UNIVERSITÀ DEGLI STUDI DI UDINE

Corso di Sistemi Mobili e Wireless

Professor Stefano Burigat

LABYRINTH
IMPLEMENTAZIONE DI UN GIOCO PER SMARTPHONE

Francesca Palù
100908

ANNO ACCADEMICO 2015-2016

Introduzione

L'applicazione *Labyrinth* realizza il gioco del labirinto da percorrere, sfruttando l'oscillazione del dispositivo. È stata scelta questa applicazione per utilizzare l'interfaccia *OpenGL* e sfruttare appieno i sensori di movimento.

Il gioco è organizzato in diversi livelli di difficoltà crescente. Per la prima metà dei livelli è previsto che, facendo oscillare il dispositivo, si muova una pallina per farle percorrere labirinti sempre più estesi entro un dato tempo. Gli ultimi 16 livelli aggiungono un'ulteriore difficoltà, dal momento che i muri del labirinto non devono essere toccati dalla pallina.

È stata prestata molta attenzione nel rendere il movimento fluido, sensibile, e facilmente controllabile dall'utente. Inoltre i labirinti sono generati automaticamente a partire da due algoritmi differenti, l'algoritmo *Depth First Search*, che produce labirinti semplici ed intuitivi, e l'algoritmo di *Kruskal*, che ne genera di più complessi, con più vie per arrivare al traguardo e molti vicoli ciechi.

Questa relazione è suddivisa in quattro capitoli: il primo presenta le caratteristiche generali dell'applicazione e delle activities presenti, il secondo offre una spiegazione dettagliata dell'activity che permette di svolgere le partite, l'activity Game, il terzo illustra nel dettaglio l'implementazione del renderer utilizzato per la visualizzazione degli elementi grafici, infine il quarto riporta le condizioni di sviluppo e di testing.

Indice

Introduzione **iii**

1 Struttura dell'applicazione **1**
1.1 Aspetti generali 1
1.2 MainActivity 2
1.3 Levels 3
1.4 Game 6
1.5 Victory 8

2 Game Activity **11**
2.1 Caratteristiche generali 11
2.2 Metodi dell'activity 11
2.3 Layout 12
2.4 Renderer 13
2.5 Sensori 13
2.6 Utilizzo del `RetainedFragment` 14

3 Il Renderer **17**
3.1 Struttura generale 17
3.2 Strutture ausiliarie 17
3.2.1 Shaders 17
3.2.2 Classe `VertexArray` 18
3.2.3 Cartella `objects` 19
3.2.4 Classe `Maze` 19
3.3 Costruttore `MyRenderer` 20
3.4 Metodo `onSurfaceCreated` 21
3.5 Metodo `onSurfaceChanged` 21
3.6 Metodo `onDrawFrame` 22
3.7 Metodo `handleTilt` 22
3.7.1 Cambio del movimento a seconda dell'orientamento . . . 25
3.7.2 Ampiezza del movimento 26

3.8 Metodo `edgeDetect` 27

4 Sviluppo e Test **29**
4.1 Ambiente di sviluppo 29
4.2 Test effettuati 29

Conclusione **31**

Elenco delle figure

1.1 Colori Layout 1
1.2 Activities 2
1.3 Activity MainActivity 3
1.4 Activity Levels 4
1.5 Activity Levels1 e Levels4 5
1.6 Activity Game 7
1.7 Activity Victory 9

2.1 Activity e RetainedFragment 15

3.1 Utilizzo di più `FloatBuffer`, uno per ogni attributo 18
3.2 Pitch e roll 23
3.3 Pallina e muri del labirinto 28

Elenco degli script

1.1 Codice utilizzato per cambiare lo sfondo dei livelli completati . . 5
1.2 Parte del metodo `onStop`, utilizzata per memorizzare il completamento del livello 6
2.1 Istruzioni per ripristinare gli ascoltatori dei sensori 14
2.2 Istruzioni per salvare e recuperare i dati dal `retainedFragment` . 16
3.1 Costruttore della classe `VertexArray` 18
3.2 Metodo `handleTilt` nella classe `MyRenderer` 24

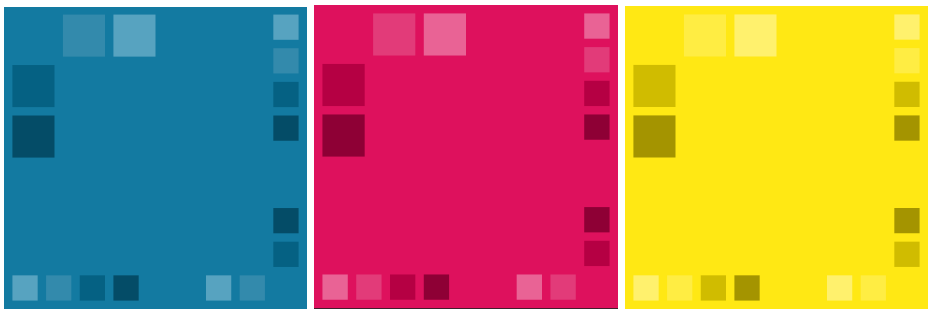
Capitolo 1

Struttura dell'applicazione

In questo capitolo illustreremo lo stile e gli elementi grafici adottati, faremo una panoramica delle varie *activities*, spiegandone la funzionalità e il loro layout.

1.1 Aspetti generali

Tutte le activities hanno un layout simile, in cui lo sfondo è blu, le scritte gialle, i pulsanti e le barre magenta. I colori sono stati scelti grazie allo strumento online *paletton* [1]. Possiamo vederli nella figura 1.1.



(a) blu: `137AA1` (b) magenta: `DE115D` (c) giallo: `FFE814`

Figura 1.1: Colori utilizzati nei vari Layout dell'applicazione

Il font utilizzato è *GoodDog Plain Font* [2], che è un font di tipo *handwritten*, irregolare e molto informale. Le immagini utilizzate per i pulsanti sono state create con *Photoshop CS6*, usando gli stessi colori del layout.

L'applicazione è costituita dalle seguenti activities:

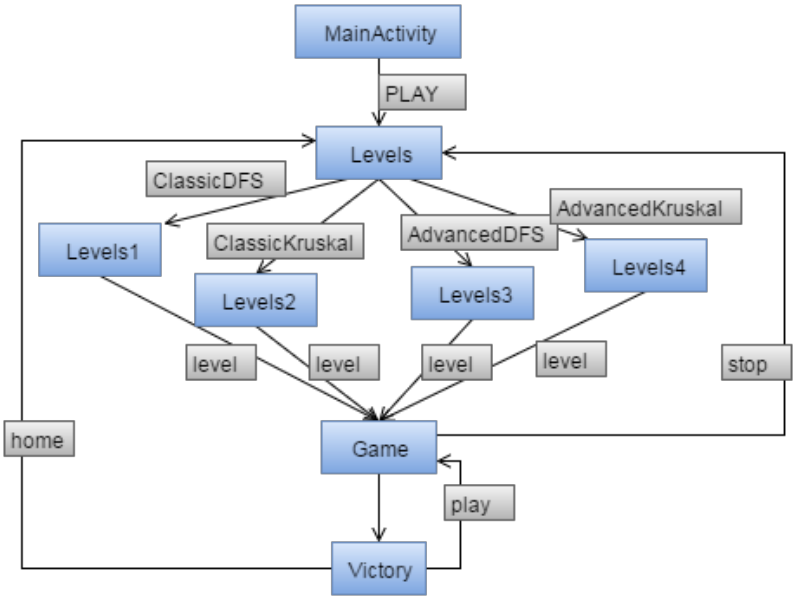


Figura 1.2: Diagramma delle activities presenti. Le activities sono rappresentate dai rettangoli azzurri, mentre nei rettangoli grigi sono riportati i pulsanti che permettono il passaggio da un'activity all'altra.

