# U N A ´ M ´

Facultad de Ingeniería

**A ´ 2D**

**´ J2ME**

**T E S I S**

que para obtener el grado de:

**I C ´**

P R E S E N T A

Nadxelle Velasco Gutiérrez

Director de Tesis: Ing. Emmanuel Hernández Hernández

Ciudad Universitaria, México D.F., 2008

# Dedicatoria

*A la U.N.A.M., mi alma mater y mi casa.*
*A mis padres, Sergio Velasco y Ma. Reyes Gutiérrez*
*A mis hermanos, Ximena y Cristian*
*Y muy especialmente a mi amado esposo Emmanuel.*

# Agradecimientos

Este trabajo hubiera sido imposible sin todas aquellas personas que siempre estuvieron ahí apoyándome. Enumerarlas me llevaría otro libro más, pero aún así me gustaría agradecer especialmente . . .

A la Universidad Nacional Autónoma de México, por el privilegio de ser parte de ella.

A mi padre Sergio Velasco, que aún hoy sigue trabajando para sacar a la familia adelante.

A mi madre Mariré que desde que era una niña me enseñó a buscar la excelencia.

Al amor de mi vida, Emmanuel, por su paciencia, su amor y por ser uno de mis mejores maestros. Sin todos sus jalones de orejas, ¡Quién sabe si hubiera terminado!

Al Dr. Jesús Savage Carmona y a todo el laboratorio de Biorobótica. El apoyo, los retos y las oportunidades que me dieron contribuyeron enormemente en mi formación.

A mis profesores de la Facultad de Ingeniería, tanto buenos como malos porque de todos ellos aprendí algo valioso y aplicable a mi vida.

A mis amigos de la carrera, mis compañeros de batalla: Carlos Alegría (mi ejemplo a seguir en cuanto a orden al programar), Silvia y Manuel (¡qué buen equipo éramos!), Jose Luis, Carelia y todos aquellos que formaron parte de mi vida y formación universitaria.

# Índice general

**Introducción**

**Notación y Terminología**

**1. Antecedentes teóricos** **1**

1.1. Tecnologías para el desarrollo de aplicaciones en dispositivos móviles 1

1.1.1. J2ME . . . . . 2

1.1.2. BREW . . . . . 6

1.1.3. Symbian . . . . . 8

1.1.4. Windows Mobile . . . . . 9

1.2. Diferencias entre dispositivos móviles y computadoras personales . . . 10

1.3. Motores Gráficos . . . . . 12

1.3.1. Tipos de Gráficas . . . . . 12

1.3.2. Ejemplos de Motores Gráficos . . . . . 14

1.3.3. Áreas de aplicación . . . . . 15

1.4. Elementos de un Motor Gráfico 2D . . . . . 18

1.4.1. Manejo de ventana y puerto de vista . . . . . 18

1.4.2. Manejo de capas o *layers* . . . . . 19

1.4.3. Manejo y soporte de *bitmaps* . . . . . 19

1.4.4. Manejo de animaciones y *Sprites* . . . . . 19

1.4.5. Manejo de texto . . . . . 21

1.4.6. Manejo de audio . . . . . 21
1.4.7. Manejo de dispositivos de entrada . . . . . 21
1.4.8. Manejo de colisiones . . . . . 22
1.4.9. Manejo de *tiles* . . . . . 23
1.4.10. Estructuras para manejo de la lógica del programa . . . . . 24
1.4.11. Manejo de física . . . . . 24
1.4.12. Manejo de comunicación vía red . . . . . 25

**2. Arquitectura del Sistema** **27**
2.1. Diseño General . . . . . 27
2.2. El *Main Loop* . . . . . 30
2.3. Módulo de Lógica . . . . . 33
2.4. Módulo de Dibujo . . . . . 36
2.4.1. Elementos Gráficos Principales . . . . . 39
2.4.2. Los *Tiles* . . . . . 40
2.4.3. Los Menús . . . . . 42
2.4.4. Los cuadros de texto o diálogos . . . . . 44
2.5. Módulo de Audio . . . . . 46
2.6. Módulo de Colisiones . . . . . 49
2.7. Módulo de interacción de usuario . . . . . 51
2.8. Módulo de Física . . . . . 52
2.9. Módulo de comunicación vía red . . . . . 52
2.10. Módulos de Funcionalidad Extra . . . . . 53

**3. Implementación de la Arquitectura** **57**
3.1. *Kansik,* un motor para la enseñanza . . . . . 57
3.2. La clase *Game* . . . . . 58
3.3. Implementación del Módulo de Lógica . . . . . 62
3.4. Implementación del Módulo de Dibujo . . . . . 63

3.5. Implementación del Módulo de Audio . . . . . 64
3.6. Implementación del Módulo de Colisiones . . . . . 65
3.7. Implementación del Módulo de Interacción de Usuario . . . . . 65
3.8. Implementación del Módulo de Funcionalidad Extra . . . . . 66

**4. Módulos de Prueba** **67**
4.1. Juego tipo *puzzle* . . . . . 68
4.2. Juego tipo RPG . . . . . 73

**Conclusiones y trabajo futuro** **77**

**A. Ejemplo de uso de Kansik** **79**
A.1. La clase MiniGameMidlet . . . . . 80
A.2. La clase MiniGame . . . . . 81
A.3. La clase Megaman . . . . . 83

**B. Instalación de Aplicaciones J2ME** **87**

**Bibliografía** **91**

# Índice de figuras

1.1. Java 2.0 . . . . . 2
1.2. MIDP 2.0 . . . . . 5
1.3. Usos de los motores gráficos . . . . . 17
1.4. Ventana y Puerto de Vista . . . . . 19
1.5. Capas o Layers . . . . . 20
1.6. Cuadros de un *Sprite* . . . . . 20
1.7. Colisión mediante rectángulos . . . . . 22
1.8. Colisión mediante rectángulos reducidos . . . . . 22
1.9. Colisión mediante verificación a nivel píxel . . . . . 23
1.10. *Tiles* . . . . . 24

2.1. Los módulos de *Kansik* . . . . . 30
2.2. El *Main Loop* . . . . . 31
2.3. Diagrama de Clases,*Updatable* . . . . . 33
2.4. Diagrama de Clases, Paquete de Lógica . . . . . 34
2.5. Diagrama de Clases, *LoadableContent* . . . . . 35
2.6. Diagrama de Casos de Uso, Módulo de Dibujo . . . . . 36
2.7. Diagrama de Clases, *Layer* . . . . . 37
2.8. Diagrama de Clases, *LayerManager* . . . . . 38
2.9. Diagrama de Clases, *IRenderRule* . . . . . 38

2.10. Diagrama de Clases, Elementos Gráficos Principales . . . . . 39
2.11. Diagrama de Clases, *Tiles* . . . . . 41
2.12. Diagrama de Clases, *ITilable* . . . . . 42
2.13. Diagrama de Clases, Menús . . . . . 43
2.14. Diagrama de Clases, Diálogos . . . . . 45
2.15. Casos de Uso, Módulo de Audio . . . . . 48
2.16. Diagrama de Clases, Módulo de Audio . . . . . 49
2.17. Diagrama de Clases, *CollitionMap* . . . . . 50
2.18. Diagrama de Clases, Paquete *RPG* . . . . . 55

3.1. Diagrama de Clases, *Game* . . . . . 59
3.2. Diagrama de Clases, *ILayers* . . . . . 64

4.1. Pantalla del *Puyo Puyo* . . . . . 69
4.2. *Framerates* del *Puyo* en Emuladores . . . . . 71
4.3. *Framerates* del *Puyo* celulares reales . . . . . 72
4.4. Pantalla del *RPG* . . . . . 73
4.5. *Framerates* del *RPG* . . . . . 75

B.1. Diagrama de Clases,*Updatable* . . . . . 88

# Introducción

Hoy en día casi todas las actividades de la vida cotidiana implican directa o indirectamente el uso de una computadora. Desde el sistema de cobro en los supermercados, los cajeros automáticos de los bancos y los sistemas de inventario de los grandes almacenes; hasta la industria del entretenimiento con los videojuegos y los efectos especiales para el cine; pasando por el control del tráfico aéreo y telecomunicaciones; cálculos científicos y visualización gráfica de resultados; comunicación remota a través de Internet, correo electrónico, *chats* y videoconferencia; además de un sin número de aplicaciones sin las cuales no se podría concebir el mundo moderno.

A esto va aunado el gran empuje que han tenido en los últimos años los dispositivos móviles, con los que se pretende traer en la palma de la mano, toda la funcionalidad de una computadora, ya sea para siempre estar comunicado y conectado con el mundo, mantener una agenda o simplemente tener un medio de entretenimiento de bolsillo. Es por esto que el desarrollo de aplicaciones para dichos dispositivos, ha adquirido mayor fuerza, atrayendo nuevas tecnologías e investigaciones para su mejora y expansión. Entre dichos dispositivos tenemos a las *PDA*[1], los localizadores, los reproductores de música digital, las consolas de videojuegos portátiles y los teléfonos celulares; estos últimos, sin lugar a dudas, de los más usados actualmente puesto que prácticamente cualquier persona sin importar su localización, su estrato social o

[1] Asistente personal digital por sus siglas en inglés: *Personal Digital Assitant*

edad posee alguno para uso personal.

Es sobre estos dispositivos móviles donde también últimamente ha habido un gran auge de aplicaciones gráficas principalmente para el entretenimiento. De fábrica, prácticamente todos los celulares tienen algún videojuego preinstalado, además con la opción de agregarle otros más pagando una cuota determinada. Las *PDA* no son la excepción y también tienen aplicaciones preinstaladas para el entretenimiento. Es más, hasta los reproductores de audio digital como el *Ipod*, traen instalados juegos de fábrica.

Lo anterior responde al increíble crecimiento que la industria de los videojuegos ha tenido en los últimos años, superando en ganancias incluso a la industria cinematográfica. Tan sólo en el año pasado (2007) las ganancias que se reportaron relacionadas con los videojuegos, según la *NPD Group* [2], superaron los 18 billones de dólares y esto exclusivamente en los Estados Unidos de Norteamérica, por lo que todavía habría que agregar lo que el resto de América, Asia y Europa sumarían.

