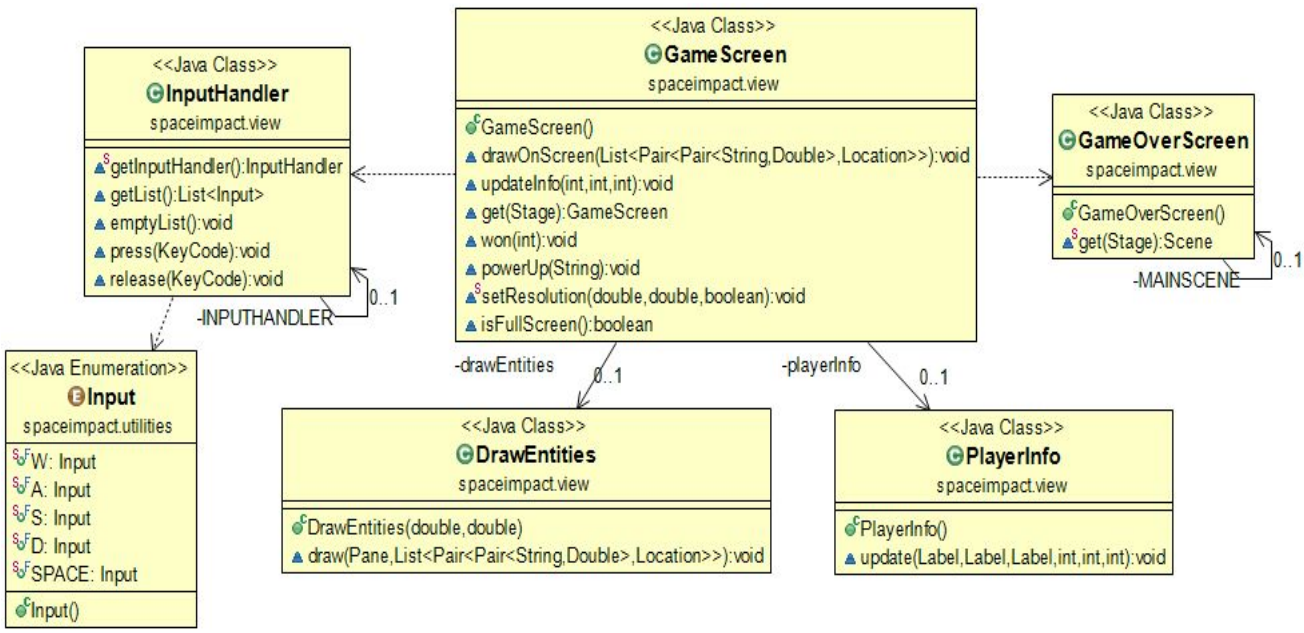
Design dettagliato

- **Bonato**

La View si occupa di gestire l'interazione con l'utente e di mostrare a video l'output prodotto dal Model durante una partita.
Per realizzarla è stato scelto, dopo un'analisi delle tecnologie esistenti, di utilizzare JavaFX, la più recente libreria grafica resa disponibile da Oracle a partire da Java 7. Questa scelta ha permesso di realizzare un'applicazione più moderna e graficamente piacevole in quanto questa tecnologia mette a disposizione diverse innovazioni rispetto alle precedenti come spiegherò in seguito.
La View è stata realizzata fin dall'inizio tenendo presente il pattern MVC, di conseguenza essa non è a conoscenza di nessun dettaglio implementativo relativo alla parte di Model e ogni comunicazione con esso passa attraverso il Controller e in particolare la ControllerInterface. Allo stesso modo, l'unico modo con cui la View comunica con il resto dell'applicativo ed in particolare con il Controller avviene attraverso la ViewInterface che rende disponibili alcuni metodi che si occupano principalmente di stampare a video i vari oggetti presenti durante una partita e di informare il Controller relativamente agli input dell'utente.
Il controller viene salvato in un campo statico nella classe View quando essa viene creata in modo che qualsiasi altra classe possa farne uso attraverso un getter.



Inizialmente la View viene avviata dal controller immediatamente dopo la creazione dell'applicativo grazie al metodo startView() che si occupa di lanciare il thread di JavaFX e di creare la finestra principale mostrando al suo interno il menù.
Per fare in modo che la View sia il più estendibile possibile è stato fatto in modo che ogni schermata sia indipendente dalle altre e che sia possibile cambiare schermata semplicemente sostituendo la scena (Scene) all'interno della finestra (Stage).
Quindi se in futuro si volesse aggiungere una nuova schermata basterebbe svilupparla indipendentemente dal resto per poi rendere disponibile la scena per essere usata dal resto della View attraverso un getter.
GameScreen è la principale classe per la gestione del gioco vero e proprio. Viene creata una sua istanza ogni volta che si inizia una nuova partita. Utilizza una serie di classi per supportare la rappresentazione a video delle informazioni e si occupa anche di rilevare gli input dell'utente. Se in futuro si decidesse di aggiungere altri tipi di navicelle o oggetti presenti durante una partita, non sarà necessario eseguire nessuna modifica alla View in quanto il metodo draw() della classe DrawEntities lavora nel modo più generale possibile stampando a video le immagini richieste in una certa posizione specifica.
Quando una partita finisce, si passa ad una schermata di "Game Over" gestita da un'altra classe.
Gli input vengono rilevati durante una partita nella classe GameScreen e vengono processati dall'InputHandler che è stato sviluppato usando il pattern Singleton in quanto è sufficiente avere un'unica istanza di esso che si occupa di gestire gli input nelle varie partite.
E' stato deciso di differenziare tra quando un tasto viene premuto e quando viene rilasciato: questo permette di avere un movimento più fluido del giocatore specialmente in un gioco come il nostro in cui è molto probabile che un tasto verrà tenuto premuto a lungo. Dopo essere stati processati dall'InputHandler, il Controller si occupa di richiederli ad ogni frame e vengono restituiti sottoforma di Lista di Input, dove Input è una enum con i tasti che possono essere usati per controllare la navicella.

