RELAZIONE PROGRAMMAZIONE AD OGGETTI

Marty's Adventure

Stefano Babini

Lorenzo Bombardini

Damiano Morandi

Stefano Tafuro

25/06/2021

INDICE

1 ANALISI 3
1.1 Requisiti: 3
1.2 Analisi e modello del dominio: 3
2 DESIGN 5
2.1 Architettura: 5
2.2 Design dettagliato: 5
2.2.1 Lorenzo Bombardini: 5
2.2.2 Stefano Babini: 7
2.2.3 Damiano Morandi: 11
3 SVILUPPO 13
3.1 Testing automatizzato: 13
3.2 Metodologia di lavoro 13
4 COMMENTI FINALI 17
4.1 Autovalutazione e lavori futuri: 17
Appendice A 18
Guida utente 18

1 ANALISI

1.1 Requisiti:

L'obiettivo del progetto è quello di realizzare un gioco di ruolo 2D, chiamato Marty's adventure, ambientato nella cittadina immaginaria di Hill Valley dove sono state girate le scene della celebre trilogia cinematografica statunitense Ritorno al futuro.

I tre scenari del gioco sono rispettivamente adattati dai film della saga, riproducendo la Hill Valley del 1955, del 2015 ed infine del 1895.

Il gioco graficamente è molto simile a un RPG retrò come Pokémon, l'ambiente di gioco infatti sarà parzialmente tridimensionale e gli sprite dei personaggi saranno animati.

Il giocatore avviando una nuova partita dovrà scegliere uno tra i tre i personaggi presenti, il quale si muoverà all'interno dei tre mondi combattendo con degli avversari di difficoltà crescente, gestiti in autonomia dal gioco.

All'interno delle mappe l'utente è incaricato di sfruttare le armi sottratte agli avversari sconfitti in precedenza per aumentare la propria potenza offensiva, battere il nemico finale ed infine accedere al nuovo mondo.

Solo una volta sconfitti tutti gli avversari di un mondo si potrà affrontare il boss e quindi passare a quello successivo (con conseguenti ricompense).

Una volta sconfitto il terzo ed ultimo boss il gioco sarà concluso.

1.2 Analisi e modello del dominio:

Marty's adventure dovrà essere in grado di permettere al giocatore la completa interazione con lo scenario, tali scenari prendono il nome di mappe (map1/2/3).

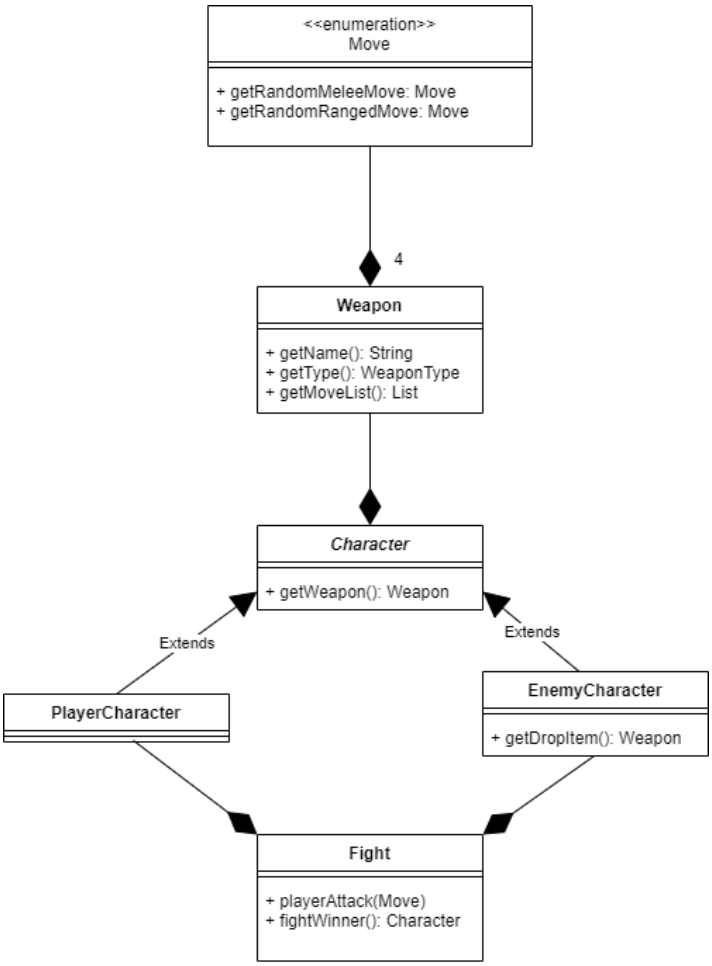
Ciascuna mappa è composta da un numero definito di avversari base, con in nome di Bulli, e un avversario finale, denominato Boss, il quale dovrà essere sconfitto per potere passare alla mappa successiva.

Durante ogni scontro verrà chiesto, al giocatore, di selezionare una mossa (Move) legata al tipo d'arma (Weapon) in suo possesso.
Ad ogni arma nel momento della sua creazione, verrà associato un set di mosse (MoveSet) che rimarrà invariato durante tutto l'utilizzo di essa.

Una volta selezionata la mossa che si vuole eseguire, verrà calcolato il danno e se l'attacco avrà successo, verranno sottratti punti vita al nemico.
Il combattimento proseguirà seguendo uno schema a turni fino alla sconfitta del giocatore (GameOver) o del nemico.

Una volta battuto un nemico, il giocatore verrà chiamato a scegliere se cambiare arma, prendendone una generata dal nemico oppure mantenere la propria.

Dopodiché, se il nemico era un boss, il gioco caricherà immediatamente la mappa successiva, altrimenti il giocatore ritornerà alla mappa corrente.



2 DESIGN

2.1 Architettura:

Marty's Adventure è stato ideato e sviluppato seguendo il pattern architetturale MVC poiché il modello gestisce in modo elegante e ben costruito la logica di dominio dell'applicazione.

Il pattern è basato sulla separazione di tre componenti:

