Relazione Programmazione ad Oggetti

Floppy Floppa

Salvatore Antonio Addimando, Benedetta Pacilli,
Valentina Pieri, Cristina Zoccola

25 Aprile 2022

Indice

1

Capitolo 1

Analisi

Floppy Floppa si ispira al gioco arcade *Flappy Bird* che nell'anno 2014 ottenne così tanto successo da diventare il gioco gratuito più scaricato sull'*Apple Store*.
Nella nostra versione, c'è un floppa a saltare lungo una mappa infinita a scorrimento orizzontale, quest'ultima però contiene più insidie di quella originale. Abbiamo infatti sviluppato ostacoli mobili e malus per intralciare il protagonista ma anche booster per aiutarlo.
Lo scopo del gioco è percorrere più metri possibili nella mappa, così da ottenere *coins* spendibili in uno *shop*, dov'è possibile acquistare nuove *skins* per il protagonista e nuove *sceneries* per lo sfondo della mappa.

1.1 Requisiti

Requisiti funzionali

- **Menù principale:** all'apertura del gioco viene mostrato il menù principale, dal quale si può avviare una nuova partita, accedere allo *shop*, alla *leaderboard*, al *tutorial*; infine si possono cancellare i dati di salvataggio, scegliere l'immagine per la *skin* del personaggio principale e quella per il *background* della mappa e uscire dal gioco.
- **Mappa di gioco:** all'avvio della nuova partita viene creata la mappa, composta da un *background* che scorre fino al termine della partita. Lungo la mappa si incontrano i vari ostacoli: quelli fissi e quelli mobili. Entrambi questi ostacoli, in collisione con il personaggio principale, ne provocano la

morte e, di conseguenza, la fine della partita. Lungo la mappa possono *spawnare* altre due entità: malus e booster. Vi sono diversi tipi di malus, ognuno di questi provoca una conseguenza spiacevole volta a mettere in difficoltà il personaggio, quando questi vi entra in collisione. Diversi tipi di booster, invece, alla collisione producono un qualche tipo di aiuto. I metri percorsi dal personaggio nella mappa, a fine partita vengono convertiti in *coins* spendibili dello *shop*.

- **Character:** il personaggio principale, caratterizzato da una skin e dall'abilità di muoversi lungo la mappa saltando. Se entra in collisione con un ostacolo fisso o mobile, muore e fa terminare la partita.
- **Shop:** il menu dello *shop* permette di visualizzare i *coins* posseduti e di spenderli in nuove *skins* per il personaggio o nuovi *backgrounds* per la mappa di gioco. Tramite il menù dello *shop* si può tornare al menù principale.
- **Leaderboard:** nel menù della *leaderboard* vengono visualizzati i punteggi, in metri percorsi, dei giocatori. Di ogni giocatore viene visualizzato solo il loro *best score*, ottenuto nelle partite svolte. Tutti i vari *best score* vengono mostrati nella *leaderboard* in ordine crescente. I dati sul *best score* vengono presi da un file di salvataggio. Tramite il menù della *leaderboard* si può tornare al menù principale.
- **End-Of-Game Menu:** viene mostrato a fine partita. Tramite questo menù si vedono i *coins* posseduti prima dell'inizio dell'ultima partita svolta e i *coins* di cui si dispone al suo termine. I *coins* guadagnati derivano principalmente dai metri percorsi in partita ma possono venire sottratti o incrementati, da rispettivamente una collisione con un determinato malus o una con uno specifico booster. Tramite questo menù si può tornare al menù principale oppure accedere sia allo *shop* che alla *leaderboard*.
- **Tutorial:** il tutorial viene autonomamente mostrato se nel file di salvataggio non ci sono dati o, quando da menù principale, vi si accede tramite apposito pulsante.
- **Skins Selection Menu:** vi si accede tramite il menù principale e permette di selezionare quale *skin* e quale

background, tra quelli già acquistati, utilizzare rispettivamente per il personaggio principale e per lo sfondo della mappa. Da questo menù si può tornare a quello principale.

- **Ostacoli:** due tipi di ostacoli, quelli fissi a forma di *pipe*, e quelli mobili che hanno invece una delle due immagini scelte come loro *skin*. In collisione provocano la morte del personaggio e la fine della partita. I fissi si trovano sui lati superiore ed inferiore della mappa mentre, quelli mobili, si muovono lungo la mappa avvicinandosi al personaggio.
- **Malus e Booster:** si muovono lungo la mappa da destra verso sinistra, avvicinandosi al personaggio. Ci sono diversi tipi sia di malus che di booster: se si entra in contatto con loro, i primi provano una qualche conseguenza spiacevole mentre, i secondi, forniscono un qualche tipo di aiuto.

Requisiti non funzionali

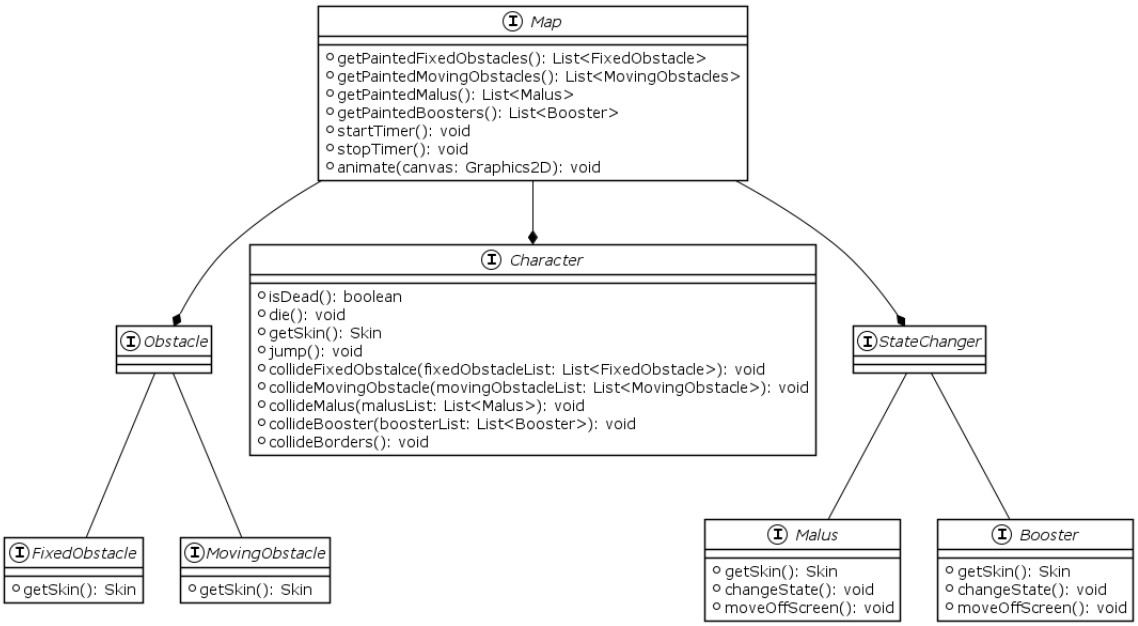
- Che il gioco risulti fluido su ogni macchina, senza *lag*, nonostante le varie entità e componenti grafiche presenti.
- Basso consumo della RAM.