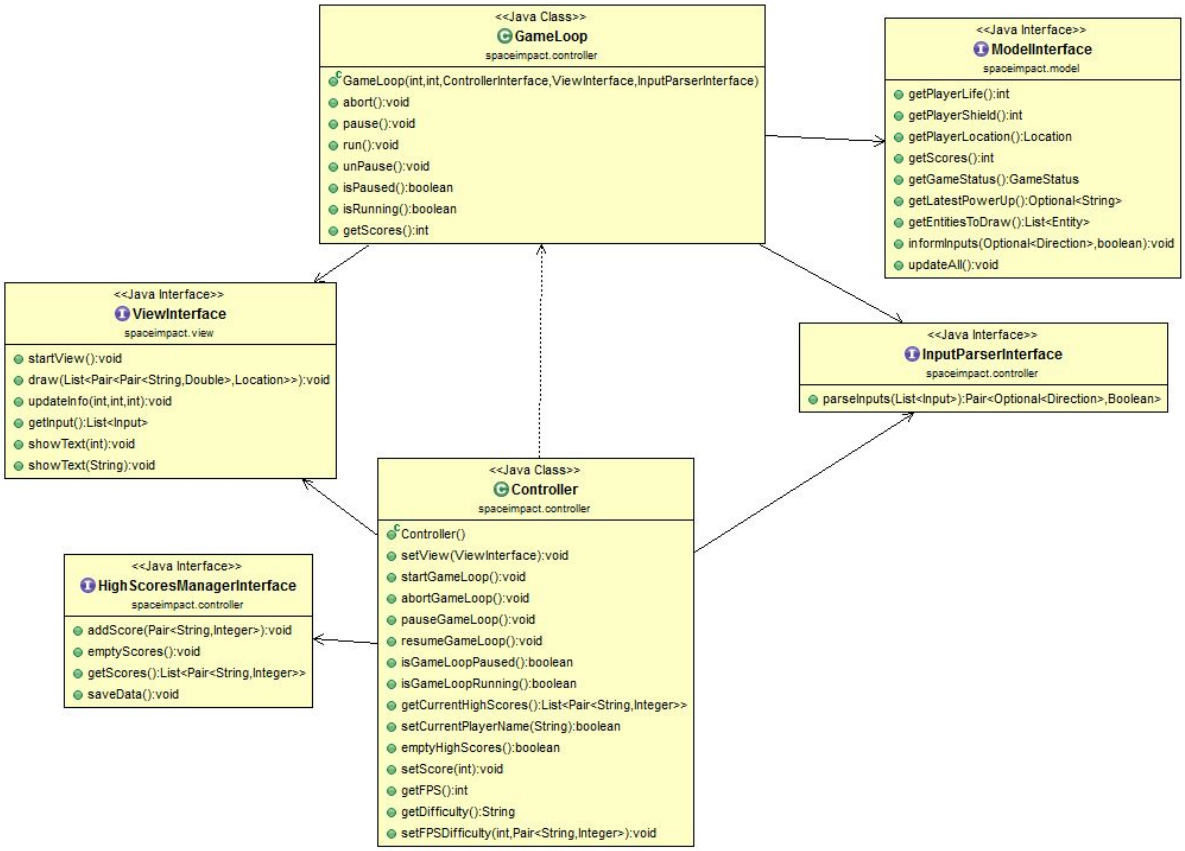


Sono stati sviluppate anche delle classi (ConfirmBox e GenericBox) generiche per la gestione delle dialog box con l'utente. Queste possono essere usate per poter far scegliere all'utente tra due possibili opzioni o per mostrare uno specifico messaggio di errore o di successo. Inoltre grazie alla enum BoxType sarebbe possibile aggiungere altri tipi di finestre per il dialogo con l'utente.
Considerando che è possibile uscire dal gioco con una varietà di opzioni (shortcuts, pulsante Exit o tasto X) è stato scelto di creare un ClosureHandler che si occupa di gestire la corretta chiusura del gioco indipendentemente dal fatto che si sia nel mezzo di una partita o nel menù principale. ClosureHandler è stato realizzato ancora una volta utilizzando il pattern Singleton dato che anche in questo caso è sufficiente avere un'istanza di questa classe che si occupi di gestire le varie richieste d'uscita.

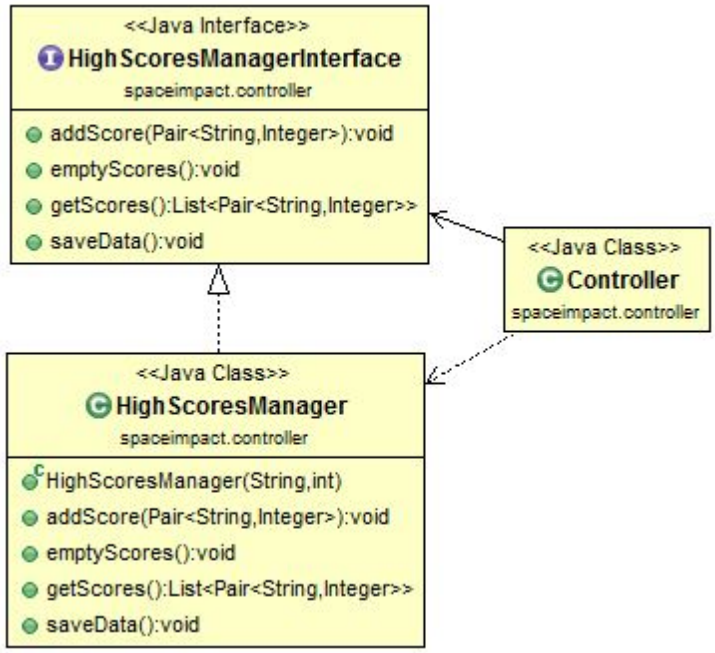
Dal punto di vista grafico è stato scelto dopo una prima analisi di avere un'unica risoluzione fissa per il menù e le altre schermate informative. Si è cercato di tenere conto delle principali risoluzioni presenti sul mercato al giorno d'oggi e si è scelto una risoluzione di 800x800px che permette di funzionare sia sui monitor più moderni sia su quelli più datati.
Al contrario, per il gioco vero e proprio è possibile scegliere una risoluzione 16:9 tra le quattro presenti nel menù "Options". Il sistema controllerà se la risoluzione scelta è valida (ovvero se non è troppo grande per lo schermo attuale) e se si accorge che l'opzione scelta corrisponde alla risoluzione attuale dello schermo passa in modalità "Full Screen".
Tutti gli elementi grafici presenti durante una partita adattano le loro dimensioni a seconda della risoluzione scelta.
E' possibile aggiungere altre risoluzioni in futuro semplicemente cambiando alcune piccole cose nella classe "Options".
Grazie a JavaFX è stato possibile utilizzare un foglio di stile (CSS) per rendere più gradevole l'esperienza grafica del gioco, in particolare l'aspetto grafico di tutti i pulsanti del gioco è stato realizzato completamente in CSS.

