A.D. 1308
unipg
DIPARTIMENTO
DI INGEGNERIA

Tesina Finale di

Programmazione di Interfacce Grafiche e Dispositivi Mobili

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2022-2023

Dipartimento di Ingegneria

docente

Prof. Luca Grilli

JPool

applicazione desktop JavaFX

studenti

330265	**Riccardo**	**Nicolini**	riccardo.nicolini1@studenti.unipg.it
329673	**Giovanni**	**Versiglioni**	giovanni.versiglioni@studenti.unipg.it

5 dicembre 2022

0. Indice

1 Descrizione del Problema **2**
1.1 Il gioco del biliardo: Palla 8 2
1.2 L'applicazione JPool 4

2 Specifica dei Requisiti **5**

3 Progetto **7**
3.1 Architettura del Software 7
3.2 Descrizione dei Pacchetti 8
3.2.1 Entities 8
3.2.2 Logic 9
3.2.3 Resources 15
3.2.4 Utils 15
3.3 Problemi Riscontrati 16

4 Appendice **18**

5 Bibliografia **23**

1. Descrizione del Problema

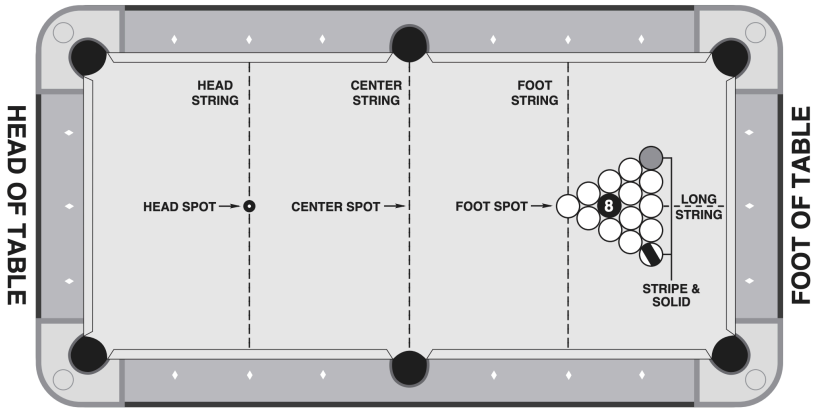
L'obiettivo di questo lavoro è lo sviluppo di un '2 Player Game', denominato JPool, che implementa una versione semplificata del già noto videogioco 8 Ball Pool.

1.1 Il gioco del biliardo: Palla 8

Il gioco 'Palla 8' è una specialità del biliardo. [1]

Si gioca con una bilia bianca e 15 bilie colorate e numerate. Di queste ultime si distinguono quelle totalmente colorate (piene), numerate dalla 1 alla 8, da quelle parzialmente colorate (spezzate), numerate dalla 9 alla 15. La bilia 8 (nera), nonostante sia rappresentata come piena, non è considerata (da un punto di vista regolamentare) né piena né spezzata.

L'obiettivo del gioco è imbucare le bilie colorate, potendo solo colpire la bilia bianca tramite la stecca. Un giocatore deve imbucare le bilie piene, l'altro le bilie spezzate. Il giocatore che per primo imbuca tutte le bilie della propria categoria, può imbucare la bilia nera. Colui che imbuca la bilia nera vince.



I. INIZIO ('BREAK')

La bilia bianca è collocata nella posizione di 'head spot'.

Le bilie colorate sono raggruppate in un triangolo equilatero (pieno) di lato pari a 5 bilie.

La bilia 1 è posizionata nella posizione di 'foot spot' e costituisce il vertice del triangolo in direzione della bilia bianca.

Nelle posizioni corrispondenti ai due vertici rimanenti sono posizionate una bilia piena e una bilia spezzata[1].

Le posizioni rimanenti del triangolo, eccetto quella centrale dove è inserita la bilia nera[2], sono riempite con le bilie restanti in modo casuale.

II. ASSEGNAZIONE

A seguito del break, se è stata imbucata almeno una bilia colorata (che non sia la nera), il giocatore che ha spaccato rimane al tavolo. Altrimenti, passa al tavolo l'altro giocatore.

A partire dal secondo tiro, al giocatore che per primo imbuca una bilia piena o spezzata, viene assegnata la categoria corrispondente. All'altro giocatore viene assegnata l'altra.[3]

Da questo momento, il giocatore ha l'obbligo di colpire per prima una delle proprie bilie e ha diritto a rimanere al tavolo finché continuerà ad imbucarle in modo regolare.

III. FALLI

Si verificano situazioni di fallo nei seguenti casi:

- la bilia bianca viene imbucata;
- la bilia bianca non colpisce nessuna bilia;
- viene colpita una bilia dell'avversario;
- viene colpita la bilia nera quando ancora non sono state imbucate tutte le proprie bilie.

[1] Le bilie che si trovano in corrispondenza dei due vertici inferiori del triangolo sono quelle con maggiore probabilità di essere imbucate al momento della spaccata. Per rendere uguale la probabilità che una bilia piena o una bilia spezzata sia imbucata durante la spaccata, si inseriscono in tali posizioni una bilia piena e una bilia spezzata.

[2] La bilia al centro del triangolo è quella con minore probabilità di essere imbucata al momento della spaccata. Per questo, ci si posiziona la bilia nera.

[3] Una qualsiasi bilia imbucata nel break non assegna la categoria corrispondente.