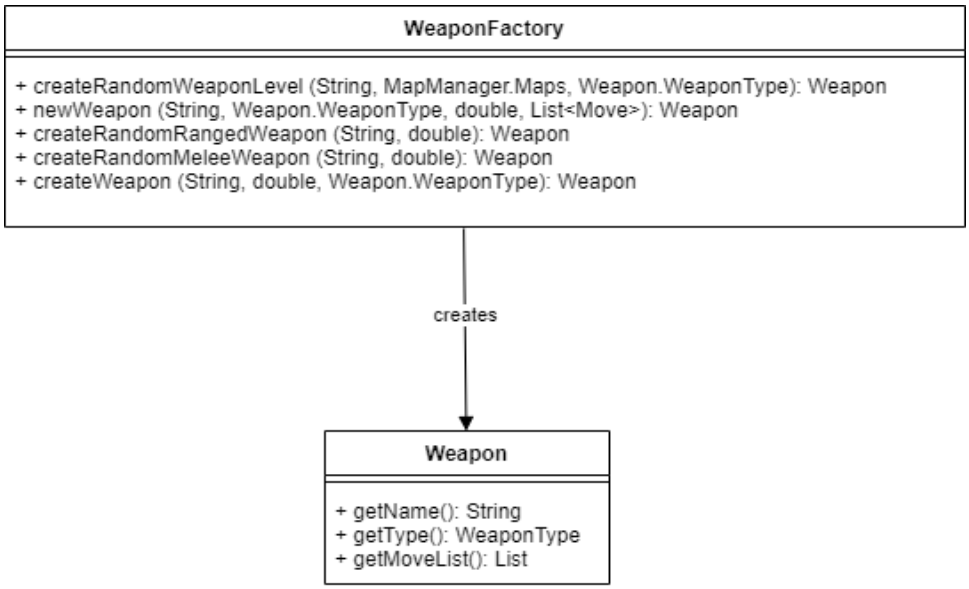
- Model: gestisce i dati e la logica di domino dell'applicazione
- View: gestisce la parte di presentazione ed interazione con l'utente
- Control: permette al giocatore di interagire col gioco

Gli input da tastiera dell'utente vengono gestiti dal PlayerInputProcessor per controllare il personaggio sulla mappa di gioco. Successivamente gli input dal control passano al model il quale si occupa di gestire i dati e le informazioni modificando lo stato in base al nuovo evento. Infine, la view si occupa di rappresentare lo stato corrente sullo schermo.

2.2 Design dettagliato:

2.2.1 Lorenzo Bombardini:

Weapon (Model)



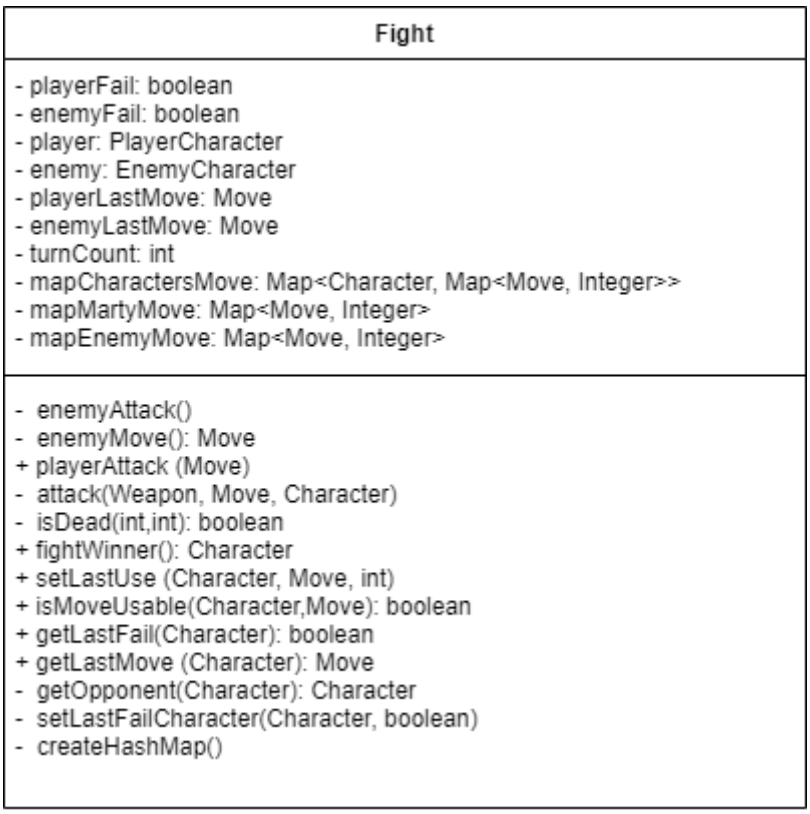
Il sistema per la gestione delle armi utilizza il pattern Static Factory, come da figura 2.1, perciò la creazione di Weapon è gestita da WeaponFactory.

Durante la creazione dell'arma viene associato ad essa una lista di mosse (Move), che non potrà subire variazioni durante l'esecuzione.

WeaponFactory permettere di creare armi personalizzandone gli attributi tramite i propri metodi.
Il metodo chiamato più frequentemente è *createRandomWeaponLevel()* che crea un'arma potendone specificare nome, tipo e la mappa in cui la si vuole generare, il metodo restituisce una Weapon che a seconda della mappa aumenterà il moltiplicatore dei danni (*damageMultiplier)* rendendola più efficace all'aumentare dei livelli del gioco.

Altri metodi pubblici sono *createRandomRangedWeapon()* e *createRandomMeleeWeapon()*, che rendono possibile la creazione di un'arma specificandone il tipo (Ranged, Melee)

Fight (Model)



Quando CombatGameScreen crea una nuova istanza di Fight, viene generata una map, mediante il metodo privato *createHashMap()*, contente per entrambi i Character (Player, Enemy), tutte le Move della propria arma e l'ultima volta che ognuna di esse viene utilizzata.
Questa map viene utilizzata per effettuare il calcolo del periodo di ricarica di ogni Move, e di conseguenza capire se essa possa essere utilizzata o meno dal Character.

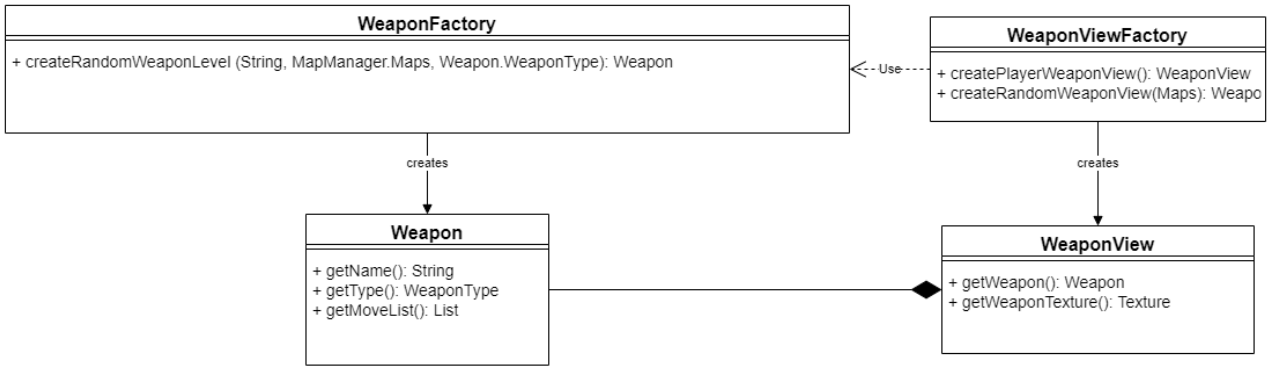
Nel momento in cui il primo attacco viene scagliato dal Giocatore, sarà il metodo *playerAttack()*, al quale verrà passata la Move dal CombatGameScreen, a chiamare sucessivamente il metodo *enemyAttack()*.

playerAttack() e enemyAttack() vanno entrambi a chiamare il metodo privato attack() che ha il compito di capire, tramite il metodo importato Move.*testFailure()* se la mossa ha successo e nel caso infliggere i danni (*Move.damage * Weapon.damageMultiplier*).

attack() prima di infliggere un danno determina, tramite il metodo privato *isDead()*, se l'attacco risulterà letale. Qualora il danno risultasse fatale il metodo *fightWinner()* restituirà il vincitore.