1.2 Analisi e modello del dominio

Il modello del dominio applicativo viene mostrato tramite l'UML in figura 1.1. Il cuore di Floppy Floppa è la mappa infinita a scorrimento orizzontale. Tramite essa si muovono ed interagiscono fra loro: *character*, ostacoli, malus e booster. Nella mappa il *character* si trova all'estrema sinistra, il giocatore può farlo muovere premendo la barra spaziatrice in modo che salti in avanti. I salti permettono al *character* di schivare ostacoli e malus. Gli ostacoli fissi, posti ai lati superiore ed inferiore della mappa, *spawnano* a coppie e con lunghezze differenti, mantenendo sempre la stessa distanza fra la *pipe* superiore e la corrispettiva *pipe* inferiore.
Gli ostacoli mobili vengono fatti *spawnare* all'estrema destra della mappa e si muovono sia in orizzontale che in verticale, creando un

movimento sinusoidale che da destra va verso sinistra, avvicinandosi al *character*. Entrambi questi elementi, in collisione con il *character* ne provocano la morte e di conseguenza, il termine della partita.
Muovendosi lungo la mappa, il *character* deve tentare di schivare anche i malus che non provocano la morte, ma una conseguenza spiacevole volta a mettere in difficoltà il giocatore. Al tempo stesso si deve cercare di entrare in collisione con i booster, i quali invece forniscono un aiuto durante o una ricompensa riscossa a fine partita. Entrambe queste entità *spawnano* all'estrema destra della mappa e si muovono orizzontalmente verso il *character*.
La mappa si ferma solo in caso di morte del personaggio, in questo caso lo spazio percorso viene tramutato in *coins*. Ai coins prodotti dai metri percorsi, si vanno ad aggiungere e sottrarre i *coins* generati dai malus e dai booster colpiti durante dal partita dal *character*. Il numero di *coins* posseduti viene quindi aggiornato e mantenuto nello *shop*, dove sono anche spendibili.

Figure 1: Fig 1.1 UML del dominio



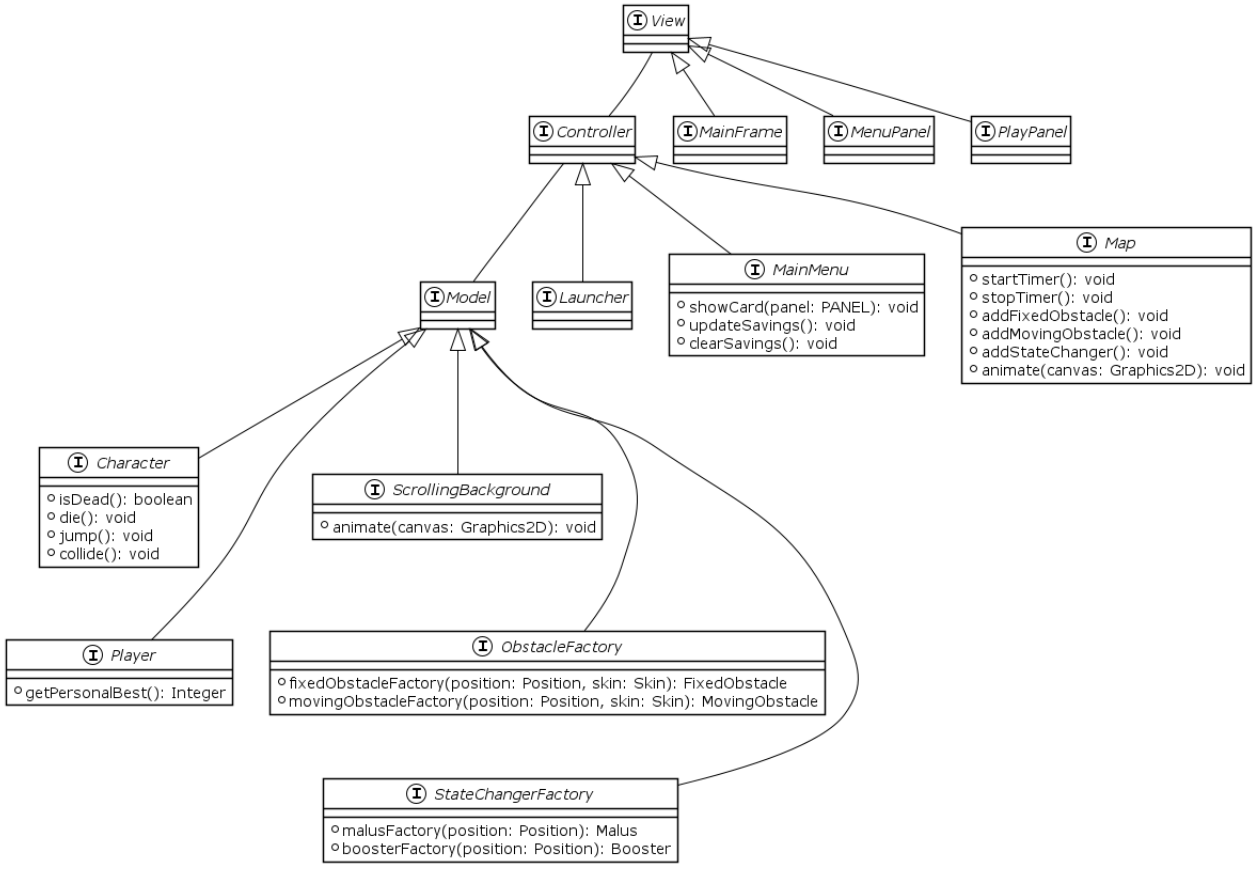
2

Capitolo 2

Design

2.1 Architettura

Figure 2: Fig 2.1 Architettura dell'applicazione



Come mostrato dall'UML in figura 2.1, *Floppy-Floppa* segue il pattern architetturale MVC.

La parte di **Model** prevede diverse entità, come descritto nella parte di Analisi. Il **Model** permette alla parte di **View** di rimanere costantemente al passo con i cambiamenti che si verificano e di mostrare così, i dati aggiornati.
La **View** permette di visualizzare il *menù principale*, lo *shop*, la *leaderboard*, il *menù* per la scelta di *skin* e *background*, la finestra per la scelta del *nickname*, i *tutorial* e infine il *menù* di fine partita.
Il **Control** è formato da diverse parti. Quelle principali sono la *mappa di gioco*, il *game engine* e il *Menù principale*. Il primo decide come e quando far apparire diverse entità di gioco quali ostacoli, booster e malus. Il secondo è il cuore del **Control** in quanto unisce e gestisce tutte le entità di **Model** e le rispettive parti di **View**. Infine il *menù principale* permette di mostrare correttamente e *on-demand* tutte le parti menzionate in **View**, ad eccezione del menù di fine partita e della finestra di scelta del *nickname*
Altre parti di **Control** sono *shop* e *leaderboard*. Il primo permette di controllare la quantità, sempre aggiornata, dell'entità *coins* e l'acquisto degli oggetti proposti in esso; il secondo mantiene il controllo e l'ordine su i dati di salvataggio delle partite.
Grazie a questa architettura la parte di **View** non intacca di alcun modo **Control** e **Model**. Ciò significa che, qualora si volesse implementare una diversa grafica, lo si potrebbe fare senza dover andare a modificare parti non strettamente appartenenti a **View**.