- **Lugaresi Secci**

Il Controller coordina Model e View: il diagramma seguente illustra i rapporti tra le classi del package controller (per motivi di chiarezza in questo diagramma mostro solo le interfacce di InputParserInterface e HighScoresManagerInterface e non le implementazioni).



Quando richiesto dalla View attraverso i metodi pubblici dell'interfaccia, il Controller utilizza un HighScoresManager per compiere operazioni sulla lista dei punteggi migliori. Attualmente i punteggi vengono salvati in locale su un file "hiscores" nella stessa directory della applicazione, è tuttavia possibile che si decida in futuro di cambiare il metodo di salvataggio. Ad esempio si potrebbe voler caricare i punteggi su un server online in modo da avere una leaderboard unica per tutti i giocatori. Per questo motivo ho utilizzato il pattern Strategy, che consente di cambiare rapidamente l'implementazione di una famiglia di algoritmi senza dover fare grandi modifiche nelle altre classi.



Sarebbe infatti sufficiente creare una nuova classe che implementi HighScoresManagerInterface e fare in modo che il Controller usi questa nuova classe al posto della precedente: cambiano le strategie di gestione degli High Scores ma non il modo in cui vengono usate dal Controller.

Quando richiesto dalla View il Controller crea un oggetto GameLoop che funge da “orologio” per sincronizzare le operazioni. Il GameLoop crea il Model e a intervalli regolari richiama i metodi delle due interfacce che si occupano di aggiornare la scena e stamparla a video; mediante altri metodi del Controller è possibile mettere in pausa o terminare il GameLoop. Considerando che stampare su schermo ad ogni frame le immagini della scena è una operazione che richiede parecchio tempo ho scelto di gestire il GameLoop sotto forma di Thread, anche in considerazione del fatto che in futuro potrebbe essere necessario rendere molto più pesante l’aggiornamento della scena operato dal model. (Ad esempio far reagire i nemici alle entità circostanti per dare al giocatore l’impressione di combattere contro entità intelligenti).

Ad ogni frame il GameLoop recupera le informazioni sulla scena dal model e ordina in un nuovo thread l’aggiornamento delle immagini su schermo. Una volta terminata l’esecuzione dei metodi del model che avanzano di un frame la scena, il GameLoop attende la terminazione del primo thread, calcola il tempo impiegato e attende un numero di millisecondi tale da rendere costante la frequenza di frame per secondo (“fps”).

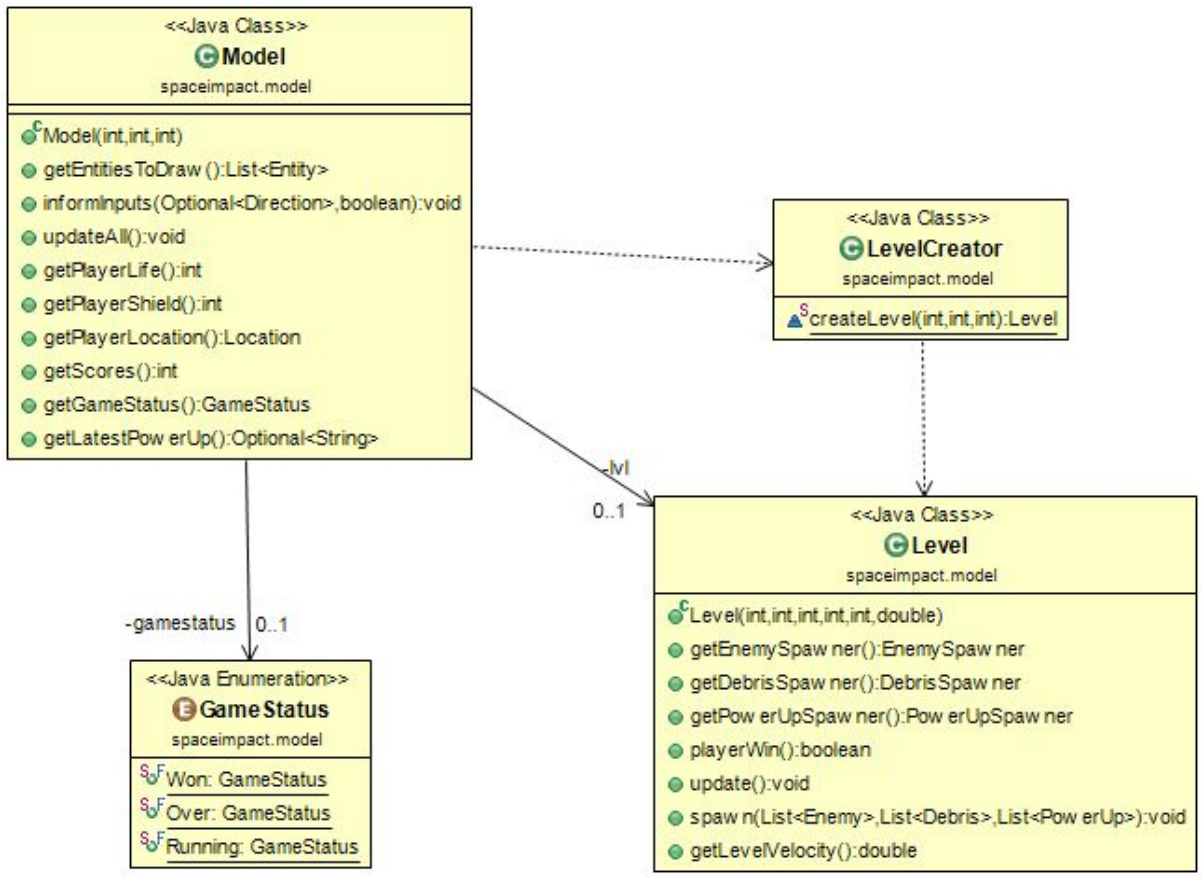
Ad ogni frame il GameLoop deve trasformare gli input “concreti” della View (la pressione sui tasti usati rispettivamente per il movimento e lo sparo) in input più “astratti” utilizzabili dal Model (ossia una direzione di spostamento e la scelta se sparare o meno). Prevedendo la possibilità futura di usare un diverso tipo di input (diversi tasti nella tastiera, o cambiare completamente periferica e usare un mouse o un gamepad) ho deciso anche questa volta di adottare il pattern Strategy creando l’interfaccia InputParserInterface.

