Tesina iniziale di

Programmazione di Interfacce Grafiche e Dispositivi Mobili

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2023-2024

DIPARTIMENTO DI INGEGNERIA

docente

Prof. Luca GRILLI

DUCK HUNT

applicazione desktop JFC/SWING

studenti

310700	**Giovanni**	**Fortunati**	giovanni.fortunati@studenti.unipg.it
330532	**Luca**	**Tribolati**	luca.tribolati@studenti.unipg.it

Data ultimo aggiornamento: 20 novembre 2024

Sommario

1. **Descrizione del Problema**

L'obiettivo di questo progetto è lo sviluppo mediante linguaggio di programmazione Java di un'applicazione desktop dal titolo Duck Hunt, che implementa una versione semplificata dell'omonimo videogioco sviluppato e pubblicato da Nintendo per il Nintendo Entertainement System (NES) nel 1984.

Il videogioco fu introdotto originariamente come bundle assieme a Super Mario Bros in alcune edizioni della NES; la sua particolarità era che faceva uso della NES Zapper, una pistola ottica che i giocatori puntano direttamente allo schermo per sparare alle anatre che appaiono volando attraverso la scena.

L'obiettivo del gioco è colpire il maggior numero di anatre possibile entro un certo limite di tempo ed utilizzando un numero massimo di 3 proiettili per ciascuna anatra.

Personaggio caratteristico del gioco è un cane da caccia (senza nome) che si presenta all'inizio della partita nella schermata di gioco per rimanere poi nascosto nell'erba fino a che il giocatore non colpisce o manca un'anatra. In entrambi i casi il cane si ripresenta al giocatore con una animazione, con l'anatra in mano se il bersaglio è stato colpito o ridendo in caso contrario.

Il gioco dispone di tre modalità:

- 1Duck, in cui compare una sola anatra da colpire;
- 2Duck, in cui le anatre sono due;
- Clay Shooting, nella quale i bersagli diventano dei piattelli lanciati in aria lontano dal giocatore.

La nostra implementazione presenterà la sola modalità 1Duck.

In questa modalità l'anatra appare dal basso dello schermo e vola casualmente dirigendosi verso la parte centrale, "rimbalzando" sui margini della schermata di gioco.

A questo punto si hanno a disposizione 3 colpi per centrarla, esauriti i quali scapperà. Si noti che c'è un tempo limite al cui termine l'anatra scappa anche senza che il giocatore abbia sparato. Il NES Zapper sarà sostituito nella nostra versione dal puntatore del mouse.

Esistono diversi tipi di anatre, ognuna delle quali conferisce uno specifico punteggio una volta colpita. Per superare il livello è necessario colpirne un numero minimo, il quale andrà ad aumentare man mano che si sale di livello.

Il gioco è strutturato in round, ognuno consistente di 10 anatre, ed è previsto un bonus se il giocatore riesce a colpirle tutte. Man mano che si sale di round aumenta la difficoltà di gioco, con le anatre che volano più velocemente e rendono i tiri via via più impegnativi.

Una demo del gioco è reperibile all'indirizzo https://www.retrogames.cc/nes-games/duck-hunt-world.html

2. Specifica dei Requisiti

L'applicazione che andremo a sviluppare presenterà le seguenti caratteristiche:

- **Interfaccia grafica**
    - A livello di interfaccia grafica saranno presenti dei contatori che tengono traccia delle variabili di gioco, quali livello, numero di anatre colpite, punteggio e colpi rimanenti;
    - Il mouse assumerà all'interno della schermata di gioco la forma di un mirino e sarà l'unico controllo con cui il giocatore dovrà interagire;
    - Un menu iniziale sarà presente all'avvio del gioco e presenterà il giocatore con le varie scelte, tra cui la possibilità di di visualizzare l'high score corrente;
    - Completa l'interfaccia grafica una schermata finale che comunica al player lo score raggiunto nella sessione di gioco appena conclusasi.
- **Meccaniche di gioco**

  Le meccaniche di gioco che saranno implementate rispecchieranno quelle del gioco originale in maniera fedele.
    - Per colpire l'anatra si ha a disposizione sia un numero di colpi che un tempo limitato;
    - Non vi é un limite teorico al punteggio ottenibile, ma l'aumento progressivo della difficoltà, implementato facendo aumentare la velocità dell'anatra che vola, rende progressivamente più difficile ottenere scores elevati;
    - In aggiunta, anche il numero di anatre colpite necessario per passare di round in round aumenta mano a mano che si progredisce nel gioco;
    - Saranno presenti dei bonus, sia sottoforma di anatre speciali che conferiscono un punteggio più elevato se colpite, sia assegnati nel caso in cui si riesca a colpire tutte le anatre previste nel round corrente;
- **Animazioni**

  Ogni personaggio presente nel gioco sarà animato in maniera fedele rispetto a come lo era nel gioco originale.

  Esempi di animazione che saranno implementati sono quella dell'anatra che vola, che viene colpita e che cade, l'animazione del cane che la prende o l'animazione del colpo andato a segno.
- **Effetti sonori**

  Saranno implementati effetti sonori specifici per ogni personaggio presente nel gioco, ossia cane, anatre e player.