No es de sorprenderse entonces que de todas partes del mundo surjan desarrolladoras con la intención de tomar, aunque sea un pequeño porcentaje, de tan enormes ganancias y México no es la excepción. Lamentablemente esta industria de entretenimiento está apenas comenzando en nuestro país, con sólo unas cuantas compañías desarrollando este tipo de aplicaciones y aunque han habido intentos por parte de la Secretaría de Gobernación para fomentar la creación de tecnología propietaria, a penas se están viendo respuestas de parte de pequeños grupos y escuelas, por lo que parece que todavía queda mucho por hacer.

Debido a la situación descrita anteriormente es importante que la Facultad de Ingeniería de la UNAM se haga de herramientas, preferentemente de código propio, para que pueda abordar en el corto plazo el desarrollo de aplicaciones gráficas para dispositivos móviles. Contar con el código fuente de las herramientas, proporciona a las nuevas generaciones, bases sobre las cuales implementar y hacer crecer el desarrollo en dichos dispositivos, lo que a su vez evitaría la dependencia de implementaciones externas, las cuales, al menos en el campo de los dispositivos móviles, son muchas veces de código cerrado, costosas y sin posibilidad de expandirse.

Es por esto que el **objetivo** de esta tesis es **crear un conjunto de librerías de programación para el desarrollo de aplicaciones gráficas interactivas en dispositivos**

[2] La antes llamada *National Purchase Diary*, es la compañía estadounidense más importante encargada de proporcionar información sobre la venta y consumo de productos en el mercado minorista

**móviles**. Estas librerías adoptarán la filosofía de trabajo de un *engine* de videojuegos en 2D.

## Organización de la tesis

La tesis está dividida en cuatro capítulos:

- El **capítulo 1**, el cual contiene los antecedentes teóricos necesarios para el desarrollo de este trabajo: comenzando con la investigación de las diversas plataformas para el desarrollo de aplicaciones para móviles; continuando con un listado de las diferencias que existen entre las computadoras de escritorio y los móviles; anotando los diferentes tipos de motores que existen y haciendo un recuento de la funcionalidad básica que éstos proporcionan.
- El **capítulo 2** corresponde al diseño de la arquitectura del motor gráfico a desarrollar, explicando cada uno de sus módulos.
- El **capítulo 3** se enfoca principalmente en la implementación del motor sobre *J2ME*[3], los problemas que se presentaron y la manera como se solventaron.
- Por último, **el capítulo 4**, muestra los módulos de prueba implementados que corroboran el correcto funcionamiento del motor desarrollado, junto con un pequeño análisis del desempeño del mismo, tanto en emuladores, como en hardware real.

Al final se incluyen las conclusiones y trabajo futuro con respecto del trabajo, así como la bibliografía del mismo.

[3] *Java 2 Micro Edition*

# Notación y Terminología

A continuación se presenta la notación utilizada a lo largo de este trabajo. En el cuadro se presentan todos los acrónimos y abreviaturas utilizados, así como su significado.

| | |
|---|---|
| **2D**. | Dos dimensiones. |
| **3D**. | Tres dimensiones. |
| **API**. | Interfaz de programación para aplicaciones, *Application Program Interface*. |
| **PDA**. | Asistente Digital Personal, *Personal Digital Assistant*. |
| **BREW**. | *Binary Runtime Environment for Wireless*. |
| **CAD**. | Diseño asistido por computadora, *Computer Assisted Design*. |
| **CLDC**. | Configuración para dispositivos con conexión limitada, *Connected Limited Device Configuration*. |
| **CPU**. | Unidad central de procesamiento, *Central Processing Unit*. |
| **GPU**. | Unidad de procesamiento de gráficos *Graphics Processing Unit*. |
| **GSM**. | Tecnología de telefonía celular, *Groupe Spécial Mobile*. |
| **J2ME**. | *Java 2 Micro Edition*. |
| **KVM**. | Máquina virtual de kilobytes, *Kilobyte Virtual Machine*. |
| **MIDI**. | *Musical Instrument Digital Interface*. |
| **MIDP**. | Perfil de J2ME para dispositivos móviles, *Mobile Information Device Profile*. |
| **MP3**. | Audio comprimido *MPEG*-1/2 capa 3. |
| **MPEG**. | Grupo encargado de dictar estándares de codificación de audio y vídeo, |

| | |
|---|---|
| | *Moving Pictures Experts Group*. |
| **NPC**. | *Non Player Characters*. |
| **PCM**. | Modulación por impulsos codificados, *Pulse Code Modulation*. |
| **RAM**. | Memoria de Acceso Aleatorio *Random Access Memory*. |
| **RPG**. | Juego de Rol, *Role Playing Game*. |
| **WAV**. | Formato de audio, apócope de *Waveform audio format*. |

# Capítulo 1

# Antecedentes teóricos

## 1.1. Tecnologías para el desarrollo de aplicaciones en dispositivos móviles

El desarrollo para dispositivos móviles está teniendo un gran auge en estos tiempos debido a la proliferación de estos pequeños aparatos, que sin lugar a dudas, ya forman parte de nuestra vida diaria. Tanto los asistentes personales (*PDA*) como los *teléfonos celulares*, son de uso común en casi todos los extractos sociales. Cada día surgen nuevas tecnologías para mejorar estos dispositivos y su mercado sigue creciendo a razones que nos recuerdan a las computadoras de hace quince años. Y como en aquellas épocas, han surgido diversas tecnologías para desarrollar aplicaciones especiales para dichos dispositivos, las cuales varían en portabilidad, desempeño, facilidad de uso y disponibilidad.

Entre las tecnologías para desarrollo de aplicaciones para dispositivos móviles más importantes se encuentran: la *J2ME*, *BREW*, *Symbian* y *Windows Mobile*.

### 1.1.1. J2ME

*J2ME* es el acrónimo de *Java 2 Micro Edition*. *J2ME* es la versión de *Java* orientada a los dispositivos pequeños con capacidades limitadas con respecto a una computadora personal; como son los teléfonos celulares, los asistentes personales (*PDA*), los sistemas embebidos, electrodomésticos, etc. Debido a que los dispositivos móviles tienen una potencia de cálculo baja e interfaces de usuario pobres, se necesita una versión específica de *Java* destinada a estos dispositivos, por tanto *J2ME* es una versión *reducida* de *J2SE* (*Java 2 Standard Edition*).

Debido a que *J2ME* está pensado para cubrir gran número de dispositivos con características muy diversas, se estructura en *configuraciones*, *perfiles* y ciertos *paquetes opcionales* ya que es necesario proveer diferentes grupos de clases para cada uno de los grupos de dispositivos que soportan *J2ME*.

En la figura 1.1 se muestra como se estructura la plataforma Java en la actualidad. Ahí también se puede apreciar que dentro de la *J2ME* tenemos dos configuraciones básicas: *CDC* (Connected Device Configuration) y *CLDC* (Connected Limited Device Configuration); y sobre ellas sus respectivos *perfiles*.

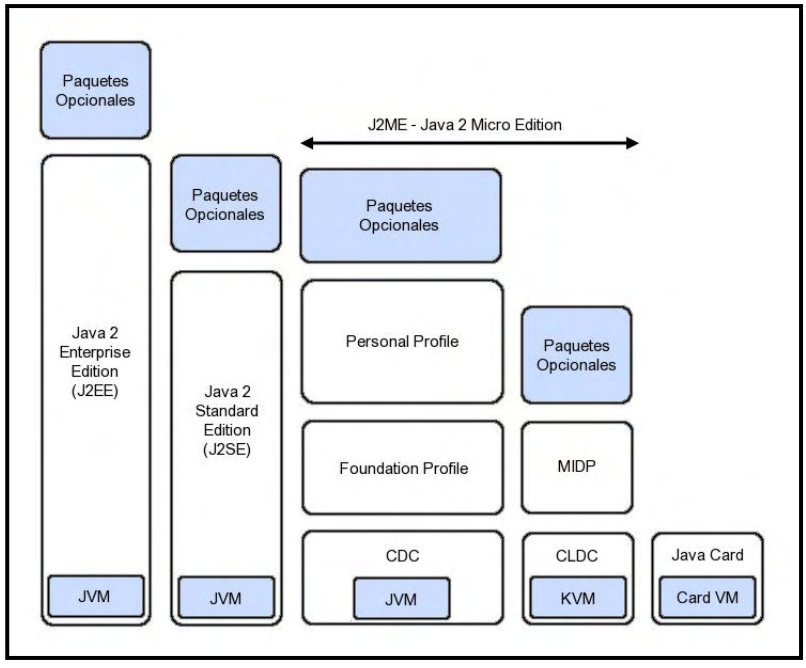



Figura 1.1: Java 2.0

### Configuraciones

Una configuración es un mínimo grupo de *API* (*Application Program Interface*), útiles para desarrollar las aplicaciones destinadas a un amplio rango de dispositivos.

La configuración estándar para los dispositivos inalámbricos con capacidades más limitadas, y también la configuración más extendida, es conocida como *CLDC* o *Connected Limited Device Configuration* por sus siglas en inglés.

*CLDC* es el conjunto de clases esenciales para construir aplicaciones para teléfonos celulares y agendas electrónicas o *PDA* con capacidades medianamente limitadas. Trabaja sobre una máquina virtual más pequeña llamada *KVM* (*Kilobyte Virtual Machine*).

Los requisitos mínimos de hardware que contempla CLDC son:

- 160KB de memoria disponible para *Java*.
- Procesador de 16 bits.
- Consumo bajo de batería.
- Conexión a red.

En cuanto a los requisitos de memoria, según *CLDC*, los 160KB se utilizan de la siguiente forma:

- 128KB de memoria no volátil para la máquina virtual de Java y las librerías del API de CLDC.
- 32KB de memoria volátil para el sistema de ejecución (Java Runtime System)

Las principales limitaciones impuestas por *CLDC* son:

- No proporciona soporte para operaciones de punto flotante.
- El método `Object.finalize` fue eliminado.
- Tiene limitado el manejo de excepciones.
- La limitación de memoria de los dispositivos.

- No se encuentran implementadas todas las clases que forman parte de la *J2SE*.

La seguridad dentro de *CLDC* es sencilla y sigue el famoso modelo de *caja de arena* (*sandbox*). Las líneas básicas de dicho modelo en *CLDC* son:

- Los archivos de clases deben ser verificados como aplicaciones válidas.
- Sólo las *API* predefinidas dentro de *CLDC* están disponibles.
- No se permite cargadores de clases definidos por el usuario.
- Sólo las capacidades nativas proporcionadas por *CLDC* son accesibles.