In quattro casi l'avversario vince direttamente la partita:

- quando viene imbucata la bilia nera ma ancora non sono state imbucate tutte le proprie bilie;
- quando viene imbucata la bilia nera in una buca diversa da quella dichiarata;
- quando la bilia bianca viene imbucata dopo la bilia nera.

Se un giocatore commette un fallo, il giocatore avversario (passato al tavolo) ha il vantaggio di poter posizionare la bilia bianca in una qualsiasi posizione del tavolo.

IV. FINE

Una volta che un giocatore ha imbucato tutte e sette le proprie bilie può colpire e imbucare la bilia nera , dichiarando la buca. Il giocatore che imbuca la bilia nera vince.

1.2 L'applicazione JPool

L'applicazione JPool implementa il gioco del biliardo attraverso una grafica bidimensionale, che rappresenta il piano del tavolo visto dall'alto.

La simulazione del colpo, effettuato tramite la stecca, è realizzata con una stecca grafica la cui punta è posizionata esattamente dove si trova la bilia bianca. Mantenendo fissa la punta, è possibile ruotare radialmente la stecca per indirizzare il colpo. Inoltre, l'utente può regolare la potenza del tiro e, graficamente, viene mostrato un adeguato caricamento della stecca.

Un giocatore può indirizzare la bilia bianca tramite il puntatore del mouse. Il videogioco mostra la linea che congiunge la bilia bianca e la posizione desiderata; e in tale posizione genera la 'ghost ball'. La 'ghost ball' rappresenta graficamente la posizione della bilia bianca al momento della collisione con la palla che si intende colpire.

Inoltre, nella modalità *Facile*, sono mostrate le traiettorie in uscita della bilia che si intende colpire e della bilia bianca a seguito di collisione. Al variare della posizione del puntatore del mouse in corrispondenza dei punti circostanti la bilia che si intende colpire, variano le traiettorie in uscita.

Al fine di rendere l'esperienza di gioco più dinamica, l'applicazione limita il tempo a disposizione per effettuare un colpo a 30 secondi. La scadenza del tempo a disposizione, prima di aver effettuato il tiro, comporta il fallo.

2. Specifica dei Requisiti

L'applicazione JPool dovrà soddisfare i seguenti requisiti:

1. Rispetto delle leggi fisiche e matematiche coinvolte: la dinamica del gioco deve tenere conto delle dinamiche reali (Legge degli Urti, Forza d'Attrito, Legge di Riflessione), ma allo stesso tempo deve realizzarne un modello approssimato per garantire un adeguato livello di difficoltà;

2. Rispetto delle regole del gioco 'Palla 8' sopra elencate;

3. Fluidità del gioco: l'animazione delle bilie in movimento deve essere fedele alle dinamiche reali, producendo un movimento lineare e privo di lag;

4. Effetto tridimensionale: la GUI deve fornire una parvenza reale del gioco mediante giochi di ombre e altre eventuali tecniche;

5. Menu di avvio

   → *Play*: consente di avviare una partita;

   → *Settings*: permette di modificare la grafica, le impostazioni audio e il livello di difficoltà relativamente alla partita avviata;

   → *How to Play*: fornisce una descrizione dettagliata dei comandi necessari per giocare;

   → *Exit*: termina ed esce dal gioco.

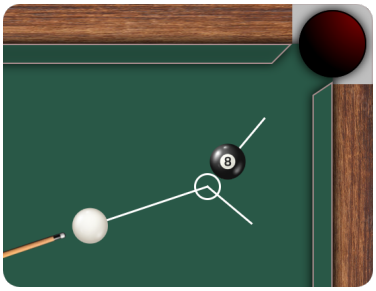
6. Personalizzazione grafica: possibilità di personalizzare il tavolo e la stecca di gioco;

7. Presenza di audio: urto tra bilie, bilia imbucata, cambio turno, fallo commesso, ticchettio tempo a disposizione, vittoria;

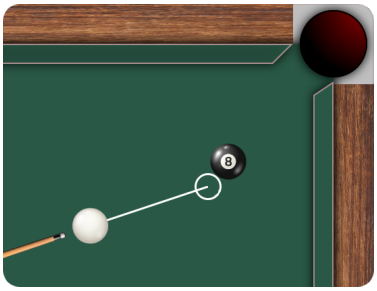
8. Diversi livelli di difficoltà

   → *Facile*: sono mostrate le anteprime delle traiettorie in uscita (Figura 2.1a)

   → *Difficile*: non sono mostrate le anteprime delle traiettorie in uscita (Figura 2.1b)



(a) Modalità Facile.



(b) Modalità Difficile.

3. Progetto

Si descrive ora la struttura dell'applicazione realizzata, illustrandone prima l'architettura software e scendendo poi nel dettaglio dei blocchi funzionali che la compongono.

3.1 Architettura del Software

Per semplificare lo sviluppo dell'applicazione si è scelto di seguire un'architettura monolitica: tutte le componenti e tutte le funzionalità del software appartengono ad un unico blocco (monolite).

Nonostante ciò, le classi e le risorse che compongono l'applicazione sono state suddivise in pacchetti; ognuno dei quali ha un preciso ruolo.

Il pacchetto `game.entities` comprende le entità direttamente coinvolte nello svolgimento del gioco: i giocatori, le bilie e la loro posizione e velocità nel piano di gioco.