3. Progetto

Nella seguente sezione andremo a fornire una descrizione del sistema software in esame, ricorrendo anche all'ausilio di diagrammi per facilitare la compresione da parte del lettore.

Ci focalizzeremo dapprima su una descrizione di alto livello del sistema software per poi passare a una descrizione più dettagliata dei metodi e delle classi implementate nel nostro progetto.

3.1 Architettura del Sistema Software

Il software, realizzato utilizzando la libreria Swing, presenta una architettura di tipo monolitico la cui caratteristica principale è la presenza di un solo package, all'interno del quale sono contenuti tutte le componenti del software. Nel nostro caso, infatti, la gestione della grafica, la logica del gioco, la gestione degli input utente e la riproduzione dell'audio sono globalmente inseriti all'interno del package '*duckhunt*'.

Per quanto riguarda la creazione del loop di gioco è stata scelta una soluzione basata sul *Delta Time Method*. Tale approccio consente di mantenere un frame rate costante, nel nostro caso fissato a 60 FPS, a prescindere dalla macchina sulla quale viene eseguito l'applicativo. Il ciclo di gioco si divide in due fasi principali, update e repaint, con i rispettivi compiti di aggiornare la logica di gioco e di ridisegnare la grafica secondo le specifiche di FPS impostate.

Il flusso di esecuzione è organizzato mediante una gestione degli stati con lo scopo di controllare il comportamento dei personaggi e l'evoluzione della partita in modo strutturato ed efficiente. Ogni entità principale del gioco, come ad esempio il cane o la papera, possiede dei propri stati interni i quali determinano il comportamento e le interazioni tra loro ed il resto del flusso.

L'alternarsi degli stati dei vari oggetti permette l'avanzamento del gioco secondo le modalità desiderate. Per poter descrivere il processo logico che ha portato all'ideazione del software ci serviamo dello schema a blocchi fornito in *Figura 3.1*, il quale schematizza un gameplay del gioco in esame. A partire dal menù principale è possibile selezionare una schermata statica di istruzioni, la possibilità di uscire dal gioco o la modalità di play, oltre la quale avviene l'inizio della partita.

Una volta avviata la partita, il gameloop è regolato attraverso il susseguirsi e l'alternarsi dei suddetti stati ed il passaggio dall'uno all'altro può avvenire sia in automatico, grazie ad esempio a dei timer, sia a seguito di input da parte dell'utente.

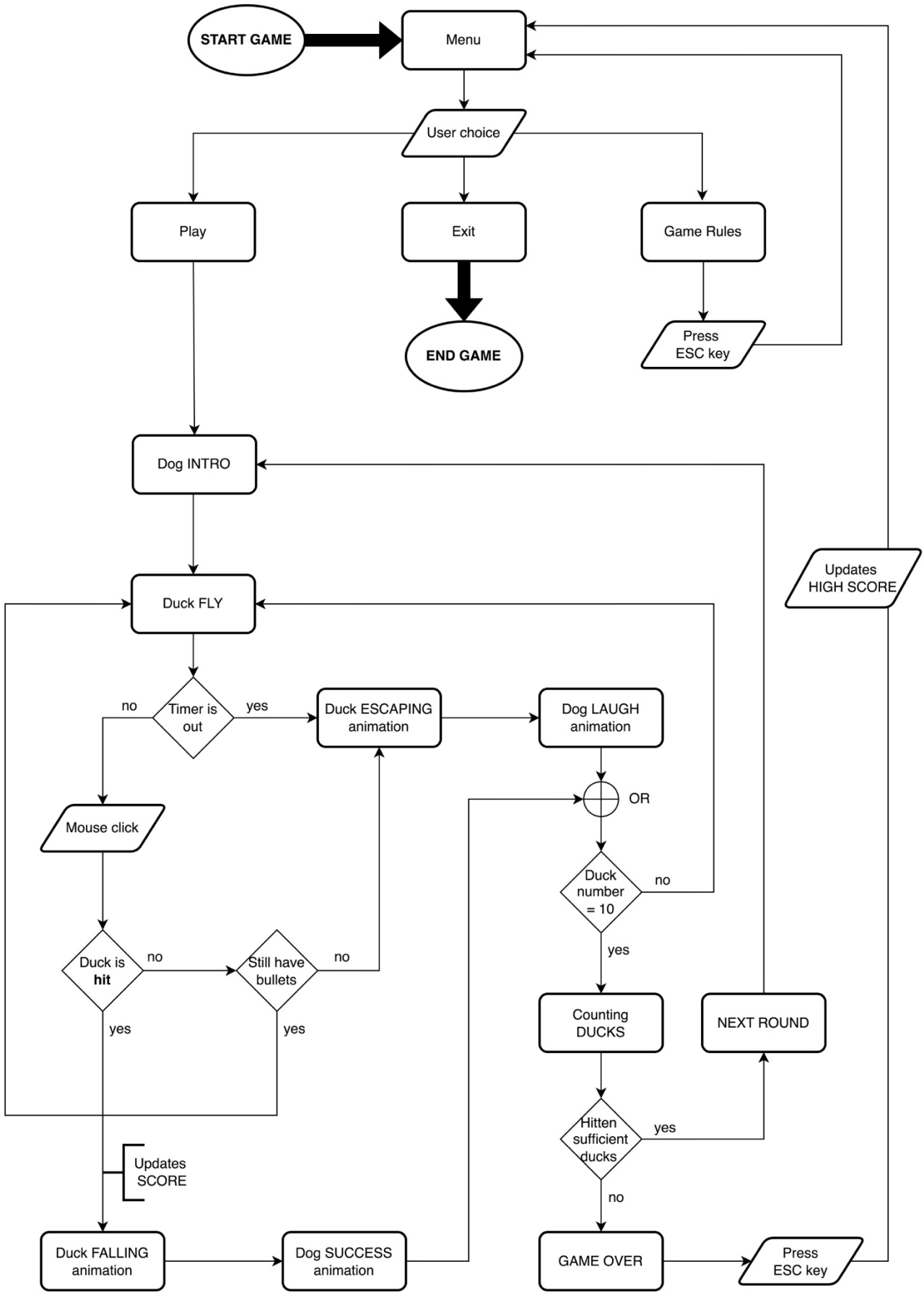
START GAME
Menu
User choice
Play
Exit
Game Rules
END GAME
Press
ESC key
Dog INTRO
Updates
HIGH SCORE
Duck FLY
Timer is
out
no
yes
Duck ESCAPING
animation
Dog LAUGH
animation
OR
Mouse click
Duck
number
= 10
no
yes
Duck is
hit
no
Still have
bullets
no
yes
yes
Counting
DUCKS
NEXT ROUND
Hitten
sufficient
ducks
yes
no
Updates
SCORE
Duck FALLING
animation
Dog SUCCESS
animation
GAME OVER
Press
ESC key