2.2.2 Stefano Babini:

Weapon (View)



Con la classe WeaponView abbiamo implementato la visualizzazione durante il combattimento dell'arma dei Character.

WeaponView contiene al suo interno un'istanza di Weapon e la texture corrispondente.

Per generare delle WeaponView in modo semplice abbiamo fatto uso di una classe che segue il pattern Static Factory chiamata WeaponViewFactory.

WeaponViewFactory al suo interno contiene le armi rappresentabili dal gioco e le genera in modo statico appoggiandosi a WeaponFactoy per la creazione dell'istanza di Weapon.

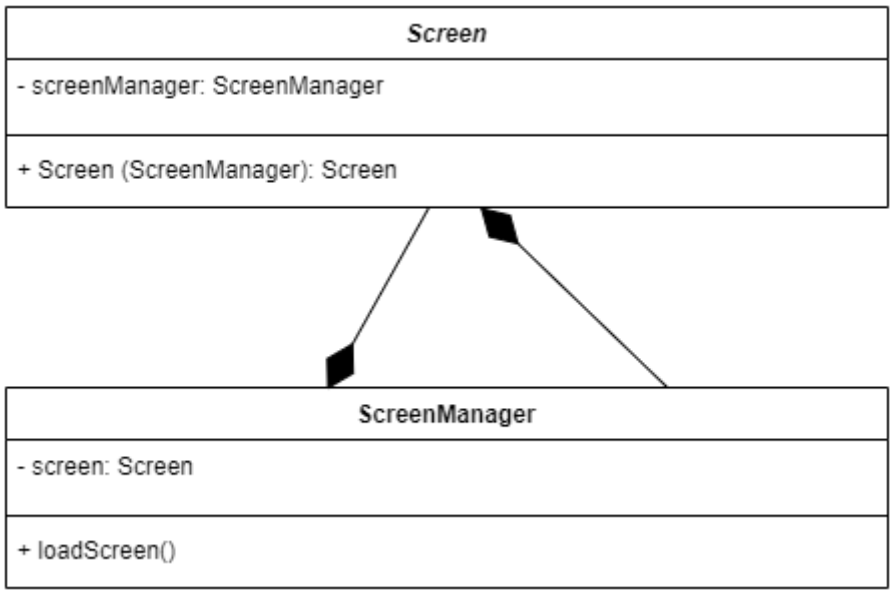
WeaponView espone due metodi importanti che sono *getWeapon() e getWeaponTexture()* i quali vengono utilizzati rispettivamente da Fight e CombatScreen per il calcolo e la visualizzazione del combattimento.

Screen (View)

Per la visualizzazione degli elementi del gioco abbiamo incontrato le seguenti difficoltà:

- Cambiare schermata ad ogni combattimento
- Mantenere in background la mappa corrente
- Cambiare mappa al termine di ogni livello
- Liberare dalla memoria di un combattimento appena terminato

Per risolvere queste problematiche abbiamo deciso di dividere il gioco in vari Screen che è un'interfaccia fornita dalla libreria LibGDX.
Per ogni elemento che richiedeva una ristrutturazione grafica degli elementi visualizzati a schermo abbiamo implementato uno Screen.

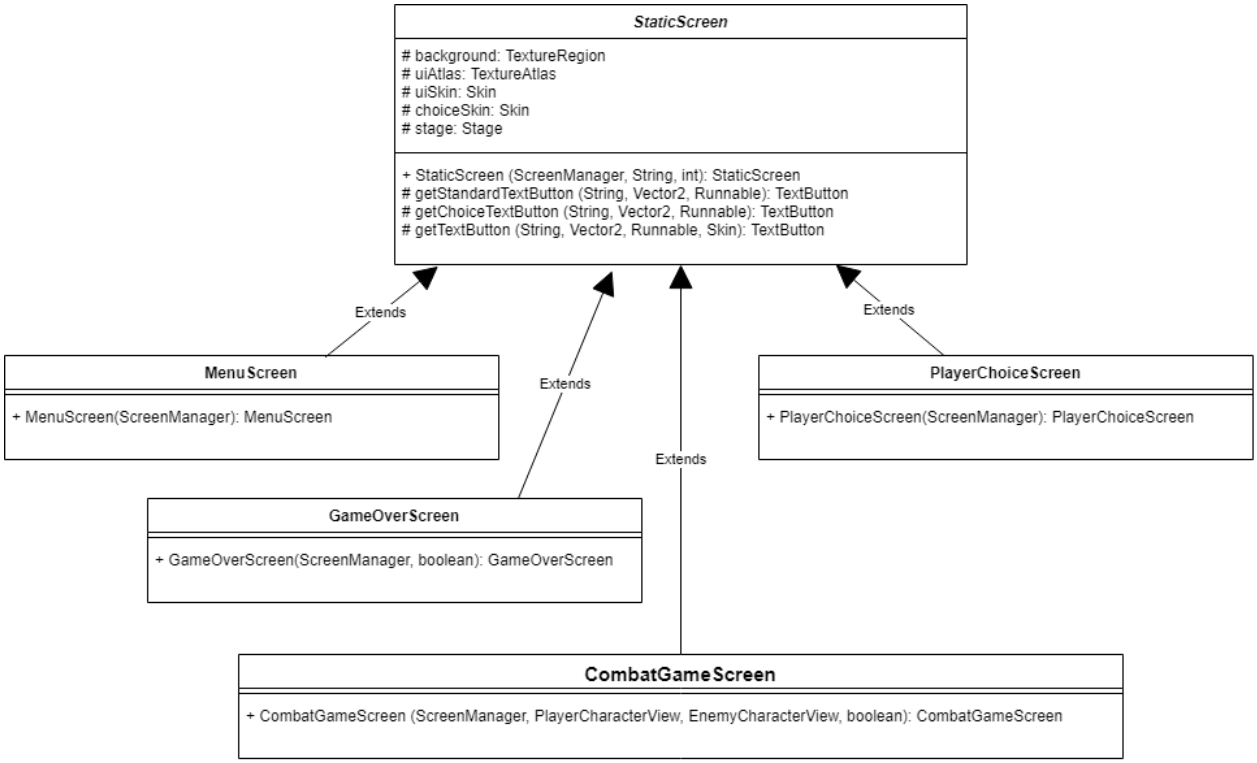


Per gestire l'alternanza su schermo degli screen abbiamo implementato uno ScreenManager. Inizialmente la classe prevedeva solo metodi statici e si limitava a gestire quale screen era attualmente visualizzato. Questo semplificava la gestione di quale screen visualizzare ma rendeva difficoltoso liberare la memoria dagli screen che non utilizzati. Quindi abbiamo deciso di passare, come attributo, ad ogni screen creato l'istanza di ScreenManager per poter chiamare la visualizzazione di altri screen al momento opportuno e allo stesso tempo permettere a ScreenManager di distruggere gli screen non più utilizzati.

