Per redigere questa tesi è stato usato il LaTeX, nella distribuzione per Windows nota come MiKTeX (versione del software: 2.4). Per la parte grafica, sono stati usati i programmi *Inkscape 0.42.2* (http://www.inkscape.org) per il disegno, e *The GIMP 2.2.8* (http://www.gimp.org) per il ritocco grafico.

Indice

1 Introduzione **7**
- 1.1 Scopo del progetto 7

2 La piattaforma J2ME **9**
- 2.1 Struttura generale 9
- 2.2 Le classi per lo sviluppo di interfacce grafiche 10
- 2.3 Le classi per lo sviluppo di giochi 10
- 2.4 Le classi per la gestione della persistenza 14
- 2.5 Le classi per la gestione dell'audio 14
- 2.6 Tecniche per la compressione delle applicazioni 14

3 Progettazione **17**
- 3.1 Requisiti 17
- 3.2 Un primo diagramma 20
- 3.3 Vincoli di efficienza 21
- 3.4 Diagramma finale 21

4 Implementazione **25**
- 4.1 Dettagli implementativi 25
  - 4.1.1 Il *game-loop* 26
  - 4.1.2 Il movimento dei nemici 28
  - 4.1.3 Implementazione dei percorsi 30
- 4.2 Funzionamento su periferiche reali 31

5 Conclusione **35**
- 5.1 Efficienza e stabilità 35
- 5.2 Ulteriori sviluppi 35

A Schema del file di descrizione **37**

Elenco delle figure

2.1 L'albero delle classi nel *package* `game` di MIDP 2.0. 11
2.2 Il concetto di *view window* 12

3.1 Un primo diagramma degli oggetti 20

4.1 Effetti dovuti all'invocazione di `resetPath()`. 29
4.2 Curve di Bezier di 2° grado. 31

Capitolo 1

Introduzione

In cui si espongono le motivazioni che hanno portato all'implementazione di questa applicazione.

1.1 Scopo del progetto

Lo scopo di questo progetto è stato quello di costruire un *motore* per piattaforme *J2ME*, che permetta di costruire un gioco interattivo a partire da un file XML che descriva tale mondo. L'idea di partenza è stata quella di ottenere una piattaforma dotata di grande flessibilità, che permetta di spostare lo sviluppo di un gioco su periferica cellulare dal sviluppatore esperto nel linguaggio Java ad un ipotetico *sviluppatore* in linguaggio XML, semplificando enormemente il processo di sviluppo dei giochi.

Capitolo 2

La piattaforma J2ME

In cui, dopo aver parlato della struttura generale della piattaforma, si centra l'attenzione sulle classi e i *package* usati nel progetto. Per riferimenti si veda [1] oppure [2]

2.1 Struttura generale

Questa piattaforma è stata progettata per periferiche con potenza limitata. Prima della sua introduzione, periferiche mobili come i telefoni cellulari non avevano (o quasi) la possibilità di installare altre applicazioni oltre a quelle presenti al momento della costruzione della periferica stessa. Con l'introduzione di *J2ME*, il discorso cambia, e si comincia a poter scaricare e installare nuove applicazioni, scritte in linguaggio Java.
Per supportare la grande varietà di periferiche che appartengono all'area di interesse di *J2ME*, è stato introdotto il concetto di *Configurazione*. Essa definisce una piattaforma Java per una certa classe di periferiche, stabilendo le caratteristiche del linguaggio e le librerie necessarie. Ciò che però viene caratterizzato in particolare da una configurazione è relativo all'estensione della memoria, alle caratteristiche dello schermo, ed al tipo di connettività di rete supportata. Per esempio, la configurazione richiesta per i telefoni cellulari, e denominata **CLDC** (*Connected, Limited Device Configuration*):

- Requisiti di memoria: almeno 128 KB per il funzionamento di Java stesso, almeno 32 KB di memoria per l'allocazione degli oggetti.
- Interfaccia utente minima.
- Connettività di rete tipicamente *wireless*, con bassa larghezza di banda e accesso discontinuo.

Come abbiamo detto precedentemente, CLDC fornisce dettagliate specifiche riguardo anche gli aspetti più intrinsechi di Java, quali le varie caratteristiche del linguaggio e della *Virtual Machine*. La Sun ha prodotto un'implementazione di quest'ultima, da considerarsi come implementazione di riferimento, denominata **KVM** (*K Virtual Machine*); K sta per *kilo*, riferendosi all'ordine di grandezza di memoria per cui è stata progettata. Le sue caratteristiche più importanti sono:

- Richiede solo 40-80 KB di memoria per sè.
- Richiede solo 20-40 KB di memoria da riservare all'*heap*.
- Può funzionare su processori a 16 bit con un *clock* di 25 MHz.

Sorge però, poiché il mercato delle periferiche mobili varia molto in base al tipo, all'azienda costruttrice, allo sviluppo della tecnologia (quest'ultimo è un fattore importante; la tecnologia migliora ad un ritmo esponenziale), il problema della *compatibilità* di un' applicazione tra una periferica e l'altra.
Per esempio, non è possibile che qualsiasi telefono cellulare abbia il supporto, ancorché minimo, per l'utilizzo di librerie grafiche 3D (infatti, al momento della scrittura di questa tesi, solo gli ultimi modelli di telefoni cellulari della Sony Ericsson possiedono tale supporto), per cui risulta impossibile scrivere codice multipiattaforma, secondo la filosofia di Java.
A causa di questi motivi, è stato introdotto un altro concetto fondamentale, che assieme al concetto di *configurazione*, costituisce lo strumento di misura per classificare le periferiche alle quali è dedicata la piattaforma *J2ME*: il *Profilo*.
Un profilo è un'estensione, ad una configurazione. Per esempio, **MIDP** (Mobile Information Device Profile) è un insieme di classi per una completa gestione di interfacce grafiche, gestione degli eventi e della persistenza, utilizzo della connettività di rete, etc. fornite allo sviluppatore di applicazioni per periferiche mobili. Alcune classi di MIDP, nella versione 2.0, sarà l'argomento dei prossimi paragrafi.