Figura 3.1

Gli stati rappresentati nello schema appartengono a diverse classi di oggetti, ma una volta combinati permettono di identificare in maniera univoca la fase in cui si trova il gioco e di conseguenza ci consentono di implementare il comportamento desiderato.

Le classi *Dog* e *Duck* rappresentano i protagonisti del gioco. I loro stati e i relativi comportamenti a loro associati, i più importanti dei quali sono stati rappresentati nello schema di cui sopra, sono la base portante su cui si basa la logica del gioco.

La classe *GamePanel* si occupa della creazione del gameloop e della gestione, sia logica che grafica, del menù di gioco. A partire da essa vengono richiamati i metodi update e draw che consentono al gioco di aggiornarsi.

La classe *GameManager* costituisce lo snodo primario, sia logico che grafico, del gioco, in quanto coordina le chiamate dei metodi update e draw contenuti nelle altre classi sulla base dei loro stati, consentendo una gestione centralizzata del flusso del gioco.

La classe *HudManager*, il cui compito è quello di fornire la logica con cui gli elementi visivi che compongono l'hud vengono aggiornati, si serve a sua volta degli stati di Dog e Duck per coordinare le informazioni visualizzate a schermo affinchè seguano l'effettivo flusso di gioco.

Le classi *HitHandler* e *KeyHandler* gesticono le interazioni con l'utente e in quanto tali sono in grado di cambiare lo stato del componente interessato dall'interazione.

Infine, le classi *ScoreManager* e *Sound* sono due classi di servizio. La prima si occupa di gestire il salvataggio e la lettura dell'highscore su un apposito file, mentre la seconda fornisce i controlli elementari che permettono la riproduzione dei suoni durante il corso del gioco.

In *Figura 3.2* è presente uno schema rappresentante ad alto livello le relazioni descritte.