Il pacchetto `game.logic` comprende le classi che codificano le regole del gioco e che gestiscono i cambiamenti di scena. Le classi di tipo 'FXML controller' sono collegate alle rispettive scene di gioco (FXML) e permettono di associare metodi alle componenti grafiche presenti in esse.

Il pacchetto `game.resources` non solo contiene le immagini e i suoni presenti nell'applicazione, ma contiene anche i file di definizione e di personalizzazione dell'interfaccia grafica presentata all'utente. In particolare, il pacchetto `game.resources.gui.def` contiene i file FXML che definiscono le componenti e le loro proprietà grafiche statiche presenti nelle scene dell'applicazione; mentre il pacchetto `game.resources.gui.style` contiene i file CSS che permettono di aggiungere proprietà grafiche in modo dinamico.

Il pacchetto `game.utils` comprende le classi rimanenti, di varia utilità nello sviluppo del software.

3.2 Descrizione dei Pacchetti

3.2.1 Entities

VECTOR

La classe `Vector.java` rappresenta un vettore bidimensionale uscente dall'origine del Pane. I metodi della classe implementano le operazioni vettoriali di:

- somma (`add(Vector v)`)
- differenza (`sub(Vector v)`)
- moltiplicazione scalare (`multiply(double k)`)
- prodotto scalare (`scalar(Vector v)`)
- normalizzazione (`normalize()`)
- vettori perpendicolari (`perpendicularLeft()`, `perpendicularRight()`)

Un vettore può anche invocare il metodo `determinant(Vector a, Vector b)`, che restituisce il determinante della seguente matrice:

$$\begin{pmatrix} \texttt{this.x - a.getX()} & \texttt{b.getX() - a.getX()} \\ \texttt{this.y - a.getY()} & \texttt{b.getY() - a.getY()} \end{pmatrix}$$

BALL

La classe `Ball.java` implementa il modello fisico (tridimensionale) di una bilia. Dato che tutte le bilie in gioco hanno stessa dimensione, i dati da specificare per la creazione di una bilia sono la sua posizione nel piano (Pane) e il suo numero.

Il metodo `drawBall()` restituisce un nodo di tipo Sphere caratterizzato da un determinato raggio e da effetti grafici quali:

- `PhongMaterial`[2], che, data una componente diffusa (grafica della superficie esterna della bilia) e una componente speculare (colore bianco), realizza la riflessione della luce.

– `DropShadow`[3], che realizza l'ombra specificandone la profondità radiale e lo spostamento lungo gli assi coordinati.

Il metodo `triangle()` crea e dispone le 16 bilie in gioco per la spaccata iniziale come da regolamento.

Il metodo `ballAnimation(int ballNum)`, collegato alla Timeline dell'applicazione, realizza l'animazione di una bilia. Nel dettaglio, chiama il metodo `spin()` per far ruotare la bilia su se stessa rispetto gli assi x e y, il metodo `bankCollision()` per modificare la velocità della bilia a seguito di un urto con una sponda e il metodo `tableFriction()` per modificare la velocità della bilia a causa della forza d'attrito del tavolo di gioco. Inoltre, grazie al controllo del metodo `collides(Ball b)`, se la bilia in input è in collisione con una qualsiasi bilia in gioco, il metodo `ballCollision()` modifica la velocità di entrambe le bilie in collisione.

Il metodo `checkPocket(int ballNum)` collegato alla Timeline dell'applicazione, controlla se la bilia in input è "contenuta" in una delle buche del tavolo di gioco e, in caso positivo, chiama il metodo `pocketed(int ballNum)` che si occupa di eliminare la bilia dal tavolo di gioco e di inserirla nel rack laterale.

PLAYER

La classe `Player.java` rappresenta un giocatore. Per la creazione di un giocatore è sufficiente fornire un nickname. I restanti attributi del giocatore sono necessari per l'implementazione delle regole del gioco; infatti ci permettono di conoscere (in ogni momento della partita) quale dei due giocatori deve colpire, la categoria di bilie assegnata a ciascun giocatore e, a fine partita, quale dei due giocatori ha vinto.

3.2.2 Logic

MENU CONTROLLER

Il controller `MenuController.java` è associato alla schermata `Menu.fxml` e fornisce tre metodi da eseguire alla pressione di ciascuno dei tre pulsanti presenti nella scena. Il pulsante START GAME, gestito dal metodo `handleStartGameButton()`, carica la scena che permette di personalizzare una partita. Il pulsante HOW TO PLAY, gestito dal metodo `handleHTPButton()`, carica la scena esplicativa delle modalità di gioco. Il pulsante EXIT, gestito dal metodo `handleExitButton()`, chiude la finestra (Stage) e termina l'applicazione.