2.2 Le classi per lo sviluppo di interfacce grafiche

2.3 Le classi per lo sviluppo di giochi

Con l'introduzione di MIDP 2.0, la piattaforma J2ME possiede alcune classi pensate appositamente per lo sviluppo di giochi per cellulari, in quanto è un

campo, quello ludico, che copre una grossa percentuale tra le varie tipologie di applicazioni mobili. Le classi in questione sono raggruppate all'interno del *package* `javax.microedition.lcdui.game` e sono strutturate come in figura. Le periferiche *wireless* possiedono processori di potenza ridotta, così

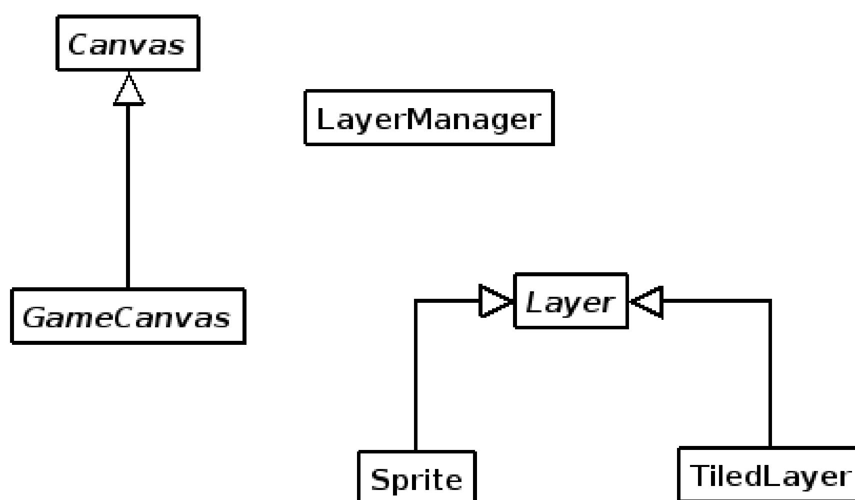


Figura 2.1: L'albero delle classi nel *package* `game` di MIDP 2.0.

tutta l'interfaccia di programmazione è strutturata per minimizzare la quantità di codice che deve scrivere l'utente; cosa che permette anche di ridurre la grandezza degli archivi delle applicazioni. Inoltre, queste classi sono strutturate in maniera così semplice anche per poter fornire un accesso più diretto all'hardware grafico (ad esempio: il *double buffering*[1]).
Il principio fondamentale su cui si basano queste API è quello dei *layers*, cioè livelli di costruzione della scena. Infatti ogni livello possiede un valore di profondità, in modo tale che viene coperto da tutti i livelli con un valore minore, e a sua volta copre tutti i livelli con un valore maggiore. Per gestire l'insieme dei livelli è stata progettata una classe apposita, `LayerManager`, che contiene una lista ordinata dell'insieme dei livelli. I metodi offerti per gestire questa lista sono i seguenti:

- `void append(Layer l)` Aggiunge un nuovo `Layer`, nella posizione più lontana dall'utente. Se questo `Layer` era già presente, viene prima rimosso.

[1]Questa tecnica permette di associare al video due *buffer*, che vengono alternativamente disegnati su schermo per ottenere l'effetto di continuità delle animazioni.

- `void insert(Layer l, int index)` Inserisce un nuovo `Layer`, dopo averlo rimosso se era già presente.

- `void remove(Layer l)` Rimuove il livello specificato.

Un'altra caratteristica importante gestita dalla classe `LayerManager` è quella della *view window*. Il metodo per impostarla è il seguente:

- `void setViewWindow(int x, int y, int width, int height)`

La *view window* specifica la regione che il `LayerManager` disegna quando ne viene invocato il metodo `paint`. Permette di controllare la regione visibile, nonché la sua locazione nel sistema di coordinate del `LayerManager`. La

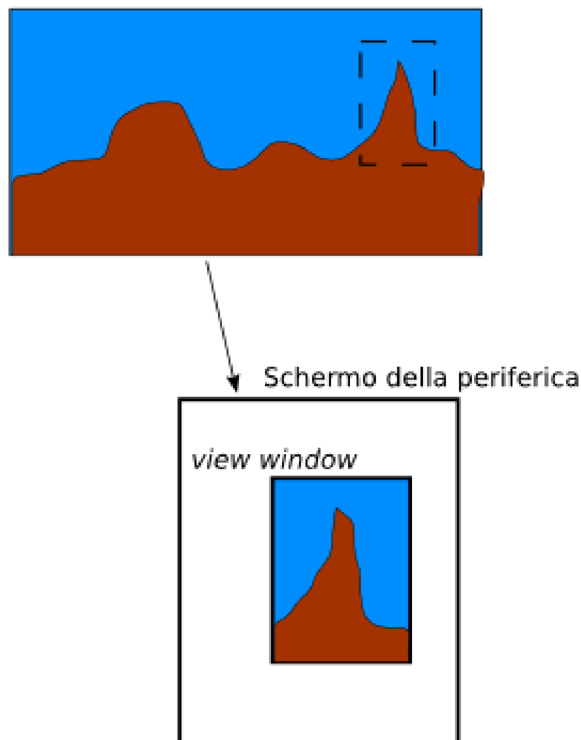


Figura 2.2: Il concetto di *view window*

classe base per rappresentare un livello è `Layer`, che è una classe astratta in cui è necessario ridefinire il metodo `void paint(Graphics g)` per ottenere

il comportamento desiderato dall'oggetto.
Le uniche due classi concrete, figlie di `Layer`, fornite da MIDP 2.0 sono `Sprite` e `TiledLayer`. Una `Sprite` è un elemento visuale che può essere disegnato utilizzando vari *frame* presenti nell'oggetto `Image` associato. Si possono così usare differenti *frame* per animare la `Sprite`. Questi *frame* possono essere ordinati nell'immagine in qualsiasi modo: per orizzontale, per verticale, o in forma tabellare, a seconda di come è più utile per lo sviluppatore. Ad ogni *frame* è assegnato un numero, a partire dallo zero, che corrisponde al *frame* in alto a sinistra nell'immagine. Il cambiamento di *frame* però non è automatico, e per ottenerlo si deve chiamare uno dei seguenti metodi:

- `void setFrame(int n)` per impostare ad un particolare *frame*
- `void prevFrame()` per impostare al *frame* precedente nella lista
- `void nextFrame()` per impostare al *frame* successivo nella lista

La lista della successione dei *frame* può essere impostata attraverso il metodo `void setFrameSequence(int[] sequence)`.
Essendo una sottoclasse di `Layer`, ogni oggetto `Sprite` possiede vari metodi per lavorare con le posizioni, come `void setPosition(int x, int y)`, `int getX()`, etc. Tutti questi metodi però si riferiscono al punto che si trova all'angolo superiore sinistro del *rettangolo di contenimento* dell'oggetto `Sprite`. A volte però è desiderabile utilizzare come punto di riferimento un altro punto a piacere; una possibilità utile soprattutto quando si lavora con le trasformazioni. Per questo ogni oggetto `Sprite` possiede i seguenti metodi:

- `void defineReferencePixel(int x, int y)` definisce il pixel di riferimento. Quando viene applicata una trasformazione, il pixel di riferimento rimane definito relativo all'angolo superiore sinistro prima della trasformazione.
- `void setRefPixelPosition(int x, int y)` posiziona l'oggetto `Sprite` cosicché il pixel di riferimento si trovi in (x, y).

Come è stato accennato precedentemente, su questo tipo di oggetti si possono applicare delle trasformazioni, che permettono di cambiarne l'aspetto. Le trasformazioni sono applicate invocando il metodo `void setTransform(int transform)`, dove l'argomento intero può assumere uno dei seguenti valori:

- `TRANS_NONE` non effettua nessuna trasformazione
- `TRANS_MIRROR` effettua un ribaltamento lungo l'asse verticale

- `TRANS_ROT90`, `TRANS_ROT180`, `TRANS_ROT270` effettua una rotazione oraria di 90, 180, 270 gradi
- `TRANS_MIRROR_ROT90`, `TRANS_MIRROR_ROT180`, `TRANS_MIRROR_ROT270` come il punto precedente, solo che preceduto da un ribaltamento lungo l'asse verticale

Un `TiledLayer` è un elemento composto da una griglia di celle che possono essere riempite con porzioni di immagine. Anche in questo caso i vari pezzi nell'immagine possono essere disposti secondo comodità. Ad ogni pezzo è assegnato un numero a partire da 1, che corrisponde al pezzo in alto a sinistra nell'immagine. Si possono creare due tipi di *tile*:

2.4 Le classi per la gestione della persistenza

2.5 Le classi per la gestione dell'audio

2.6 Tecniche per la compressione delle applicazioni

Bibliografia

[1] John W. Muchow. *Core J2ME Technology & MIDP*. Prentice Hall PTR, 2001.

[2] Vartan Piroumian. *Wireless J2ME Platform programming.* Prentice Hall PTR, 2002.

Capitolo 3

Progettazione

In cui si vogliono mostrare tutti i vari passaggi compiuti nella fase iniziale di progettazione, partendo da un modello di dominio indipendente dalla piattaforma finale e arrivando al diagramma delle classi finale.

3.1 Requisiti

Riprendendo quanto introdotto nel Capitolo 1, lo scopo del progetto è stato quello di creare un *motore* per applicazioni ludiche su piattaforma *J2ME*. Il desiderio era di creare una piattaforma autonoma, dotata di grande flessibilità, che semplifichi enormemente lo sviluppo di giochi su periferica cellulare. Non si voleva che il motore, scritto in linguaggio Java, ponesse alcun vincolo sulla semplicità del file di descrizione e sulla sua potenza espressiva. In particolare, era desiderio fare in modo che, una volta scritto il motore in Java, creare nuovi giochi comportasse solo il cambiare la descrizione testuale del gioco.
Per soddisfare tali requisiti, è sembrato ovvio partire, nella progettazione, dall'ideazione della struttura del file di descrizione. La cosa fondamentale era che il file doveva essere di testo, quindi aperto; non doveva essere inventato nessun formato binario particolare, anche se l'uso di un formato testuale avrebbe influito sulla *performance*. Fatta questa precisazione, la prima cosa che è venuta in mente per risolvere il problema di ideare un formato per il file di descrizione, è stata quella di usare uno standard largamente diffuso, che ha anche il pregio di essere direttamente supportato da alcune periferiche cellulari: XML. La definizione del mondo in formato XML obbliga a pensare ad esso come ad un *albero della scena*, il che permette anche di imporre una gerarchia tra gli oggetti del mondo. A questo punto, possiamo definire la struttura di un qualsiasi gioco alla maniera seguente:

Un gioco è costituito da n livelli, i quali possono contenere vari tipi di oggetti, compreso il giocatore.

Successivamente, è stato necessario decidere quali tipi di oggetti vogliamo far gestire al motore. Se si guarda il panorama dei giochi di medio[1] livello esistenti, si vede che ogni gioco si basa sull'interazione dei seguenti oggetti:

- Il giocatore, che ha sempre un punto di partenza in ogni livello,
- Alcuni oggetti che possono essere raccolti, sia per aumentare il punteggio che per ottenere dei *bonus*,
- Alcuni oggetti che si muovono in maniera autonoma, magari in maniera ciclica lungo un percorso prestabilito, e che spesso danneggiano il giocatore in caso di collisione,
- Oggetti che vengono lanciati, come per esempio i proiettili,
- Oggetti su cui il giocatore può camminare e/o arrampicarsi

Volendo creare qualcosa particolarmente generico, si è deciso di prendere in considerazione i seguenti tipi di oggetti:

- Il *giocatore*, le cui coordinate di partenza siano specificabili nel file di descrizione. Inoltre, il giocatore si può muovere all'interno del mondo, e può saltare secondo una curva prestabilita.
- Gli *oggetti generici*, che possono essere raccolti dal giocatore, e che aumentano il punteggio. Per ragioni di semplicità si è deciso di non prendere in considerazione oggetti che aggiungano potenzialità al giocatore. Le coordinate della posizione sono specificate nel file di descrizione.
- Gli *oggetti statici*, in cima ai quali può camminare il giocatore, e li può scalare ai lati. Le coordinate della posizione sono specificate nel file di descrizione.
- Gli *oggetti dinamici*, con i quali il giocatore può navigare il mondo, accedendo a posizioni altrimenti inaccessibili. Nel file di descrizione viene specificato l'intervallo bidimensionale in cui si può muovere liberamente.