2.2 Design dettagliato

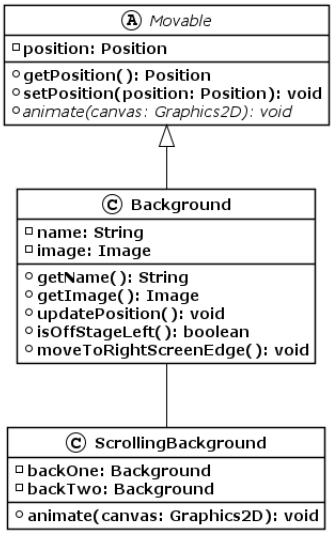
In questa parte ogni membro del gruppo presenta dettagliatamente le parti da sé implementate, con relativi UML e spiegazioni sulle soluzioni adottate o da loro stessi concepite.

Salvatore Antonio Addimando

- **Map**
- **ScrollingBackground**
  Per la realizzazione dello sfondo della mappa sono stati utilizzati due *Decorator Pattern*. C'era la necessità di creare

un'immagine di sfondo per la mappa che si muovesse, per simulare la mappa a scorrimento infinito.
Un primo *Decorator* è stato utilizzato per fare un *wrapping* di Image con Background, il quale, aggiunge ad Image la funzionalità di potersi muovere costantemente verso sinistra.
L'altro *Decorator* è stato invece impiegato per *wrappare* Background in uno ScrollingBackground; quest'ultimo fa uso di due Background: li utilizza in modo tale che la situazione iniziale, a inizio partita, abbia un Background che occupa lo schermo per intero e un secondo che comincia all'estremità destra dello schermo.
Si muovono in contemporanea verso sinistra in modo da susseguirsi continuamente (il Background più a sinistra una volta fuori dalla visualizzazione torna all'estrema destra dello schermo per ripetere il ciclo), così da simulare lo scorrimento infinito.

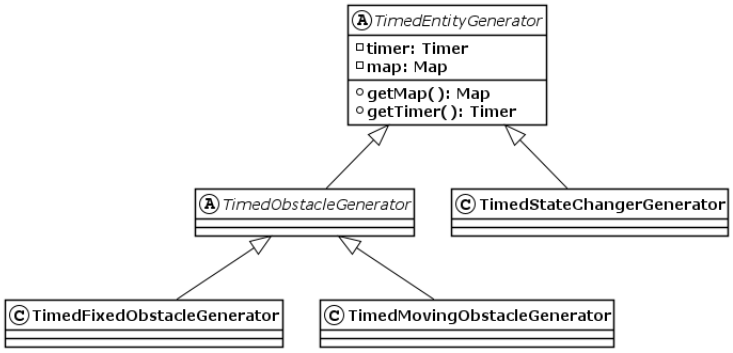
Figure 3: Fig 2.2 Background e ScrollingBackground



- **Generator**
  La funzionalità principale della mappa è quella di posizionare correttamente le varie entità al suo interno, durante una partita. Al fine di fare questo sono stati realizzati diversi Generator seguendo il *Composite Pattern*.
  Alla base del *Composite* c'è TimedEntityGenerator, tutti gli

altri partono da questo usandolo come modello. I due rami principali che partono dalla radice sono: TimedObstacleGenerator e TimedStateChangerGenerator. TimedStateChangerGenerator crea un oggetto di tipo StateChanger, invece, TimedObstacleGenerator fa da intermediario tra: TimedEntityGenerator e il livello con TimedFixedObstacleGenerator e TimedMovingObstacleGenerator. Questo per categorizzare gli ultimi due generator menzionati, che rispettivamente creano un oggetto di tipo FixedObstacle e uno di tipo MovingObstacle.

Figure 4: Fig 2.3 Composite di Generator



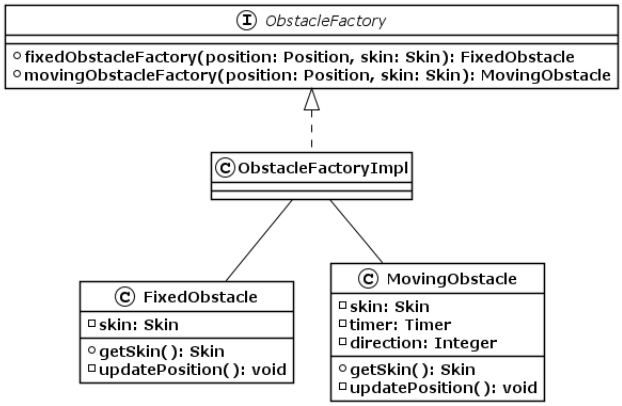
Gli UML per lo Scrolling Background e i Generator sono rispettivamente in figura 2.2 e 2.3.

- **Leaderboard**
  Ai fini della realizzazione della Leaderboard questa è stata divisa in una parte di **Control** e in una parte di **View**.
  La parte di **Control** gestisce un elenco di *Player* (unità fondamentale della Leaderboard), in modo che appaiano in ordine decrescente di *personal best* nel LeaderboardPanel (**View**).
  Questa suddivisione permette, in futuro, di modificare la **View** senza intaccare minimamente la parte di **Control**.

Benedetta Pacilli

- **Moving Obstacle**
  Come mostrato nell'UML in figura 2.4 i *moving obstacle* (ostacoli mobili) vengono creati sfruttando il *Factory Pattern*. Questo perché c'era bisogno di implementare diversi tipi di *obstacle* senza però perdere il legame tra il tipo generico *Obstacle* e i vari ostacoli che ne sarebbero derivati. Il *Factory Pattern* permette inoltre di poter continuare ad aggiungere, in futuro, ulteriori tipologie di ostacoli, senza dover andare ad effettuare grande cambiamenti nel codice.