Trattandosi di una interfaccia funzionale ho implementato il metodo parseInputs direttamente nel costruttore del Controller usando una lambda. Per cambiare il tipo di input basta implementare in modo diverso il metodo parseInputs (volendo anche in una classe separata invece che nella lambda) oltre che ovviamente cambiare la enum Input.
Ho realizzato inoltre il metodo draw della classe view.DrawEntities e il metodo statico getImage di model.entities.EntityType: il primo metodo viene chiamato dalla View ad ogni frame per stampare su schermo le immagini delle entità, il secondo viene richiamato dal GameLoop per scegliere l'immagine giusta per ogni entità.

- **Giacomini**

MODEL

Il Model si occupa di gestire tutte le entità, applicare gli input utente passati dal Controller alla entità Spaceship ed eseguire tutti i controlli necessari per far funzionare correttamente il gioco.
Il Model è creato quando necessario dentro al GameLoop del Controller.
Esso rappresenta astrattamente, insieme alla classe Level, il livello corrente a cui l'utente sta giocando.
Il model, nel suo costruttore, richiama il metodo statico createLevel della classe LevelCreator, implementata usando il pattern Factory, con i parametri passati dal Controller (difficoltà, identificatore livello e framerate).
Questo metodo crea e inizializza un nuovo Level il quale contiene principalmente delle variabili generali per la definizione del livello, un metodo per verificare se il livello è stato completato e 3 tipi di spawner, già configurati adeguatamente rispetto alla difficoltà dei gioco.

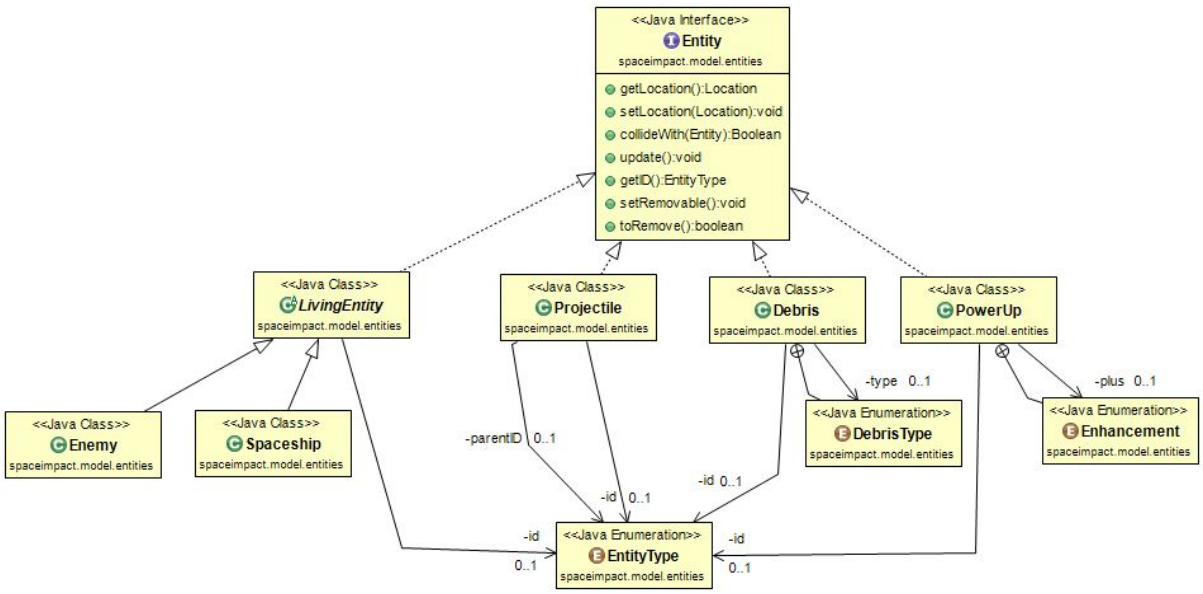


Ad ogni tick del GameLoop vengono richiamati i metodi informInputs, che serve al controller per comunicare gli input da tastiera dell'utente al model, e successivamente il getEntitiesToDraw, che restituisce la lista completa di entità attive nell'istante di gioco, che dovranno successivamente essere mostrate dalla view.