[1] Ci si riferisce a giochi non troppo vecchi (e semplici) e né troppo recenti e particolari, che magari richiedono particolari API, come quelle per il 3D.

- I *nemici*, che sono oggetti che si muovo nella scena in maniera autonoma, lungo un percorso definito file di descrizione attraverso 3 coppie di coordinate (x, y).

Per capire meglio come funziona la descrizione, è utile fare un esempio. Pertanto, si consideri la descrizione seguente (per il *DTD* del documento, si veda l'Appendice):

```
<game name="...">
   ...
   <level number="..." background="...">
      <player xpos="..." ypos="..." image="..."/>
      <static xpos="..." ypos="..." image="..."/>
      <dynamic startx="..." starty="..." endx="..."
                endy="..." image="..."/>
      <object xpos="..." ypos="..." hide="false" image="..."/>
      <enemy startx="..." starty="..." mediumx="..."
            mediumy="..." endx="..." endy="..." image="..."/>
   </level>
   ...
</game>
```

Qui viene descritto un livello, in cui il giocatore parte dalla posizione $(xpos, ypos)$, c'è un solo oggetto statico nella posizione $(xpos, ypos)$, un solo oggetto dinamico il cui *intervallo di movimento* è $[endx - startx, endy - starty]$, un solo oggetto da raccogliere in $(xpos, ypos)$ e visibile, ed un solo nemico. Una caratteristica che si nota subito è rappresentata dal fatto che in tutti i tipi di oggetti c'è un attributo `image`. La sua presenza è dovuta ad un'altra scelta di progetto: per ottenere una grande generalità si è deciso di far sì che l'immagine associata ad ogni oggetto sia specificabile nel file di descrizione. In questa maniera con un oggetto statico possiamo rappresentare sia un palazzo, se il giocatore si trova in una città, oppure anche una palma, se il giocatore si trova nel deserto, e in generale qualsiasi oggetto che vogliamo stia fermo nella scena. L'unica particolarità da evidenziare riguarda l'immagine da associare al giocatore. In questo caso, infatti, l'immagine di input non deve essere costituita da un singolo *frame*, ma da ben 11, relativi alle varie posizioni del giocatore:

1. il giocatore sta fermo

2. due frame da alternare nel movimento verso destra

3. due frame da alternare nel movimento verso sinistra

4. due frame da alternare nella scalata sul lato sinistro
5. due frame da alternare nella scalata sul lato destro
6. salto verso destra
7. salto verso sinistra

3.2 Un primo diagramma

Partendo dai requisiti esposti nel paragrafo precedente, è stato progettato un primo diagramma degli oggetti coinvolti nell'applicazione. E' da tenere presente che questo diagramma è concepito come un diagramma iniziale, che non è costruito in riferimento alla piattaforma *J2ME*, ma solo come un diagramma generico degli oggetti. Questo diagramma rispecchia molto

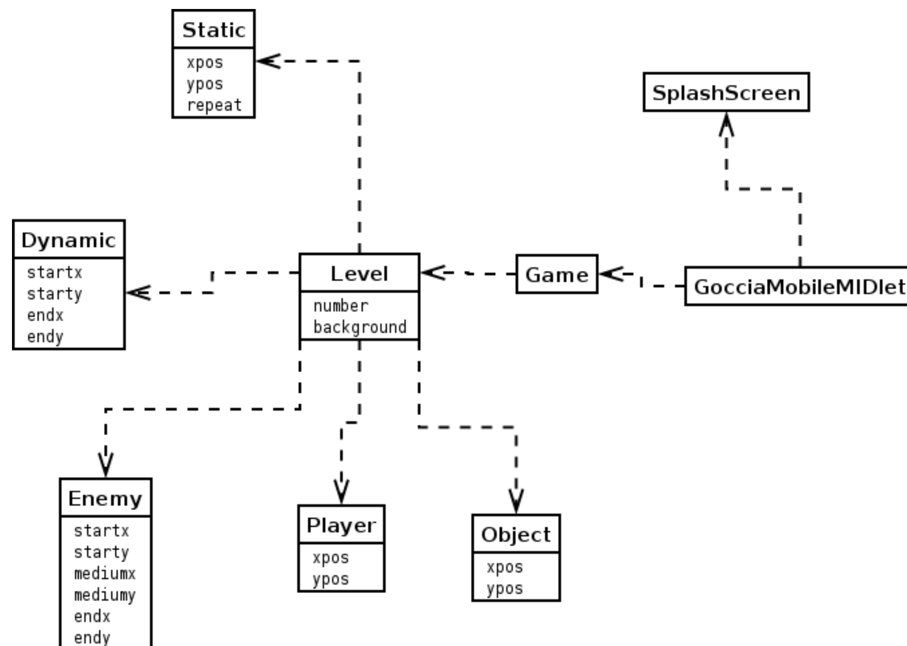


Figura 3.1: Un primo diagramma degli oggetti

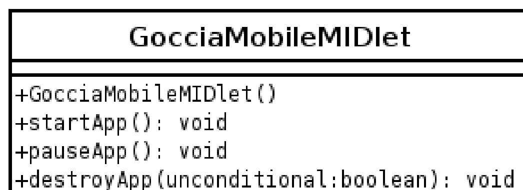
la struttura del file XML di descrizione, ed infatti rappresenta quelle che saranno le classi fondamentali della piattaforma.