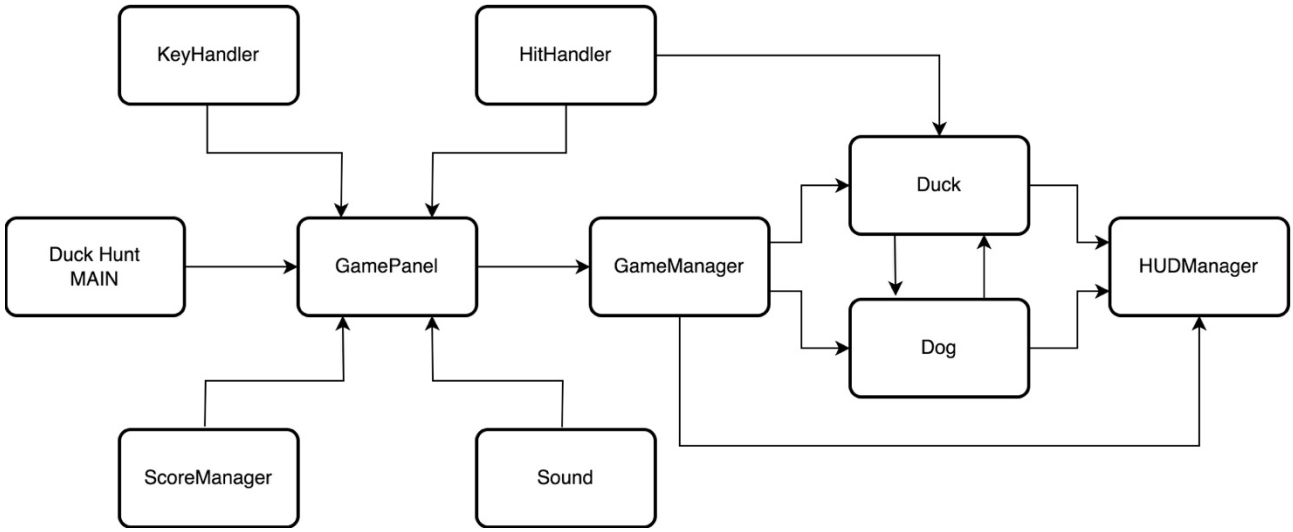

Figura 3.2

3.2 Descrizione delle classi

Descriviamo ora più in dettaglio le singole classi e il loro ruolo all'interno dell'applicativo, iniziando dalla classe *GamePanel*.

GamePanel

La classe *GamePanel*, estendendo *JPanel*, va a costituire la finestra all'interno della quale vengono visualizzati tutti gli elementi di gioco e attraverso la quale vengono recepiti gli input dell'utente.
All'interno di essa vengono instanziate tutte le classi presenti nel package, tra cui *KeyHandler* e *HitHandler*, che si occupano rispettivamente di gestire gli input da tastiera e da mouse.

L'implementazione dell'interfaccia *Runnable* rende inoltre possibile la creazione e l'esecuzione di un metodo *run()* , attraverso la quale è stato realizzato il loop di gioco secondo il modello *Delta Time Method*.
Ciò ci ha permesso di ottenere un frame rate costante pari a 60 FPS e di poter dunque coordinare in maniera esatta i tempi di svolgimento del gioco. All'interno del game loop vengono richiamate le due funzioni portanti su cui si basa la struttura del gioco, ossia *update()* e *repaint()*.
Quest'tultima richiama autonamamente il metodo *paintComponent* contenuto in *GamePanel* per disegnare a schermo le risorse grafiche del menù caricate mediante il metodo *loadImages().*

Il *GamePanel* si occupa della gestione logica e grafica del menu iniziale, della schermata delle istruzioni e del pulsante exit, mentre una volta avviato il gameplay il controllo, sia logico che grafico, viene passato agli appositi metodi *update()* e *draw()* della classe *GameManager*.
Tutte le operazioni di posizionamento a schermo e coordinamento tra gli elementi grafici che costituiscono il gioco sono state eseguite facendo riferimento alla risuluzione nativa del gioco originale, pari a 256x224 pixel, per poi procedere a una scalatura finiale fino alla dimensione desiderata grazie al metodo *scale()* della classe *Graphics2D*.

GamePanel si serve infine delle classi *Sound*, per definire una serie di metodi per la gestione degli effetti sonori, e *ScoreManager*, per leggere e scrivere da file il punteggio massimo conseguito e visualizzarlo nel menù iniziale.

GameManager

La classe *GameManager* ha il compito di regolare sia il flusso del gioco sia l'ordine con cui vengono renderizzate le componenti grafiche.

I suoi due metodi principali, *update()* e *draw()*, chiamati rispettivamente nelle equivalenti classi di *GamePanel*, ricevono da quest'ultimo le istanze delle classi da controllare.
Tramite un susseguirsi di stati, rappresentati dalla variabile *phase*, il flusso del gioco viene controllato facendo corrispondere ad ogni particolare fase diversi cambiamenti degli stati delle altre entità, quali *Dog*, *Duck* e *HUDManager*.

Con riferimento al metodo *update()*, a titolo di esempio, possiamo considerare il caso in cui il giocatore non abbia superato il round. In questo caso la classe GameManager si occupa, nell'ordine, di settare lo stato del cane affinché egli salga con la corretta animazione, in particolare l'animazione LAUGH, resettare lo stato della papera e far partire l'effetto sonoro corrispondente.

Duck
La classe *Duck* rappresenta il cuore del gioco: essa gestisce la logica di volo, l'alternarsi dei diversi tipi di papera con i rispettivi punteggi secondo le rispettive probabilità e la memorizzazione in appositi attributi delle variabili di gioco, quali numero di papere colpite e numero del round corrente.

Tra gli attributi più importanti della classe *Duck* troviamo la posizione, la velocità e gli stati che la descrivono, quali *DuckState*, *DuckType* e *DuckAnimation*, rispettivamente per lo stato (es. HIT), per il tipo (es. BLACK) e l'animazione da visualizzare (es. FLYSTRAIGHTRIGHT).
Ancora una volta la classe dispone dei metodi *update()* e *draw()*, richiamati al momento opportuno all'interno della classe *GameManager*.

Per far volare la papera si utilizza una serie di condizioni contenute nel metodo *update()* di *Duck* grazie a cui sono stati definiti alcuni angoli di attacco a seguito dell'urto del personaggio contro una parete ed assegnati ad essi i corrispettivi angoli di ripartenza desiderati.
L'aggiornamento delle animazioni della papera a seguito di un urto è gestito nel metodo *flightManager()*, che seleziona il set di sprites corretto rispetto all'angolo di ripartenza.