La configuración *CDC* (*Connected Device Configuration*) y los elementos relacionados con la misma, están diseñados para trabajar sobre *PDA* avanzadas, terminales de televisión, así como para sistemas embebidos avanzados. Sigue siendo limitada, pero ya utiliza una máquina virtual estándar.

**Perfiles**

En la arquitectura de *J2ME*, por encima de la configuración, tenemos el llamado *perfil* (*profile*). El perfil es un grupo más específico de *API* desde el punto de vista del dispositivo. Es decir, una configuración se ajusta a una familia de dispositivos mas o menos amplia, mientras que el perfil se orienta hacia cierto grupo de dispositivos dentro de la configuración a la que pertenece. Un perfil añade funcionalidades adicionales a las proporcionadas por la configuración.

Sobre la configuración *CDC* tenemos el perfil *FP* (*Fundation Profile*) que constituye el de más bajo nivel; el *PBF* (*Personal Basis Profile*) la cual permite diseñar interfaces gráficas, y el *PP* (*Personal Profile*) que está diseñada para generar interfaces de usuario mucho más sofisticadas (contiene una versión completa de *AWT* o *Abstract Windowing Toolkit*, la API de Java para manejo de ventanas).

Sobre la configuración *CLDC* se presenta la especificación *MIDP* (*Mobile Information Device Profile*) diseñada para trabajar sobre teléfonos celulares y *PDA*. Dentro de la especificación, se define a un dispositivo *MIDP* como un dispositivo pequeño de recursos limitados, móvil y con una conexión inalámbrica. Es decir, las funcionalidades que provee hacen referencia a conexiones de red limitadas en velocidad y estabilidad e interfaz de usuario reducida.

A continuación se presenta una imagen (figura 1.2) que muestra la arquitectura funcional de alto nivel que provee MIDP 2.0.

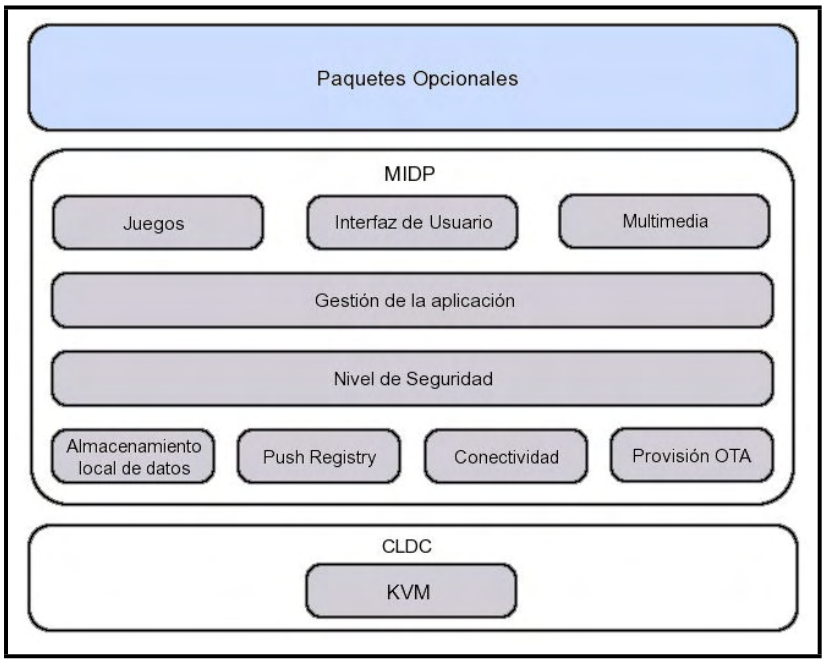



Figura 1.2: MIDP 2.0

**MIDLet**

Las aplicaciones *J2ME* desarrolladas bajo la especificación de *MIDP* se denominan *MIDLets*. Las clases de un *MIDLet* son almacenadas en el formato definido por los bytecodes de *Java*, dentro de un archivo `.class`. Estas clases deben ser verificadas antes de su puesta en marcha, para garantizar que no realizan ninguna operación no permitida una vez que la aplicación está funcionando dentro del dispositivo. Esta preverificación debe hacerse antes de instalarse en el dispositivo ya que la *KVM* del mismo no tiene la capacidad suficiente para hacerlo en tiempo de ejecución.

Los *MIDLets*, al igual que los *applets* [1], son empaquetados en archivos `.jar`, pero además necesitan datos extra para la puesta en marcha de la aplicación. Esta información se almacena en un archivo de definición o manifiesto que va incluido

[1] aplicación que se ejecuta en el contexto de otro programa o contenedor, por ejemplo un navegador web

dentro del `.jar` y en otro archivo descriptor con extensión `.jad`. Típicamente un archivo `.jar` se compondrá de:

- Clases del *MIDLet*
- Clases de soporte
- Recursos (imágenes, sonidos...)
- Manifiesto (archivo `.mf`)

Cuando un archivo `.jar` contiene muchos *MIDLets* se le denomina *MIDLet Suite* y permite compartir recursos (imágenes, sonidos...) entre los mismos, por lo que optimiza los recursos del dispositivo.

#### Paquetes opcionales

Los paquetes opcionales corresponden a implementaciones de las llamadas *JSR* (*Java Specification Requests*) para ciertos modelos y marcas de dispositivos. Entre ellos se encuentran paquetes que en la actualidad se utilizan para el desarrollo de aplicaciones gráficas como la *JSR-184 – Mobile 3D Graphics API for J2ME* – que permite dibujar en tercera dimensión.

Cabe mencionar que aunque muchos de los paquetes sólo funcionan para ciertos modelos y marcas de dispositivos, están teniendo una gran aceptación en el mercado. Cada día salen a la venta más aparatos que presentan dicha funcionalidad, por lo que se puede asegurar que en un futuro no muy lejano también lleguen a ser parte del estándar de *MIDP*.

### 1.1.2. BREW

*BREW* es una plataforma de desarrollo de aplicaciones para teléfonos celulares desarrollada por la compañía *Qualcomm*. *BREW* es un acrónimo que significa: *Binary Runtime Environment for Wireless*. Aunque en un principio fue desarrollada para celulares de tecnología *CDMA* (*Code division multiple access*) hoy en día ya es soportada por teléfonos de tipo *GSM*(*Global System for Mobile Communications (Groupe Spécial Mobile)*), *GPRS*(*General Packet Radio Service*) y *UMTS*(*Universal Mobile Telecommunications System*).

Desde el punto de vista de un desarrollador de software, *BREW* puede ser entendida primero, como una plataforma o serie de *API* para desarrollar aplicaciones en teléfonos celulares; pero también es una forma de vender y entregar productos a los clientes.

Dentro de los celulares, *BREW* esta construido entre la aplicación desarrollada y el chip de sistema operativo propio de cada modelo de teléfono o *ASIC* – *Application Specific Integrated Circuit* – por lo que los desarrolladores no tienen porque preocuparse más que de que el modelo del dispositivo sea compatible con *BREW*.

*BREW* utiliza como lenguaje de programación *C* y *C++* que al estar directamente compilado a código nativo es mucho más veloz que su contraparte de *J2ME* y por lo tanto es una plataforma que se ha hecho popular entre las grandes compañías desarrolladoras de aplicaciones gráficas interactivas como son *EA* (*Electronic Arts*) y *Gameloft* en sus ramas dedicadas al los dispositivos móviles.

Del sitio propio de *BREW*, se puede obtener de manera gratuita el *BREW SDK*, el cual se anexa al *Microsoft Visual Studio* para poder comenzar a desarrollar. Cabe mencionar que a pesar de que el *SDK* viene con un emulador, los programas que se prueban en éste no se ligan a código nativo de celular sino a código de computadora personal. De esta manera, la única forma para hacer una depuración completa y adecuada es adquiriendo un compilador específico para el microprocesador del celular e instalar la aplicación físicamente en un teléfono compatible. Es aquí donde se presenta entonces la mayor dificultad en el uso *BREW*, al menos para personas de recursos económicos limitados.

Para poder instalar una aplicación *BREW* en un celular es necesario que la misma este *firmada* de lo contrario será automáticamente borrada. Y los únicos autorizados para hacer estas *firmas digitales* son la gente que tengan un contrato con *BREW* para obtener opciones a firmas, el cual cuesta, en su versión más económica, aproximadamente cuatrocientos dólares, con un límite de cien firmas o un año de pruebas (lo que suceda primero). Este contrato también garantiza la entrada al *BREW Developer Extranet* donde ya se puede tener acceso a herramientas para generar un *ID* específico de nuestra aplicación (*BREW ClassID Generator*); el programa que genera las firmas válidas (*BREW TestSig Generator*) y la aplicación para instalar el programa en el celular (*BREW AppLoader*).

Pero los costos no se detienen ahí. Como se mencionó anteriormente, el *SDK* gratuito que se une al *Visual Studio*, sólo compila para procesadores *x86* y compatibles.

Si en realidad se quiere compilar a código fuente del celular, es necesario comprar un compilador para *ARM* (el procesador que utilizan los celulares compatibles con *BREW*) el cual cuesta alrededor de mil quinientos dólares en su licencia individual por un año.

Además, si el usuario desea que su aplicación sea distribuida para venta, es necesario que pase por el *True BREW Testing* (el cual también tiene un costo), donde la propia *Qualcomm* comprueba la *fiabilidad* del producto, para así comenzar a distribuirlo entre sus clientes a través del *BDS* (*Brew Distribution System*) garantizando la expansión del mismo.

Estas restricciones económicas en cuanto a las pruebas y distribución del producto, son ventajosas para las grandes compañías ya que las empresas pequeñas no representan competencia, pues carecen de los recursos necesarios para utilizar dicha tecnología. De la misma manera dejan fuera a cualquier grupo o entusiasta que desee aventurarse a desarrollar aplicaciones sobre BREW sólo por pasatiempo o simple curiosidad.

Estas mismas razones fueron las que se tomaron en cuenta para no realizar el presente trabajo de tesis en la plataforma *BREW*, sino en *J2ME*, la cual es mucho más amigable y abierta hacia todo tipo de desarrolladores, tanto a nivel comercial, como educativos y hasta de pasatiempo.