3.3 Vincoli di efficienza

Nel passare dal diagramma iniziale a quello finale è stato tenuto conto di vari aspetti, dovuti al fatto che l'applicazione avrebbe dovuto girare su dispositivi dotati si scarsa potenza. Innanzitutto è stata fatta la scelta di non usare un numero troppo elevato di classi, in quanto la creazione di nuovi oggetti è un'operazione molto costosa. Al tempo stesso si è scelto di utilizzare in maniera estensiva l'ereditarietà, cosicché, sfruttando il riutilizzo del codice, si è ridotta il più possibile la quantità di codice da scrivere.

3.4 Diagramma finale

Oltre ai vincoli di efficienza esposti nel paragrafo precedente, bisogna tenere presente una caratteristica della piattaforma *J2ME*, cioè quella che non aiuta lo sviluppo di applicazioni che siano costruite secondo i buoni principi della progettazione ad oggetti. In particolare, la piattaforma *J2ME* induce lo sviluppatore a creare applicazioni che siano altamente accoppiate, soprattutto in rapporto con la piattaforma stessa, e quindi dotate di bassa coesione. Il motivo è evidente: fornendo la maggior parte possibile di caratteristiche nelle librerie, è possibile conseguire una maggiore *performance*, in quanto possono essere inserite in fase di costruzione della periferica, e in un modo efficiente. La classe principale dell'applicazione è `GocciaMobileMIDlet`, con il suffis-



so *`-MIDlet`* usato per una convenzione pressocché universale; come i piccoli programmi Java caricabili nel browser sono *Applet* e e le classi princiapli di un Application Server Java sono *Servlet*, le classi principali di un applicazione sulla piattaforma *J2ME* si chiamano *MIDlet*. Questa classe, essendo appunto quella principale, deve estendere la classe `MIDlet` all'interno del *package* `javax.microedition.midlet`, e deve ridefinirne i metodi astratti `startApp`, `pauseApp` e `destroyApp`. Praticamente, il contenitore crea un nuovo oggetto `GocciaMobileMIDlet`, e subito dopo ne invoca il metodo `startApp`.
La classe principale inoltre, implementa anche l'interfaccia `CommandListener` del *package* `javax.microedition.lcdui`. Questa interfaccia espone solo un

metodo, `void commandAction(Command c, Displayable d)`, che viene invocato ogni volta che l'utente sceglie una voce nel menu dell'applicazione, raggiungibile tramite uno dei due *softkey*. Questo menu è costituito da un insieme di oggetti `Command` a cui è associata una stringa, che viene mostrata sullo schermo. Ogni volta che l'utente sceglie una voce del menu, viene passato a `commandAction` un'istanza del comando scelto e l'istanza dell'oggetto grafico a cui è associato il comando stesso. Per esempio, nel caso della nostra applicazione, a commandAction viene sempre passato come secondo argomento l'istanza dell'oggetto `Game`, che estende la classe `Displayable` attraverso la classe `GameCanvas`.
Dato che un requisito è quello che all'avvio dell'applicazione venga mostrato uno *splash*, il metodo `startApp` mostra una nuova istanza della classe `SplashScreen`, che mostra il logo dell'azienda presso cui è stata sviluppata l'applicazione e che dura fino a quando l'utente non preme un tasto o siano trascorsi 6 secondi. Dopo lo *splash*, viene subito caricata una nuova istanza di Game, e parte l'esecuzione del suo metodo `run()` in un *thread* separato, che si occupa fondamentalmente del *rendering* della scena e registra continuamente lo stato dei tasti di azione.
I tasti azione che vengono gestiti dall'applicazione sono i seguenti: *left*, *right*, *up*, *down* e *fire*. I primi quattro tasti vengono premuti dall'utente per far muovere il giocatore a destra o a sinistra, oppure, se è possibile, per farlo salire e scendere lungo i lati degli oggetti `Static`. Il tasto *fire* invece, viene premuto quando l'utente desidera far saltare il giocatore, il quale segue un percorso definito da una particolare istanza della classe `Curve`.
Una volta che nel metodo `run()` di `Game` viene riconosciuta la pressione di uno dei tasti azione, viene richiesto l'aggiornamento dello stato all'istanza corrente della classe `Level`, la quale ha infatti i seguenti metodi:

- `void updateToLeft()` se è possibile sposta la posizione del giocatore a sinistra di 1 pixel. Inoltre, sposta anche la *view window* se, essendo la larghezza L_l del livello maggiore della larghezza L_s dello schermo, il giocatore si trova nell'intervallo $[\frac{L_s}{2}, L_l - \frac{L_s}{2}]$.

- `void updateToRight()` se è possibile sposta la posizione del giocatore a destra di 1 pixel. Per lo spostamento della *view window* si veda il punto precedente.

- `void updateToUp()` se il giocatore si trova lungo uno dei due lati verticali di un oggetto `Static` o si trova a bordo di un oggetto `Dynamic`, sposta il giocatore verso l'alto di 1 pixel. Nel secondo caso, lo spostamento avviene solo se l'oggetto dinamico si trova all'interno del suo *intervallo di movimento*, altrimenti non succede niente.

- `void updateToDown()` vale lo stesso discorso del punto precedente, tranne che lo spostamento è di 1 pixel verso il basso.

- `void updateWithFire()` sposta il giocatore di una posizione lungo un percorso rappresentato da un'istanza della classe `Curve`. Il salto viene effettuato verso destra o verso sinistra a seconda che l'ultimo movimento orizzontale del giocatore sia stato verso destra o verso sinistra. Dopo aver effettuato lo spostamento viene verificato che ciò non provochi una collisione con un oggetto `Static` lungo i suoi lati laterali o il lato inferiore, non provochi una collisione con un oggetto `Dynamic` lungo gli stessi lati, e non provochi una collisione con il bordo destro o sinistro del livello. In tutti questi casi il salto viene disabilitato e viene impostata una condizione di caduta che sposterà il giocatore lungo la verticale fino al bordo inferiore dello schermo, al lato superiore di un oggetto `Static`, oppure al lato superiore di un oggetto `Dynamic`.