Per visualizzare la corretta animazione della papera ad ogni suo diverso stato si sono usati diversi sprites per ciascuna classe di movimento, facendoli alternare opportunamente in maniera ordinata.

La classe *Duck* interagisce sia con la classe *Dog* che con la classe *HUDManager*, al fine di coordinare le animazioni della papera con quelle del cane e di visualizzare al momento giusto le informazioni desiderate all'interno dell'hud di gioco, come ad esempio il numero di papere colpite o la papera corrente.

Dog
Anche in questo caso, come per la classe *Duck*, disponiamo dei metodi *update()* e *draw()*, i quali sono utilizzati alla stessa maniera per rappresentare le animazioni del personaggio durante le diverse fasi del gioco. Troviamo in maniera simile alla classe precedentemente descritta gli attributi della posizione del cane nello spazio di gioco, del suo stato e della animazione corrente da visualizzare.

La classe *Dog* e la classe *Duck* interagiscono tra di loro in diversi punti, come ad esempio nel momento in cui *Dog* , passando allo stato WAIT al termine della sua animazione, effettua un chiamata al metodo *resetDuckTime()*, necessario per resettare il timer che misura il tempo massimo che il giocatore ha a disposizione prima che la papera scappi.

HUDManager
La classe *HUDManager*, al contrario di quelle precedenti, non possiede degli stati propri, bensì coordina le sue azioni sulla base della conoscenza degli stati in cui si trovano le altre entità.
Essa gestisce le animazioni e la visualizzazione delle informazioni che avvengono a livello di HUD in tutte le fasi del gioco, come i proiettili rimanenti, il numero delle papere colpite (segnate in rosso), l'animazione delle paperelle rosse posizionate in basso sullo schermo a seguito di una vittoria/sconfitta ed il variare del punteggio nel corso della partita.
Interagisce con la classe *Duck* per la visualizzazione del punteggio e dell'avanzamento del gioco, con *Dog* per la sincronizzazione dei vari label di GameOver e NextRound e con *GameManager*

per la renderizzazione tramite *Graphics2D*.

HitHandler
La classe *HitHandler* gestisce la meccanica principale del gioco, ovvero quella dello sparo mediante click del mouse. Per permettere l'"ascolto" da parte del pannello di gioco di eventuali interazioni con l'utente è stata realizzata implementando l'interface *MouseListener*.

Possedendo un'istanza della classe *Duck* è in grado di verificare, a partire dalla posizione in ciascun momento della papera, se il click del mouse è avvenuto all'interno della hitbox relativa alla sua posizione ed, in caso affermativo, di impostare lo stato della papera ad HIT.

La classe interagisce inoltre con *HUDManager*, per l'aggiornamento del numero di proiettili rimanenti, e con la classe *GamePanel*, per la gestione dei suoni relativi allo sparo.

KeyHandler
La classe implementa l'interface *KeyListner* per permettere la ricezione di input da tastiera da parte dell'utente all'interno del menu e durante le fasi del gioco.
Per vincolare gli input solo alle fasi di gioco desiderate e prevenire comportementi indesiderati abbiamo bisogno di fornire a *KeyHandler* le istanze di *GamePanel*, *GameManager*, *Dog* e *Duck*, cosicchè possa accedere ai loro stati.

Ad esempio, il comando ESC per tornare al menu iniziale a seguito di un GameOver non deve poter essere premuto mentre la papera sta volando o mentre il cane sta eseguendo l'animazione di ingresso.

ScoreManager
I metodi della classe *ScoreManager* sono chiamati all'interno di *GamePanel* ad ogni avvio del gioco oppure al termine di ogni partita per permettere la lettura e/o scrittura del file di testo contenente l'high score corrente e per permetterne la sua visualizzazione nel menu iniziale.

Grazie al metodo *getResPath()*, la classe *ScoreManager* è in grado di ottenere il percorso assoluto della cartella contenente il file su cui memorizzare l'high score e di consentire così l'esecuzione dei propri metodi *saveHighScore* ed *updateHighScore* da parte di *GamePanel*.

Sound
Questa classe fornisce i controlli elementari, come *setFile()* o *play(),* a partire dalla quale sono realizzati i metodi definiti in GamePanel per la riproduzione dei suoni durante il gioco.

Segue uno schema delle classi dettagliato realizzato secondo la specifica UML in *Figura 3.3*.