- `MainActivity`: è l'activity di apertura;
- `Levels`: presenta le diverse modalità di gioco;
- `Levels1`,`Levels2`,`Levels3`,`Levels4`: presentano i diversi livelli di gioco;
- `Game`: raccoglie il gioco vero e proprio;
- `Victory`: riporta il risultato della partita.

In figura 1.2 è possibile vedere i passaggi da un'activity all'altra.
Vediamole più nel dettaglio, nell'ordine in cui compaiono.

1.2 MainActivity

L'activity di apertura è costituita dal pulsante `play`, che porta all'activity Levels.
Il layout utilizzato è un `RelativeLayout`, con il pulsante al centro.

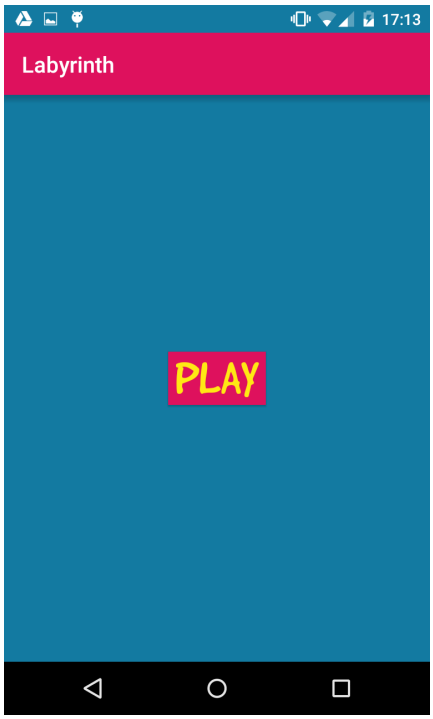


Figura 1.3: Activity MainActivity

1.3 Levels

In questa activity è possibile scegliere il tipo di labirinto che si vuole percorrere, se quello generato con l'algoritmo *DFS* o di *Kruskal*, e la modalità di gioco, normale o avanzata. La scelta viene effettuata premendo su uno dei quattro pulsanti, che riportano all'activity con i livelli del tipo scelto. In figura 1.4 è possibile vedere la schermata dell'activity Levels.

L'algoritmo DFS genera labirinti facili da percorrere, mentre l'algoritmo di Kruskal crea molti vicoli ciechi. Per una spiegazione più dettagliata del funzionamento dei diversi algoritmi si rimanda a [3]. Nella modalità avanzata, si aggiunge l'ulteriore regola di non toccare i muri del labirinto, altrimenti si perde immediatamente. Una volta premuto uno dei quattro pulsanti, si viene rimandati in una delle quattro activities contenenti i livelli veri e propri.

Queste activities utilizzano un layout comune, `res.layout.content_levels1.xml`, costituito da un messaggio centrale di istruzioni, adatto ad ogni tipo di labirinto, ed otto pulsanti disposti a griglia, ognuno corrispondente a un diverso livello. Con uno *swipe* verso destra o verso sinistra è possibile vedere la classe precedente o successiva di livelli. Quindi se sono nell'activity Level1, dei labirinti *Classic DFS*, con uno swipe verso destra passo all'activity Level4, dei labirinti *Advanced Kru-*

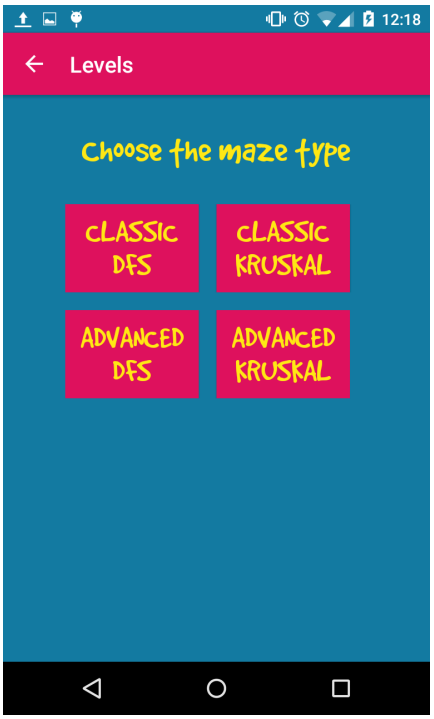


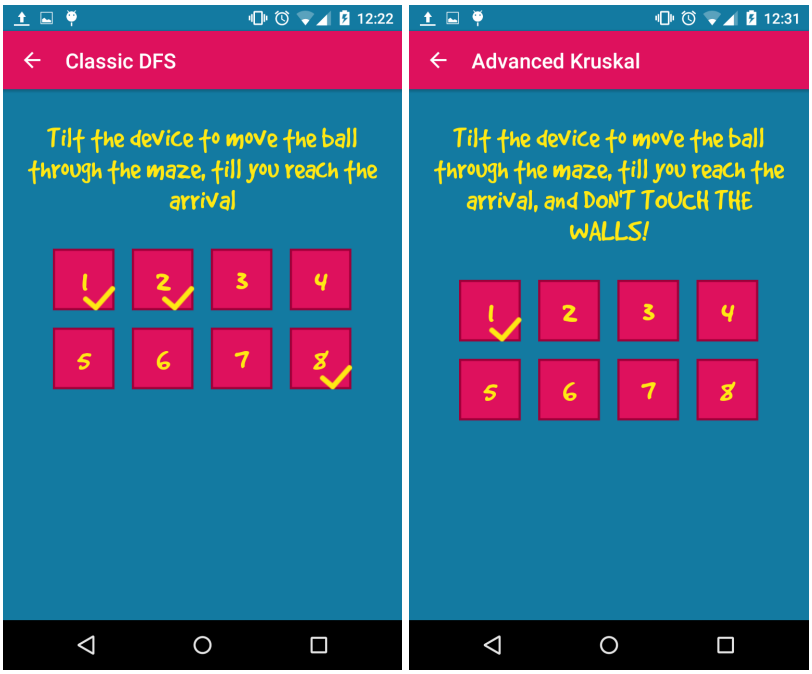
Figura 1.4: Activity Levels

skal, mentre verso sinistra passo all'activity Level2, dei labirinti *Classic Kruskal*. Per la realizzazione delle animazioni è stata seguita la guida [4], mentre il codice della classe `OnSwipeTouchListener`, utilizzato per riconoscere la *gesture*, è stato preso da [5].

Non è necessario affrontare i livelli in ordine, ma quelli completati saranno riconoscibili per lo sfondo diverso. In entrambi i casi lo sfondo è un elemento *drawable* realizzato con *Photoshop*. In figura 1.5 è possibile vedere la schermata dell'activity *Levels1* e *Levels4* in cui alcuni livelli risultano completati, e altri no.

Il layout a griglia è stato creato utilizzando un RelativeLayout che contiene un LinearLayout per ogni riga di pulsanti. I comandi `padding` e `layout_margin` sul file `content_level.xml` definiscono gli spazi tra i vari pulsanti. Lo sfondo dei pulsanti è stato definito dalla classe `Levels`, in modo da cambiare a seconda del completamento o meno del livello, recuperato con l'oggetto `SharedPreferences` [6], che permette di salvare e recuperare coppie chiave valore. In questo caso la chiave è la stringa *tipo di labirinto* + *livello corrente*, e il valore è un booleano che ne indica il completamento. Lo script 1.1 riporta il codice per l'assegnamento dello sfondo in base al completamento del livello.