Screen manager è quindi una classe che è molto simile al pattern creazionale Simple Factory ma oltre a creare gli Screen si occupa anche della loro visualizzazione e distruzione.

StaticScreen (View)

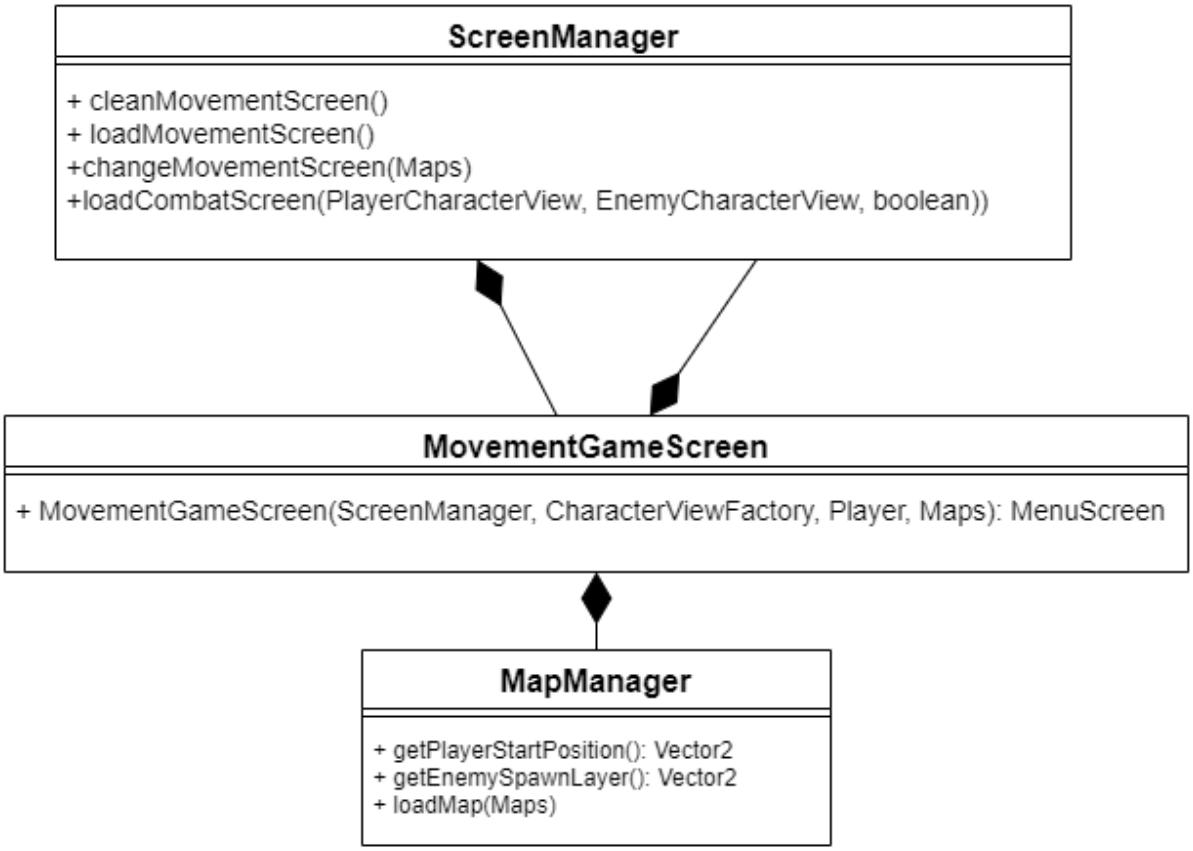
Per tutti gli screen che non richiedono un input da tastiera e creano una UI basata su bottoni e label abbiamo creato una classe astratta StaticScreen, che implementa molti metodi comuni tra questi, questo permette anche di definire uno stile grafico coerente tra tutti gli screen. Come lo stile grafico dei bottoni e dei titoli.



StaticScreen è stato esteso dalle seguenti classi:

- MenuScreen, primo screen visualizzato all'avvio del gioco permette l'uscita dal gioco, regolazione del volume e accesso al PlayerChoiceScreen
- PlayerChoiceScreen, permette la selezione del personaggio giocabile
- GameOverScreen, comunica il game over, varia in caso di vittoria o sconfitta
- CombatGameScreen, visualizza il combattimento in corso tra due Character

MovementGameScreen (View)



MovementGameScreen è la classe che si occupa del render della mappa, dello spawn dei personaggi e della gestione delle collisioni. Viene istanziato da ScreenManager e come argomento gli viene passata la mappa da visualizzare. La classe si appoggia a MapManager per il caricamento della TiledMap.

Questa, tramite la CharacterViewFactory crea e posiziona un Enemy per ogni rettangolo presente in EnemySpawnLayer che è un attributo della mappa corrente. Similmente, mette il Player alla sua posizione iniziale e il Boss nella sua posizione finale. Inoltre, si occupa anche di chiamare *loadCombatScreen()* di ScreenManager quando avviene una collisione con un Enemy e di conseguenza viene creato un CombatScreen.

La collisione con il Boss è gestita solo quando tutti gli Enemy della mappa sono stati sconfitti. Una volta sconfitto il Boss, c*hangeMovementScreeen()* istanzierà il MovementGameScreen con la mappa successiva e libererà la memoria dalla mappa corrente.

2.2.3 Damiano Morandi:
Character (View):

La view character si occupa della rappresentazione a schermo dei character model.