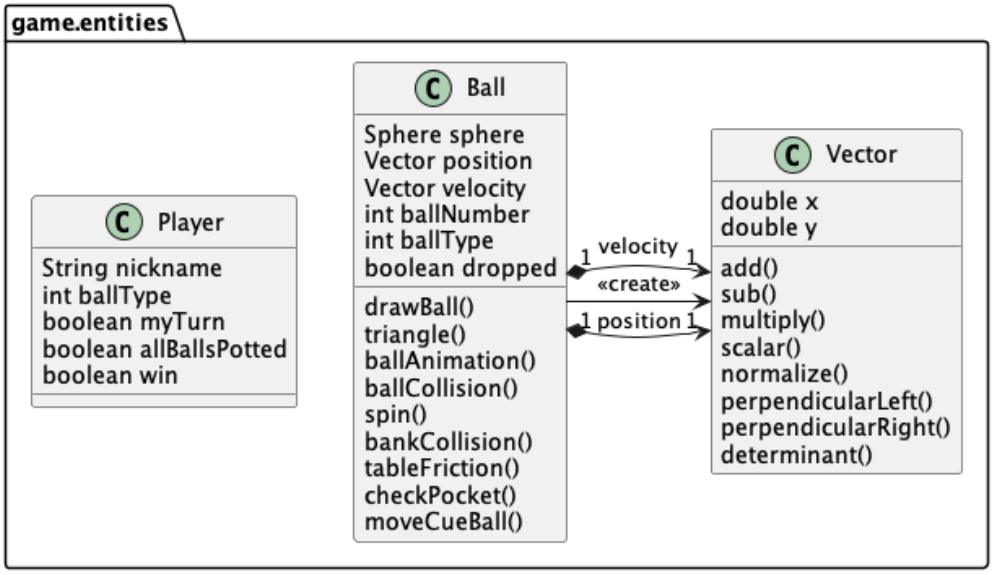


Figura 3.1: Diagramma UML (semplificato) del package `game.entities`.

SETTINGS CONTROLLER

Il controller `SettingsController.java` è associato alla schermata FXML delle impostazioni di gioco e gestisce gli input dell'utente relativamente alla personalizzazione di una partita. Per la scelta del tavolo, della stecca e della difficoltà di gioco si è scelto di utilizzare la componente grafica Pagination. Quest'ultima permette di definire una sottoscena contenente diverse pagine indicizzate. Ogni pagina può contenere diversi componenti ed è selezionabile dall'utente tramite appositi pulsanti grafici. Le tre paginazioni `tableMenu`, `cueMenu`, `modeMenu` sono definite nel metodo `initialize()` sfruttando il metodo `setPageFactory(...)`, che permette di specificare un Node da inserire in ogni pagina. Una proprietà fondamentale di un oggetto Pagination è quella di poter chiamare in ogni momento il metodo `getCurrentPageIndex()` per conoscere la pagina selezionata dall'utente e, di conseguenza, selezionare la grafica desiderata.

GAME CONTROLLER

Il controller `GameController.java` è associato alla schermata `Game.fxml` e, oltre a dirigere la partita nel tempo, contiene metodi che implementano l'esecuzione di un tiro.

Il metodo `initialize()` si occupa di impostare correttamente la grafica e la logica (variabili di controllo) all'inizio di ogni partita. Si occupa principalmente di:

– stabilire quale dei due giocatori deve iniziare, comunicando con il controller delle impostazioni di gioco;

– caricare la stecca di gioco selezionata, comunicando con il controller delle impostazioni di gioco;

– configurare le bilie per la spaccata iniziale, invocando il metodo `triangle()` della classe `Ball.java`.

La grafica dell'esecuzione di un tiro è implementata tramite le seguenti componenti della scena FXML:

– `Line guidelineToBall`, cioè la linea che congiunge la bilia bianca e la bilia che si vuole colpire;

– `Circle ghostBall`, cioè la previsione della bilia bianca al momento dell'urto;

– `Line guidelineFromBall`, cioè la linea che mostra la traiettoria della bilia colpita dopo la collisione;

– `Line guidelineFromCue`, cioè la linea che mostra la traiettoria della bilia bianca dopo la collisone.

I metodi che permettono all'utente di visualizzare la grafica di un tiro e poi di poterlo effettuare sono i seguenti.

Il metodo `guidedTrajectory(MouseEvent event)` mostra all'utente, al variare della posizione del puntatore del mouse, la grafica del tiro. Questo metodo svolge principalmente due compiti:

– aggiorna la grafics mostrando all'utente la `guidelineToBall`, la `ghostBall`, la `guidelineFromBall` e la `guidelineFromCue`;

– aggiorna la grafica ruotando la stecca intorno alla bilia bianca adeguatamente rispetto alla direzione del colpo (posizione del mouse).

Il metodo `fixTrajectory(MouseEvent event)` è associato alla scena FXML tramite l'azione di rilascio del mouse e memorizza le componenti x e y di rilascio del mouse in due apposite variabili d'istanza.

Il metodo `cueLoading()` permette di caricare il colpo, regolandone la potenza. Questo metodo si occupa di aggiornare la grafica mostrando all'utente la potenza del tiro che si vuole effettuare in base a quanto ha trascinato l'apposito slider verso il basso. Allo stesso tempo, fissata la posizione radiale della stecca intorno alla bilia bianca, la si trasla adeguatamente rispetto alla potenza scelta

in direzione opposta rispetto a quella del tiro che si vuole effettuare; emulando il reale caricamento di un colpo.