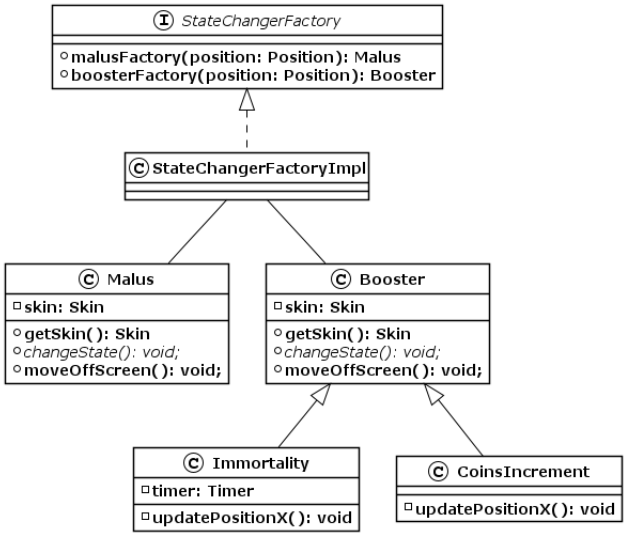
Figure 5: Fig 2.4 Factory di MovingObstacle



- **Booster**
  Come si vede dall'UML in figura 2.5, anche i *booster* sfruttano il *Factory Pattern*. I booster derivano dal tipo *State Changer* in quanto ciò che fanno è modificare un parametro del gioco o la situazione del *Character*.
  Al contrario degli ostacoli, i vari tipi di *Booster* vengono implementati tramite sue sottoclassi. Questa soluzione è stata adottata per permettere di mantenere il rapporto con il tipo iniziale: *State Changer* e, al tempo stesso, per differenziare i vari tipi di *Booster*: a *booster* diversi corrispondono cambiamenti di stato diversi.
  Il cambiamento di stato viene fornito dal metodo

ChangeState, astratto nella classe *Booster*, così da poter essere implementato differentemente dalle sottoclassi.
Sfruttare il *Factory Pattern* permette di aggiungere facilmente e senza cambiamenti al resto del codice, ulteriori sottoclassi di *Booster* che causino un diverso cambiamento di stato nel gioco.

Figure 6: Fig 2.5 Factory di Booster



- **Priced Skin e Priced Background**
  Per la realizzazione della classe *Shop*, che permette di comprare degli oggetti di tipo *Skin* e *Background*, serviva associare a questi ultimi un campo prezzo che ne permettesse l'acquisto. Usando il *Decorator Pattern*, *Priced Skin* è diventato un wrapper di *Skin*, stessa cosa per *Priced Background* che è divenuto wrapper di *Background*. Usare il *Decorator* ha permesso di mantenere il riferimento alla super classe (e di poter ereditare i suoi metodi), e di poter aggiungere informazioni in più all'oggetto, così da specializzarlo.
  Gli UML nelle figure 2.6 e 2.7 mostra come i *Decorator* sono stati realizzati.

Figure 7: Fig 2.6 PricedBackground

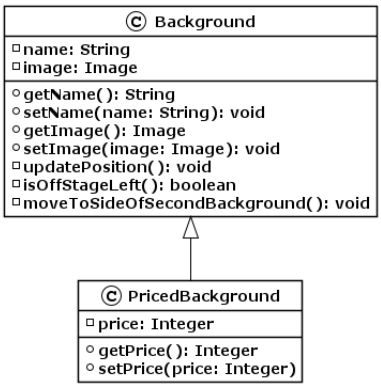
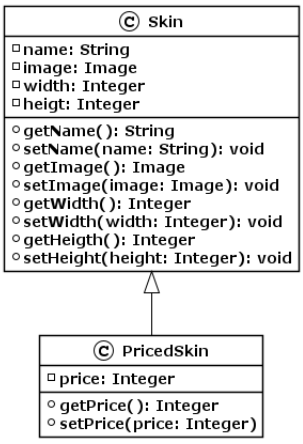
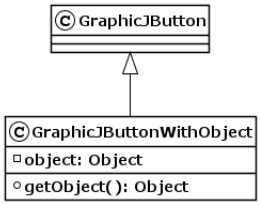


Figure 8: Fig. 2.7 Skin e PricedSkin



- **GraphicJButtonWithObject**
  Una semplice classe che specializza la classe GraphicJButton, così da aggiungergli un campo *Object* sfruttando il *Decorator Pattern*. Questa soluzione è stata adottata per avere dei pulsanti BUY nello *Shop* che permettessero facilmente di simulare l'azione dell'acquisto. Avendo un campo *Object* che mantiene l'oggetto al quale il pulsante BUY fa riferimento, è semplice accedervi per segnarlo come comprato.

Figure 9: Fig 2.8 GraphicJButtonWithObject



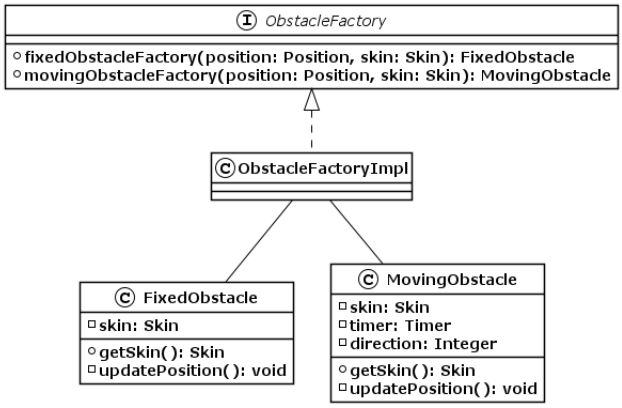
- **Shop e ShopGUI**
  Lo *Shop* è stato diviso in una parte di **View** e in una di **Control**. Questa divisione tra grafica e logica permette di poter effettuare modifiche alla grafica, senza intaccare la parte di controllo. Nessun pattern di programmazione è stato individuato per lo sviluppo di questa parte del programma, i quali dettagli, vengono mostrati nella parte di Analisi, in quella riguardante l'Architettura.

Valentina Pieri

- **Obstacle Factory e Fixed Obstacle**
  Come si può osservare nell'UML in figura 2.9, è stata creata l'interfaccia Obstacle Factory, utilizzando il *Factory Pattern*. L'utilizzo del *Factory Pattern* è stato dettato dal fatto che il nostro obbiettivo era quello di poter creare vari tipi di ostacoli, i quali però avessero tutti caratteristiche base comuni, rappresentato dalla componente Obstacle. *Factory Pattern* è molto utile anche nell'eventuale problema futuro di voler implementare nuovi tipi di Obstacle.
  FixedObstacle (ostacolo fisso) crea una coppia di *pipes*, le quali vengono generate una sopra all'altra e separate da uno

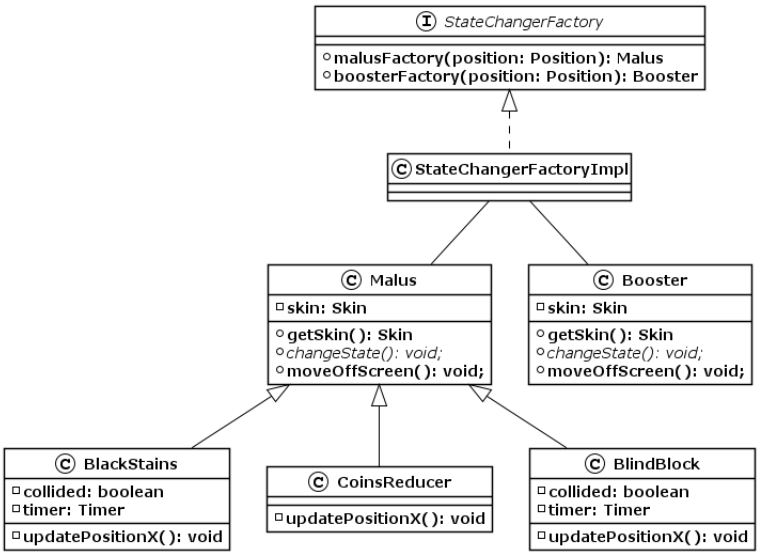
spazio, per permettere il passaggio del personaggio. I FixedObstacle vengono creati con l'apposito metodo factory in ObstacleFactory.