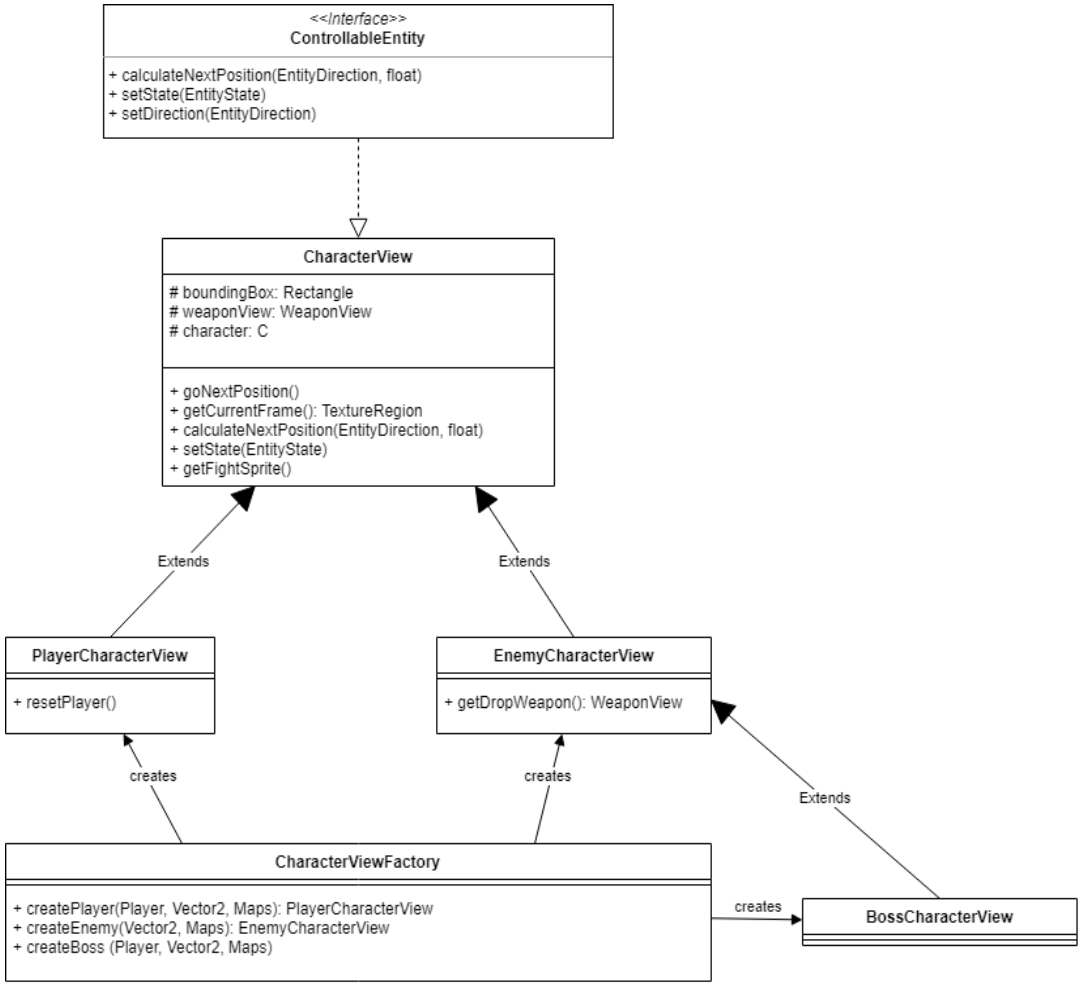
È composta da CharacterView e le sue specializzazioni, CharacterViewFactory e le tre classi di supporto: MapData, AnimationPack e l'enumerazione *Player*.

- CharacterView è una classe astratta, la quale contiene al suo interno la maggior parte della logica condivisa dai personaggi. In particolare: tiene traccia della posizione sulla mappa, l'accelerazione, le collisioni, la direzione e la gestione delle animazioni con relative texture che sono strettamente legate allo stato del movimento. Implementa inoltre l'interfaccia ControllableEntity, che definisce una classe la cui posizione sulla mappa è controllabile dalle classi specificate nel controller.
- PlayerCharacterView estende CharacterView aggiungendo la possibilità di cambiare l'arma del giocatore. Essa implementa la logica per gestire il caricamento e scaricamento delle texture e specificando i punti vita, la velocità e l'accelerazione massima dei personaggi giocabili.
- EnemyCharacterView deriva anch'essa da CharacterView, aggiungendo accelerazione e velocità massime dei nemici del gioco, nonché la view dell'arma che il personaggio rilascia una volta sconfitto.
- BossCharacterView aggiunge ad EnemyCharacteView la gestione del caricamento e disposizione delle texture del personaggio.

I costruttori di queste classi sono package-private, e le loro istanze essere ottenute attraverso una CharacterViewFactory.

Questa contiene i percorsi alle texture dei vari personaggi, i punti vita dei nemici e quale dev'essere il boss nemico per ogni personaggio giocabile, il tutto indicizzato per mappa di gioco.

I metodi *createPlayer()*, *createEnemy()* e *createBoss()* allocano i modelli e le view degli omonimi personaggi con le texture relative alla mappa specificata.



Per le classi di supporto:

- AnimationPack permette di sezionare una texture secondo i parametri specificabili al costruttore e accedere all'animazione e frame appropriati per all'istante di accesso.
- MapData raccoglie i dati associati con una mappa da CharacterViewFactory.
- Player elenca in una enumerazione i personaggi giocabili.

Entity (Controller)

L'entity controller si occupa di gestire l'input da tastiera dell'utente e tradurlo in input utili per interfacce implementabili da classi di più alto livello.

ControllableEntity è un'interfaccia implementabile per definire un'entità che è può essere orientata in uno spazio 2D in una direzione (su - giù - destra - sinistra) impostabile con *setDirection()*, che può essere messa in uno stato di movimento o meno con *setState()* e che calcoli la propria prossima posizione dopo un cambiamento di stato invocando *calculateNextPosition()*.

Ad operare su essa è il singletone della classe PlayerInputProcessor, la quale si frappone tra il framework di LibGDX e la ControllableEntity del giocatore traslando le pressioni e i rilasci dei tasi W, A, S e D in direzioni e cambiamenti di stato.

3 SVILUPPO

3.1 Testing automatizzato:

Il gruppo ha effettuato il testing automatizzato utilizzando versione 5.7.1 del framework JUnit

per la verifica del funzionamento di diversi componenti chiave.

Nello specifico troviamo:

- Per la gestione dei combattimenti (testFight, testMove)
- Per la corretta creazione delle armi (testWeapon, testWeaponFactory)
- Per il corretto caricamento delle mappe (testToolbox, testMapManager) per la loro implementazione abbiamo deciso di prendere spunto da una classe presente su GitHub (vedi 3.3 feature avanzate)
- Per la gestione del input (testPlayerInputProcessor)

3.2 Metodologia di lavoro

Durante la fase preliminare del progetto abbiamo cooperato nella progettazione di un primo UML, il quale prevedeva una divisione delle classi ben definita in cui ognuno di noi era chiamato a lavorare autonomamente.
In seguito allo sviluppo degli Assets e l'implementazione delle classi precedentemente suddivise, abbiamo collaborato al fine di correggere e migliorare alcuni aspetti dei lavori svolti singolarmente, ultimando il progetto grazie a un reciproco aiuto.

Per lo sviluppo del progetto su Github si è adottata una variante della metodologia proposta nelle slide di lezioni.

Localizzato un repository principale, ogni studente ha creato un proprio branch sui cui produrre le classi e funzionalità decise nelle periodiche riunioni: il detentore del repository ha creato un branch development, mentre tutti gli altri hanno creato una fork del progetto e hanno lavorato sui rispettivi master.

Una volta che un obbiettivo era stato raggiunto, lo studente creava una pull request (PR) su Github dal proprio branch verso il branch master del repository principale. Il detentore di questi procedeva a fare una revisione delle modifiche proposte e decideva se accettare la pull request o richiedere dei cambiamenti prima della merge sul ramo principale allo scopo di garantire che il ramo master risultasse sempre funzionante e compilasse correttamente.