Il livello corrente viene recuperato dal tag associato ad ogni pulsante all'interno del layout, mentre il tipo del labirinto è una costante di tipo String definita in



(a) Levels1 (b) Levels4

Figura 1.5: Activity Levels1 e Levels4

ognuna di queste activities. Questi due elementi, che serviranno all'activity Game, le vengono inviati associando un `bundle` all'intent che invoca l'activity, quando si preme il pulsante del livello. La preferenza viene aggiornata nel momento in cui il livello viene completato con successo, nel metodo `onStop` dell'activity Game, altrimenti di default resta `false`. Questa parte del metodo `onStop` la riportiamo nello script 1.2.

Script 1.1: Codice utilizzato per cambiare lo sfondo dei livelli completati

```
// Restore preferences
for (int i=0; i<Constants.maxLevels; i++){
     SharedPreferences level = getSharedPreferences
        (mazeType + (i+1), 0);
     boolean completed = level.getBoolean("completed",
        false);
     //change buttons appearance if level is completed
     if (completed) {
          buttons[i].setBackground(ContextCompat.
          getDrawable(this, R.drawable.levelcompl));
```

```
        }
        else {
            buttons[i].setBackground(ContextCompat.
            getDrawable(this, R.drawable.level));
        }
        buttons[i].setTypeface(font);
}
```

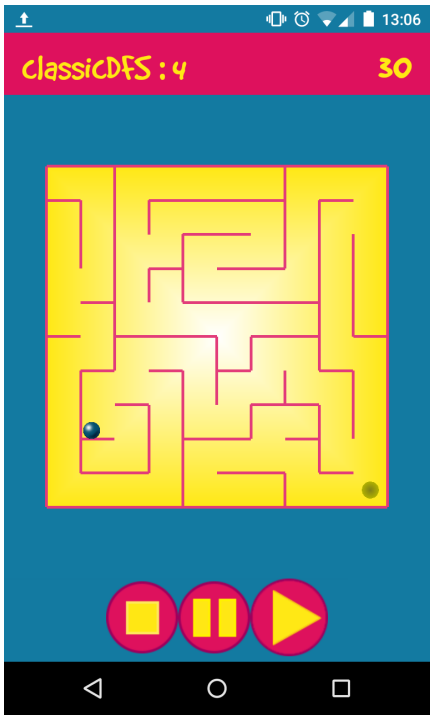
Script 1.2: Parte del metodo `onStop`, utilizzata per memorizzare il completamento del livello

```
// Save level completed
// if game won
if (renderer.goal) {
    // Save completed level on preferences
    SharedPreferences.Editor editor =
        currentLevel.edit();
    editor.putBoolean("completed", true);
    // Commit the edits!
    editor.commit();
}
```

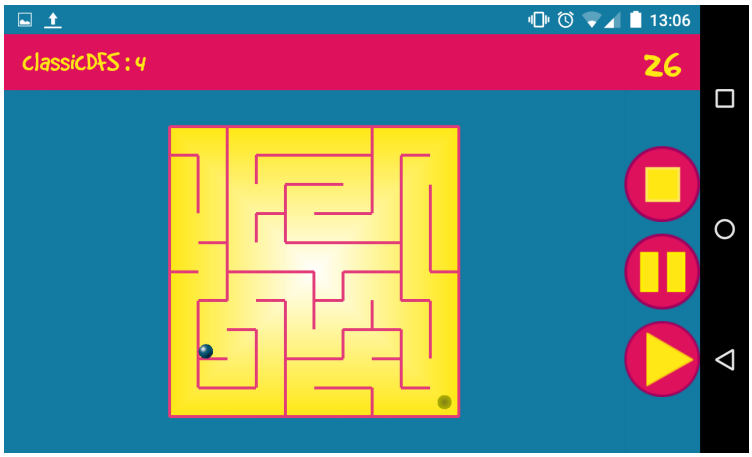
1.4 Game

Questa è l'activity più complessa. Riporta il livello corrente, il conto alla rovescia, il piano di gioco, con il labirinto, la pallina e il traguardo, e tre pulsanti, uno per interrompere la partita e tornare all'activity `Levels` (da usare nel caso si volesse cambiare il livello), uno per interrompere il timer, e uno per farlo ripartire.

Il layout cambia a seconda dell'orientamento del dispositivo, salvando ogni volta lo stato della partita, anche se in realtà si ha un'esperienza di gioco migliore impedendo il cambio dell'orientamento. È stata volutamente lasciata all'utente la scelta di impostare o no l'orientamento fisso, per permettergli di sperimentare anche la modalità di gioco landscape. Inoltre, nel momento dell'implementazione, è stato possibile provare e verificare l'uso dei frammenti per mantenere un oggetto, senza usare scorciatoie che permettono di semplificare il codice evitando il problema. Nel capitolo 2 vedremo più nel dettaglio alcuni aspetti implementativi. In figura possiamo vedere una schermata tipica dell'activity, nella modalità *portrait* e *landscape*.



(a) Portrait



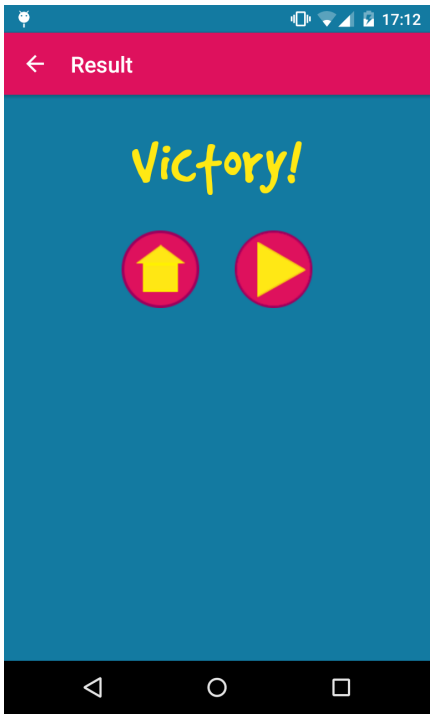
(b) Landscape

Figura 1.6: Activity Game

1.5 Victory

L'activity `Victory` mostra il risultato della partita, che può essere positivo, e quindi permette di passare al livello successivo, o negativo, nel cui caso dà la possibilità di ritentare il livello. Il risultato ed il livello da completare vengono recuperati dal bundle associato all'intent, che contiene il parametro `result`, 0 nel caso di sconfitta e 1 per la vittoria, e `level`. Dal bundle viene recuperato anche il tipo di labirinto (Classic DFS, Classic Kruskal, Advanced DFS o Advanced Kruskal) per poterlo passare all'activity Game.

Il layout è costituito da un `RelativeLayout`, che contiente una *TextView* per il messaggio, e un LinearLayout per contenere due pulsanti, uno per tornare all'activity Levels, e uno per iniziare una nuova partita.



(a) Partita vinta

(b) Partita persa

Figura 1.7: Activity Victory

Capitolo 2

Game Activity

In questo capitolo viene descritta l'activity Game, il suo layout, gli elementi grafici, il *renderer*, la gestione del movimento del dispositivo.

2.1 Caratteristiche generali

L'activity Game è la più importante dell'applicazione. È costituita da una barra orizzontale, con il livello corrente ed il timer, la superficie in cui giocare, e tre tasti per uscire dalla partita, metterla in pausa, e farla ripartire. Le regole per vincere sono semplici: nella modalità classica, bisogna muovere la pallina attraverso il labirinto fino al traguardo in basso a destra entro un dato tempo. L'utente controlla il movimento oscillando il dispositivo. Nella modalità avanzata c'è anche l'obbligo di non toccare i muri del labirinto.

Per semplificare il controllo della pallina, essa rimane nella posizione iniziale per un secondo quando inizia la partita. Invece, quando raggiunge il traguardo e si vince la partita, rimane sul traguardo per 100 millisecondi e l'utente avverte una vibrazione, prima di passare all'activity Victory. Nella modalità avanzata c'è la vibrazione e la pallina si ferma per 100 millisecondi anche quando tocca un muro.