Figure 10: Fig 2.9 Factory di FixedObstacle



- **State Changer Factory e Malus**
  Guardando l'UML in figura 2.10, StateChangerFactory è l'interfaccia che modella StateChangerFactoryImpl, la quale serve per generare le sottoclassi Malus e Booster, secondo il *Factory Pattern*. Anche qui il Factory Method è utilizzato per poter in futuro aggiungere nuove tipologie di State Changer. All'interno di StateChangerFactoryImpl vengono definiti i metodi per creare i vari State Changer, Malus e Booster, che vanno a modificare il *Character* o la partita. Ad ogni Malus corrisponde un cambiamento di stato diverso, quindi per questo motivo i Malus sono implementati tramite delle sottoclassi; le quali mantengono con State Changer un legame. Il metodo che attua le modifiche è *changeState*, il quale rimane astratto in Malus, per essere poi implementato nelle sottoclassi.

Figure 11: Fig 2.10 Factory di Malus



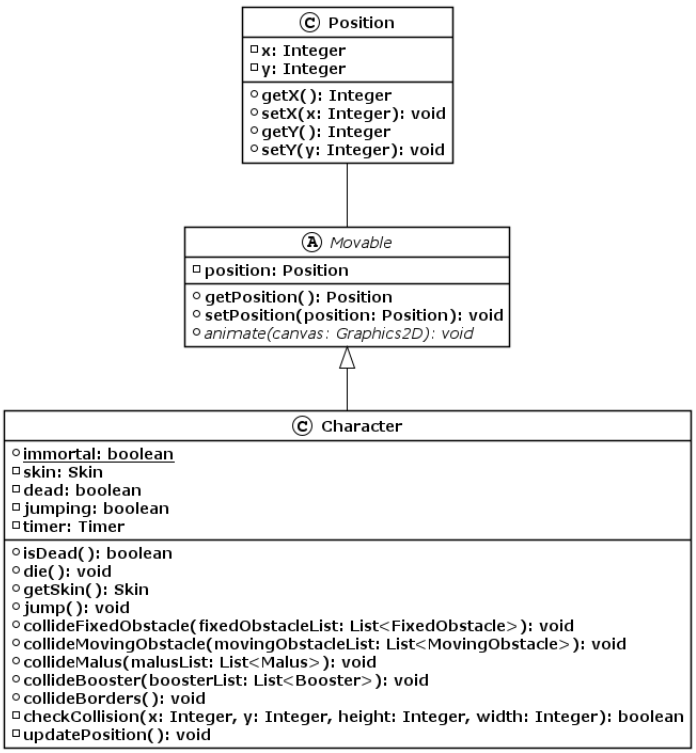
- **Main Menu e MenuPanel**
  Il Menu principale è diviso in una parte di **View**, MenuPanel e in una di **Control**, MainMenu.
  Questa divisione permette allo sviluppatore di poter modificare la parte di **View**, senza andare ad intaccare MainMenu; dove è stata implementata la gestione dei vari JPanel grazie all'utilizzo di *CardLayout*. Nessun pattern di programmazione è stato utilizzato per lo sviluppo di questa parte del programma. Per maggiori informazioni rifarsi alla parte di Architettura e di Analisi.

- **SelectionPanel e GameSettings**
  La classe SelectionPanel gestiste l'equipaggiamento da parte del giocatore dei prodotti comprati nello Shop. SelectionPanel rappresenta la parte di **View** e GameSettings quella di **Control**. Nessun pattern di programmazione è stato utilizzato per lo sviluppo di questa parte del programma. Per maggiori informazioni rifarsi alla parte di Architettura in Analisi.

Cristina Zoccola

- **Movable e Position**

Figure 12: Fig. 2.11 Movable, Position e Character

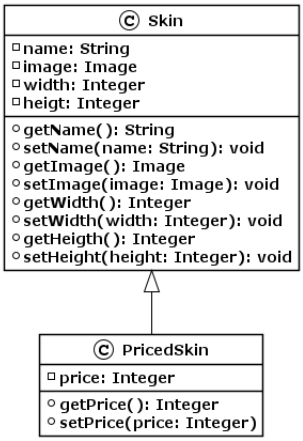


La classe *Movable* racchiude le caratteristiche comuni tra tutti gli oggetti che verranno poi animati all'interno della mappa, questa classe è stata implementata per seguire il pattern di programmazione *Composite*, abbiamo deciso di usarlo perchè permette di racchiudere in una struttura ad albero tutti gli oggetti che verranno poi animati tramite il metodo comune *animate*; questa classe infatti verrà poi specializzata dalle sottoclassi delle diverse entità animabili.
Abbiamo scelto di implementarla come classe astratta per poter utilizzare anche il pattern *Flyweight*, abbiamo deciso di usarlo perchè ci permette di risolvere il problema della "pesantezza" del programma, questo viene fatto mantenendo in questa classe il campo comune a tutti *position* di tipo

Position e non dovendolo quindi mantenere in ogni singolo oggetto.
La posizione è implementata con la classe *Position* con cui si tiene traccia delle relative coordinate x e y.
L'UML in figura 2.11 mostra la loro realizzazione.

- **Skin**
  La classe *Skin* è stata implementata sfruttando il pattern *Decorator*, facendola diventare un wrapper della classe già presente in java *Image*.
  Usando questo pattern è stato possibile aggiungere informazioni rispetto alla semplice immagine associata ad ogni oggetto presente nel gioco, come il proprio nome, l'altezza e la larghezza; l'aggiunta dell'altezza e della larghezza ci ha permesso di gestire al meglio le collisioni tra le entità.
  La sua implementazione è mostrata nell'UML in figura 2.12.

Figure 13: Fig. 2.12 PricedSkin



- **Character**
  La classe *Character* implementa la parte di **Model** del personaggio principale del gioco.
  Vengono gestite le collisioni tra il personaggio e i booster, i malus e gli ostacoli, mobili e fissi, cambiando di conseguenza lo stato del personaggio.
  Inoltre viene gestita la gravità del personaggio, che cade verso il basso, ma risale sfruttando la meccanica del salto sempre implementata in questa classe.