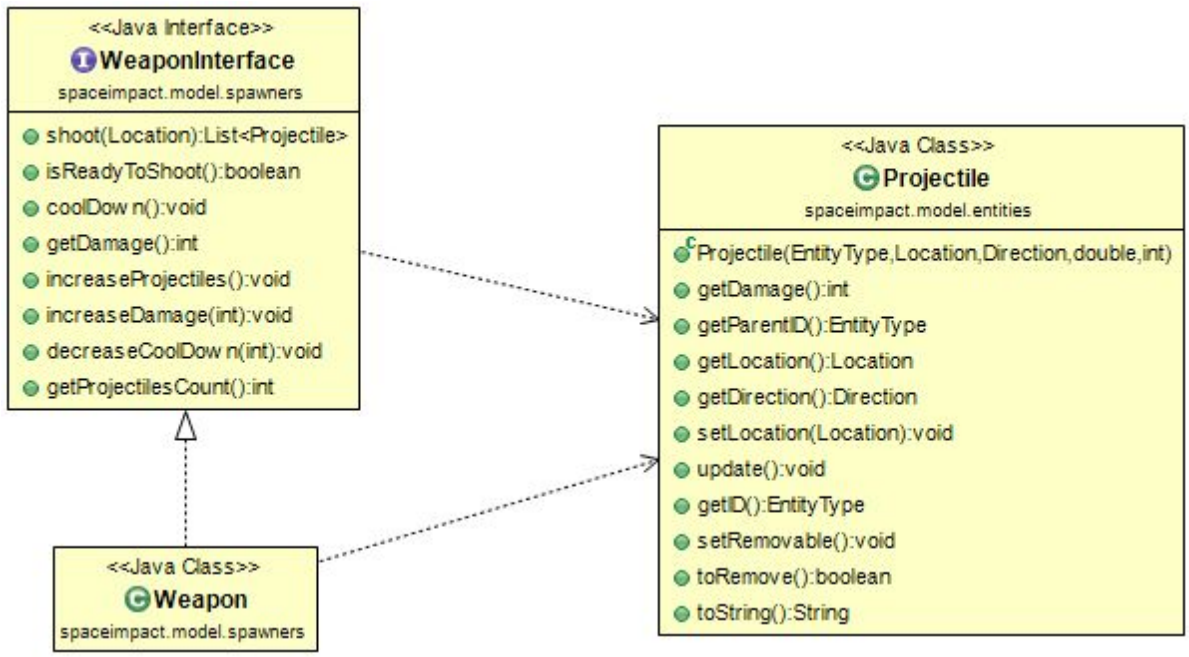
Il model, prima di poter inviare la lista di entità, esegue diverse operazioni per aggiornare lo stato e la posizione di tutte le entità attive.
Come prima cosa è richiamato il metodo updateAll che si occupa inizialmente di aggiornare la posizione di tutte le entità attive e in particolare del player usando i dati forniti dal controller tramite informInputs, e successivamente di richiedere nuove unità agli spawner del Level.
Dopo che l'updateAll è stato completato il model lancia un secondo metodo per controllare se hanno avuto luogo collisioni tra le entità appena mosse.
Nel caso in cui il Model rilevi collisioni genera, nella stessa Location in cui è avvenuto lo scontro, nuove unità Debris di tipo Explosion, nel caso in cui la collisione ha provocato la morte di una o entrambe le entità, di tipo Hit, se una unità ha provocato danno ad un'altra senza distruggerla, o un tipo Sparkle, nel caso in cui il player si sia scontrato con un PowerUp (in questo caso viene richiamato il metodo applyEnhacement per applicare il potenziamento al giocatore).
Finito il controllo sulle collisioni vengono raggruppate e restituite tutte le liste di unità dentro il model.
Come si può intuire, il controllo delle collisioni è la parte computazionalmente più intensa di tutto il model e si è cercato quindi di limitare il più possibile il numero di controlli da eseguire eliminando a prescindere coppie di entità che effettivamente non potrebbero mai collidere, oppure entità già morte.
Per gestire adeguatamente le unità "morte" o non più attive, si è implementato una sorta di Garbage Collector che si occupa di cancellare tutte le entità morte o non più utili da tutte le liste di unità presenti nel model facendo il modo di limitare il più possibile il numero di unità attive.

ENTITIES

Le entità sono state implementate partendo da una interfaccia EntityInterface che definisce metodi generali che tutte le entità dovrebbero possedere.
Tutti i tipi di entità son definiti nella enumerazione EntityType e per ogni entità esiste un metodo per conoscerne il tipo (getID).



Sono definiti due principali gruppi di entità: il gruppo LivingEntity (classe astratta) e il gruppo formato da entità statiche o quelle entità che fan parte dello scenario (Asteroids). Del primo gruppo fanno parte Enemy e Spaceship, che sono entità che possiedono vita e/o scudo, che sono in grado di effettuare azioni complesse come sparare, spostarsi randomicamente o seguendo gli input del giocatore nelle 8 direzioni definite dalla Enumerazione Direction. Entrambe sfruttano la Classe Weapon, implementata seguendo un pattern Factory, che mette a disposizione diversi metodi per potenziare le statistiche dell'arma e un metodo richiamato dalle entità viventi che le permette di generare e restituire nuovo/i Projectile con direzione uguale a quella dell'entità che ha sparato.