2.2 Metodi dell'activity

L'activity Game contiene i seguenti metodi:

- `onCreate`, che attua i seguenti passaggi:
  - istanzia l'oggetto `glSurfaceView`, a cui viene associato il FrameLayout definito nel layout `content_game`,

  - inizializza i dati, creandoli da zero o prendendoli dal `RetainedFragment`,
  - crea il renderer. Questo oggetto sarà spiegato in dettaglio nel capitolo 3,
  - controlla se il sistema supporta OpenGL ES 2.0,
  - assegna il renderer alla `glSurfaceView`,
  - prepara i sensori necessari;

- `onSensorChanged`, che invia al renderer i cambiamenti registrati dai sensori;
- `onPause`, che memorizza i dati da ripristinare;
- `onResume`, che ripristina lo stato dell'applicazione;
- `onStop`, che, se il livello è stato registrato con successo, lo registra nelle preferenze;
- `onRestart`, che ripristina lo stato dell'applicazione;
- `onDestroy`, che interrompe l'applicazione e salva i dati necessari sul `RetainedFragment`;
- i metodi associati ai tre pulsanti presenti:
  - `stop`, che interrompe la partita e porta all'activity Levels,
  - `pause`, che blocca il timer e la pallina nella posizione corrente,
  - `play`, che fa ripartire il timer e permette di nuovo il movimento della pallina;
- `generateMaze` infine genera l'oggetto `Maze` in base al livello e al `mazeType`, necessario al renderer per creare il labirinto desiderato, e ci associa i secondi adatti per il timer.

2.3 Layout

Questa activity utilizza due layout, uno per la modalità portrait, e uno per la modalità landscape. Il caricamento di un layout diverso per l'orientamento landscape è reso possibile dal file `res.layout-land.content_game.xml`. Entrambi i layout presentano un RelativeLayout che contiene un LinearLayout per la TextView, un FrameLayout per la GLSurfaceView, e un LinearLayout per i tre pulsanti.

La barra superiore riporta il tipo di labirinto, il livello corrente ed i secondi rimanenti per vincere la partita. I primi due parametri sono passati all'activity Game dal bundle dell'intent, mentre per visualizzare il testo del timer viene creato un oggetto di tipo `CountDown`, classe che estende `CountDownTimer`. Questa classe fa sì che quando il timer scade, venga dato inizio all'activity Victory, per segnalare però la sconfitta all'utente.

L'altezza della barra superiore è gestita dall'attributo `?attr/actionBarSize`, in modo da essere uguale all'`AppBarLayout` presente negli altri layout.

Come spiegato in [7], il cambio dell'orientamento distrugge e ricrea l'activity, quindi, per permettere all'utente di continuare la partita, vengono salvati i dati necessari con l'utilizzo dei `RetainedFragment`. Per la spiegazione si rimanda alla sezione 2.1.

2.4 Renderer

L'implementazione del renderer sarà approfondita nel capitolo 3. Esso rappresenta gli oggetti sulla scena, il piano del labirinto quadrato, giallo ai bordi e bianco al centro, i muri del labirinto come linee color magenta, la pallina avrà invece il disegno di una texture. All'inizio della partita sarà in alto a sinistra, poi a seconda di come viene mosso il dispositivo, percorrerà il labirinto, ed il renderer la rappresenterà in tempo reale nella posizione corretta.

2.5 Sensori

I sensori adatti al nostro caso sono quelli che ci permettono di capire quando il dispositivo viene inclinato e subisce delle oscillazioni. Seguendo il suggerimento dell'esempio [8], per ottenere la matrice di rotazione del dispositivo abbiamo bisogno dei sensori:

- `TYPE_ACCELEROMETER`: misura l'accelerazione in m/s^2 lungo gli assi x, y, e z;
- `TYPE_MAGNETIC_FIELD`: misura il campo geomagnetico in μT sugli assi x, y, e z.

Con questi dati possiamo creare la matrice di rotazione, e ricavare gli spostamenti che ci interessanno, che sono chiamati *pitch* e *roll*, e quindi inviarli al renderer, che li gestirà. Rimandiamo alla sezione 3.7 per ulteriori dettagli.

Questi dati, la prima volta che la superficie viene creata a inizio partita, non vengono inviati immediatamente, ma dopo un secondo, ossia quando la variabile

`sleep` viene portata a `false`, in modo da permettere all'utente di prepararsi a giocare.

Quando l'activity Game è in pausa, o distrutta, o la partita è stata interrotta con il metodo `pause`, vengono tolti gli ascoltatori dei sensori, con l'istruzione `sensorManager.unregisterListener(this)`. Saranno poi ripristinati nei metodi `play`, `onRestart` e `onResume` con le istruzioni presenti in 2.1.

Script 2.1: Istruzioni per ripristinare gli ascoltatori dei sensori

```
sensorManager.registerListener(this, accelerometer,
   SensorManager.SENSOR_DELAY_GAME);
sensorManager.registerListener(this, magnetometer,
   SensorManager.SENSOR_DELAY_GAME);
```

Con la costante `SENSOR_DELAY_GAME`, la frequenza di campionamento dei dati viene impostata a 20000 microsecondi, che si è visto empiricamente che permette di ottenere un movimento sufficientemente fluido e controllabile. Per ulteriori spiegazioni si rimanda alla pagina dedicata del sito ufficiale per lo sviluppo Android [9].

2.6 Utilizzo del `RetainedFragment`

I *Fragment* possono essere utilizzati per salvare istanze di oggetti, in modo da poter essere recuperati per ricostruire l'activity. Il frammento non sarà distrutto con l'activity, quindi manterrà tutti i dati. In figura 2.1 possiamo vedere uno schema dell'activity e del fragment di supporto.

I dati che necessitiamo di memorizzare sono:

- l'oggetto di tipo `Maze`, che servirà per ricostruire i muri del labirinto;
- i secondi rimanenti allo scadere del timer;
- la posizione corrente della pallina.

Per creare un fragment adatto al nostro scopo, è stata creata la classe `RetainedFragment` secondo le indicazioni forniti dal sito ufficiale per lo sviluppo in Android [10] e dalla guida all'utilizzo dei RetainedFragment [11].

Quindi è stata creata una classe per rappresentare i dati che dobbiamo memorizzare, `data.RetainedData`, i cui campi sono:

- `maze` di tipo `Maze`;
- `seconds` di tipo `long`;

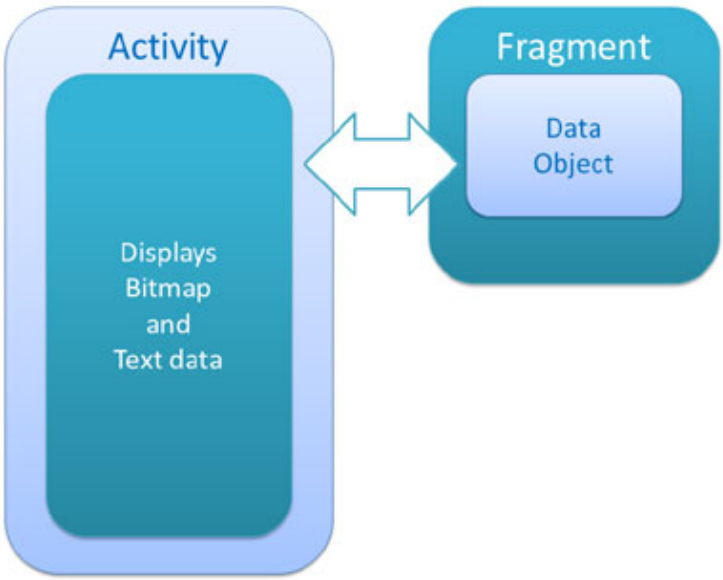


Figura 2.1: Activity e RetainedFragment

- `ballPosition` di tipo `Point`.