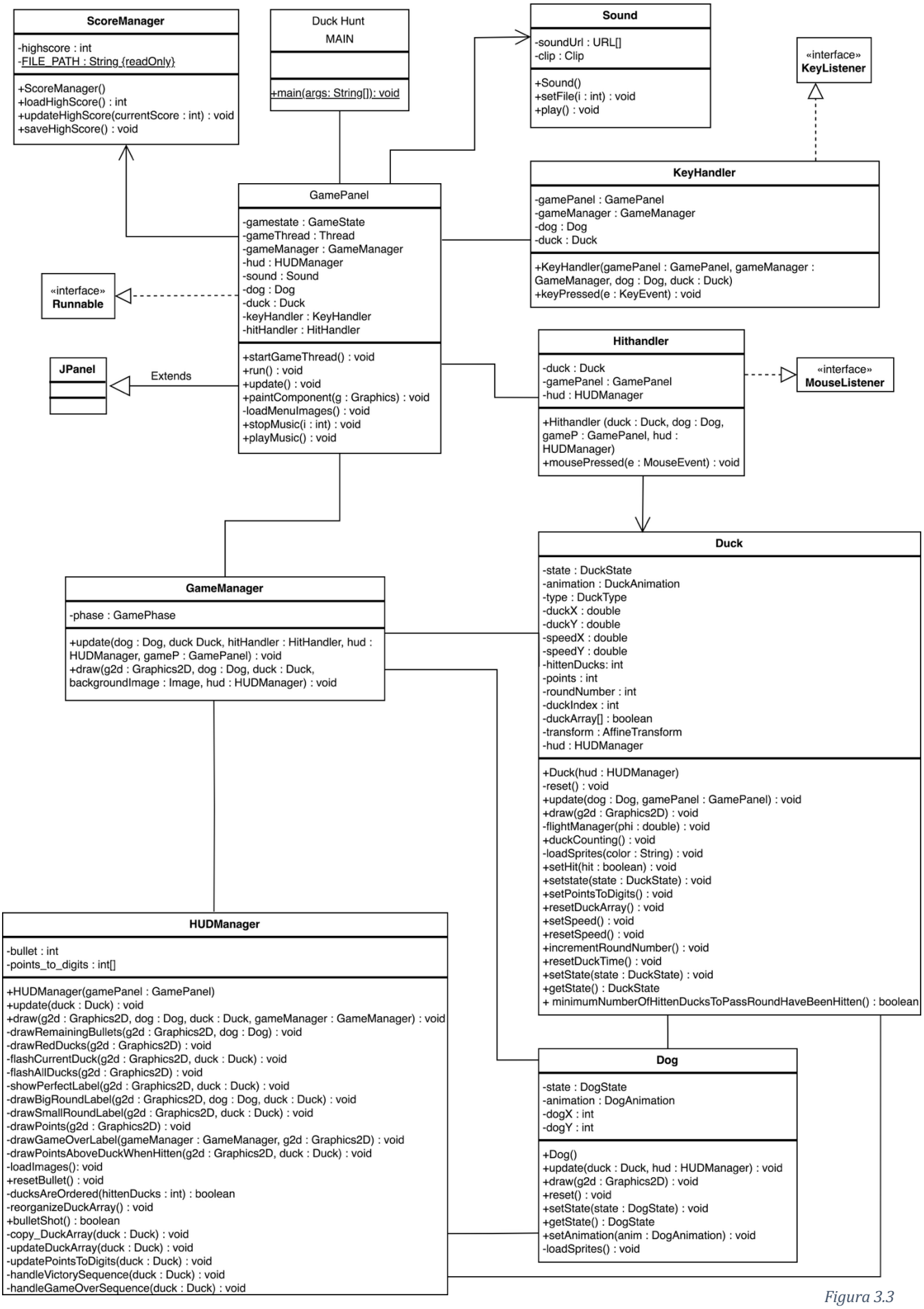
ScoreManager
-highscore : int
-FILE_PATH : String {readOnly}
+ScoreManager()
+loadHighScore() : int
+updateHighScore(currentScore : int) : void
+saveHighScore() : void
Duck Hunt
MAIN
+main(args: String[]): void
Sound
-soundUrl : URL[]
-clip : Clip
+Sound()
+setFile(i : int) : void
+play() : void
«interface»
KeyListener
KeyHandler
-gamePanel : GamePanel
-gameManager : GameManager
-dog : Dog
-duck : Duck
+KeyHandler(gamePanel : GamePanel, gameManager : GameManager, dog : Dog, duck : Duck)
+keyPressed(e : KeyEvent) : void
GamePanel
-gamestate : GameState
-gameThread : Thread
-gameManager : GameManager
-hud : HUDManager
-sound : Sound
-dog : Dog
-duck : Duck
-keyHandler : KeyHandler
-hitHandler : HitHandler
+startGameThread() : void
+run() : void
+update() : void
+paintComponent(g : Graphics) : void
-loadMenuImages() : void
+stopMusic(i : int) : void
+playMusic() : void
«interface»
Runnable
JPanel
Extends
Hithandler
-duck : Duck
-gamePanel : GamePanel
-hud : HUDManager
+Hithandler (duck : Duck, dog : Dog, gameP : GamePanel, hud : HUDManager)
+mousePressed(e : MouseEvent) : void
«interface»
MouseListener
Duck
-state : DuckState
-animation : DuckAnimation
-type : DuckType
-duckX : double
-duckY : double
-speedX : double
-speedY : double
-hittenDucks: int
-points : int
-roundNumber : int
-duckIndex : int
-duckArray[] : boolean
-transform : AffineTransform
-hud : HUDManager
+Duck(hud : HUDManager)
-reset() : void
+update(dog : Dog, gamePanel : GamePanel) : void
+draw(g2d : Graphics2D) : void
-flightManager(phi : double) : void
+duckCounting() : void
-loadSprites(color : String) : void
+setHit(hit : boolean) : void
+setstate(state : DuckState) : void
+setPointsToDigits() : void
+resetDuckArray() : void
+setSpeed() : void
+resetSpeed() : void
+incrementRoundNumber() : void
+resetDuckTime() : void
+setState(state : DuckState) : void
+getState() : DuckState
+ minimumNumberOfHittenDucksToPassRoundHaveBeenHitten() : boolean
GameManager
-phase : GamePhase
+update(dog : Dog, duck Duck, hitHandler : HitHandler, hud : HUDManager, gameP : GamePanel) : void
+draw(g2d : Graphics2D, dog : Dog, duck : Duck, backgroundImage : Image, hud : HUDManager) : void
HUDManager
-bullet : int
-points_to_digits : int[]
+HUDManager(gamePanel : GamePanel)
+update(duck : Duck) : void
+draw(g2d : Graphics2D, dog : Dog, duck : Duck, gameManager : GameManager) : void
-drawRemainingBullets(g2d : Graphics2D, dog : Dog) : void
-drawRedDucks(g2d : Graphics2D) : void
-flashCurrentDuck(g2d : Graphics2D, duck : Duck) : void
-flashAllDucks(g2d : Graphics2D) : void
-showPerfectLabel(g2d : Graphics2D, duck : Duck) : void
-drawBigRoundLabel(g2d : Graphics2D, dog : Dog, duck : Duck) : void
-drawSmallRoundLabel(g2d : Graphics2D, duck : Duck) : void
-drawPoints(g2d : Graphics2D) : void
-drawGameOverLabel(gameManager : GameManager, g2d : Graphics2D) : void
-drawPointsAboveDuckWhenHitten(g2d : Graphics2D, duck : Duck) : void
-loadImages(): void
+resetBullet() : void
-ducksAreOrdered(hittenDucks : int) : boolean
-reorganizeDuckArray() : void
+bulletShot() : boolean
-copy_DuckArray(duck : Duck) : void
-updateDuckArray(duck : Duck) : void
-updatePointsToDigits(duck : Duck) : void
-handleVictorySequence(duck : Duck) : void
-handleGameOverSequence(duck : Duck) : void
Dog
-state : DogState
-animation : DogAnimation
-dogX : int
-dogY : int
+Dog()
+update(duck : Duck, hud : HUDManager) : void
+draw(g2d : Graphics2D) : void
+reset() : void
+setState(state : DogState) : void
+getState() : DogState
+setAnimation(anim : DogAnimation) : void
-loadSprites() : void