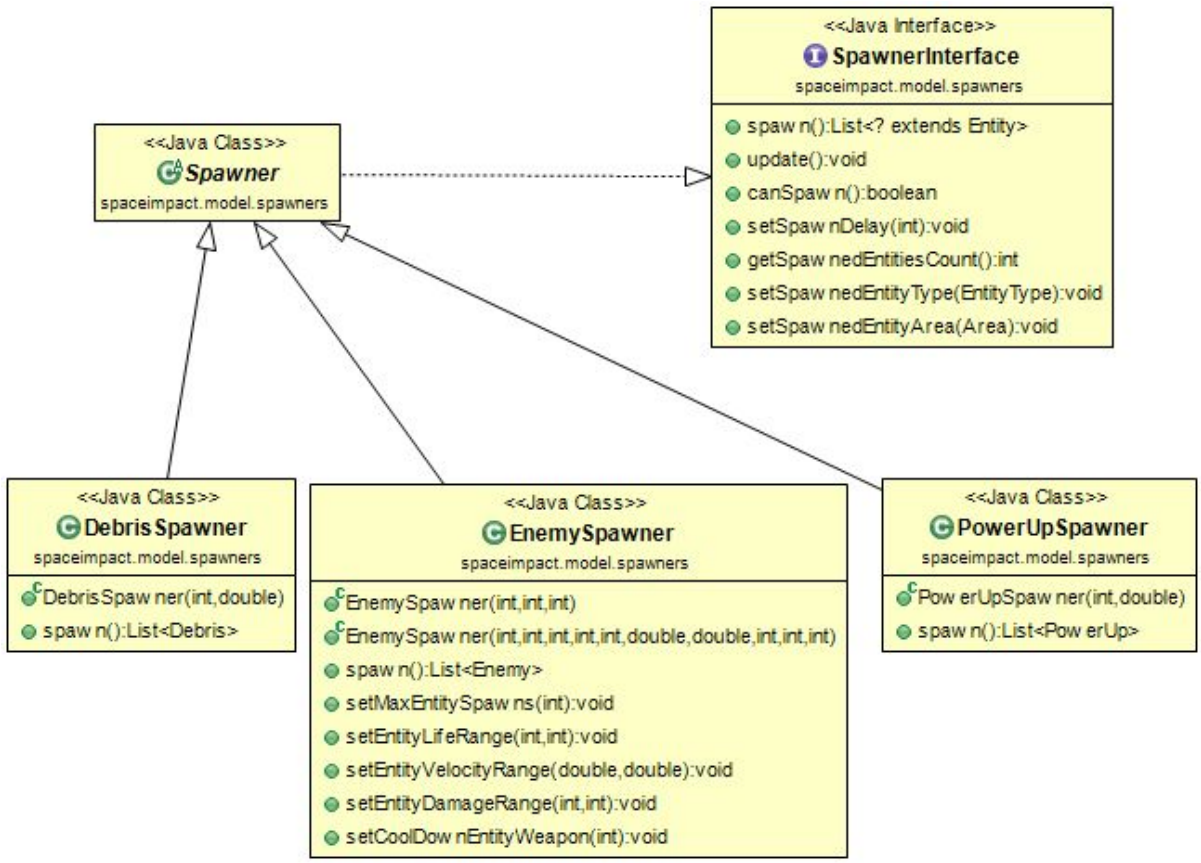


Il secondo gruppo di entità è formato invece da Debris, Projectiles e PowerUp. Sono tutte entità che non possiedono vita e non possono muoversi randomicamente ma solo in una

sola direzione e con una velocità costante. Come detto in precedenza i Projectiles non generati dalla classe Weapon e sono utilizzati dal giocatore per distruggere Enemy o viceversa. I powerup non sono altro che Potenziamenti che se “presi” dal giocatore ne aumentano le statistiche o ne migliorano l’arma. I potenziamenti sono definiti da una enum interna alla classe PowerUp. Infine i Debris possono essere quelle entità generate casualmente come Asteroidi che possono collidere col giocatore, oppure effetti, con lifetime limitato, come Explosions, Hit o Sparkle, rispettivamente generati alla morte di una Entità, alla collisione tra due oggetti o quando il giocatore acquisisce un powerup.

SPAWNERS

Quasi tutte le entità presenti nel model vengono generate attraverso spawner. I metodi di uno spawner generico sono rappresentati da SpawnerInterface che è implementata seguendo un pattern factory nella classe astratta Spawner. I vari tipi di spawner estendono Spawner e ne particolarizzano lo scopo.
Ogni spawner possiede un metodo spawn() che se richiamato genera entità, con alcune proprietà definite dai campi interni allo spawner e altre generate randomicamente entro limiti prestabiliti, e le ritorna sotto forma di lista.



Per lo spawner EnemySpawner son stati implementati anche metodi per cambiare i parametri per lo spawn. Così facendo è stato possibile definire il gioco come una serie, anche infinita, di livelli, in cui, per ognuno di essi, è variabile la forza, il numero e le statistiche generiche dei nemici.

Sviluppo

Testing automatizzato

Abbiamo creato alcuni test JUnit per controllare il funzionamento dei seguenti componenti: model.entities.Enemy, model.entities.Spaceship, model.spawners.Weapon (modifica di vita e scudi, controllo collisioni, sparo e spostamento), controller.GameLoop (creazione, pausa e terminazione di una partita, controllo precisione del framerate) e controller.HighScoresManager (save/load dei punteggi su file).
Abbiamo inoltre testato manualmente l'applicazione per verificare il corretto funzionamento dell'interfaccia grafica nei seguenti sistemi operativi: Windows 7, Windows 8.1, Windows 10, Lubuntu 15.10, Ubuntu 14.04 e Mac OS X 10.10 (a 64 bit); Trisquel 6 (a 32 bit).

Metodologia di lavoro

Il gruppo ha inizialmente discusso insieme la suddivisione in classi delle componenti principali, e si è accordato sui metodi delle interfacce da implementare per la comunicazione tra M, V e C. A questo punto ad ogni membro del gruppo è stata assegnata una certa quota di lavoro (cercando il più possibile di dividere il tutto equamente) e il gruppo ha iniziato a lavorare individualmente sulle rispettive classi.
Nel corso dello sviluppo l'aggiunta di features ci ha costretto alcune volte a rivedere i metodi convenuti nelle interfacce (aggiunte o modifiche in corso d'opera) tornando momentaneamente in fase di design.
Le modifiche locali sono state caricate frequentemente su Bitbucket usando Mercurial in modo che tutti i membri del gruppo avessero una copia aggiornata su cui lavorare.
Occasionalmente si sono formate più "teste" di sviluppo, ma dato che la maggior parte delle classi è stata sviluppata individualmente salvo in rari casi non abbiamo avuto conflitti.
Abbiamo fatto uso di Slack[3] per comunicare e coordinare le modifiche quando necessario.