Il metodo `cueShot()` è associato alla scena FXML tramite l'azione di rilascio dello slider e si occupa di far partire il colpo. Nello specifico, imposta il vettore velocità della bilia bianca in direzione della bilia che si vuole colpire (coordinate x e y mouse released) e lo moltiplica scalarmente per la velocità della bilia bianca impostata tramite lo slider di potenza.

```
cueBallVelocity = powerSlider.getValue();

double angle = Math.atan2(yMouseReleased - ball[0].
   getPosition().getY(), xMouseReleased - ball[0].
   getPosition().getX());

setCueVelocity(cueBallVelocity * Math.cos(angle),
   cueBallVelocity * Math.sin(angle));
```

La funzione `Math.atan2(y, x)` permette di determinare l'angolo formato da un vettore `(x, y)` con l'asse x di un piano con polo nell'origine. Per determinare la direzione e il verso della bilia bianca, si vuole determinare l'angolo formato dal vettore `(xMouseReleased, yMouseReleased)` con l'asse delle ascisse di un piano cono polo nel centro della bilia bianca `(ball[0].getPosition().getX(), ball[0].getPosition().getY()`.

Noto l'angolo, si scompone la velocità della bilia bianca lungo x e y sfruttando le funzioni trigonometriche `Math.cos()` e `Math.sin()` e la si moltiplica scalarmente per la velocità impostata dallo slider.

Dato che una partita si gioca nel tempo, è necessario definire un "asse temporale" a cui siano associati i metodi necessari alla creazione delle animazioni e allo svolgimento della partita. A tal proposito, si è scelto di utilizzare la classe `Timeline`. Contestualmente al caricamento del file `Game.fxml` nel metodo `initialize()` viene creata la timeline e viene chiamato il metodo `startGame()`. Quest'ultimo imposta il periodo di aggiornamento di ogni frame a 0.015 secondi [1] e imposta la lunghezza della timeline come indefinita, in modo che la partita possa continuare fino a che non si determini un vincitore. Quindi, avvia la timeline (inizia l'animazione) e ad ogni aggiornamento di frame viene invocato il metodo `update()`.

Il metodo `update()` svolge diversi compiti, ma principalmente:

[1] La scelta del frame rate è stata fatta consultando altri progetti di videogiochi e facendo diversi tentativi.

– invoca i metodi `ballAnimation(int ballNum)` e `checkPocket(int ballNum)` per tutte le bilie in gioco, in modo da aggiornare la posizione delle bilie e, se si trovano in una delle buche del tavolo, mandarle in buca;

– controlla se tutte le bilie in gioco sono ferme e, in tal caso, aggiorna la logica e la grafica del gioco sulla base di ciò che si è verificato al turno precedente (bilie imbucate, falli).

RULES

La classe `Rules.java` contiene metodi (collegati alla Timeline) che permettono di controllare l'andamento della partita e il rispetto delle regole di gioco (falli).

Il metodo `checkFoul()` svolge diversi compiti necessari al corretto svolgimento di una partita come da regolamento ed è infatti richiamato dal metodo `update()` alla fine di ogni turno. Si occupa principalmente di gestire i cambi di turno, di gestire l'assegnazione delle bilie e di controllare e segnalare i falli.

Il metodo `checkPotted()` controlla se tutte le bilie di una categoria assegnata ad un giocatore sono state imbucate e in tal caso assegna al giocatore tale proprietà.

BOARD

La classe `Board.java` contiene metodi (collegati alla Timeline) che mostrano all'utente informazioni utili al rispetto delle regole di gioco. Nello specifico, aggiorna i seguenti Label presenti nella scena di gioco: `scoreboardLabel`, `centralboardLabel` e `foulboardLabel`. Il primo specifica in ogni momento quale dei due giocatori è al tavolo, il secondo notifica l'assegnazione delle bilie ai giocatori, la vittoria di un giocatore ed è anche utilizzato quando si vuole terminare la partita; il terzo notifica i falli rilevati e ne specifica il tipo.

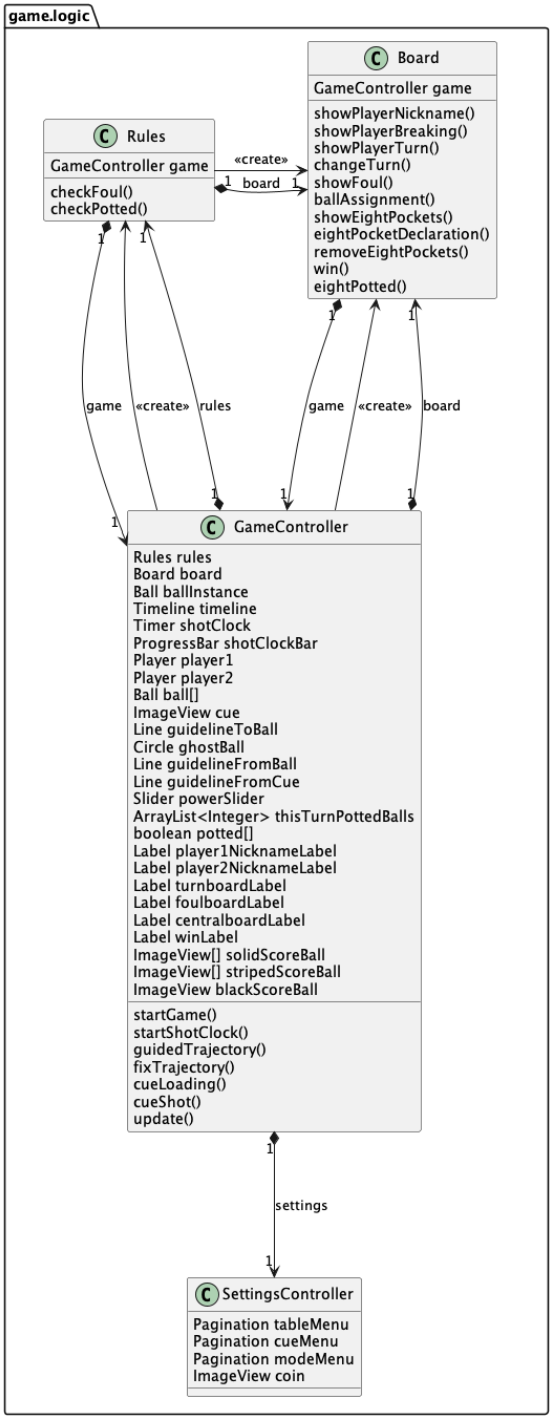


Figura 3.2: Diagramma UML (semplificato) del package `game.logic`.