Non è stato utilizzato nessun pattern specifico per la sua realizzazione, ma estendendo la classe *Movable* fa parte dello schema ad albero formato grazie al pattern *Composite*
Nell'UML in figura 2.7 mostra più nello specifico la sua realizzazione.

- **Tutorial**
  La classe *TutorialPanel* gestisce la **View** del *Tutorial* che verrà mostrato all'utente, non è stato utilizzato nessun pattern specifico per la sua realizzazione.
  La sua implementazione è meglio mostrata nel'UML 2.1 dell'Architettura.

3

Capitolo 3

Sviluppo

3.1 Testing Automatizzato

In questa sezione verranno elencati tutti i test automatizzati, effettuati allo scopo di verificare il corretto funzionamento del programma. Per la creazione di ogni test è stato utilizzato *JUnit*.

- **Salvatore Antonio Addimando**

  - TestTimedEntityGenerator:
    In questa classe viene testato l'effettivo funzionamento degli spawner regolati da timer. In particolare per ogni Generator viene controllato il numero di elementi presenti nella rispettiva lista presente nella classe Map.
  - TestLeaderboard:
    Qui viene controllata la corretta inizializzazione della leaderboard partendo dal file savings di base, definito dalla una costante della classe Constants, l'inserimento di un nuovo player e l'update di un Player già esistente.

- **Benedetta Pacilli**

  - Test di CoinsIncrement
    Il test controlla se il booster in questione si muove come previsto, dando in input posizioni di partenza diverse. Per la realizzazione di questo test è stata creata un'apposita classe innestata dentro la classe CoinsIncrement, così da poter usare il metodo privato updatePositionX().

- Test di Immortality
  Il test controlla se il booster in questione si muove come previsto, dando in input posizioni di partenza diverse. Viene inoltre controllato il suo comportamento ovvero, quello di rendere il *Character* immortale alle collisioni con gli ostacoli, sia fissi che mobili. Per la realizzazione di questo test è stata creata un'apposita classe innestata dentro la classe Immortality, così da poter usare il metodo privato updatePositionX().

- Test di MovingObstacle
  Il test controlla se l'ostacolo mobile si muove come previsto, dando in input posizioni di partenza diverse. Per la realizzazione di questo test è stata creata un'apposita classe innestata dentro la classe MovingObstacle, così da poter usare il metodo privato updatePosition().

- TestShop
  Il test dello *Shop* controlla che a inizio legga correttamente il file di salvataggio. Lo shop infatti necessita per il suo corretto funzionamento di leggere, la prima volta che si apre il gioco (quindi non vi sono ulteriori dati oltre a quelli di base), dei dati dal file di salvataggio. Questo per impostare correttamente i *PurchaseStatus* degli oggetti dello shop.

• Valentina Pieri

- TestFactoryObstacleImpl
  Il test di ObstacleFactoryImpl controlla che le factory utilizzate, sia per fixedObstacle sia per MovingObstacle, funzionino correttamente. All'interno della classe sono presenti due test: TestFixedObstacle, per verificare se factoryFixedObstacle ritorna un fixedObstacle, e TestMovingObstacle, che verifica se factoryMovingObstacle ritorna un MovingObstacle.

- Test di FixedObstacle
  Questo test controlla se l'ostacolo fisso si muove come previsto, dando come input una position. Si osserva se il

metodo updatePosition() ritorna correttamente la nuova posizione del fixedObstacle. Il test si trova sotto forma di classe, innestata all'interno di FixedObstacle, così da poter controllare updatePosition(), essendo quest'ultimo privato.

- Test di CoinsReducer
  Questo test controlla se il malus si muove come previsto, dando come input più position diverse. Si osserva se il metodo updatePositionX() ritorna correttamente la nuova posizione di CoinsReducer. Il test si trova sottoforma di classe, innestata all'interno di CoinsReducer, così da poter controllare updatePositionX(), essendo quest'ultimo privato.

- Test di BlindBlock
  Questo test controlla se il malus si muove come previsto, dando come input più position diverse. Si osserva se il metodo updatePositionX() ritorna correttamente la nuova posizione di BlindBlock. Il test si trova sottoforma di classe, innestata all'interno di BlindBlock, così da poter controllare updatePositionX(), essendo quest'ultimo privato.

- Test di BlackStain
  Questo test controlla se il malus si muove come previsto, dando come input più position diverse. Si osserva se il metodo updatePositionX() ritorna correttamente la nuova posizione di BlackStain. Il test si trova sottoforma di classe, innestata all'interno di BlackStain, così da poter controllare updatePositionX(), essendo quest'ultimo privato.

• Cristina Zoccola

- TestCommonMethods
  In questo test viene controllato il corretto comportamento dei diversi metodi usati da tutti i componenti del gruppo.
  In particolare se viene correttamente generato un numero random, se vengono calcolati i corretti *pixel* considerando la dimensione dello schermo e se vengono correttamente presi i file dalle risorse del gioco.

- TestCharacter
  In questo test viene controllato il corretto funzionamento delle collisioni tra il personaggio e le diverse entità sulla mappa. In particolare sono stati creati 3 test: uno per gli ostacoli fissi, uno per tutte le altre entità che si muovono (ostacoli mobili, booster e malus), perché al loro interno utilizzano tutte lo stesso metodo privato, e uno per i bordi della mappa.

- TestUpdatePosition
  In questo test viene controllato il corretto aggiornamento della posizione del personaggio, sia quando cade che quando salta, questa classe è stata creata innestata in *Character* perché il metodo per aggiornare la posizione è privato.

Sono stati poi svolti dei test manuali per verificare la parte grafica e il corretto funzionamento del gioco su diversi sistemi operativi(Windows e Linux) e su diverse risoluzioni di schermo.

3.2 Metodologia di Lavoro

Per la realizzazione di *Floppy-Floppa* il lavoro è stato suddiviso al momento dell'Analisi, cercando di rendere la suddivisione il più equa possibile. Per meglio organizzarci, sempre durante l'Analisi, abbiamo creato un UML generale che racchiudesse tutte le varie entità da sviluppare insieme alle dipendenze tra queste.
Come DVCS abbiamo utilizzato Git. Ognuno di noi ha creato, partendo dal dev, tutti i *feature branch* necessari allo sviluppo delle proprie parti. Questi *feature branch* sono stati eventualmente diramati in ulteriori *sub-feature branch*.
Per quanto riguarda la risoluzione di problemi, venivano usati degli *hotfix branch* mentre, i rami dedicati esclusivamente al miglioramento del codice sono creati degli *enhancement branch*.
Tutti i branch di sviluppo (ovvero quelli in cui è stato scritto del codice), sono stati fatti confluire al dev, che alla fine per la creazione della *release*, è stato *mergiato* al main.
Di seguito viene elencato la suddivisione del lavoro:

Salvatore Antonio Addimando: implementazione della mappa a scorrimento infinito (*Map*, *Background*, *ScrollingBackground*),

creazione e gestione della leaderboard e dello *score management* (salvare il *nickname* del *player*, mantenere la leaderboard aggiornata, in ordine decrescente di punteggio e per ogni *player/nickname* mantenere solo il suo *best score*). Per mantenere la leaderboard ordinata è stato usato il *Binary Search*.
Infine l'inserimento della musica di background.