Questa classe contiene anche i rispettivi metodi per recuperare ed impostare i campi.

A questo punto facciamo in modo che, nell'activity Game, i dati vengano salvati nel `RetainedFragment` e da lì recuperati. Creiamo quindi:

- un oggetto di tipo `RetainedFragment`, chiamato `dataFragment`;
- un oggetto di tipo `RetainedData`, chiamato `retainedData`.

Fatto questo, nell'activity Game effettuiamo i seguenti passaggi, per assicurarci che il frammento venga salvato e recuperato al bisogno:

- nel metodo `onCreate`:
  - un `FragmentManager` controlla se esiste un frammento `RetainedFragment`;
  - un controllo verifica se l'activity viene creata per la prima volta, nel qual caso viene creato il frammento e, col metodo `generateMaze`, i dati, che vengono anche memorizzati dal frammento,
  - altrimenti, se i dati del frammento erano stati caricati correttamente, li recuperiamo e li assegnamo alle nostre variabili,

  - se il frammento c'era, ma senza dati caricati, li generiamo e li carichiamo col metodo `generateMaze`;

- a questo punto, il metodo `onDestroy` salva i dati che sono stati modificati dal corso della partita nel frammento. Questi dati saranno i secondi rimanenti e la posizione corrente della pallina.

Lo script 2.2 riporta i passaggi relativi al metodo `onCreate` per il salvataggio ed il recupero dei dati dal `retainedFragment`.

Script 2.2: Istruzioni per salvare e recuperare i dati dal `retainedFragment`

```
//get the saved fragment on activity restart
FragmentManager fm = getSupportFragmentManager();
dataFragment = (RetainedFragment)
   fm.findFragmentByTag("data");
//running the activity for the first time
if (savedInstanceState == null){
     //create fragment and data for the first time
     dataFragment = new RetainedFragment();
     //add the fragment
     fm.beginTransaction().add(dataFragment,
        "data").commit();
     generateMaze();
}else{
     if(isDataLoaded){
          //we are recreating the activity
          //get the data object out of the fragment
          retainedData = dataFragment.getData();
          maze = retainedData.getMaze();
          seconds = retainedData.getSeconds();
          startBallPosition =
             retainedData.getBallPosition();
     } else {
          generateMaze();
     }
}
```

Capitolo 3

Il Renderer

In questo capitolo verrà spiegata l'implementazione del renderer, fatta seguendo le indicazioni e gli esempi di *OpenGL ES 2 for Android* [12].

3.1 Struttura generale

Il renderer è un oggetto di tipo `MyRenderer`. Questa classe, oltre al costruttore, presenta tutti i metodi che permettono:

- la creazione e visualizzazione degli elementi grafici,
- i cambiamenti degli elementi grafici in risposta ai segnali provenienti dai sensori, secondo le regole del gioco,
- il riconoscimento dello stato di fine partita, con vittoria o sconfitta,
- la memorizzazione e il ripristino della posizione della pallina.

3.2 Strutture ausiliarie

Il renderer necessita di strutture dati e programmi ausiliari, quali gli shaders, i `VertexArray` e gli oggetti da renderizzare.

3.2.1 Shaders

Per rappresentare un oggetto grafico in OpenGL abbiamo bisogno di un *vertex shader*, che genera la posizione finale dei vertici dell'oggetto 3D, e un *fragment shader*, che genera il colore di ogni frammento. Nella cartella `res.raw` sono

presenti i vertex shader e i fragment shader utilizzati. Questi shaders saranno poi compilati grazie alla classe `util.ShaderHelper`.

`util.TextResourceReader` permetterà di leggere gli shaders. Per collegare invece gli shaders tra loro in un *OpenGL program* (*linking*), vengono utilizzate le classi presenti nella cartella `programs`, che sono `ColorShaderProgram` e `TextureShaderProgram`, entrambi che estendono `ShaderProgram`.

3.2.2 Classe `VertexArray`

La classe `data.VertexArray` si occupa di copiare i dati dalla *Java's Memory Heap* alla *Native Memory Heap*. Essa trasforma l'array contenente le posizioni dei nostri oggetti in un `FloatBuffer`. È stato scelto di creare un `FloatBuffer` per ogni attributo, quindi uno per le posizioni dei vertici, e uno per i colori. In figura 3.1 è possibile vedere come vengono creati più `FloatBuffer`, uno per ogni attributo. Nello script 3.1 riportiamo anche il costruttore della classe `data.VertexArray`. Per ulteriori approfondimenti si rimanda sempre a *OpenGL ES 2 for Android* [12].

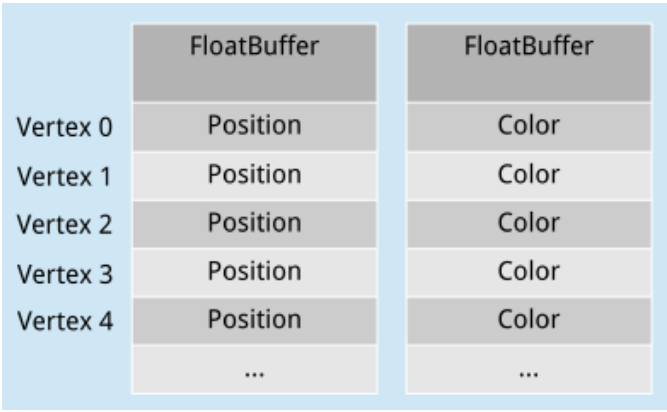


Figura 3.1: Utilizzo di più `FloatBuffer`, uno per ogni attributo

Script 3.1: Costruttore della classe `VertexArray`

```
private final FloatBuffer floatBuffer;
private final FloatBuffer colorBuffer;
public FloatBuffer vertexData;
public FloatBuffer colorData;
//convert float[] to FloatBuffer
public VertexArray(float[] coords, float[] colors) {
    //position buffer
    vertexData = ByteBuffer
            .allocateDirect(coords.length *
                BYTES_PER_FLOAT)
```

```
                .order(ByteOrder.nativeOrder())
                .asFloatBuffer();
        //color buffer
        colorData = ByteBuffer
                .allocateDirect(colors.length *
                   BYTES_PER_FLOAT)
                .order(ByteOrder.nativeOrder())
                .asFloatBuffer();
        floatBuffer = vertexData.put(coords);
        colorBuffer = colorData.put(colors);
}
```

3.2.3 Cartella `objects`

Prima di creare i nostri oggetti `FloatBuffer`, vengono creati degli oggetti di tipo `float[]`, una classe per ogni elemento presente sulla superficie di gioco. Queste classi sono nella cartella `objects`, e sono:

- `Table`: crea il piano del labirinto;
- `Lines`: dato un oggetto `Maze`, crea i muri del labirinto, costituito da segmento orizzontali (`horLines`) e verticali (`verLines`);
- `Arrival`: identifica il punto d'arrivo all'interno del labirinto, come un'ombra scura circolare. Deve richiamare l'idea del solco in cui dovrà entrare la pallina, quindi è più scura al centro;
- `Ball`: crea la pallina. Alla pallina viene assegnata una texture, disegnata con Photoshop. In questo caso gli shaders utilizzati saranno il `texture_vertex_shader` e il `texture_fragment_shader`, compilati dalla classe `util.TextureShaderHelper` e per il linking si userà la classe `programs.TextureShaderProgram`.

Queste classi contengono ognuna un metodo `draw` per disegnare gli oggetti sulla `glSurfaceView`.