3.2.3 Resources

Oltre alle immagini e ai suoni, questa sezione contiene i file di definizione (FXML) e personalizzazione (CSS) della GUI.

FXML

Ad ogni schermata di gioco corrisponde un file FXML che ne definisce la struttura grafica e che contiene i metodi da eseguire in risposta agli input dell'utente (eventi). Per la realizzazione della parte statica delle schermate di gioco si è utilizzato il software SceneBuilder, fornito dal framework JavaFX. [4]

CSS

Contestualmente al caricamento di ogni schermata tramite FXMLLoader, viene aggiunta la 'stylesheet' `standard.css`. Quest'ultima contiene una serie di proprietà grafiche, relative alle varie componenti delle schermate, che caratterizzano lo stile dell'applicazione. Le proprietà grafiche sono specificate principalmente su classi di componenti, piuttosto che su di una singola componente; perché si vuole creare uno stile grafico, preciso ed univoco per ogni componente dell'applicazione. Ad esempio, la classe dei pulsanti dell'applicazione fa riferimento allo stile mostrato in Figura 3.3.

Figura 3.3: Stile grafico dei pulsanti di JPool.

Sono inoltre presenti quattro file CSS per la personalizzazione grafica del tavolo di gioco. In base alla scelta dell'utente nella schermata di impostazioni di gioco, viene caricata, dinamicamente, una precisa configurazione che modifica il legno esterno del tavolo e il colore del piano di gioco e delle sponde. Le componenti da modificare sono incluse in una precisa 'CSS style class' (table, bank), in modo da poter essere referenziate nella 'CSS stylesheet' caricata (`table1,2,3,4.css`).

3.2.4 Utils

La classe `Constants.java` contiene costanti utili allo sviluppo del gioco.

Ad esempio, nella sezione `//SPLIT` sono tabulate tutte le coordinate x e y necessarie al raggruppamento delle bilie nel triangolo del break.

La classe `Sounds.java` permette di riprodurre effetti sonori.

3.3 Problemi Riscontrati

Durante lo sviluppo del videogioco sono stati riscontrati problemi di diversa natura.

Problemi di natura tecnica, cioè riguardanti la sintassi e la semantica della tecnologia utilizzata (JavaFX), sono stati risolti consultando la documentazione ufficiale (JavaFX API) oppure consultando domande (e relative risposte) inerenti nel sito stackoverflow.com.

Problemi di natura logica, cioè relativi al corretto funzionamento del software e all'assoluzione delle specifiche, sono stati trattati caso per caso con diversi metodi.

TRAIETTORIA BILIA BIANCA

Il calcolo della traiettoria della bilia bianca, dopo un'ipotetica collisione con un'altra bilia, ha richiesto particolare attenzione. Infatti, l'utente sposta la ghost ball intorno a una bilia che vuole colpire per visualizzare le traiettorie post collisone e, solo dopo aver trovato la direzione desiderata, carica e rilascia il colpo, assegnando una certa velocità alla bilia bianca. Il problema è che, quando l'utente sta decidendo la direzione del colpo, non esiste un vettore velocità della bilia bianca (da applicare al centro della ghost ball) che ci permetta di calcolare le velocità finali come nel metodo `ballCollision(int ballNum)`.

Il vettore traiettoria delle bilia che si intende colpire (`ballFinalVelocity`) è stato calcolato assumendo una velocità scalare della bilia bianca (`cueBallVeloctiy`) pari a 50.

Per il vettore traiettoria della bilia bianca (`cueFinalVelocity`), si è deciso di calcolarlo come uno dei due vettori perpendicolari al vettore `ballFinalVelocity`. In effetti, in una collisione bidimensionale tra due bilie, è sempre vero che le bilie seguono direzioni tra di loro ortogonali. Resta solo da stabilire adeguatamente il verso del vettore ortogonale, infatti può essere tanto in una direzione quanto nell'altra.

La soluzione a quest'ultimo problema è stata trovata notando che: ruotando la ghost ball attorno alla bilia da colpire, questa dovrà andare in un verso se il suo centro si trova alla sinistra della linea `guidelineToBall` (che unisce la bilia bianca e la bilia da colpire), nel verso opposto altrimenti. La condizione che, dati tre punti, permette di determinare se un punto si trova a sinistra o a destra di una retta (individuata dagli altri due) è fornita dal segno del determinante della matrice:

$$\begin{pmatrix} \texttt{x - x1} & \texttt{x2 - x1} \\ \texttt{y - y1} & \texttt{y2 - y1} \end{pmatrix}$$

dove $P = (x, y)$ è il punto da verificare e $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ sono i punti che individuano la retta.[5][6]

Il metodo `determinant(Vector a, Vector b)`, trattato nella sezione 3.2.1, è stato realizzato a tale scopo. Il punto (vettore) che invoca il metodo è il punto che si vuole verificare (centro ghostball), mentre i due punti in input individuano la retta (centro bilia bianca, centro bilia da colpire).