### 1.1.3. Symbian

*Symbian* es un sistema operativo abierto para móviles por lo que cualquier manufacturador puede licenciarlo. *Symbian* fue desarrollado por *Symbian Ltd*, un consorcio de empresas desarrolladoras de teléfonos celulares que incluye a *Motorola*, *Nokia*, *Panasonic* y *Sony-Ericsson*. *Symbian* es actualmente soportado por un amplio rango de teléfonos celulares gracias a que su licenciamiento es relativamente sencillo.

Para *Symbian* se pueden desarrollar aplicaciones en varios lenguajes que incluyen *C++*, *Java* y *Visual Basic*. La ventaja de usar *C++* es que permite que éstas sean compiladas a código nativo, lo que las convierte en aplicaciones más rápidas y mejor integradas al sistema operativo.

El problema con *Symbian* radica en su portabilidad ya que, al ser sólo un tipo de sistema operativo, desarrollar para él limita a las aplicaciones a correr sólo en

dispositivos que usen Symbian; a diferencia de *J2ME* que está incluido prácticamente en todos los dispositivos móviles actuales.

### 1.1.4. Windows Mobile

*Microsoft* ha desarrollado para los dispositivos móviles una versión de su sistema operativo *Windows compactada* llamada *Windows Mobile*, la cual se encuentra instalada en las *PocketPC* y los *Smartphones*.

Gracias a la tecnología de desarrollo *.Net* de *Microsoft* en su versión *Compact Framework*, se pueden desarrollar aplicaciones para estos dispositivos usando *C++*, *Visual Basic* y *C#* casi como si se tratara de una aplicación para *PC*; haciendo uso de herramientas como *Visual Studio*.

El problema radica en que los *Smartphones* con *Windows Mobile*, al igual que los que usan *Symbian*, sólo constituyen un pequeño grupo entre la amplia variedad de dispositivos móviles que actualmente hay en el mercado, sin olvidar que también son de los más costosos.

## 1.2. Diferencias entre dispositivos móviles y computadoras personales

En nuestros días, la tecnología avanza tan rápidamente que los componentes electrónicos van reduciendo su tamaño y costo dramáticamente. El poder de las computadoras personales de la década pasada, podría decirse que es equivalente al de los dispositivos móviles actuales, con la ventaja de que el tamaño de estos últimos, no rebasa la palma de la mano, además de que continúan haciéndose más pequeños cada vez. Y aunque es claro que muestran ya bastante poderío computacional, sería un tanto absurdo imaginar que los dispositivos móviles se podrían programar de la misma manera que una computadora personal actual, ya que existen bastantes diferencias que no sólo se limitan a los recursos de memoria o procesador.

Si se va desarrollar una aplicación para dispositivos móviles se tienen que tener bien claras las diferencias que existen con respecto a las computadoras personales para evitar errores en el diseño e implementación, por lo que a continuación se presenta una lista con las principales diferencias que podemos encontrar:

1. **Procesadores (*CPU*)**. Actualmente las computadoras manejan procesadores principales de hasta 3.0 Ghz y de uno hasta cuatro núcleos. En cambio los dispositivos móviles llegan a lo mucho a los 600 Mhz (las *PDA* más poderosas), pero en general se mantienen en un promedio bajo (a penas sobrepasando los 100 Mhz).

2. **Procesadores gráficos (*GPU*)**. Las computadoras modernas tienen tarjetas de video con *GPU* sumamente poderosos (*NVIDIA 8800 GTX*), además de poseer tecnología que permite el uso de varios *GPU* funcionando en paralelo (sin olvidar que el nuevo sistema operativo de *Microsoft*, *Windows Vista*, exige que la computadora tenga una tarjeta para procesamiento de gráficos). Pero los *GPU* en los dispositivos móviles actuales son la excepción a la regla, pues aunque sí existen, todavía es costoso incluirlos.

3. **Memoria disponible**. En la actualidad, es normal que las computadoras tengan arriba de un gigabyte de memoria principal (*RAM*) y cientos de gigabytes de memoria secundaria. Los móviles en cambio llegan a tener en promedio a penas unos pocos megabytes de *RAM* y unos cuantos cientos de megabytes de almacenamiento secundario.

4. **Resolución de pantalla**. Este aspecto muestra una de las razones del porqué los móviles pueden ser muy diferentes a una computadora de escritorio. Con cientos de configuraciones, los móviles tienen resoluciones de pantalla que van desde los 128x128 píxeles hasta 800x480, pasando por un sin fin de resoluciones intermedias no estandarizadas (ni siquiera en su *proporción de aspecto*). En cambio las computadoras de escritorio actuales manejan mucho mayor cantidad de píxeles en pantalla, que van desde 1024x728 hasta 1600x1200 en promedio, manteniendo su proporción de aspecto ya sea en 16:9 o 4:3.

5. **Sistemas operativos**. Las computadoras de escritorio tienen a su disposición diferentes tipos de sistemas operativos, pero en general pueden estar bien acotados ya que corren en arquitecturas muy semejantes y hasta podríamos decir que tienen dividido el mercado en usuarios de *Microsoft Windows*, de *MacOS* y de *Linux*. En cambio, los dispositivos móviles corren una gran variedad de sistemas operativos que van cambiando entre cada compañía y hasta en modelos que son de la misma marca; pueden compartir funcionalidad, pero en esencia son particulares de cada dispositivo.

6. **Ancho de Banda de Red**. Hoy en día, estar conectado a la gran red de información es parte de la vida diaria y, en ciertos sectores, hasta se ha convertido en indispensable. Las computadoras de escritorio ya cuentan como parte de su configuración básica, posibilidad para conectarse en red, que en estos momentos de forma local, asciende hasta el Gigabit de velocidad. En cambio, sólo los dispositivos móviles más avanzados (*Pocket PC*) traen conexión *wi-fi* cuya velocidad máxima es de 54Mbit/s; pero, en general, la tecnología de más uso actualmente (*GSM*), usa transferencias de apenas unos cuantos Kbits/s. La tecnología en expansión *3G*, promete velocidades mayores (de hasta 3Mbits/s) pero como se puede observar, no se compara a las velocidades de transferencias entre computadoras de escritorio, aunque ya es equiparable a la velocidad estándar de Internet de alta velocidad en América (1Mbit/s).

## 1.3. Motores Gráficos

**Definición 1** *Un motor o engine gráfico es el* ***componente central*** *de una* ***aplicación interactiva*** *y de* ***eventos concurrentes*** *que dibuja* ***gráficos en tiempo real*** [2]*. Provee las tecnologías subyacentes, simplifica el desarrollo y muchas veces permite que la aplicación sea ejecutada en diferentes plataformas. En general un engine nos proporciona una serie de herramientas de desarrollo y componentes de software reutilizables. También son conocidos como* ***librerías de programación*** *que proporcionan la funcionalidad básica para aplicaciones gráficas de manera flexible y sencilla, ayudando a lograr un desarrollo mucho más veloz y eficiente.*

### 1.3.1. Tipos de Gráficas

La graficación por computadora es el campo de la informática visual, donde se utiliza a las computadoras tanto para generar imágenes visuales sintéticamente, como para integrar o cambiar la información visual y espacial probada del mundo real. Se considera pionero en el campo al Dr. *Ivan Sutherland* por su tesis de doctorado *Sketch Path*, en el año de 1962.

Por los elementos necesarios para crear una imagen, las gráficas por computadora se pueden divider en dos grandes grupos: *los gráficos en dos dimensiones* y los de *tres dimensiones*. Por su forma de almacenar la información gráfica, los gráficos por computadora también pueden dividirse: *modelos vectoriales* y *modelos descriptivos*.

***Gráficos en dos dimensiones.*** Los gráficos 2D consisten en la generación por computadora de imágenes digitales basándose en su mayoría, en modelos también de dos dimensiones como serían: líneas, polígonos, texto e imágenes digitalizadas. Estas gráficas se utilizan principalmente en aplicaciones que fueron originalmente desarrolladas usando técnicas tradicionales de impresión y dibujo tales como: tipografía, cartografía, visualización de datos bidimensionales (histogramas, gráficas de pastel y barras), además de la gran parte de las interfaces de usuario actuales.

[2] Se considera **tiempo real** cuando no se alcanzan a percibir retrasos en el dibujo de las imágenes generadas sobre el dispositivo de despliegue

***Gráficos en tres dimensiones.*** Como su nombre lo indica, son imágenes generadas por computadora de elementos en tres dimensiones. En vez de que la computadora almacene la información sobre puntos, líneas y curvas de un plano bidimensional, la computadora guarda la posición de puntos, líneas y típicas caras (para construir un polígono) de un espacio de tres dimensiones. Las figuras tridimensionales son la base de prácticamente todos los gráficos 3D realizados en computadora. Por consiguiente, la mayoría de los motores de gráficos 3D están basados en el almacenaje de vértices (por medio de 3 simples coordenadas dimensionales X,Y,Z); líneas que conectan aquellos grupos de puntos, las caras que son definidas por las líneas, y luego una secuencia de caras que crean las figuras o modelos tridimensionales. También incluyen todo tipo de operaciones que se puedan realizar sobre dichos vértices y caras, como son: rotaciones, traslaciones, iluminación y texturizado.

***Modelos vectoriales.*** Las gráficas vectoriales son aquellas que usan primitivas geométricas tales como puntos, líneas, curvas y polígonos para representar imágenes. Por lo anterior pueden ser escaladas fácilmente sin perder calidad. Se usan comúnmente en el dibujo de texto (fuentes), logotipos y diagramas.

***Modelos descriptivos.*** Los modelos descriptivos pueden ser de dos tipos: los basados en vértices o mallas y los basados en píxels (llamados comúnmente mapas de bits o *Bitmaps*). Estos últimos están constituidos por una rejilla uniforme de píxeles, o puntos de color. Cada píxel tiene un valor específico como por ejemplo color, brillo, transparencia en color o una combinación de tales valores. Un mapa de bits está caracterizado por la cantidad de píxeles que tiene a lo largo y ancho de la imagen (*resolución*) y por la cantidad de bits que se requieren para guardar la información de un píxel (*profundidad de color*). Son los más indicados a utilizar cuando se intentan obtener imágenes fotorealistas. Los modelos basados en mallas contienen una descripción extensiva de todos los vértices, reglas de unión, caras, normales, texturas utilizadas para poder reproducir un modelo.