- `void update()` effettua l'aggiornamento della scena quando non c'è interazione da parte dell'utente.

La classe `Level` possiede altri metodi con il prototipo del tipo `void add...(... obj)`, dove al posto dei puntini ci sono i nomi dei vari tipi di oggetti dlla scena. La classe `Level` possiede internamente un elenco dei vari oggetti presenti nel livello, e questi metodi vengono invocati dalla classe `Parser` ogni volta che costruisce uno di questi oggetti a partire dal file XML di descrizione.
Per quanto riguarda il *parsing* del file XML, esso viene effettuato utilizzando le librerie specificate in *JSR 172*, il quale definisce un *parser* di tipo *SAX*, ma minimale per ragioni di efficienza. La nostra classe `Parser` estende la classe `DefaultHandler` del *package* `org.xml.sax.helpers`, ridefinendo i due metodi `void startElement(...)` e `void endElement(...)`. Il primo metodo accetta i seguenti argomenti:

1. `String uri` è il *namespace* dell'elemento, o la stringa vuota se non è stato specificato.

2. `String localName` è il nome del *tag* senza il *namespace*, o la stringa vuota se quest'ultimo non è stato specificato.

3. `String qName` è il nome *completamente qualificato* del *tag*, o la stringa vuota se i nomi completamente qualificati non sono ammessi.

4. `Attributes attrs` è l'elenco degli attributi di questo elemento. Ogni attributo è incapsulato in una classe `Attribute` che possiede alcuni metodi per ottenerne il nome ed il valore.

Il metodo `endElement()` accetta gli stessi argomenti di `startElement` meno `attrs`, e con gli stessi significati. Questi due metodi vengono rispettivamente chiamati quando il *parser* incontra un *tag* di apertura e uno di chiusura. Nel primo caso viene costruito di volta in volta un nuovo oggetto relativo al *tag* incontrato.

Capitolo 4

Implementazione

In cui dapprima si parla di alcuni scelte implementative, e poi del *porting* dell'applicazione su dispositivi reali.

4.1 Dettagli implementativi

In questa sezione si vuole dare uno sguardo ad alcuni punti del codice, per permettere una maggiore comprensione del funzionamento interno dell'applicazione.
Innanzitutto, come si può vedere dai diagrammi mostrati nel capitolo precedente, è stato fatto un largo uso della tecnica del *multithreading*; questa tecnica permette ad un'applicazione di poter eseguire più compiti *contemporaneamente* e, sui computer dotati di più di un processore, è un meccanismo che astrae la suddivisione dei compiti tra di essi [1]. Comunque anche su dispositivi dotati di singolo processore il guadagno in termini di *performance* è alto [1]. Ed è per questo che anche se i dispositivi mobili sono dotati di potenza ridotta e hanno caratteristiche limitate (ad esempio non sono presenti unità *floating point* per il calcolo decimale), una caratteristica presente, e largamente utilizzata dagli sviluppatori soprattutto di applicazioni *real-time* come alcune tipologie di giochi, è proprio il *multithreading.*
Nella nostra applicazione, ci sono fondamentalmente due compiti che devono essere eseguiti contemporaneamente:

- Acquisizione dello stato dei tasti di azione,
- *Rendering* della scena.

Inoltre, il secondo punto può essere suddiviso nei seguenti due compiti:

- *Rendering* della posizione del giocatore e della *view window*,

- *Rendering* della posizione dei nemici (oggetti `Enemy`), che hanno un movimento indipendente.

Questa struttura è necessaria: aspettare che l'utente prema un tasto non deve bloccare l'esecuzione del gioco! Quindi, anche se mentre si aspetta il giocatore sta fermo, non rimangono fermi i nemici, che hanno un loro percorso da seguire; e se il giocatore fa una mossa non può bloccare l'acquisizione di un nuovo stato dei tasti di azione. In pratica, ci sono più comportamenti *paralleli*.

4.1.1 Il *game-loop*

Per non incorrere in questi problemi, nella nostra applicazione ci sono due classi che implementano l'interfaccia `Runnable`:

- `Game`, che si occupa del *rendering* della scena
- `Enemy`, che si occupa di spostare la posizione dei nemici

Il compito dell'acquisizione dello stato dei tasti di azione è eseguito dal *thread* principale, cioè lo stesso che ha istanziato la classe `GocciaMobileMIDlet`.
Il metodo `run()` della classe `Game` contiene il codice seguente:

```
1     ... // setup iniziale
2     while(loadNextLevel())
3     {
4         ...
5         System.gc();
6         ...
7         prevLevel.moveEnemies();
8         while(!prevLevel.isEndLevel())
9         {
10            int ks;
11
12            ks = getKeyStates();
13            switch(ks)
14            {
15                /* chiama i metodi appropriati della classe
16                   Level per aggiornare la posizione
17                   del giocatore */
18                ...
19            }
20            graphics.drawImage(...);
```

```
21            prevLevel.paint(graphics, 0, 0);
22            flushGraphics();
23            try
24            {
25                gameThread.sleep(20);
26            }
27            ...
28        }
29        ... // codice per la schermata alla fine di un livello
30    }
31    ... // codice per la schermata finale
```

Come si può vedere dal codice, il *game-loop* si basa su un doppio ciclo *while*; se c'è un altro livello da caricare, che è indicato dal valore di verità restituito dal metodo privato `loadNextLevel()`, lo carichiamo, e una delle prime cose che viene fatta è richiedere un ciclo di *garbage-collection* per liberare le risorse occupate dal caricamento e dal gioco dell'eventuale livello precedente. Subito prima di inizire il *rendering* della scena vengono fatti partire i *thread* di esecuzione di tutti i nemici presenti nel livello corrente (linea 7). Il ciclo più interno viene eseguito fintantoché il livello non è finito (condizione generata dalla collisione del giocatore con il bordo destro del livello[1]), ed è strutturato secondo uno schema classico:

1. verifica dello stato dei tasti azione, ed esecuzione dei necessari aggiornamenti
2. *rendering* della scena in base agli aggiornamenti eseguiti
3. *flushing* del *buffer* interno
4. pausa del *thread* corrente per permettere l'esecuzione degli altri *thread*, tra cui quello principale

Nel caso particolare della nostra applicazione, il punto numero 1 corrisponde alle linee di codice 12–19 (si tenga presente che `ks = getKeyStates()` *non acquisisce* un nuovo stato dei tasti di azione, *ma* lo *verifica*, in quanto l'acquisizione è compito del *thread* principale, N.d.A.); il punto numero 2 corrisponde alle linee 20–21; il punto numero 3 alla linea 22; il punto numero 4 alla linea 25. Se non fosse eseguito il passo numero 4 non si potrebbe far muovere il giocatore, mentre se non fosse eseguito il passo numero 3 non

[1] Si ricordi che quello che si vede sullo schermo è solo una porzione dell'intero livello. La collisione anche con il bordo destro dello schermo è solo una conseguenza, ed è dovuta al concomitante movimento della *view window* e del giocatore.

verrebbe mai visualizzato niente sullo schermo. Il valore passato al metodo `sleep(...)` alla linea 25 rappresenta l'intervallo, in millisecondi, durante il quale il *thread* corrente viene sospeso; in questo modo il controllo può andare temporaneamente ad altri *thread*[2]. Cambiare questo numero implica un cambiamento nella velocità di gioco.

4.1.2 Il movimento dei nemici

Come è stato detto all'inizio di questa sezione anche la classe `Enemy`, che rappresenta i nemici del giocatore e che hanno movimento indipendente, implementa l'interfaccia `Runnable`. Quindi, se nel file di descrizione sono specificati n nemici, avremo in totale nella nostra applicazione $2 + n$ *thread* di esecuzione[3]. Il movimento dei nemici viene fatto iniziare con la chiamata, nel *game-loop*, al metodo `prevLevel.moveEnemies()`, che a sua volta invoca il metodo `startAnim()` su ogni oggetto `Enemy` del livello corrente.

```
1   void startAnim()
2   {
3       enemyThread.start();
4   }
```

Il codice che viene eseguito nel *thread* separato è il seguente:

```
1   public void run()
2   {
3       int[] pos = null;
4
5       while(continuePath)
6       {
7           c.resetPath();
8           ...
9           while(c.hasMoreElements())
10          {
11              pos = (int[])c.nextElement();
12              move(pos[0], pos[1]);
13              try
14              {
15                  enemyThread.sleep(200);
16              }
```

[2]Questo metodo non si comporta come il metodo `yield()`; durante l'intervallo in cui il *thread* è addormentato non perde nessun *lock*.

[3]1 *thread* principale + 1 *game-loop* + n nemici $= 2 + n$.

```
17                  ...
18              }
19          }
20      }
```

Ogni nemico continua a muoversi lungo il suo percorso in maniera ciclica, almeno fino a quando la variabile `continuePath` rimane vera. E quest'ultima possiede tale valore fino a che non è finito il livello corrente. A quel punto prima di distruggere l'oggetto relativo al livello finito il sistema invoca il seguente metodo, che rende falsa la variabile, su ogni oggetto `Enemy`:

```
1   void stopAnim()
2   {
3       continuePath = false;
4   }
```

In questa modo il *thread* termina in maniera pulita. Dopo essere entrati nel primo ciclo, viene reimpostato il percorso, rappresentato dalla variabile `c` (per una trattazione dei percorsi si veda la sottosezione successiva); questo poiché il percorso deve essere seguito indefinite volte. Per maggiore chiarezza, viene presentata la seguente immagine: In questa immagine, la freccia

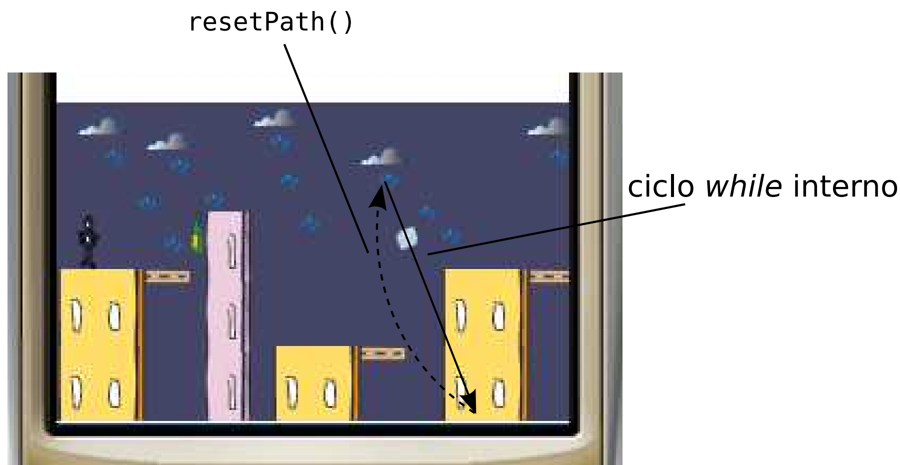


Figura 4.1: Effetti dovuti all'invocazione di `resetPath()`.

continua indica il percorso seguito dal blocco di ghiaccio, ed il cui codice è quello del ciclo `while` più interno; la freccia tratteggiata invece, rappresenta l'incovazione di `resetPath()`, che riporta il blocco di ghiaccio nella posizione di partenza.