Figura 3.3

3.3 Problemi riscontrati

Un primo aspetto piuttosto delicato in cui ci siamo imbattuti è sicuramente quello del rimbalzo dell'anatra contro i bordi del pannello di gioco. Questo perché il personaggio non si limita ad un rimbalzo qualsiasi, ma lo fa entro certi angoli a seconda dell'angolo di attacco con cui arriva a scontrarsi con il bordo. In un primo momento abbiamo provato ad implementare, all'interno della classe *Duck*, un rimbalzo con un angolo di 180° rispetto all'asse giaciente lungo il bordo interessato, non riscontrando tuttavia un risultato soddisfacente. Questo poiché l'anatra poteva ripartire verso la stessa direzione da cui arrivava, o ancora poteva ripartire seguendo una traiettoria perfettamente orizzontale o verticale, non seguendo in maniera fedele le dinamiche del gioco orginale alla quale ci siamo ispirati. Abbiamo allora deciso di procedere con un approccio più elaborato andando a dividere i possibili angoli di attacco e di ripartenza in settori ed assegnando ad ogni angolo di arrivo un particolare range da cui poter estrarre un angolo di ripartenza randomico. La scelta dell'animazione della papera è in funzione dell'angolo con cui rimbalza; ne sono presenti infatti di due tipi diversi, una che mostra il personaggio volare in orizzontale, per angoli di rimbalzo che fanno si che la papera si muova parallelemente al terreno, ed una raffigurante l'anatra in una posizione obliqua, utilizzata per angoli di ripartenza che fanno si che la pepera segua una traiettoria obliqua rispetto alla schermata di gioco. Una logica simile è stata usata anche per calcolare degli angoli di partenza randomici ad ogni respawn dell'anatra.

Un altro problema incontrato è stato quello della velocità con cui l'anatra rimbalza sui bordi. Considerando la velocità come scomposta in due componenti, orizzontale e verticale, inizialmente abbiamo fatto corrispondere ad ogni rimbalzo su una specifica parete un cambio di verso della rispettiva componente semplicemente moltiplicandola per (-1). Ad esempio, se ci si trova a sbattere contro la parete sinistra, indifferentemente arrivando dall'alto o dal basso, basta cambiare il verso della componente orizzontale della velocità per far cambiare direzione all'anatra mantenendo un rimbalzo 'lineare'. Una volta introdotta la logica per cui il rimbalzo dovesse variare non solo verso ma anche angolo di ripartenza della papera questo approccio si è rivelato essere fallace. La papera, infatti, oltre che cambiare verso, aumentava o diminuiva la sua velocità di rimbalzo in rimbalzo. Per risolvere questo problema abbiamo pensato sì di scomporre in componenti il vettore velocità, ma questa volta considerando quest'ultima come un raggio di modulo costante e moltiplicandola per il seno ed il coseno dell'angolo di ripartenza rispettivamente per calcolare la componente verticale ed orizzontale del vettore. Utilizzando quest'ultimo approccio è stato possibile implementare la logica del rimbalzo casuale entro certi angoli scelti, pur mantenendo la velocità caratterizzante il round in esame costante ad ogni rimbalzo.