Los modelos descriptivos se diferencian de los vectoriales en que en lugar de usar curvas, polígonos y geometrías para generar el modelo, utilizan información punto por punto según la resolución del gráfico. Lo anterior provoca que se requiera de mucha memoria para guardar la información de un modelo de alta calidad. Además, los modelos descriptivos no pueden ser escalados a una mayor o menor resolución sin que se pierda calidad en el mismo.

### 1.3.2. Ejemplos de Motores Gráficos

Gracias a la creciente industria de los videojuegos, los motores más conocidos del mercado están relacionados directamente con el desarrollo de este medio de entretenimiento. Es más, los motores para juegos actuales son de las aplicaciones más complejas que hay, pues figuran docenas de finos sistemas que interactúan entre sí para lograr una mejor experiencia al usuario. Entre las principales funcionalidades que cubren están: el render [3] de gráficas 2D y 3D, física, detección de colisiones, sonido, scripting, animación, inteligencia artificial, manejo de redes y envío de datos, manejo de memoria, manejo de multiprocesos e hilos, manejo de dispositivos de entrada/salida y optimización para diferentes plataformas.

Entre los motores comerciales para gráficas y juegos en 3D más conocidos están:

- Torque
- Unreal Engine
- CryENGINE
- RenderWare
- Id Tech 4

También existen engines 3D de código abierto como son:

- Ogre3D
- Crystal Space
- Nebula

En cuanto a motores 2D existen varios pero ya no tan difundidos por la proliferación de gráficos en 3D. Como ejemplo tenemos a *Torque2D*, *CRM32Pro*, *FIFEngine*.

En el caso de motores gráficos para dispositivos móviles, tenemos a:

[3] dibujar

- ***Herocraft Hitech Mobile Dragon.*** Está escrito en *C++* con soporte a *OpenGL ES*. Por lo anterior sólo corre en *Microsoft Windows Mobile* para *PocketPC* y *Smartphones*, *Symbian* y *PalmOS*. Implementa tanto gráficos 2D como 3D.

- ***Ideaworks3D Airplay 3.0.*** También escrito en *C++*, es un motor para gráficas 3D en dispositivos móviles. Puede ser utilizado en dispositivos con *BREW*, *Microsoft Windows Mobile*, *Symbian* y *PalmOS*.

- ***In-fusio EGE.*** Escrito en *J2ME*, provee un motor de gráficas 3D con posibilidad de juego multijugador. Por estar escrito en *Java* se puede utilizar en una gran cantidad de dispositivos móviles.

### 1.3.3. Áreas de aplicación

Hoy en día existen múltiples aplicaciones que hacen uso de de las gráficas por computadora y que podrían tener como componente principal a un motor gráfico. Dichos casos se enumeran a continuación.

***Interfaces de usuario.*** Hoy en día prácticamente todas las plataformas, desde teléfonos celulares hasta estaciones de trabajo multiproceso, usan interfaces gráficas para manipular actividades de manera simultanea (usando ventanas o pantallas), además de seleccionar menús, íconos y objetos en pantalla. Teclear sólo se utiliza para introducir el texto que será almacenado o manipulado; o como comandos de acción rápida que no usan más de dos o tres teclas. Esto también incluye implícitamente el manejo de los dispositivos de entrada del usuario y los manejadores de evento para las acciones que éste realice.

***Visualización de Datos.*** Otro de los usos comunes de las gráficas por computadora hoy en día es graficar funciones matemáticas, físicas y económicas en 2D o 3D; crear histogramas, gráficas de barras y de pay; diagramas de planeación, inventarios y producción, etc. Todos estos son usados para presentar tendencias y patrones en los datos de manera más significativa y concisa, además de dar claridad a fenómenos complejos y facilitar una toma de decisiones informada.

***Diseño asistido por Computadora.*** En los procesos de diseño se hace un uso importante de las gráficas por computadora, en particular para sistemas de ingeniería y arquitectura, sin embargo, en la actualidad casi todos los productos se diseñan

usando una computadora. Los métodos de *CAD* (*Computer Assisted Design*), ahora se utilizan de forma rutinaria en el diseño de construcción de automóviles, aeronaves, embarcaciones, naves espaciales, computadoras, telas, edificios, circuitos eléctricos y chips, sistemas ópticos, redes tanto telefónicas como de computadoras y en un sin fin de disciplinas más. Pero cada día se busca con frecuencia, que el énfasis se centre en interactuar con el modelo computacional del componente o sistema para probarlo y optimizarlo de manera más sencilla y menos costosa (Figura 1.3(a)).

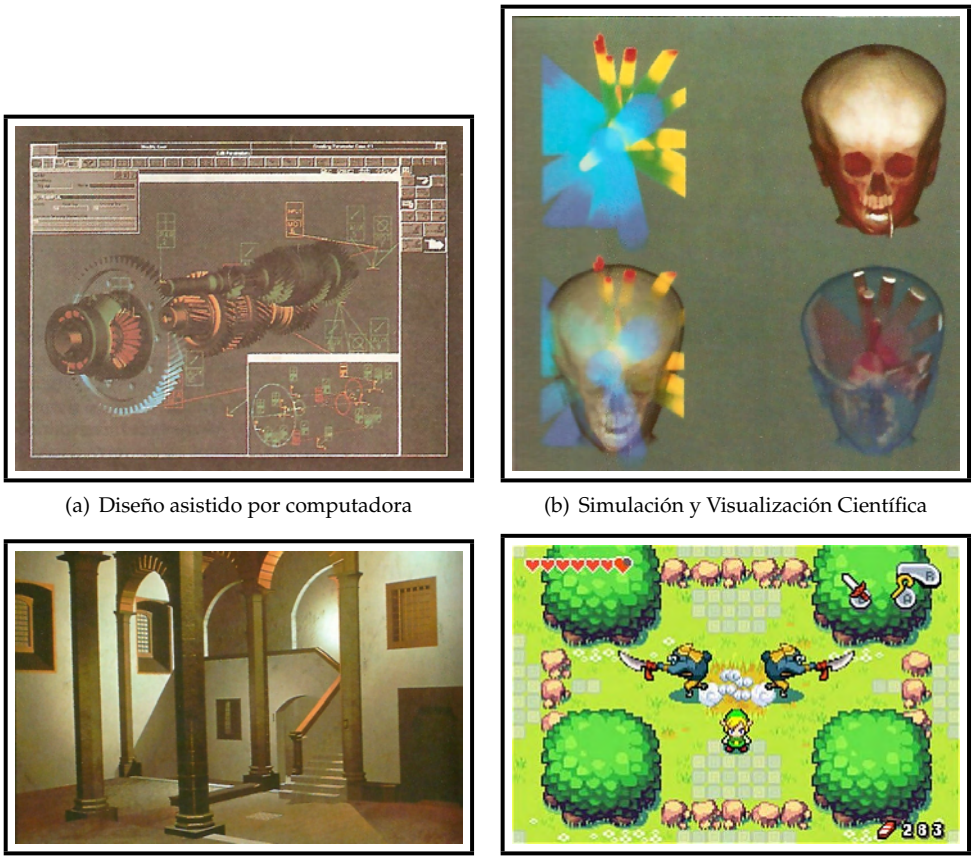
***Simulación y Visualización Científica.*** Científicos, ingenieros, personal médico, analistas comerciales y otros con frecuencia necesitan analizar grandes cantidades de información o estudiar el comportamiento de ciertos procesos. Las gráficas por computadora se pueden usar para mostrar simulaciones de fluidos, reacciones químicas y nucleares, fenómenos de relatividad, sistemas fisiológicos y funcionamiento de órganos, deformaciones de estructuras mecánicas sometidas a diferentes cargas, el flujo de aire sobre la superficie de una cápsula espacial, el modelado numérico de tormentas, modelado de proteínas y otras muchas más aplicaciones que hoy en día permiten a la comunidad científica estudiar más a fondo los fenómenos de su interés (Figura 1.3(b)).

***Arte por Computadora.*** Los métodos de gráficas por computadora se utilizan en forma generalizada tanto en aplicaciones de bellas artes como en aplicaciones de arte comercial. Los artistas utilizan una gran variedad de métodos computacionales, incluyendo hardware para propósitos especiales (*drawing tablets*), programas artísticos de brocha de pintar del artista (como *Lumena*), paquetes de pintura, software desarrollado de manera especial, paquetes de matemáticas simbólicas, paquetes *CAD*, software de edición electrónica de publicaciones y paquetes de animaciones que proporcionan los medios para diseñar formas de objetos y especificar movimientos de objetos (Figura 1.3(c)).

***Control de procesos.*** Mientras que los simuladores o los videojuegos permiten a los usuarios interactuar con la representación de un mundo real o imaginario, existen muchas otras aplicaciones que dejan al los usuarios interactuar con otros aspectos del mundo real. Pantallas de estado en refinerías, plantas de energía eléctrica, pozos petroleros y redes de computadoras muestran datos de sensores dispuestos en los componentes críticos del sistema, para que los operadores puedan responder en caso de problemas. Un ejemplo serían los radares de los

aeropuertos.

***Entretenimiento.*** (Figura 1.3(d)) En la actualidad se utilizan comúnmente métodos de gráficas por computadora para producir contenidos y aplicaciones de entretenimiento. Hoy día los productores, tanto de películas, programas de televisión y videos musicales, hacen uso de las técnicas de graficación por computadora para crear contenidos y efectos especiales. Las caricaturas de ahora difícilmente son dibujadas a mano como en antaño, sino que se animan directamente en computadora. Además, en la actualidad, los juegos que usan realidad virtual y los videojuegos son más comunes y de mayor uso entre toda la población.



(a) Diseño asistido por computadora

(b) Simulación y Visualización Científica

(c) Arte

(d) Entretenimiento

Figura 1.3: Usos de los motores gráficos

## 1.4. Elementos de un Motor Gráfico 2D

Todos los motores 2D que existen en el mercado actual están constituidos por elementos en común bien definidos que le dan al usuario una funcionalidad básica. Pueden subdividirse en varios módulos específicos enfocados en el render y dibujo, la lógica de la aplicación, la organización de la información, la interacción con el usuario y con otros programas; además de módulos de funcionalidad extra para aplicaciones específicas.