4.1.3 Implementazione dei percorsi

Sia nel caso, appena esposto, degli oggetti `Enemy`, che nel caso del salto del giocatore, a seguito della pressione del tasto *fire*, si è parlato spesso di - percorso. Proprio a motivo dell'importanza che riveste questo concetto nella nostra applicazione, si è deciso durante l'implementazione di dargli uno *status* particolare, e ispirandosi al *pattern Strategy*[4][2] si è incapsulato il percorso in una classe `Curve`. Sia i nemici che il giocatore hanno un oggetto interno che è istanza, costruita a partire da opportuni parametri, di questa classe.
La caratteristica principale che era necessario avesse un'implementazione del percorso era che avrebbe dovuto utilizzare solo qualche punto per disegnarlo; affinché, anche in questo caso, un ipotetico sviluppatore di un nuovo gioco avrebbe potuto svincolarsi il più possibile dalla rappresentazione in pixel della scena. Inoltre, dato che i dispositivi cellulari non sono dotati di unità per il calcolo decimale (e quindi, in Java, non sarebbero stati disponibili i tipi di dato `double` o `float`), si voleva discretizzare la curva del percorso in parti uguali, da far corrispondere alle varie posizioni del giocatore durante il salto o dei nemici durante il loro movimento. Per venire incontro a tutte queste esigenze, si è optato per una particolare tipologia di parametrizzazione delle curve, che è quella delle *curve di Bezier*. Questa parametrizzazione è di tipo polinomiale, ed è funzione del numero dei punti di controllo. La formula generale di questo tipo di parametrizzazione è la seguente:

$$C(u) = \sum_{i=0}^{n} \binom{n}{i} u^i (1-u)^{n-i} p_i \tag{4.1}$$

con il parametro u che varia nell'intervallo $[0, 1]$. Per esempio, 4 punti di controllo possono essere approssimati con una curva polinomiale di grado 3:

$$C_5(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u)p_2 + u^3 p_3 \tag{4.2}$$

l pregi di questa rappresentazione coincidono proprio con le caratteristiche richieste; ad esempio, non è necessario ricalcolare ogni volta i valori $u^i(1-u)^{n-i}$ se si cambiano i p_i[5] (chiamati *punti di controllo* della curva).
Nella nostra applicazione si è scelto di permettere allo sviluppatore di fornire 3 parametri che definiscano l'aspetto della curva[6]: punto iniziale, punto finale e tangente iniziale. Ad esempio: La curva inoltre è stata discretizzata con 20 valori equidistanti di u, i quali sono calcolati preliminarmente.

[4] Del *pattern Strategy* si è sfruttata l'idea che anche i comportamenti sono oggetti.

[5] Questa importante proprietà delle curve di Bezier si chiama *invarianza affine*.

[6] Questo vale per i nemici, perché l'ampiezza del salto di Goccia è gestita internamente. Anche in questo caso si fa comunque uso di un oggetto `Curve`.

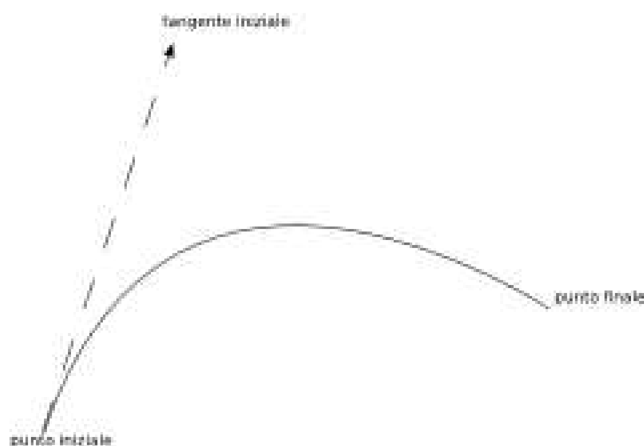


Figura 4.2: Curve di Bezier di 2° grado.

4.2 Funzionamento su periferiche reali

Verso la fine del progetto, una volta che tutto funzionava sugli emulatori, buona parte del tempo è stata spesa per risolvere i problemi causati dal *porting* dell'applicazione su periferiche reali. Sebbene infatti *J2ME* sia una piattaforma sempre più diffusa e stabile sulle periferiche mobili, ancora molti problemi si presentano poiché non tutti i costruttori scelgono di rispettare appieno i vari standard, e ciò costringe i singoli sviluppatori o di dotarsi di molti dispositivi per fare *testing*, oppure di operare una scelta su quali periferiche supportare nelle proprie applicazioni. Una delle poche eccezioni in questo panorama così diversificato, è il caso degli ultimi cellulari della Sony Ericsson, la quale produce dispositivi molto al passo con gli standard, e per i quali è molto facile sviluppare nuove applicazioni.
Nel caso dell'applicazione in esame, la scelta fatta sin dall'inizio è stata quella di supportare principalmente i cellulari Nokia. Il motivo di questa scelta è dovuta al fatto che questa azienda produce dispositivi dotati di grande *usabilità*, la quale a sua volta ha prodotto una larga diffusione di questi dispositivi.
Tornando all'applicazione in esame, i due grossi problemi avuti nel passare su periferica reale è stato l'utilizzo di due librerie, quella per il *parsing* di sorgenti scritti in formato XML, e quella che permette l'utilizzo di *file audio* nelle applicazioni. Il problema con la libreria *JSR 172* è stato risolto includendola interamente nel package dell'applicazione. Invece il problema con l'altra libreria ??????

Bibliografia

[1] Daniel J. Berg Bill Lewis. *Multithreaded prgramming with Java technology*. Prentice Hall PTR, 1999.

[2] Johnson Vlissides Gamma, Helm. *Design Patterns*.

Capitolo 5

Conclusione

In cui si considera la situazione attuale del progetto, e si evidenziano le basi necessarie per un eventuale proseguio del progetto.

5.1 Efficienza e stabilità

In cui si parla dell'efficienza del progetto in termini di memoria e la sua stabilità rispetto a varie possibili situazioni non previste dall' implementazione.

5.2 Ulteriori sviluppi

Per far vedere come aggiungere ad esempio altri tipi di oggetti nel gioco e quindi altri tag al file XML.

Appendice A

Schema del file di descrizione

Ciò che segue è la definizione secondo lo standard *XML Schema* del file di descrizione dell'applicazione.