Una volta soddisfatti i requisiti di qualità del codice della PR, il detentore del ramo principale accettava la Pull Request. Successivamente, gli altri studenti effettuavano una rebase del proprio ramo sul master principale, in modo da operare sempre su una versione aggiornata e funzionante del codice.

Babini Stefano

Durante la fase di progettazione ha preso in carico la parte riguardante il movimento del personaggio, il caricamento delle mappe la schermata di combattimento.

Finita la parte di progettazione di gruppo ha sviluppato in autonomia la parte del MapManager, confrontandosi spesso con Morandi, incaricato del Toolbox per quanto riguarda il caricamento dell'assets. Dopo che i test hanno dato esito positivo e le classi indispensabili per il MovementGameScreen erano state implementate ha iniziato lo sviluppo di quella classe.

Durante lo sviluppo di MovementGameScreen sono state effettuati vari refactor per rimuovere porzioni di codice duplicati e alcune correzioni insieme a Morandi riguardo alcuni metodi di CharacterView.

Per velocizzare la creazione dei personaggi ha poi creato la CharacterViewFactory inizialmente non prevista, ma successivamente discussa e approvata dal gruppo.

Successivamente ha iniziato e terminato lo sviluppo di CombatGameScreen e ClasseScreenManager per gestire l'alternanza delle due schermate.

Ha sviluppato anche il MenuScreen e PlayerChoiceScreen, in seguito al refactor del codice da parte di Morandi che ha creato la classe astratta StaticScreen a partire dai metodi di CombatGameScreen.

Successivamente è stato deciso di arricchire il CombatGameScreen con le immagini delle armi, questo ha richiesto la creazione di WeaponView e WeaponViewFactory che estendono e utilizzano le classi già implementate in precedenza da Bombardini

In conclusione, è stato deciso di far comparire un banner all'ingresso di ogni nuova mappa e la musica, quindi ha creato la classe WorldBannerFactory e implementato il suo utilizzo in MovemenGameScreen e la classe MusicController.

Alla fine del progetto si è dedicato al testing e debugging dell'applicativo per quanto riguarda il ridimensionamento delle finestre e il movimento del personaggio.

Bombardini Lorenzo

Durante la fase di progettazione ha preso in carico la parte riguardante la gestione del combat system e di tutti i suoi componenti

In seguito alla fase di progettazione ha sviluppato, in completa collaborazione con Tafuro, tutti gli asset (mappe, personaggi, schermate di combattimento) in modo da renderli subito disponibili agli incaricati della view.

Successivamente ha implementato le classi legate al combattimento, progettate in precedenza.

Per poter sviluppare la classe principale Fight ha dovuto implementare e testare le classi che ne componevano lo svolgimento (Move, Weapon e WeaponFactory).

Dopo che i test hanno dato esito positivo e le classi indispensabili per il Fight erano state implementate ha iniziato lo sviluppo di quella classe.

La classe Fight una volta terminata è stata aggiornata frequentemente per poter adattarsi al meglio al gioco. Grazie alle attente richieste di Babini è stato possibile definire una classe in grado di gestire al meglio i combattimenti.

Successivamente è stato deciso di modellare diversamente gli attributi di Move andando a correggere alcuni errori commessi in fase di progettazione e di conseguenza migliorandone l'implementazione

In conclusione, si è dedicato a bilanciare i combattimenti all'interno del gioco, cercando di aumentarne la difficoltà proporzionandola ai livelli.

Alla fine del progetto testando l'applicativo ha riscontrato alcuni errori commessi nella fase di modellazione degli asset e di conseguenza ha ricostruito alcune CollisionBoxs che causavano problemi grafici.

Morandi Damiano

Detentore del repository principale su github, nelle fasi iniziali ha creato e configurato l'ambiente di sviluppo del progetto, per poi occuparsi di revisionare le PR verso il master principale per tutta la durata del progetto.

Si è inoltre dedicato allo sviluppo dei componenti riguardanti l'input da tastiera (PlayerInputController e ControllableEntity) e ha contribuito inizialmente al pacchetto character della view implementando CharacterView e AnimationPack, e parte del pacchetto screen implementando GameOverScreen.

Si è occupato poi della gestione delle risorse, sviluppando Toolbox e condotto vari refactor nei pacchetti Screen (introducendo StaticScreen) e Character (introducendo BossCharacterView e MapData) per assicurare che le risorse manualmente gestite del framework LibGDX fossero correttamente liberate e cercare di migliorare l'organizzazione del codice e rimuovere parti duplicate all'interno dei pacchetti.

Ha introdotto le classi di supporto nei test GdxTestRunner e WaitableRunnable.

Tafuro Stefano

Si è occupato della creazione degli asset in collaborazione con Bombardini e dell'implementazione delle classi model character

3.3 Note di sviluppo

Feature avanzate utilizzate:

Babini Stefano:

- Utilizzo libreria esterna LibGDX
- Utilizzo e modifica dei file di build di Gradle
- Lambda expressions
- Implementazione libreria esterna gdx-testing

Bombardini Lorenzo:

- Utilizzo ***List*** per la gestione dei *MoveList* di ogni arma
- Utilizzo ***HashMap*** per la gestione del sistema di ricarica di una *Move*
- Utilizzo factory design pattern per la creazione delle armi

Morandi Damiano:

- Utilizzo libreria esterna LibGDX
- Sviluppo di un semplice Test Runner Junit 5 con Future per eseguire certi test nel thread di rendering OpenGL
- Utilizzo di costruttori statici
- Utilizzo del pettern singletone
- Gestione dinamica delle risorse a gestione manuale di LibGDX adottando una strategia di ownership
- Aggiunta di dipendenze con gradle

Fonti di ispirazione e materiale pubblico utilizzato:

- Colonna sonora: https://www.youtube.com/watch?v=XgUx0pWo4rM&t=79s
- Design concepts for sprites: https://twitter.com/ahutton8/status/657227626004787201
- Libro consultato durante alcune fasi dello sviluppo: https://www.amazon.it/Mastering-LibGDX-Game-Development-Patrick/dp/1785289365/ref=tmm_pap_swatch_0?_encoding=UTF8&qid=1624611053&sr=8-1
- Libreria adattata per toolbox testing: https://github.com/TomGrill/gdx-testing
- Design utilizzato come modello per lo sviluppo 2D: https://3dwarehouse.sketchup.com/model/aba2af0135571d0b2eaf6afcc2dcd556/Back-to-the-Futures-Hill-Valley-Courthouse-Square?hl=it

4 COMMENTI FINALI

4.1 Autovalutazione e lavori futuri:

Stefano Babini

Nel complesso mi ritengo soddisfatto della riuscita del progetto. Come gruppo, siamo riusciti ad implementare tutte le funzionalità minime concordate, e siamo riusciti anche a implementare parte delle funzionalità opzionali. Personalmente ritengo di aver svolto il mio compito adeguatamente contribuendo in modo significativo alla riuscita del progetto. Ho necessità di fare ancora esperienza con i linguaggi OO, in particolare per quanto riguarda la pulizia e chiarezza del codice, che rischia di diventare caotico. A prescindere dalle difficoltà riscontrate, in generale reputo questa esperienza positiva, in quanto mi ha dato la possibilità di lavorare in team, cosa che non era mai stata possibile e mi ha dato l'occasione di mettermi alla prova in nuovi ambiti che difficilmente si riescono ad affrontare con brevi esercitazioni. Inoltre, ho avuto modo di sperimentare ed utilizzare in maniera sempre più consapevole il linguaggio Java, la cui conoscenza e padronanza offre grandi potenzialità ed utilità in molti ambiti.