A seguito di aver implementato la nuova logica sopra descritta, tuttavia, ci siamo ritrovati con un nuovo problema in cascata. La necessaria conversione del tipo delle variabili *duckX* e *duckY* contenenti la posizione della papera da *int* a *double* ci ha portato ad alcune difficoltà nell'usare il metodo *g2d.drawImage(Image img, int duckX, int duckY, ImageObserver imObv)* della classe *Graphics2D* all'interno del metodo *draw()* della classe *Duck*. Dopo aver dapprima provato con un semplice cast esplicito, senza successo, abbiamo optato per usare la classe *AffineTransform* per convertire le coordinate in modo tale da poter fornire al *drawImage()* di *g2d* dei valori adeguati a rappresentare il personaggio secondo le traiettorie impostate.

Per quanto riguarda la gestione degli effetti sonori si è scelto di utilizzare la libreria nativa *javax.sound.sampled* piuttosto che librerie esterne, come ad esempio *JLayer*. Per i nostri scopi, ossia la riproduzione di effetti sonori di breve durata in formato *.wav*, non abbiamo infatti riscontrato la necessità di usare una libreria più complessa, in quanto *Clip* offre nativamente tutti i comandi di cui abbiamo avuto bisogno per riprodurre i vari effetti sonori durante il gioco. Inoltre l'utilizzo di una libreria nativa di java garantisce sia una maggiore portabilità sia la compatibilità con qualsiasi versione di Java SE, riducendo la complessità del progetto ed eliminando la necessità di dipendenze esterne. L'unica problematica nella gestione dei suoni mediante *Clip* è emersa nel momento in cui si è andato ad aggiungere il suono dello sparo sovrapponendolo a quello della papera che vola. Per evitare di creare più istanze di *Clip* si è andato, nel momento in cui il giocatore spara, ad interrompere il suono dello papera che vola per poi riprenderlo dal punto in cui si era interrotto una volta terminato l'effetto sonoro del colpo.

4. Possibilità di personalizzazione

È stata prevista la possibilità di alcune personalizzazioni per quanto riguarda la dimensione della finestra e la velocità della papera nei round.

I background sono stati forniti all'applicativo nella loro risoluzione nativa, ovvero quella propria della console NES di 256x224 pixel ed allo stesso modo gli sprites hanno una dimensione ad essa rapportata. Le risoluzioni notevolmente più elevate dei monitor in nostro possesso oggi, tuttavia, non permetterebbero una visualizzazione soddisfacente della schermata di gioco, la quale risulterebbe essere estremamente piccola. Per ingrandire ad una dimensione accettabile è stato usato all'interno del *GamePanel* il metodo *scale()* della classe *Graphics2D* il quale, a valle di tutte le operazioni di logica e grafica del software, scala tutto al fattore di scala specificato, nel nostro caso SCALE_FACTOR = 3. La risoluzione ottenuta risulta in questo modo essere di 768x732 pixel. Nel caso in cui ci si trovi ad utilizzare l'applicativo in macchine diverse da quella in cui è stato progettato e si senta conseguentemente il bisogno di cambiare le dimensioni della schermata di gioco, è possibile farlo semplicemente cambiando il fattore di scala ad un intero più grande, se la si vuole ingrandire, o più piccolo, se si vuole ottenere l'effetto contrario.

La velocità della papera, allo stesso modo, è fatta aumentare nel corso dei diversi round da un valore iniziale di 2.3 di 0.8 per ogni round. Nel caso in cui non si fosse soddisfatti della durata del gameplay, stimata attualmente attorno agli 8-10 minuti massimi, si può tranquillamente agire sul parametro speed, contenente la velocità iniziale, o sul metodo *setSpeed()* della classe *Duck*, responsabile dell'aumento della velocità al passare dei round. Diminuendo il fattore incrementale all'interno di *setSpeed()*, ad esempio, si può allungare il tempo di durata del gameplay fino ad un minutaggio soddisfacente.

5. Bibliografia

https://www.retrogames.cc/nes-games/duck-hunt-world.html

https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics2D.html

https://docs.oracle.com/javase/8/docs/api/java/awt/geom/AffineTransform.html

https://docs.oracle.com/javase/8/docs/api/javax/sound/sampled/package-summary.html

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html

https://docs.oracle.com/javase/8/docs/api/java/io/BufferedWriter.html

https://docs.oracle.com/javase/8/docs/api/java/io/File.html

https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

https://docs.oracle.com/javase/7/docs/api/java/awt/event/KeyListener.html

https://docs.oracle.com/javase/7/docs/api/java/awt/event/MouseListener.html

https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics.html

https://docs.oracle.com/javase/8/docs/api/javax/imageio/ImageIO.html

È stato inoltre fatto riferimento alle dispense messe a disposizione nella pagina Unistudium del corso.