- **Bonato**

Ho realizzato tutte le classi presenti all'interno del package view, ad eccezione di DrawEntities che ho sviluppato insieme a Nicola Lugaresi Secci. In particolare, per quanto riguarda la classe in questione, io mi sono preoccupato di realizzare la parte relativa alla traslazione dell'immagine di sfondo.
Inoltre ho sviluppato una prima versione dell'ImageLoader poi migliorata da N. Lugaresi Secci.
La scelta delle immagini da utilizzare è stata fatta in gruppo scegliendo quelle che ci sembravano più consone per il nostro gioco.
Nella parte iniziale di analisi c'è stato un confronto con N. Lugaresi Secci (Controller) per decidere in che modo le nostre due parti avrebbero comunicato tra loro e in particolare di che metodi avremo avuto bisogno nelle rispettive interfacce.

[3] https://slack.com/

- **Giacomini**

Mi sono occupato della realizzazione di tutte le classi e interfacce dentro al package model, incluse quelle presenti nei sottopackage model.entities e model.spawners. Il model è stato implementato in modo tale da essere il più possibile indipendente da tutte le altre parti dell'applicazione. Il confronto con gli altri membri è stato minimo. È bastato infatti comunicare al controller quanti e quali tipi di entità fossero presenti nel model e alcuni loro cambi modificabili, come nel caso di PowerUp e Projectile per poter cambiare texture a seconda dell'effetto o della potenza.
N. Lugaresi Secci ha contribuito nell'implementazione di alcuni metodi nelle enumerazioni interne alle classi Debris e PowerUp per far si che per ogni enumerazione fosse possibile conoscere, attraverso un metodo appunto, il nome del file immagine da passare alla view. Un secondo aiuto è stato dato anche nella enumerazione Direction, con l'implementazione di un metodo per poter convertire l'enumerazione in un valore in gradi.

- **Lugaresi Secci**

Ho realizzato in autonomia le classi e le interfacce del package controller, i metodi di view.DrawEntities e model.entities.EntityType relativi alla stampa su video delle entità e la classe utilities.Pair. Purtroppo nella spartizione dei compiti ho sopravvalutato il mio carico di lavoro, e per riequilibrare ho lavorato anche alle classi utilities.ImageLoader, view.InputHandler (con T. Bonato), model.Direction, model.entities.Debris e model.entities.PowerUp (con D.Giacomini).

Note di sviluppo

Per quanto riguarda la View, parte del CSS relativo all'aspetto dei pulsanti è stato preso da vari tutorial presenti su internet per poi essere adattato al nostro applicativo.

Commenti finali

Autovalutazione e lavori futuri

Il tempo non eccessivo a nostra disposizione ci ha impedito di aggiungere parti di programma come avremmo voluto, come i suoni e i boss di fine livello, ma riteniamo che si possano aggiungere nuove features senza grandi sconvolgimenti. Il team di sviluppo è complessivamente soddisfatto del risultato conseguito sia dal punto di vista delle feature realizzate che sul piano della "pulizia" del codice, anche se chiaramente non possiamo definirlo perfetto. Il lavoro di gruppo è stato complessivamente positivo nonostante l'impossibilità nell'incontrarsi di persona durante lo sviluppo dell'applicativo. Come già rilevato nelle parti precedenti di questa relazione c'è ampio spazio per modifiche e miglioramenti successivi (suoni, rudimentali IA, boss di fine livello, livelli non randomici, salvataggio del profilo dell'utente…).

- **Bonato**

Sono abbastanza soddisfatto del mio lavoro specialmente per quanto riguarda l'impatto grafico.
Nello sviluppo della View la cosa a cui ho cercato di dare più importanza è il modo in cui è possibile passare da una schermata all'altra senza grosse difficoltà e devo dire che anche se non sono sicuro che si tratti della migliore soluzione sono contento del risultato ottenuto.
Inoltre sono molto soddisfatto anche di come vengono gestiti gli input dell'utente in quanto permettono un esperienza molto fluida al giocatore.
Il lavoro in gruppo è stato molto positivo e mi ha permesso di capire i motivi per cui la parte di analisi del problema è molto importante. Infatti avendo un idea di come si intende affrontare lo sviluppo ho trovato molto più semplice implementare le relative soluzioni.
Purtroppo non sono riuscito ad implementare alcune cose che avrebbero migliorato l'esperienza di gioco (come il suono) in quanto ho superato le 100 ore previste dal progetto.
Comunque credo che a questo punto sia relativamente semplice come aggiunta e in futuro mi piacerebbe provare ad affrontarla personalmente o insieme al resto del gruppo.

- **Giacomini**

Fin da subito ho capito che era vitale una corretta gestione delle entità e la loro suddivisione in sottocategorie.
Ho ragionato quindi soprattutto sulla gerarchia delle entità e il metodo migliore per poter verificare facilmente le collisioni, con le dovute ottimizzazioni, e cercare un modo per ridurre al minimo il numero di entità nelle collezioni del model, un punto critico in quanto dal model dipendono, in gran parte, le performance dell'applicazione.
Per risolvere questo problema ho implementato, sulla falsa riga di quello di java, un mio particolare Garbage Collector per pulire ad ogni refresh le liste di entità del Model.
Proseguendo ritengo, in relazione al tempo concesso, di aver fatto un lavoro discreto nel complesso, anche se, certamente, ci sono diversi punti in cui è possibile operare miglioramenti.
Per esempio una ulteriore ottimizzazione sarebbe stata quella di inviare al controller solamente le entità che sono effettivamente visibili nell'area di gioco e non tutte quelle attive, sarebbe bastato infatti un controllo per ogni Location di ogni entità.
Sempre per mancanza di tempo non sono riuscito ad implementare metodi per poter inviare al controller gli effetti sonori ad ogni generazione di Debris Explosion, Hit o Sparkle.
Per concludere, mi pare di aver implementato con sufficiente attenzione all'espandibilità il model e tutte le entità in esso e i relaviti spawner e credo che l'organizzazione delle classi e dei miei package risulti sufficientemente buona.