3.2.4 Classe `Maze`

Il costruttore `Lines` vuole come argomento un oggetto di tipo `Maze`. Questo oggetto contiene i dati necessari per creare i muri del labirinto. È realizzato dalla classe astratta `Maze`, estesa, a seconda del tipo di labirinto richiesto, dalla classe `DFS` e `Kruskal`, il cui costruttore, dati due valori interi x e y che rappresentano le dimensioni del labirinto, realizza un array di dimensione $x \times y$ contenente i dati

necessari per realizzarne il disegno. Per l'applicazione *Labyrinth* sono stati considerati solo labirinti quadrati, quindi con $x = y$. I labirinti vengono generati ogni volta che parte l'activity Game in modo casuale, sempre diversi.

L'implementazione di queste tre classi ha preso spunto da [13] per la classe `DFS`, e da [14] per la classe `Kruskal`.

3.3 Costruttore `MyRenderer`

Analizziamo ora il costruttore della classe `MyRenderer`. Esso richiede i seguenti argomenti con cui inizializzare i suoi campi:

- `Context context`: il contesto dell'activity Game, che servirà da argomento per gli OpenGL programs e per iniziare l'activity Victory al termine della partita;
- `int level`: il livello corrente, che sarà passato all'activity Victory per indicare il prossimo livello da eseguire;
- `String mazeType`: il tipo di labirinto, che sarà passato all'activity Victory, che dovrà a sua volta passarlo all'activity Game, come spiegato in 1.5;
- `CountDown timer`: è il timer che andrà annullato una volta conclusa la partita;
- `Maze maze`: è l'argomento necessario al costruttore `Lines` per creare i muri del labirinto richiesto;
- `Handler handler`: questo oggetto permette l'esecuzione del codice dopo un certo ritardo, deve essere passato da un'activity, e servirà per passare all'activity Victory dopo 100 millisecondi;
- `Point ballPosition`: questo oggetto permette di portare la pallina al punto a cui era stata lasciata nel caso di un cambio di orientamento dello schermo, infatti è uno dei dati salvati dal `RetainedFragment`.

Viene inoltre inizializzata la vibrazione, che verrà attivata al raggiungimento del traguardo e, nella modalità avanzata, quando la pallina toccherà i bordi.

Dalla prossima sezione analizzeremo i metodi presenti.

3.4 Metodo onSurfaceCreated

Questo è il metodo che viene invocato al momento della creazione della glSurfaceView. Esso prevede i seguenti passaggi:

- ripulitura dello schermo;
- inizializzazione degli oggeti Table, Lines, Ball e Arrival;
- inizializzazione degli shaders ColorShaderProgram e TextureShaderProgram;
- caricamento della texture utilizzata per l'oggetto Ball;
- calcolo della posizione iniziale della pallina, in alto a sinistra se è la prima volta che viene fatta partire l'activity, altrimenti la posizione sarà quella recuperata dal RetainedFragment;
- calcolo della posizione del traguardo, in cui sarà disegnato l'oggetto Arrival, in basso a destra;
- inizializzazione della variabile goal a false. Questa variabile indica il completamento o meno della partita corrente con successo;
- inizializzazione dell'oggetto ballBoundingSphere, un cerchio che circonda il punto d'arrivo, più grande di arrival, su cui si testerà se la pallina ha raggiunto il traguardo o no.

3.5 Metodo onSurfaceChanged

Il metodo onSurfaceChanged viene chiamato quando la superficie viene inizializzata e ogni volta che essa cambia, per esempio ad una rotazione dello schermo.

Questo metodo fa in modo che il *viewport* copra l'intera superficie, quindi crea la matrice per calcolare la proiezione ortografica, a seconda che si sia in modalità *landscape* o *portrait*. È sufficiente la matrice ortografica e non quella prospettica dal momento che gli oggetti sono sullo stesso piano. In questo modo, alla rotazione dello schermo gli oggetti non si adattano alla superficie modificata, per esempio stirandosi orizzontalmente o verticalmente, ma mantengono le proporzioni corrette.

Viene inoltre impostato il valore della variabile booleana landscape, che ci sarà utile per capire l'orientamento del dispositivo.

3.6 Metodo `onDrawFrame`

Il metodo `onDrawFrame` viene invocato ogni volta che un nuovo frame deve essere disegnato. Questo metodo esegue i seguenti passaggi:

- ripulitura della superficie di rendering;
- invio della matrice di proiezione ortografica, calcolata col metodo `onSurfaceChanged`, al vertex shader, per il calcolo della posizione finale dei vertici;
- impostazione del `ColorShaderProgram` come *OpenGL program*, in modo da colorare i *fragment* secondo i colori indicati;
- rappresentazione della `table` nella posizione corretta;
- rappresentazione di `lines`;
- rappresentazione del punto di arrivo nella posizione corretta;
- impostazione del `TextureShaderProgram` come *OpenGL program*, in modo da colorare i *fragment* della pallina in base alla texture indicata;
- rappresentazione dell'oggetto `Ball` nella posizione corretta;

3.7 Metodo `handleTilt`

Il metodo `handleTilt` gestisce i dati registrati dai sensori, inviati al renderer dal metodo `onSensorChanged` dell'activity Game. I dati ricevuti sono:

- *pitch*, che indica la rotazione del dispositivo attorno all'asse x espressa in gradi;
- *roll*, che indica la rotazione del dispositivo attorno all'asse y espressa in gradi.

In figura 3.2 è possibile vedere come sono posizionati gli assi rispetto al dispositivo.

Ricevuti questi dati, il metodo calcola lo spostamento della pallina, secondo quanto spiegato nella sezione 3.7.2, adattandola in base all'orientamento del dispositivo, come vedremo in sezione 3.7.1.

La modifica della posizione viene corretta aggiungendo dei vincoli, ossia che non può uscire dal piano, nè oltrepassare i muri del labirinto. Nella modalità

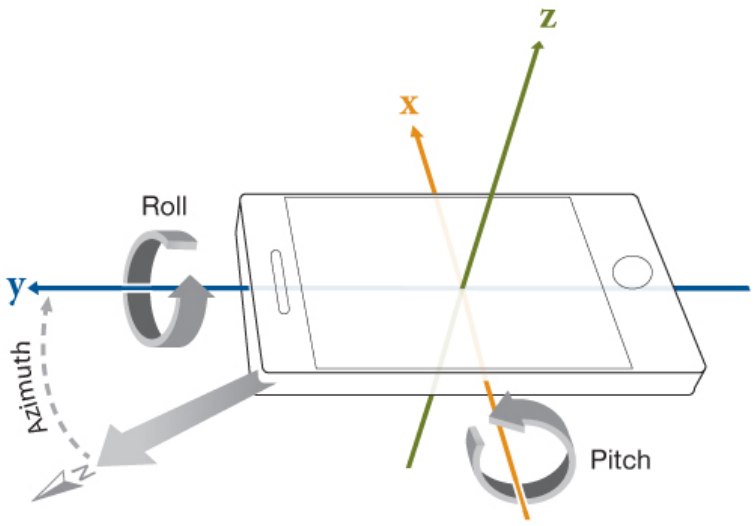


Figura 3.2: Rappresentazione grafica degli assi rispetto al dispositivo e dei movimenti *pitch* e *roll*

avanzata questi elementi non dovranno invece essere toccati dalla pallina. Questi controlli sono effettuati dal metodo `edgeDetect`, spiegato nella sezione 3.3.

A questo punto, aggiornata in modo corretto la posizione della pallina, si controlla se il traguardo è stato raggiunto, verificando se la posizione della pallina si trova all'interno della `ballBoundingSphere`, creata più grande di `arrival`, dal momento che con `ballPosition` consideriamo un punto e non tutta la sfera.