Se analizaron los motores 2D mencionados en la sección 1.3.2 y se concluyó que los principales elementos que presentan dichos motores son: el manejo de ventana y puerto de vista, manejo de capas, manejo y soporte de *bitmaps*, manejo de animaciones y *sprites*, manejo de texto, manejo de audio, manejo de dispositivos de entrada/salida, manejo de *tiles*, manejo de física, manejo de colisiones y manejo de la estructura lógica del programa. Todas estas funcionalidades son las que se buscarán implementar dentro de la arquitectura que esta tesis propone, cumpliendo así con el propósito de dar una herramienta poderosa para futuros desarrollos.

### 1.4.1. Manejo de ventana y puerto de vista

Podemos definir como *el mundo* al ambiente donde toda la aplicación gráfica se va a desarrollar. En términos prácticos constituye un rectángulo bien definido en anchura y altitud. Sus dimensiones están determinadas haciendo referencia a coordenadas mundiales.

Un área de coordenadas mundiales que se selecciona para desplegarla se llama *ventana*. Un área en un dispositivo de salida en el que se mapea una ventana se denomina *puerto de vista*. La *ventana* define *qué* se debe ver mientras que el *puerto de vista* define *dónde* se debe desplegar (figura 1.4). Una transformación de ventana a puerto de vista consiste en convertir de coordenadas mundiales a coordenadas del dispositivo.

El manejo de ventanas requiere implícitamente que se desarrollen algoritmos de *recorte* para definir los elementos que se encuentran dentro y fuera de las mismas. Un recorte permite la extracción de la parte de una escena definida para verla.

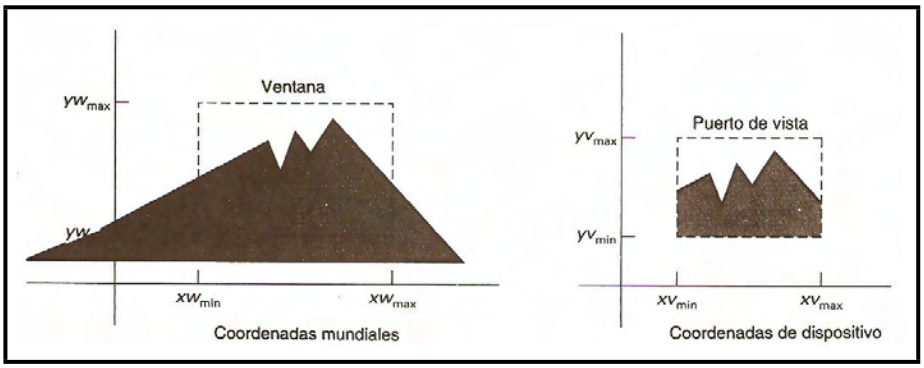



Figura 1.4: Ventana y Puerto de Vista

### 1.4.2. Manejo de capas o *layers*

Un *layer* o capa es un elemento gráfico que se encuentran a cierto nivel de profundidad relativa. La profundidad indica cuál elemento se ve más cercano al observador y cuáles otros se encuentran detrás de él. La profundidad, llamada también *orden en Z* (Z-order), indicará cual de estas capas se pinta primero (figura 1.5). Ejemplos de capas pueden ser: *sprites*, imágenes, texto y en general, cualquier elemento gráfico 2D.

Cada capa puede tener su propia posición dentro de la pantalla y la posibilidad de ser rotada, escalada y desplazada según sea necesario.

### 1.4.3. Manejo y soporte de *bitmaps*

Un bitmap es una imagen rectangular definida por cierto número de píxeles. Cada píxel tiene su propia posición y características, como brillo y color, las cuales son guardadas con cierto formato según el tipo de archivo de mapa de bits que se esté utilizando.

Actualmente existen varios formatos de mapas de bits, los cuales pueden tener compresión o no. En general, un motor presenta soporte para varios tipos de imágenes siendo entre los más comunes: *png*, *gif*, *jpeg*, *bmp*, *raw* y *tiff*.

### 1.4.4. Manejo de animaciones y *Sprites*

Una animación es una serie de imágenes dibujadas en secuencia con un retraso entre cada una. Cada imagen o cuadro es ligeramente diferente al anterior por lo que,

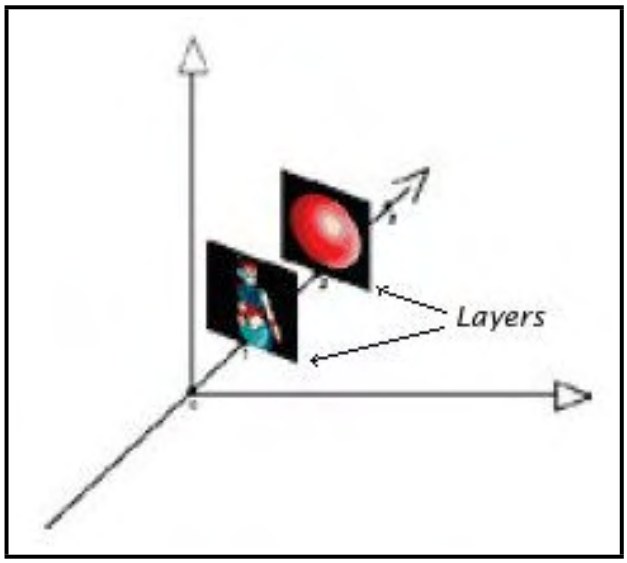



Figura 1.5: Capas o Layers

cuando son reproducidos uno en sucesión del otro, dan la apariencia de movimiento.

Para este trabajo, los *sprites* son elementos gráficos bidimensionales que tienen una animación asociada. Por lo anterior desempeñan un papel principal dentro de las aplicaciones gráficas en 2D. En general representan cualquier elemento con movimiento de la aplicación.

Los *sprites* están formados por un conjunto de imágenes que constituyen los cuadros de animación del mismo (figura 1.6). Además tienen características muy particulares como son su cuadro actual, su velocidad y su dirección.

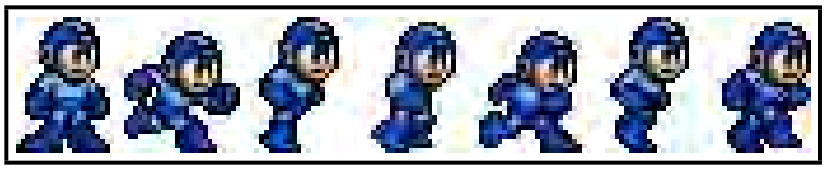


Figura 1.6: Cuadros de un *Sprite*

### 1.4.5. Manejo de texto

En general, todas las aplicaciones gráficas necesitan dibujar en pantalla texto. Por lo anterior, los motores gráficos deben proporcionar funcionalidad para dibujarlo, escalarlo, rotarlo, cambiarle el color, la orientación, el tipo de letra, además todas aquellas acciones que tengan que ver con su manipulación.

### 1.4.6. Manejo de audio

Cuando se trata de aplicaciones gráficas enfocadas al entretenimiento como son los videojuegos, el audio y la música representan un papel muy importante para mejorar la experiencia del usuario. Hoy en día existen varias tecnologías de manejo de audio, que van desde sonidos de un solo canal, hasta el audio en 3 dimensiones – sonido envolvente o *surround* – de hasta 8 canales. Además, el sonido digital puede presentarse en modalidades de audio comprimido y sin comprimir, el cual se utiliza como audio interactivo cuando los sonidos son reproducidos al momento de que el usuario realiza alguna acción; o de ambientación, cuando ocurre algo en el *mundo* de la aplicación.

Una librería que pretenda dar funcionalidad para el desarrollo de aplicaciones interactivas, deber proporcionar herramientas adecuadas para manejar audio en diferentes formatos y modalidades para así facilitar y agilizar la implementación de software más rico y profesional.

Entre los formatos más comunes de audio tenemos al: *wav* (apócope de *Waveform audio format*, codificado en *PCM*[4] sin comprimir), al *MIDI* (*Musical Instrument Digital Interface*), *mp3* (audio comprimido MPEG-1/2 capa 3), *ogg Vorbis* (audio de alta compresión) y *wma* (audio comprimido de *Windows*).

### 1.4.7. Manejo de dispositivos de entrada

Todas las aplicaciones que sean interactivas, necesitan permitir que el usuario proporcione información a las mismas a través de los distintos dispositivos de entrada de que el aparato disponga. En el caso de los móviles, tenemos teclados y botones, pantallas táctiles y cojinetes de direcciones (*control pads*).

[4]El PCM se obtiene de digitalizar la señal de audio usando la técnica de modulación por impulsos codificados o *Pulse Code Modulation* por sus siglas en inglés

### 1.4.8. Manejo de colisiones

Consiste en cualquier método para determinar que dos elementos gráficos han *chocado* entre sí, es decir, cuando han interactuado físicamente entre ellos. Un *engine* gráfico proporciona, por lo menos, un método para detectar colisiones entre los diferentes objetos, siendo el más sencillo usar rectángulos alrededor de los elementos gráficos y verificar si éstos se traslapan entre sí (figura 1.7).