Benedetta Pacilli: creazione degli ostacoli mobili (*MovingObstacle*), creazione dei Booster (classe *Booster*, sottoclassi *Immortality* e *CoinsIncrement*), creazione del Panel mostrato alla fine di una partita in cui vengono mostrati i *coins* aggiornati, creazione e gestione dello shop (*Shop* e *ShopGUI*),

Valentina Pieri: creazione degli ostacoli fissi (*FixedObstacle*), creazione dei Malus (classe *Malus*, *CoinsReducer*, *BlindBlock*, *BlackStain*), creazione e gestione del Menu principale che *on-demand* mostra gli altri Panel disponibili. Creazione di un panel per la selezione delle *Skins* del *Character* e dei *Backgrounds* per la mappa.

Cristina Zoccola: creazione del *Character* (il movimento del *jump*, la sua relazione con tutti i componenti della mappa: quindi le collisioni letali con gli ostacoli e quelle non letali con malus/booster, infine la relazione con la mappa stessa: la collisione fatale con i bordi superiore ed inferiore), creazione del Panel in cui mostrare il Tutorial in caso di Leaderboard vuota.

In comune: sono state svolte le classi Constants e CommonMethods e tutti i componenti del game engine ovvero: Launcher, MainFrame, NicknamePanel, PlayPanel.

3.3 Note di Sviluppo

Di seguito vengono elencate le feature di linguaggio avanzate che ogni membro ha utilizzato.

- Salvatore Antonio Addimando
  - Lambda Expressions

  - java.awt.Graphics2D, javax.swing.Timer e javax.sound.sample
- Benedetta Pacilli
  - Generici
  - Reflection
  - Lambda Expressions
  - java.awt.Graphics2D e javax.swing.Timer
- Valentina Pieri
  - Lambda Expressions
  - java.awt.Graphics2D e javax.swing.Timer
- Cristina Zoccola
  - Lambda Expressions
  - java.awt.Graphics2D e javax.swing.Timer

Per la realizzazione di un metodo che permettesse la rotazione delle immagini è stato preso spunto da qui:
https://stackoverflow.com/a/8639615/10600358
Per utilizzare il *Binary Search* è stato fatto uso di Collections.BinarySearch.

4

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

- **Salvatore Antonio Addimando**
  Commento personale Questo progetto mi ha permesso di mettere alla prova le mie capacità di collaborazione e di organizzazione di un progetto. La parte che mi è piaciuta di più è stata l'organizzazione e il briefing sull'utilizzo dei branch con gli altri membri del gruppo. Peccato di esserci inoltrati nell'implementazione senza pensare all'utilizzo di più thread per ogni componente. Sono comunque contento che il gioco funzioni anche su un unico thread.

- **Benedetta Pacilli**
  Mi ritengo soddisfatta del progetto realizzato. Mi ha permesso di applicare e consolidare le conoscenze sugli utilizzi più avanzati di Git appresi a lezione e di sviluppare, per la prima volta, un progetto abbastanza consistente in gruppo. Penso che sarebbe carino, in un futuro, realizzare un versione per Android di Floppy-Floppa migliorandone anche le performance o aggiungendo nuovi tipi di ostacoli, malus, booster o nuovi items da comprare nello shop.
  Un problema che abbiamo dovuto affrontare è quello di aver usato un solo Thread. Non aver usato più thread penso sia un punto di debolezza del programma ma, al tempo stesso, è stata una challenge che ci ha permesso di metterci in gioco riuscendo alla fine a rendere comunque il programma fluido.

- **Valentina Pieri**
  Questo progetto mi ha dato la possibilità di mettere alla prova le mie capacità organizzative, sia personali che di gruppo, e di collaborazione. Alcune difficoltà sono venute fuori per via della malattia da Covid prima di due componenti del gruppo e successivamente di altre due; però a discapito

delle difficoltà sono contenta di come siamo riusciti ad organizzarci e a comunicare tra di noi. Dal punto di vista tecnico sono stata molto contenta di aver potuto approfondire accuratamente la mia conoscenza di Git e I suoi vari utilizzi, già precedentemente presentati a lezione; è stato anche molto stimolante il poter creare un progetto insieme ad altre persone e sperimentare per la prima volta la creazione di un progetto di programmazione ad oggetti in gruppo, apprendendo insieme o singolarmente nuove conoscenze. Riassumendo, mi reputo soddisfatta del progetto realizzato.

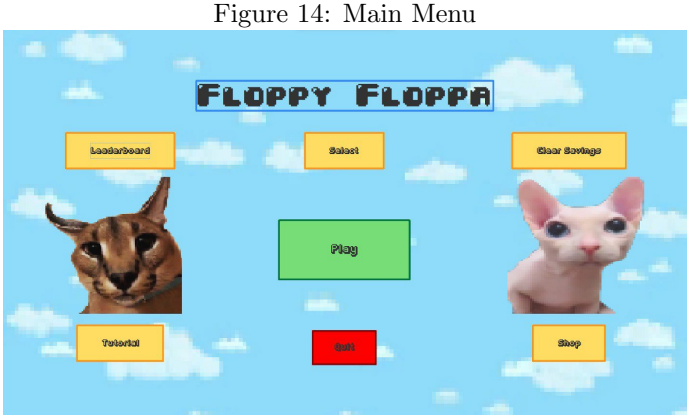
- **Cristina Zoccola**
  Nel complesso sono soddisfatta della realizzazione di questo progetto, ho potuto imparare a lavorare al meglio in gruppo, cosa che prima non avevo mai fatto in un progetto di queste dimensioni.
  Sono contenta della collaborazione che c'è stata tra i componenti del gruppo e ho trovato particolarmente utile l'iniziale momento di organizzazione comune che mi ha permesso di capire al meglio come poi progettare e sviluppare la mia parte di lavoro tenendo conto dell'interazione con le parti sviluppate dagli altri.
  Inoltre sono molto felice di aver potuto espandere le mie conoscenze per quanto riguarda l'uso di Git.

Appendice A

Guida utente

Il Menù principale:

Figure 14: Main Menu



Il menù principale è composto da diversi bottoni:

- PLAY: crea una nuova partita
- LEADERBOARD: mostra i punteggi di tutti i giocatori, vengono mantenuti i personal best di ognuno in ordine decrescente.
- SELECT: Permette di scegliere una skin o un background a piacere, tra quelli già comprati. I pulsanti sono abilitati solo per gli item già acquistati e rimangono invariati se un item viene selezionato o meno.
- SHOP: Si vedono i coins posseduti(comuni a tutti i player) ed è possibile spenderli negli items mostrati nello shop. Di base i pulsanti nello Shop sono tutti attivi(non è comunque permesso l'acquisto se i coins non sono sufficienti) e si disattivano quando il relativo item viene comprato. Se non ci sono dati salvati, risultano acquistati la skin e il background di base.