Se la pallina ha raggiunto l'arrivo, e quindi `goal = false`, si verificano i seguenti passaggi:

- il dispositivo vibra;
- la posizione della pallina viene fatta coincidere con quella dell'arrivo;
- viene annullato il timer;
- viene creato un nuovo intent, che, se ci sono ulteriori livelli, avrà associato un bundle contenente il livello corrente, il risultato positivo della partita, e il tipo di labirinto, e porterà all'activity Victory. Se non ci sono ulteriori livelli, non avrà associato nulla e porterà all'activity Levels;
- infine viene lanciata la nuova activity, ma dopo 100 millisecondi, in modo tale che l'utente si renda conto di aver raggiunto il traguardo.

Lo script 3.2 riporta il codice del metodo. Approfondiamo ora nelle prossime sezioni gli aspetti implementativi appena nominati.

Script 3.2: Metodo `handleTilt` nella classe `MyRenderer`

```
//Handle tilt movement in portrait and landscape mode
public void handleTilt(float pitch, float roll) {
     Point touch = new Point(ballPosition.x,
        ballPosition.y, ballPosition.z);
     //gravitational acceleration
     Double g = 9.8;
     float portPitch = pitch + 15;
     float landRoll = roll + 15;
     //mass of the ball
     float m = ball.radius/10;
     //pitch
     if (landscape){
          //left right landscape mode
          float deltaY =
          m*(float)(g*(Math.sin(Math.toRadians(pitch))));
          touch.x = ballPosition.x - deltaY;
     }else {
          //top bottom portrait mode
          float deltaY =
          m*(float)(g*(Math.sin(Math.toRadians(portPitch))));
          touch.y = ballPosition.y + deltaY;
     }
     //roll
     if (landscape) {
          //top bottom landscape mode
          float deltaX =
          m*(float)(g*(Math.sin(Math.toRadians(landRoll))));
          touch.y = ballPosition.y + deltaX;
     }else {
          //left right portrait mode
          float deltaX =
          m*(float)(g*(Math.sin(Math.toRadians(roll))));
          touch.x = ballPosition.x + deltaX;
     }
     //check the maze wall
     edgeDetect(touch);
     ballPosition = new Point
               (clamp(ballPosition.x, leftBound +
                  ball.radius, rightBound -
                  ball.radius),
               clamp(ballPosition.y, bottomBound +
                  ball.radius, topBound - ball.radius),
```

```
                0f);
        // If the ball intersect the arrival, goal = true
        goal = Geometry.intersects(ballBoundingSphere,
           ballPosition);
        if (goal){
              v.vibrate(50);
              // Visualize the ball in the arrival position
              ballPosition = arrivalPosition;
              timer.cancel();
              // Call new activity if there are other level
              if (level<Constants.maxLevels) {
                    intent = new Intent(context,
                       Victory.class);
                    Bundle b = new Bundle();
                    b.putInt("level", level);
                    b.putInt("result", 1);
                    b.putString("maze", mazeType);
                    intent.putExtras(b);
              } else {
                    intent = new Intent(context,
                       Levels.class);
              }
              // Execute some code after 100 milliseconds
                 have passed
              handler.postDelayed(new Runnable() {
                    public void run() {
                          // invoke startActivity on the
                             context
                          context.startActivity(intent);
                    }
              }, 100);
        }
}
```

3.7.1 Cambio del movimento a seconda dell'orientamento

Sono stati riconosciuti due casi, la modalità portrait e la modalità landscape, che vengono riconosciute dalla variabile `landscape` impostata dal metodo `onSurfaceChanged`.

Modalità portrait

In questa modalità le variazioni del dato pitch condizionano gli spostamenti verticali della pallina, e quindi, a seconda dell'ampiezza del movimento, varia il campo *y* dell'oggetto `ballPosition`, che assegna la posizione della pallina nel metodo `onDrawFrame` (vedi sezione 3.6).

Invece le variazioni del dato roll ne condiziona gli spostamenti orizzontali, e quindi il campo *x* di `ballPosition`.

Per tener conto dell'inclinazione verso l'utente che ha il dispositivo quando viene tenuto in mano, al dato `pitch` in input sono stati aggiunti 15 gradi, quando si è in questa modalità.

Modalità landscape

Nella modalità landscape i movimenti sono invertiti, ossia le variazioni del dato di roll condizionano i movimenti verticali della pallina, in questo caso di nuovo il campo *y* di `ballPosition`, invece le variazioni del dato pitch ne condizionano il dato *x*, per assegnare uno spostamento orizzontale alla pallina.

In questo caso, per tener conto dell'inclinazione che l'utente dà al dispositivo quando lo tiene in mano, sono stati aggiunti 15 gradi al dato `roll`.

3.7.2 Ampiezza del movimento

Un corpo quando corre lungo un piano inclinato, subisce l'accelerazione gravitazionale, producendo un movimento uniformemente accelerato. Inoltre, altri fattori che incidono direttamente sulla velocità del corpo sono l'inclinazione del piano e la massa del corpo stesso. L'angolo di inclinazione, dato dal pitch e dal roll, aumentano la velocità, la massa invece viene assegnata in base al raggio della pallina. Questo valore si è scelto di dividerlo per 10 in modo da rendere il movimento abbastanza lento da permettere di riconoscere dove si trova la pallina rispetto alla linea rappresentante il muro del labirinto. Dividendo il raggio per valori più bassi, a movimenti molto bruschi del dispositivo (*shaking*) la pallina a volte oltrepassava il muro.

Si è quindi adottata la formula 3.1 per il calcolo dello spostamento della pallina:

$$\begin{cases} s = \text{spostamento;} \\ m = \text{massa del corpo} = \texttt{ball.radius}/10; \\ g = \text{accelerazione gravitazionale} = 9.8; \\ \alpha = \text{angolo d}'\text{inclinazione;} \\ s = m * g * sin(\alpha). \end{cases} \tag{3.1}$$

3.8 Metodo edgeDetect

Questo metodo modifica i valori di ballPosition, che serviranno ad aggiornare la posizione dell'oggetto ball, in modo da mantenerla entro i muri del labirinto. Il tutto avviene per ogni limite, superiore, inferiore, destro e sinistro, accedendo ai campi lines.verlines e lines.horlines, che avevamo nominato nella sezione 3.2.3, nel seguente modo:

- dato quello che sarà il nuovo punto della pallina, chiamato touch, si controlla, per ogni segmento di lines, se c'è un segmento orizzontale che potrebbe costituire un confine per la pallina, per cui (lines.horLines[i]<= touch.x) && (touch.x<=lines.horLines[i+2]);
- determinato il segmento di interesse, si verifica se costituirà un limite superiore o inferiore, confrontando la coordinata *y* di ballposition con le coordinate *y* del segmento orizzontale considerato. Se (ballPosition.y <= lines.horLines[i+1]), il segmento sarà il limite superiore, altrimenti quello inferiore;
- quindi si calcola la distanza tra touch e il segmento;
- se questa è minore del raggio della pallina, vuol dire che la pallina supererebbe il muro, quindi la futura posizione della pallina viene calcolata in modo che non lo oltrepassi;
- inoltre viene impostata la variabile edge = true, dal momento che il muro è stato toccato;
- lo stesso viene fatto considerando i segmenti verticali, determinati invece con la condizione (lines.verLines[i+3]<=touch.y) && (touch.y<= lines.verLines[i+1]), che costituiranno il limite destro e sinistro; nel primo caso sarà verificato (ballPosition.x <= lines.verLines[i]), altrimenti avremo un limite sinistro.

Per rendere più chiaro il calcolo delle coordinate coinvolte, rimandiamo all'immagine 3.3.