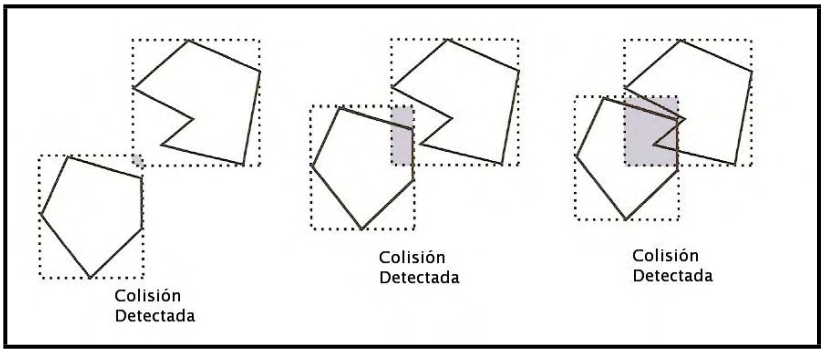



Figura 1.7: Colisión mediante rectángulos

Como se observa en la figura 1.7 este método no es muy preciso a menos que los elementos gráficos que se estén utilizando sean también rectangulares. Una forma de mejorar esta técnica es reducir los rectángulos de colisión un poco para reducir el error. Su problema radica en que se pueden presentar casos en que los objetos se sobreponen unos a otros y la colisión no es detectada. (Ver figura 1.8)

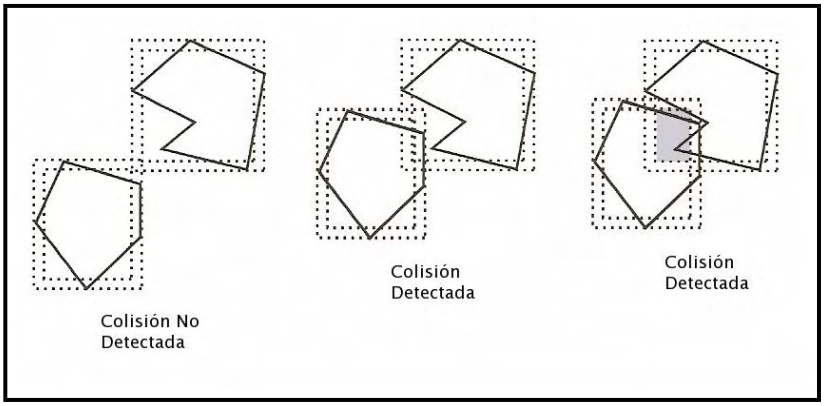



Figura 1.8: Colisión mediante rectángulos reducidos

La técnica de detección más precisa es entonces detectar la colisión basándose en la propia imagen del elemento gráfico, lo cual implica verificar a nivel de píxel si los elementos transparentes o de imagen en sí se sobreponen unos a otros. (Ver figura 1.9).

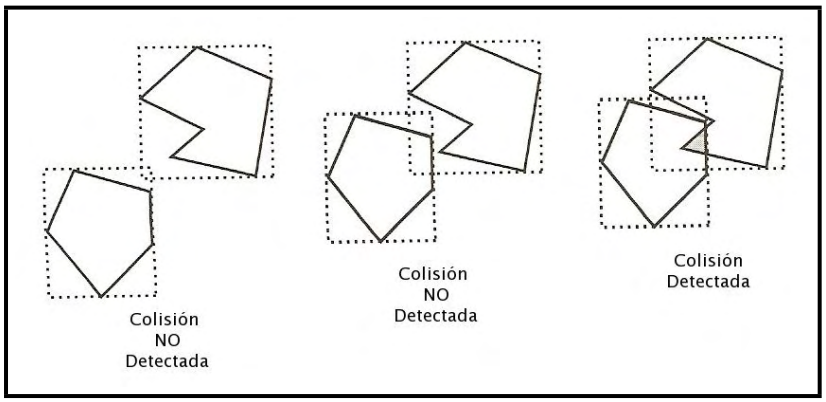



Figura 1.9: Colisión mediante verificación a nivel píxel

Desgraciadamente ésta técnica (figura 1.9) requiere de mayor tiempo de procesamiento que la detección por medio de rectángulos por lo que puede crear un cuello de botella en el rendimiento de la aplicación.

### 1.4.9. Manejo de *tiles*

Un *tile* es un mosaico compuesto por imágenes más pequeñas que al estar ordenadas de cierta manera, generan una imagen compuesta mucho más compleja. Es una técnica muy usada para generar fondos de manera dinámica y sin hacer uso de todos los recursos que se requerirían para guardar imágenes más grandes para cada caso en particular.

Como la figura 1.10 lo muestra, un conjunto pequeño de *tiles* pueden ser arreglados para construir imágenes mucho más complejas e interesantes. Por lo tanto, no es difícil imaginar que con un conjunto mayor de imágenes, es posible construir mosaicos mucho más ricos que pueden formar todo un mundo virtual en 2D.

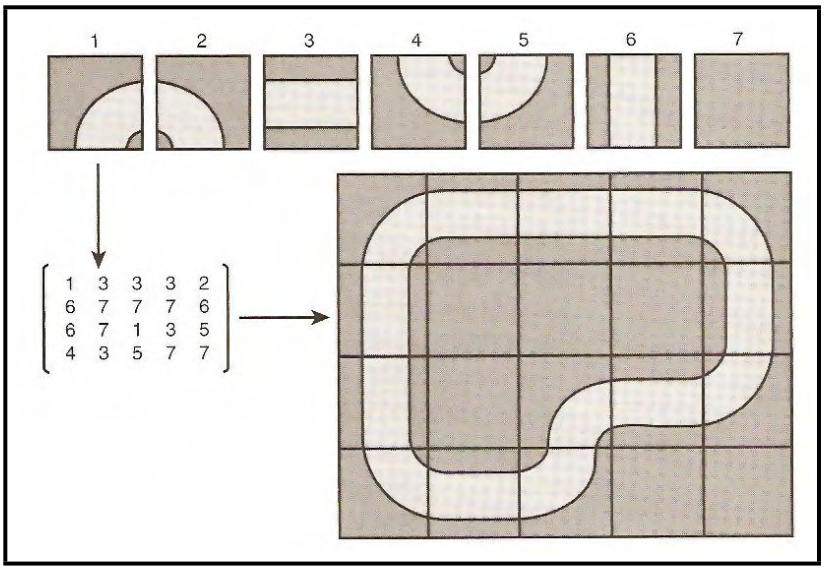



Figura 1.10: *Tiles*

### 1.4.10. Estructuras para manejo de la lógica del programa

Además de toda la funcionalidad gráfica, es común que los motores modernos proporcionen elementos para facilitar el desarrollo de las acciones de la aplicación, que no necesariamente, tienen que ver con desplegar imágenes en pantalla. Para esto proporcionan manejadores de eventos, máquinas de estado, soporte de *scripting*[5] y la implementación de algoritmos de inteligencia artificial, entre otros.

También los *engines* están estructurados de tal manera que permiten que el código fuente quede ordenado para que sus modificaciones y expansiones se realicen de manera sencilla. Esto se logra gracias a la estructura en módulos del propio motor.

### 1.4.11. Manejo de física

Cuando en las aplicaciones gráficas se buscan simular fenómenos reales, es necesario proporcionar herramientas que se dediquen a hacer los cálculos pertinentes referentes a velocidades, trayectorias, deformaciones, rebotes y demás acciones que se busquen representar.

[5]El *scripting* es un lenguaje de programación independiente que permite controlar y configurar al programa principal

Esto incluye implementar algoritmos matemáticos para el cálculo de senos y cosenos, operaciones con matrices, operaciones con vectores, operaciones con polinomios, potencias, raíces cuadradas y demás artilugios matemáticos que se usen en el cálculo de cualquier variable física que se vaya a representar

### 1.4.12. Manejo de comunicación vía red

Hoy en día la comunicación entre dispositivos es una de las acciones más comunes que muchas aplicaciones realizan, ya sea para compartir información o recursos de procesamiento. Las aplicaciones que permiten a los usuarios comunicarse con otros para transmitir mensajes, archivos o hasta jugar con otras personas se hacen cada vez más populares.

Esto no es la excepción para las aplicaciones que corren en dispositivos móviles por lo que, aunque estén limitadas de recursos, como se mencionó en el apartado 1.2, generalmente presentan opciones para comunicarse con otros dispositivos, ya sea por *wi-fi*, *bluetooth*, *infrarrojo* o las propias redes de datos de la telefonía celular (*GSM*, *3G*).

Por lo tanto, proporcionar los elementos básicos para establecer la comunicación entre dispositivos, se ha convertido en una parte importante de cualquier motor que pretenda dar una funcionalidad más completa.

# Capítulo 2

# Arquitectura del Sistema

En el capítulo 1 se definieron todos los conceptos básicos que involucran el diseño e implementación de un motor gráfico en 2D, partiendo desde las tecnologías para el desarrollo de aplicaciones en dispositivos móviles, hasta cada uno de los elementos que lo conforman. Todos estos elementos constituyen la guía de diseño e implementación de la arquitectura que se presenta en esta tesis.

A continuación se mostrarán las consideraciones de diseño que se tomaron en cuenta para la realización de éste motor gráfico, comenzando desde su concepción y continuando a través de todos sus diferentes módulos; explicando las funciones que éstos realizan y como interactúan entre cada uno de ellos.

## 2.1. Diseño General

Como se apuntó en la definición 1 un motor gráfico es el componente central de una aplicación interactiva y de eventos concurrentes que dibuja en tiempo real, por

lo que empezaremos describiendo a este tipo de aplicaciones, así como todas las consideraciones de diseño que se necesitan tomar en cuenta para su implementación.

Una aplicación interactiva permite al usuario en cualquier momento proporcionar información para que ésta modifique el estado del programa. Ésta información se puede recibir a través del teclado, ratón, pantalla táctil, botones, palancas y cojinetes de direcciones. Si dicha aplicación realiza distintas actividades al mismo tiempo, se dice que es de eventos concurrentes. Además, si a cada momento puede modificar su estado y actualizarse a sí misma, realizando los cálculos y acciones pertinentes en tiempo de ejecución, es una aplicación de tiempo real.

Para proporcionar la interactividad es necesario que la aplicación nos de soporte para los dispositivos de entrada que queramos utilizar, que en el caso de los dispositivos móviles, se limita a su teclado y pad de direcciones, pero además, en algunos casos, a su pantalla táctil.

Lo más complicado se presenta al momento de querer diseñar una aplicación de eventos concurrentes, ya que se requiere que se puedan ejecutar múltiples instrucciones al mismo tiempo. Lo ideal sería que se contara con un procesador independiente para cada instrucción ya que así la aplicación sería 100 % eficiente y sumamente veloz, pero la realidad es que sólo hasta hace apenas un par de años, los procesadores con múltiples núcleos se han venido masificando. Ahora las computadoras caseras ya pueden tener hasta cuatro núcleos en su procesador, permitiendo así manejar un multiproceso real. El problema es que los dispositivos móviles difícilmente cuentan con una arquitectura multinúcleo, por lo que tenemos que idear una forma de poder ejecutar varias tareas al mismo tiempo, o que al menos se simule que así es.

La manera más sencilla de simular esta concurrencia de actividades es actualizar, con una frecuencia que sea imperceptible para el usuario, la información que éste recibe a través de nuestra aplicación. Cada determinado tiempo se registrarán sus interacciones, se cambiará el estado interno del programa y se mandará pintar en pantalla el resultado de las anteriores acciones; creando la ilusión de que nuestra aplicación es concurrente, interactiva y se ejecuta en tiempo real. Usar este tipo de técnica produce la sensación de que actualiza sus eventos y acciones como si se tratara de una película cinematográfica o animación, la cual está compuesta de cuadros o *frames* que se van cambiando conforme el usuario interactúe con la aplicación.