Figure 15: Leaderboard

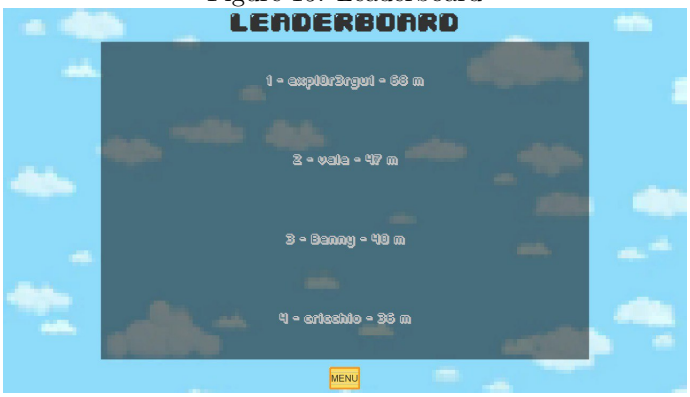


Figure 16: Selection Panel

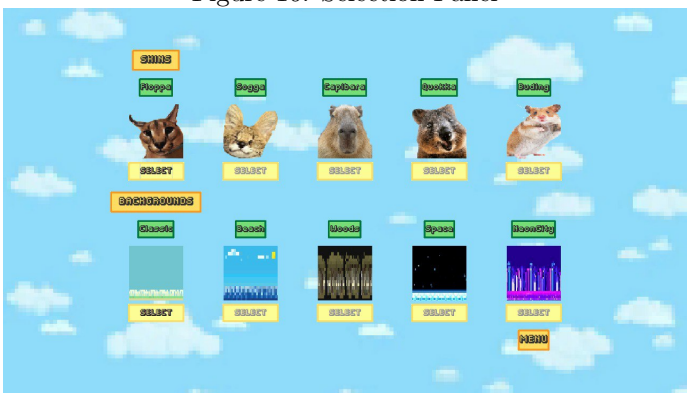
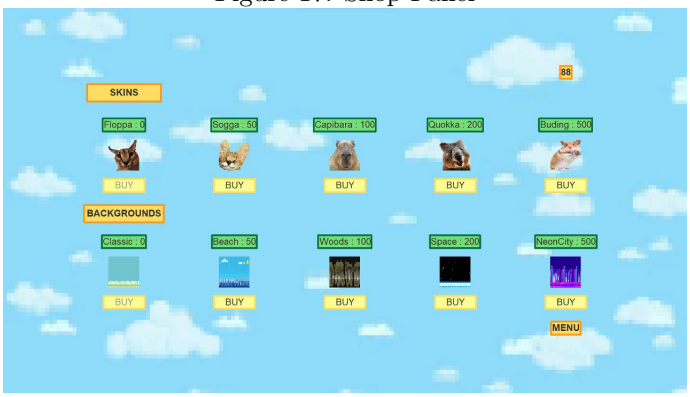


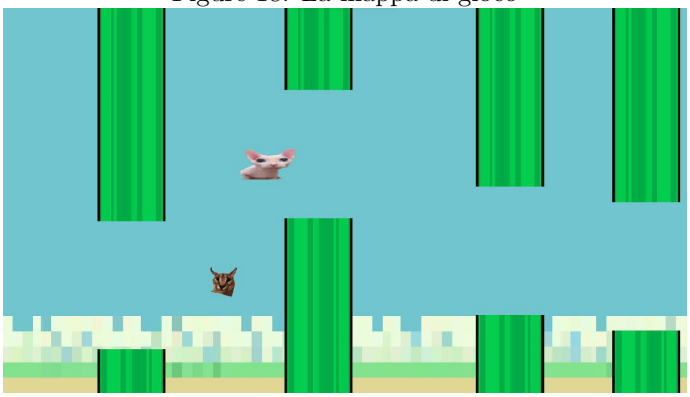
Figure 17: Shop Panel



- TUTORIAL: Quando è la prima volta che si apre il gioco, quindi non esiste un file di salvataggio, viene automaticamente mostrato il tutorial, raggiungibile anche tramite l'apposito pulsante del menù.
- CLEAR SAVINGS: Elimina i salvataggi.
- QUIT: Quando si preme si esce dal gioco e vengono salvati automaticamente i dati sul file.

Il gioco:

Figure 18: La mappa di gioco

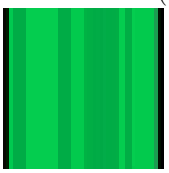


Lo scopo del player è di saltare tra le pipes senza colpire alcun tipo di ostacolo per rimanere il partita il più a lungo possibile. Per far fare un salto al character basta premere la barra spaziatrice. Se si tiene premuta la barra spaziatrice il character continua a muoversi verso l'altro mentre, se non si preme niente cade verso il basso. Una collisione tra il character e i bordi superiore ed inferiore della mappa, fa morire il character e di conseguenza, finire la partita.

Gli ostacoli:

- Ostacoli fissi: a forma di tubo e di diverse dimensioni, spawnano dall'alto e dal basso della mappa. In collisione con il character lo fanno morire.

Figure 19: Un ostacolo fisso (FixedObstacle)



- Ostacoli mobili: presenti nel gioco con due skin diverse(Bingus e Walter), spawnano all'estrema destra della mappa e si muovono verso il character con movimento sinusoidale. In collisione con il character lo fanno morire.

Figure 20: Bingus

Figure 21: Walter

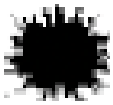
I malus:

Coins Reducer: Quando il character entra in contatto con questo malus, viene generata una quantità random di coins che viene sottratta dal numero di coins posseduti, considerando anche quelli guadagnati durante l'ultima partita. In collisione con il character scompare dalla mappa.

Figure 22: Il Malus Coins Reducer

Black Stain: Quando il character entra in contatto con questo malus, compare una macchia nera al centro dello schermo che riduce la visuale del player. In collisione con il character si teletrasporta dietro il character.

Figure 23: Il Malus Black Stain



Blind Block: Quando il character entra in contatto con questo malus, compare un blocco nero alla fine dello schermo che riduce la visuale del player. In collisione con il character si teletrasporta dietro il character.

Figure 24: Il Malus Blind Block

I booster:

Immortality: Quando il character entra in contatto con questo booster, il character viene reso immortale per 10 secondi. In collisione con il character scompare dalla mappa.

Figure 25: Il Booster Immortality

Coins Increment: Quando il character entra in contatto con questo booster, viene generata una quantità random di coins che viene aggiunta al numero di coins posseduti, considerando anche quelli guadagnati durante l'ultima partita. In collisione con il character scompare dalla mappa.

Figure 26: Il Booster Coins Increment