Una volta calcolata la posizione della pallina, il metodo edgeDetect, se siamo nella modalità avanzata e la pallina ha toccato un muro, fa partire l'activity Victory, che riporterà stavolta il messaggio *Game over*. Questo passaggio si svolge in modo simile a quanto presente nello script 3.2, con b.putInt(level, level - 1) e b.putInt(result, 0).

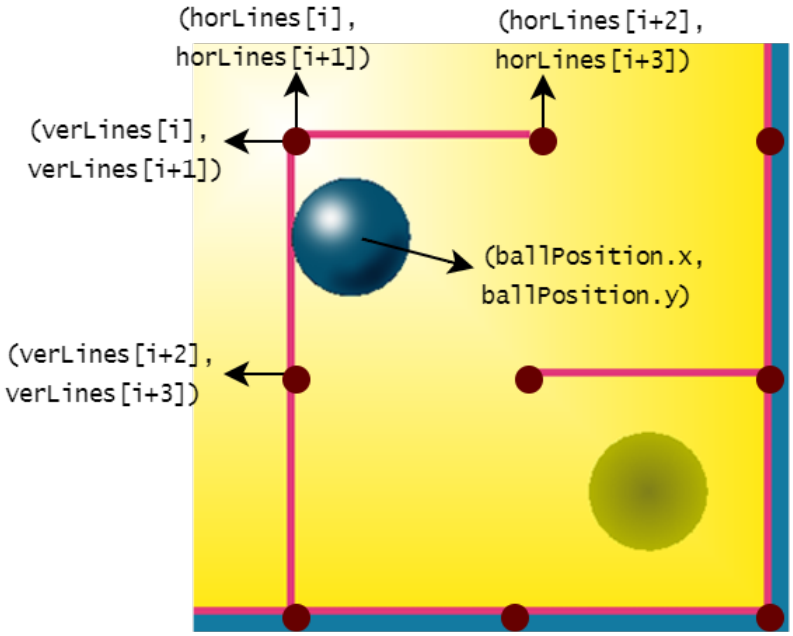


Figura 3.3: Rappresentazione grafica delle coordinate di `ballPosition`, `lines.horlines`, `lines.verlines`

Capitolo 4

Sviluppo e Test

In questo capitolo sarà spiegato l'ambiente di sviluppo utilizzato, la versione Android supportata dall'applicazione, e i dispositivi su cui sono stati svolti i test, con i relativi esiti.

4.1 Ambiente di sviluppo

Il progetto è stato sviluppato con Android Studio 1.5.1. La minima versione di Android supportata è la 5.0.1, corrispondente all'SDK 21, secondo le impostazioni date al momento della creazione. La *target SDK* è la versione 23, corrispondente alla versione di Android 6.0. Infine la *compiled SDK* è di nuovo la versione 23.

4.2 Test effettuati

L'applicazione non è stata testata con l'emulatore, per la presenza dei componenti OpenGl e per l'impossibilità di testare i sensori utilizzati.

Sono stati eseguiti i seguenti test su dispositivi:

- *LG Nexus 4*, con versione Android 5.1.1, in cui i test si sono dimostrati tutti positivi, e da cui sono state prese tutte le immagini utilizzate nella relazione;
- *Motorola MotoG 2013*, con versione Android 5.1.1, in cui i test si sono conclusi tutti in modo positivo;
- *LG Spirit*, con versione Android 5.0.1, in cui il non veniva visualizzato il font corretto, e al suo posto c'era quello di default, che, a causa delle dimensioni diverse, portava a visualizzazioni poco piacevoli. Per il resto i test sono stati positivi;

- *LG g3*, con versione Android 5.0.1, ha presentato lo stesso problema sulla visualizzazione del font;
- *htc One m7*, con versione Android 5.0.2, che ha presentato il problema della corretta visualizzazione del font;
- *htc desire 310*, con versione Android 5.0.1, che ha di nuovo presentato il problema di visualizzazione del font;
- *htc Sensation*, con versione Android 4.4, che era stato sbloccato in modo da avere privilegi di *root*, in cui i test sono stati conclusi in modo positivo;

Conclusione

Questa applicazione mi ha dato modo di apprendere molti concetti non solo di Android, ma anche di OpenGL. Infatti prima di procedere all'implementazione, è stata seguita un'attenta fase di documentazione e progettazione, in modo da produrre un codice robusto, facile da estendere e modificare con ulteriori funzionalità, e privo di elementi deprecati.

Questo tipo di applicazione infatti si presta a diversi sviluppi futuri, quali per esempio l'aggiunta di caratteristiche grafiche alla superficie di gioco, la memorizzazione di ulteriori dati, come il tempo impiegato al completamento dei livelli, l'aggiunta di ulteriori livelli con nuovi ostacoli, la condivisione online dei punteggi, una modalità di gioco multiplayer, e molto altro.

Dal punto di vista dell'interazione, l'interfaccia è semplice ma curata, immediata da usare e da capire. Sono stati inseriti molti elementi (sensori, fragment, gesture, memorizzazione di dati, supporto grafico, texture) perchè ritenuti utili e fondamentali ad una esperienza utente piacevole e che soddisfacesse le aspettative dell'utente.

Il tipo di gioco scelto, anche se già utilizzato in altre applicazioni, è risultato adatto allo scopo di esplorare molti aspetti della programmazione Android e di OpenGL.

Bibliografia

[1] *Paletton, Color Scheme Designer*, reperibile presso `http://www.paletton.com`.

[2] *GoodDog Plain Font*, creato da *Fonthead Design*, reperibile presso `http://www.1001fonts.com/gooddog-plain-font.html`.

[3] Raccolta e spiegazione dei principali algoritmi per la generazione automatica di labirinti casuali, reperibile presso `https://en.wikipedia.org/wiki/Maze_generation_algorithm`.

[4] Guida all'implementazione delle animazioni, reperibile presso `http://www.mysamplecode.com/2013/02/android-animation-switching-activity.html`.

[5] Codice utilizzato per la classe `OnSwipeTouchListener`, reperibile presso `http://stackoverflow.com/questions/4139288/android-how-to-handle-right-to-left-swipe-gestures/19506010#19506010`.

[6] Guida ufficiale allo sviluppo di applicazioni Android, *SharedPreferences*, reperibile presso `http://developer.android.com/guide/topics/data/data-storage.html#pref`.

[7] Guida ufficiale allo sviluppo di applicazioni Android, *Recreating an Activity*, reperibile presso `http://developer.android.com/training/basics/activity-lifecycle/recreating.html`.

[8] Codice di esempio per la costruzione di una bussola, reperibile presso `http://www.codingforandroid.com/2011/01/using-orientation-sensors-simple.html`.

[9] Guida ufficiale allo sviluppo di applicazioni Android, *Sensor Overview*, reperibile presso `http://developer.android.com/guide/topics/sensors/sensors_overview.html`.

[10] Guida ufficiale allo sviluppo di applicazioni Android, *Handling Runtime Changes*, reperibile presso `http://developer.android.com/guide/topics/resources/runtime-changes.html`.

[11] Guida all'utilizzo dei frammenti per il recupero di oggetti da parte dell'activity, reperibile presso `http://www.101apps.co.za/index.php/articles/fragments-configuration-changes-and-retaining-objects.html`.

[12] Brothaler Kevin, *OpenGL ES 2 for Android*, The Pragmatic Programmers, First edition, 2013.

[13] Codice per la generazione di un labirinto con l'algoritmo DFS in Java, reperibile presso `http://rosettacode.org/wiki/Maze_generation#Java`.

[14] Codice per la generazione di un labirinto con l'algoritmo di Kruskal in Java, reperibile presso `https://github.com/psholtz/Puzzles/tree/master/mazes/maze-04/java`.