Lorenzo Bombardini

Sin da quando abbiamo deciso il tema del progetto sono subito stato entusiasta poiché sono un grande appassionato della trilogia, sicuramente questo mi ha aiutato a svolgere il mio compito al meglio puntando ad ottenere un buon risultato. Terminato il progetto mi posso ritenere soddisfatto, personalmente credo di aver svolto il mio compito adeguatamente, cercando di implementare al meglio le classi a me assegnate. Ritengo che, come team, siamo stati in grado di cooperare al meglio, puntando a dividere in modo equo tutta la mole di lavoro. Ciononostante, ho ancora molto da imparare per quanto riguarda il lavoro in gruppo, poiché mi sono trovato spesso discutere con i membri del mio team a causa di opinioni divergenti. Il progetto nel suo complesso mi ha aiutato sicuramente, evidenziando problemi organizzativi che con esercitazioni di più breve durata non sarebbero emersi. Reputo questo progetto fondamentale per l'apprendimento del linguaggio Java e del corso di OOP.

Damiano Morandi

Seppur l'andatura del progetto è stata un attimo più accidentata di quanto mi ero inizialmente aspettato, alla fine siamo riusciti a raggiungere gli obbiettivi previsti e direi di potermi ritenere soddisfatto del prodotto finale.

Questo progetto è stato sicuramente utile per imparare collaborare in remoto durante questo difficile periodo di pandemia, dimostrandomi, nella pratica, quanto sia necessario creare un piano di sviluppo e una definizione dei vari componenti di un progetto prima di iniziare a stendere il codice.

Stefano Tafuro

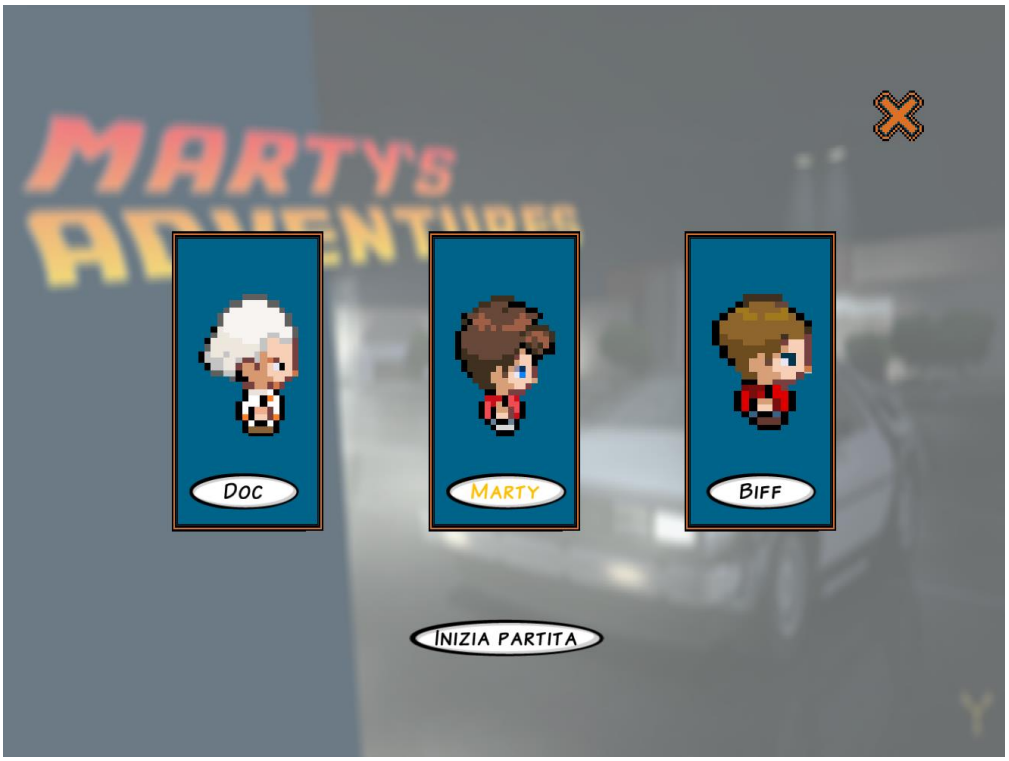
Purtroppo, il mio lavoro non è stato alla pari di quello dei miei colleghi, infatti posso dire di aver seguito solo la parte di creazione degli asset e l'implementazione delle classi model del Character. Ciononostante, il gruppo è stato in grado di coprire egregiamente le mie mancanze e a portare a termine la deadline del progetto.