Se il determinante è positivo, allora il punto si trova a sinistra della retta e quindi il vettore `cueFinalVelocity` è il perpendicolare sinistro di `ballFinalVelocity`. Se il determinante è negativo, allora il punto si trova a destra della retta e quindi il vettore `cueFinalVelocity` è il perpendicolare destro di `ballFinalVelocity`. Se il determinante è nullo, allora il punto giace sulla retta e quindi il vettore `cueFinalVelocity` ha stessa direzione e verso di `ballFinalVelocity`, per questo si imposta come nullo.

SOVRAPPOSIZIONE SUONI

L'applicazione prevede l'esecuzione di effetti sonori in diverse circostanze. Una di queste è l'urto tra due bilie.

Al momento della spaccata, le bilie sono raggruppate in modo compatto in un triangolo. Quando uno dei due giocatori spacca, colpendo (in generale) la bilia sulla punta del triangolo, si verificano molti urti tra bilie.

Dato che ogni urto riproduce l'effetto sonoro di collisione, si sovrappongono in un breve intervallo temporale tanti suoni. Testando l'applicazione, si è notato che, talvolta, questa fitta riproduzione e sovrapposizione di suoni manda in blocco il software.

Si è risolto il problema disattivando l'effetto sonoro di collisone per il primo tiro, avviando un suono di spaccata iniziale.

4. Appendice

In questo capitolo si entra nel dettaglio di alcuni metodi di particolare interesse nello sviluppo dell'applicazione, analizzandone i concetti matematici, fisici e logici che li rendono operativi.

URTO BILIA-BILIA

```
public void ballCollision(Ball b) {

        Vector n1 = position.sub(b.position);
        n1.normalize(); // un1
        double v1n = velocity.scalar(n1);
        n1.multiply(v1n); // v1n (vector)
        Vector v1t = velocity.sub(n1);

        Vector n2 = b.position.sub(position);
        n2.normalize(); // un2
        double v2n = b.velocity.scalar(n2);
        n2.multiply(v2n); // v2n (vector)
        Vector v2t = b.velocity.sub(n2);

        velocity = v1t.add(n2);
        b.velocity = v2t.add(n1);

}
```

La collisione tra due bilie[7] è un urto di tipo elastico. Questo significa che nella collisione si conservano sia la quantità di moto che l'energia cinetica.

Le equazioni di conservazione in uno spazio monodimensionale sono le seguenti:

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2'$$

$$\frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 = \frac{1}{2} m_1 v_1'^2 + \frac{1}{2} m_2 v_2'^2$$

Dalle quali si ricavano le velocità finali:

$$v_1' = \frac{v_1(m_1 - m_2) + 2m_2 v_2}{m_1 + m_2} \qquad v_2' = \frac{v_2(m_2 - m_1) + 2m_1 v_1}{m_1 + m_2}$$

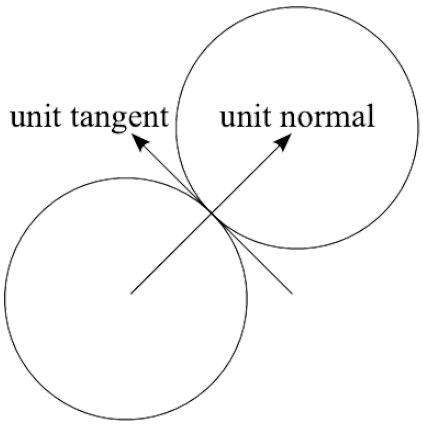
Per il contesto di interesse è possibile fare l'ipotesi $m_1 = m_2 = 1$, visto che tutte le bilie hanno stessa massa, quindi le velocità finali diventano:

$$v_1' = v_2 \qquad v_2' = v_1$$

In una collisione bidimensionale si proiettano le velocità lungo due direzioni: una direzione perpendicolare alla superficie dell'impatto (normale), una direzione parallela alla superficie dell'impatto (tangente). Questo significa che ogni vettore velocità sarà caratterizzato da una componente normale e da una tangenziale.

La componente tangenziale del vettore velocità rimane costante nella collisione perché non c'è alcuna forza in tale direzione.

La componente normale, invece, cambia seguendo la legge unidimensionale trattata sopra.



Il metodo, sfruttando le operazioni definite nella classe `Vector.java`, calcola le velocità normali v_1^n, v_2^n e le velocità tangenziali v_1^t, v_2^t[1], per poi determinare le medesime dopo la collisione e, quindi, calcolare le velocità finali.

In particolare, per ogni bilia:

1. si determina la direzione normale sottraendo i vettori

[1]Le velocità normali v_1^n, v_2^n sono di fatto i vettori n_1, n_2, visto che i metodi utilizzati non restituiscono un vettore, ma modificano l'invocante.