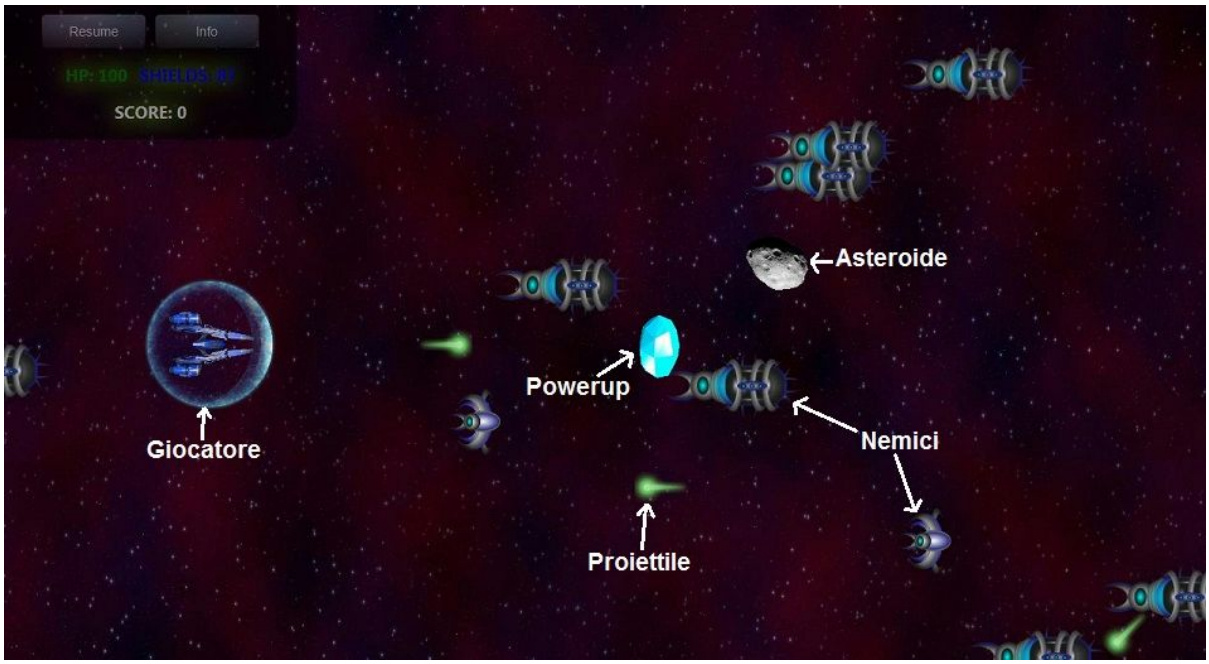
- **Lugaresi Secci**

In qualità di responsabile del Controller ho dovuto confrontarmi spesso con i compagni del team per ritoccare i metodi delle interfacce. Penso che avrei potuto gestire in modi diversi (probabilmente più efficienti) la sincronizzazione del GameLoop con gli altri thread, tuttavia mi reputo soddisfatto delle mie parti di codice e dell'organizzazione generale delle classi.
Reputo che le componenti del programma siano sufficientemente protette grazie all'incapsulamento, e che il codice allo stato attuale permetterebbe l'aggiunta di nuove features senza grandi difficoltà.

Guida utente

All’avvio dell’applicazione viene mostrato il “Menu principale” (vedi figura).

Il primo pulsante (“New Game”) consente di iniziare una nuova partita con le opzioni correnti. “High Scores” mostra la lista dei dieci punteggi migliori realizzati localmente; “Options” consente di modificare la difficoltà del gioco e altre opzioni di tipo grafico (risoluzione finestra e numero di frames per secondo). “Info” riepiloga brevemente i tasti per giocare e cita gli autori del gioco e delle immagini utilizzate (è possibile aprire questa finestra anche durante il gioco). Infine, analogamente al pulsante per la chiusura della finestra, il tasto “Exit” serve per chiudere il gioco.
L’immagine successiva rappresenta un possibile momento di gioco.



Il giocatore può muovere la sua astronave usando i tasti [W], [S], [A] e [D].
Il tasto [Space] (barra spaziatrice) serve per sparare: i proiettili lanciati dal giocatore danneggiano i nemici e gli asteroidi, i proiettili lanciati dai nemici danneggiano scudi e vita del giocatore. Il giocatore è danneggiato anche se tocca asteroidi e nemici.
I nemici si spostano per la schermata sparando: l'aspetto dei nemici e il colore identifica quanto sono potenti, meglio stare lontani da quelli più forti.
In alto a sinistra sono indicati scudi, vita e punteggio del giocatore, insieme a un pulsante per mettere in pausa il gioco e uno per mostrare l'elenco dei comandi.
È possibile mettere in pausa anche premendo [P], tornare al menù principale premendo [BACK SPACE] o uscire completamente dal gioco premendo [ESC].
Occasionalmente appaiono i "Powerup", gemme colorate che potenziano la nave del giocatore se toccate.
Scopo del gioco è distruggere tutti i nemici del livello senza che la propria nave venga distrutta.
Alla fine della partita avrai la possibilità di salvare il tuo punteggio per vederlo nella pagina dei punteggi migliori: si ottengono più punti distruggendo i nemici e giocando a livelli di difficoltà elevata.