APPENDICE A

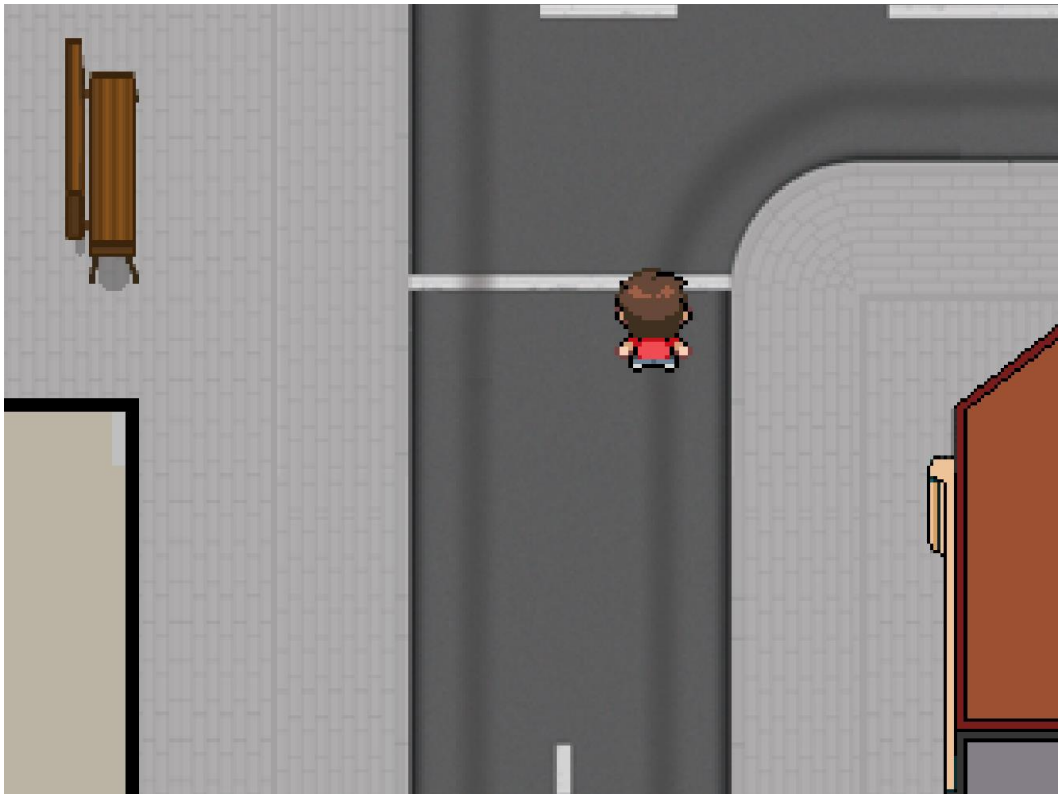
Guida utente

Il gioco si apre con il menu dal quale è possibile iniziare una nuova partita, uscire o regolare le opzioni.

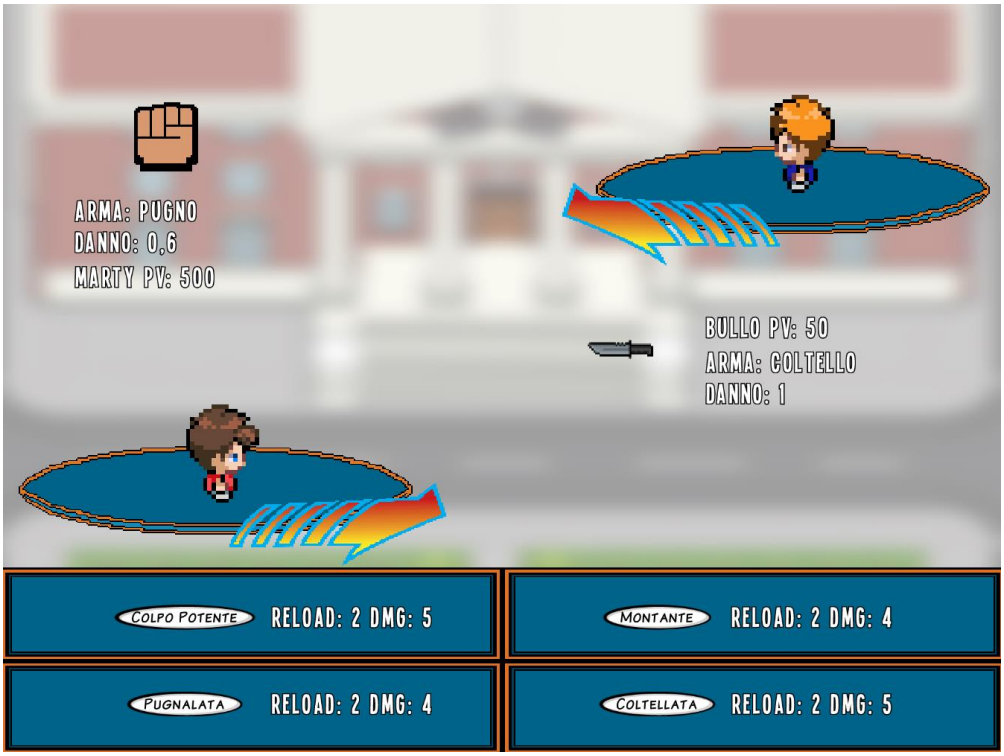
Per ogni partita è possibile scegliere uno dei tre personaggi giocabili, i nemici cambiano di conseguenza al personaggio scelto



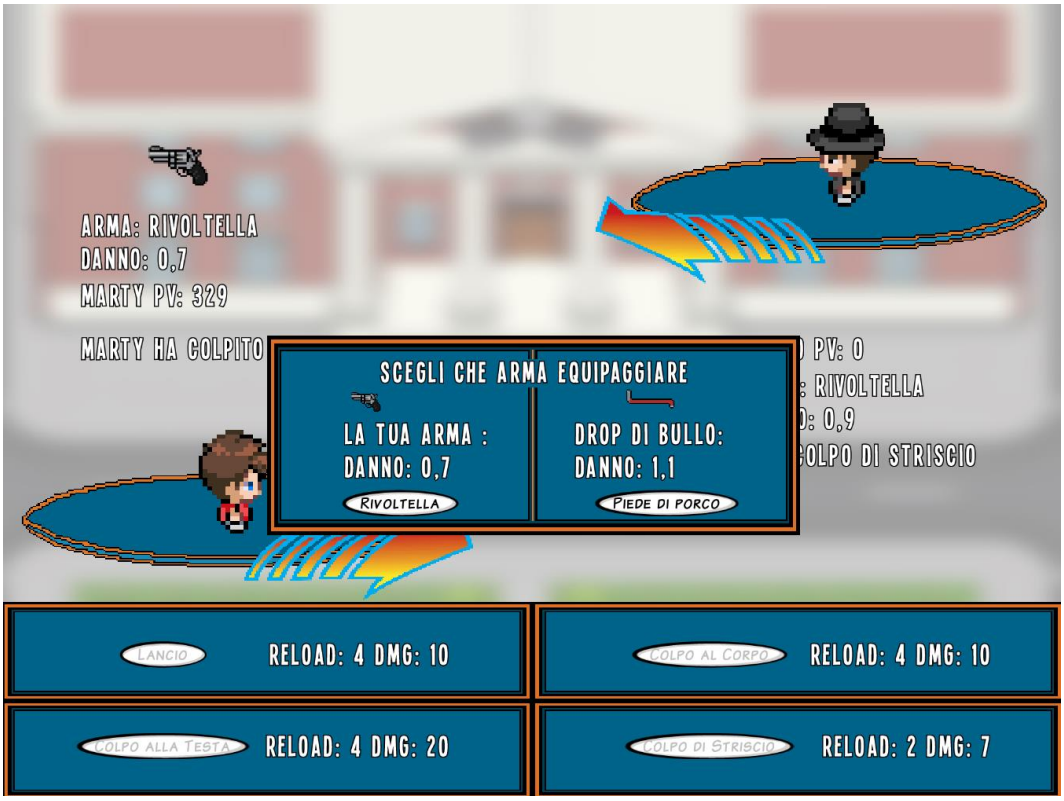
Nella schermata di movimento ci si muove utilizzando i tasti W, A, S e D, alla collisione con un nemico si entra in combattimento. Il Boss di ogni livello è affrontabile solo dopo aver sconfitto tutti i Bulli del livello



Nella schermata di combattimento tramite il mouse si seleziona con quale mossa attaccare l'avversario



Al termine del combattimento è possibile scegliere quale arma equipaggiare: tenersi la propria oppure equipaggiare il drop del nemico



In caso di sconfitta verrà visualizzata la seguente schermata e sarà possibile tornare al menù per iniziare una nuova partita