2. si normalizza il vettore, ottenendo l'unità normale

3. si proietta (prodotto scalare) la velocità della bilia lungo l'unità normale, ottenendo il valore scalare della velocità della bilia lungo la componente normale

4. si moltiplica (scalarmente) l'unità normale per il risultato ottenuto al punto 3, ottenendo $v^n_{1,2}$

5. si determina la velocità tangenziale ($v^t_{1,2}$) sottraendo alla velocità la velocità normale

Infine, il metodo determina i vettori finali `velocity` (velocità bilia invocante) e `b.velocity` (velocità bilia in input) sommando la velocità normale e la velocità tangenziale della bilia dopo la collisione. I vettori v^n_1, v^n_2 (nel metodo n_1, n_2) sono scambiati in accordo con la legge della dinamica monodimensionale; mentre i vettori v^t_1, v^t_2 si conservano nella collisione.

URTO BILIA-SPONDA

```
public void bankCollision() {

        double x = position.getX();
        double y = position.getY();
        double r = Constants.BALL_RADIUS;

        // LEFT BANK (A)
        // REGION A1
        if (x - r <= Constants.A_MARGIN && (y >= Constants.
           A_UP_CORNER_START && y <= Constants.A_UP_CORNER_END)
           ) {
            velocity.setY(-velocity.getSize());
            velocity.setX(0);
        }
        // REGION A2
        else if (x - r <= Constants.A_MARGIN && (y >= Constants
           .A_UP_CORNER_END && y <= Constants.
           A_DOWN_CORNER_START)) {
            velocity.setX(Math.abs(velocity.getX()));
        }
        // REGION A3
        else if (x - r <= Constants.A_MARGIN && (y >= Constants
           .A_DOWN_CORNER_START && y <= Constants.
           A_DOWN_CORNER_END)) {
            velocity.setY(velocity.getSize());
            velocity.setX(0);
        }

        ...

}
```

Per gestire l'urto di una bilia con una sponda del tavolo, si è deciso di suddividere ogni sponda in tre zone distinte. Infatti, dato che ogni sponda è compresa tra due buche del tavolo, è possibile individuare una regione in prossimità di una buca, un'altra in prossimità dell'altra e un'altra ancora compresa tra le due regioni. Ogni regione in prossimità di una buca è fatta in modo da direzionare la bilia che la colpisce nella direzione della buca.

Per semplificare il problema, si è deciso di far riferimento alle sponde con letterali, come indicato dalla Figura 4.1.

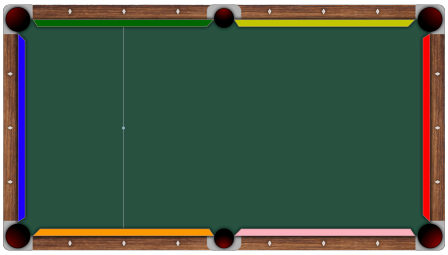


Figura 4.1: Blu (A), Rosso (B), Verde (C), Giallo (D), Arancione (E), Rosa (F)

Per ogni sponda, le relative regioni sono sempre considerate in ordine crescente rispetto agli assi del piano (Pane). Cioè per le sponde A e B le regioni A1, A2, A3 e B1, B2, B3 sono individuate andando nel verso crescete dell'asse y (dall'alto verso il basso), mentre per le sponde C, D, E, F le regioni C1, C2, C3, D1, D2, D3, E1, E2, E3 e F1, F2, F3 sono individuate andando nel verso crescente dell'asse x (da sinistra verso destra).

Per semplicità, si è riportato solo il codice che controlla l'urto con la sponda A, ma i ragionamenti sono del tutto analoghi per le sponde rimanenti.

Se una bilia urta la regione A2, allora per la Legge di Riflessione [2] si conserva il modulo della velocità. Inoltre, la componente y conserva il verso della velocità (ad esempio una bilia che urta provenendo dall'altro, cioè con velocità verso il basso, continuerà ad andare verso il basso dopo l'urto). Lungo x, invece, il verso della velocità sarà sicuramente verso destra, quindi la velocità lungo x viene impostata positiva.

Se una bilia urta la regione A1 o A3, la velocità lungo x sarà sicuramente nulla; visto che la bilia viene indirizzata verso la rispettiva buca. La velocità lungo y, invece, sarà negativa (verso) e pari al modulo della velocità per la regione A1 (visto che la bilia deve andare verso l'alto) e sarà positiva (verso) e pari al modulo della velocità per la regione A3 (visto che la bilia deve andare verso il basso).

[2] L'urto di una bilia contro una sponda è assimilabile alla riflessione della luce su uno specchio piano.

5. Bibliografia

[1] VNEA. 8 ball pool. http://www.vnea.com/Data/Sites/12/photos/8-BallRules.pdf.

[2] Phongmaterial (javafx 8), 2015. https://docs.oracle.com/javase/8/javafx/api/javafx/scene/paint/PhongMaterial.html.

[3] Dropshadow (javafx 2.2), 2013. https://docs.oracle.com/javafx/2/api/javafx/scene/effect/DropShadow.html.

[4] Javafx. https://openjfx.io/.

[5] Calculate on which side of a straight line is a given point located? https://math.stackexchange.com/questions/274712/calculate-on-which-side-of-a-straight-line-is-a-given-point-located.

[6] How to tell whether a point is to the right or left side of a line. https://stackoverflow.com/questions/1560492/how-to-tell-whether-a-point-is-to-the-right-or-left-side-of-a-line.

[7] Chad Berchek. 2-dimensional elastic collisions without trigonometry, 2009. https://www.vobarian.com/collisions/2dcollisions2.pdf.