De este parecido con los cuadros de una animación, surge el concepto de

cuadros por segundo o *framerate* de una aplicación gráfica; el cual se refiere al número de cuadros que se pueden pintar, atendiendo a la interacción del usuario y la propia lógica de la aplicación, en un segundo de tiempo. Estos cuadros pueden ser ligeramente diferentes a los anteriores para simular una animación.

En general, a un mayor *framerate*, se obtendrá una aplicación que se denote más fluida y de mejor respuesta que una con un *framerate* bajo. Los humanos, por ejemplo, podemos percibir una fluidez muy aceptable a los 60 cuadros por segundo, además de que muchas pantallas y monitores refrescan su pantalla a esa frecuencia, por lo que se considera el *framerate* óptimo para una aplicación gráfica.

Pero muchas veces, debido al poderío del dispositivo que este ejecutando la aplicación, obtener 60 cuadros por segundo no resulta factible, por lo que se opta en reducir el *framerate* a uno menor que no afecte la funcionalidad del programa. En estos casos se buscan frecuencias arriba de los 15 cuadros por segundo, ya que proporcionar un *framerate* menor provocaría una sensación de lentitud y poca fluidez en la aplicación.

Ahora bien, como la *interacción*, la *lógica* y el *pintado* son constantes que ocurren durante toda la ejecución del programa, los eventos concurrentes son controlados repitiendo el registro de la interacción del usuario, actualizando la lógica y mandando pintar dentro de un ciclo principal al que llamaremos: *Main Loop*. Este ciclo es el que se repetirá con una frecuencia equivalente al *framerate* y constituye el corazón de nuestra aplicación interactiva.

Además del *Main Loop*, el *engine* cuenta con otros módulos con funciones mucho más específicas a los cuales podemos clasificar en: el módulo de audio, el módulo de dibujo, el módulo de colisiones, el módulo de interacción de usuario, el módulo de lógica, el módulo de física, el módulo de comunicación vía red y los módulos de funcionalidad extra que se quieran agregar.

En la figura 2.1 se muestra el diagrama de bloques que representa a nuestro motor con los diferentes módulos implementados. Aquellos que se encuentran con línea punteada también forman parte del *engine*, pero no han sido diseñados a detalle, por lo que su implementación se deja para posteriores revisiones de este trabajo.

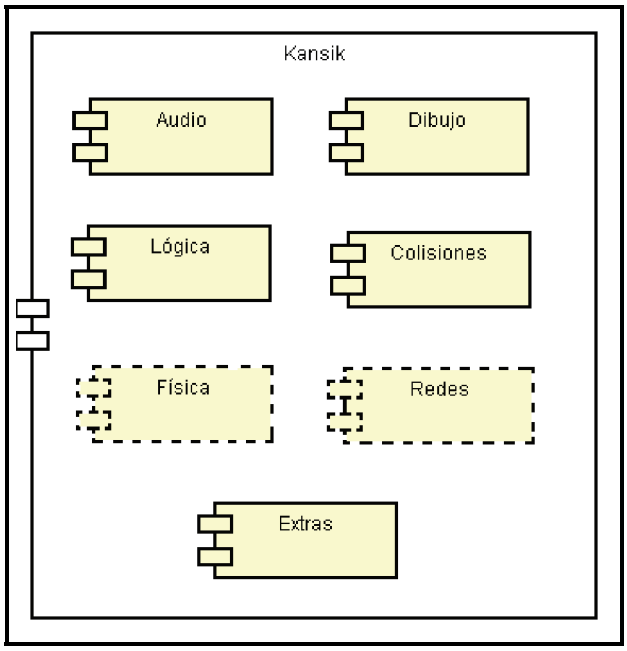



Figura 2.1: Los módulos de *Kansik*

## 2.2. El *Main Loop*

Como ya se mencionó, el *Main Loop* es el elemento principal de toda nuestra aplicación y sobre él se ejecutan la mayoría de las acciones que nuestro programa realiza. Los demás módulos que componen al *engine* se dedican entonces a complementar todas estas acciones y a agregar funcionalidad que, muy probablemente, también será ejecutada dentro de este ciclo principal.

En la figura 2.2 se muestra el diseño general del ciclo principal (*Main Loop*) en donde se presentan todas las acciones principales que deben realizarse dentro del mismo.

El primer paso consiste en *inicializar* todos los recursos que se van a usar dentro del ciclo y que robarían tiempo precioso si se cargarán dentro del *Main Loop*. En este momento corresponde inicializar los manejadores de audio y video, los dispositivos de entrada, los recursos de red y todo aquello que vayamos a utilizar dentro de nuestra aplicación. En general esta inicialización se realiza una sola vez cuando el programa empieza y no se vuelve a repetir. Un ejemplo de recursos que en este mo-

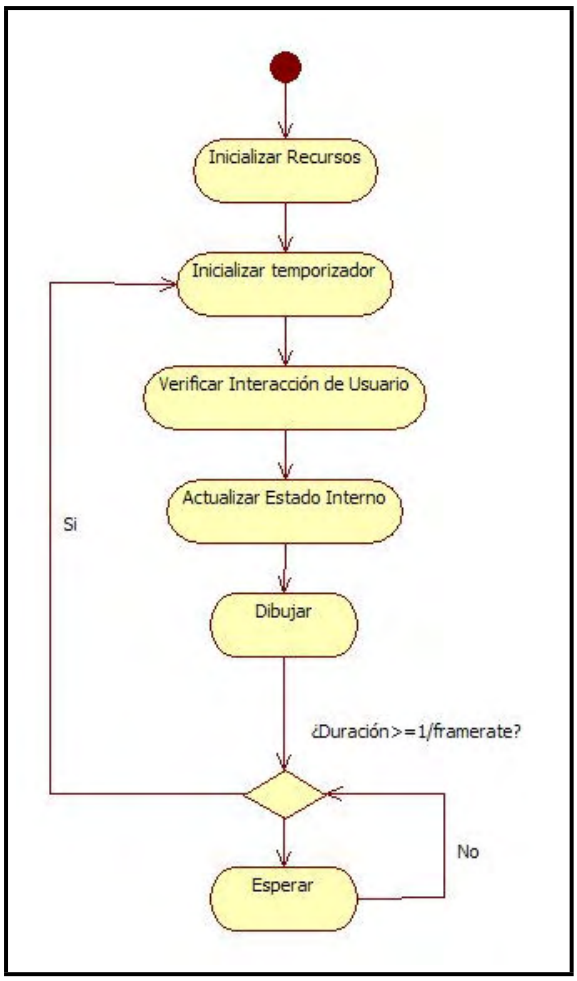



Figura 2.2: El *Main Loop*

mento se inicializan, correspondería al grupo de *drivers* encargados del audio, video, red y de los distintos dispositivos tanto de entrada como de salida que se vayan a usar; así como reservar la memoria que el programa utilizará.

Ya que tenemos todos los recursos que necesitaremos disponibles, podemos comenzar con el ciclo principal. En primer lugar tenemos que inicializar un contador de tiempo, esto con el fin de regular a un *framerate* fijo la ejecución del ciclo principal. Así, en caso de que el dispositivo pueda desplegar un mayor número de cuadros por segundo, todo se muestra a la misma velocidad. Para esto, al final del ciclo, se verificará si el tiempo transcurrido en segundos es mayor o igual al inverso del *framerate* y en caso contrario, se mantendrá en espera al ciclo.

Como se mencionó anteriormente, dentro del *Main Loop* se realizarán acciones que se repiten durante toda la ejecución del programa, las cuales son verificar la interacción del usuario, actualizar el estado interno de la aplicación y dibujar en el buffer de trabajo [1]. A continuación se describirán cada una de ellas y los papeles que representan dentro del programa.

***Interacción de Usuario.*** En este apartado es donde se revisan todos los dispositivos de entrada que la aplicación inicializó, para verificar los datos que el usuario haya proporcionado. Aquí se obtienen lecturas del teclado, las pantallas táctiles, los botones, los *pads* de direcciones y todas las demás entradas para así alimentar la aplicación con la interacción del exterior. En el caso de que se trate de un programa con interacción entre otros dispositivos, en este momento también se registran todos los datos de la red necesarios.

El leer los dispositivos que proporcionan datos externos, también implica reservar memoria para registrar dichas entradas y así poderlas utilizar dentro de la aplicación al momento de revisar si va ser necesario modificar su estado.

***Actualizado del Estado Interno*** Es aquí donde se ejecuta la lógica del programa y se verifican todos los comportamientos de cada uno de los objetos que forman la aplicación, tomando en cuenta las entradas del exterior registradas en el paso anterior.

A grandes rasgos, en este momento es cuando se ejecutan todos los cálculos de movimiento, cambios de estado, operaciones matemáticas y todas las acciones

[1] El buffer de trabajo es el área dentro de la memoria principal de la computadora designada a almacenar una copia de la imagen que esta siendo mostrada en pantalla actualmente

que el programa debe realizar en cada ciclo. En el caso particular de los videojuegos, es en este paso donde se verifican colisiones, se ejecuta la inteligencia artificial, la lógica del juego y se realizan todos los movimientos, rotaciones y traslaciones de los elementos que componen la escena.

***Render*** Ya que se actualizó el estado interno de la aplicación tomando en cuenta las entradas del usuario, dentro del *Main Loop* se procede a dibujar en el *buffer* de trabajo todos los elementos visuales de la aplicación, reflejando el nuevo estado interno del programa.

Es aquí donde se usan todos los algoritmos de gráficas para el render de imágenes, *sprites*, *tiles*, texto y demás gráficos 2D que el programa requiera. En este momento también se realizan los recortes y operaciones de ventana a puerto de vista; se aplican transparencias y se dibujan las capas en el *z-order* establecido.

## 2.3. Módulo de Lógica

Como se explicó en la sección anterior (2.2), dentro del *Main Loop* se requiere actualizar el estado interno de la aplicación y verificar el comportamiento de cada uno de los objetos que la componen. Para lograr esto se necesita una clase abstracta, a la que llamaremos ***Updatable***, que describa todas las acciones que una clase debe implementar para ser actualizada durante cada ciclo del *Main Loop*. En la figura B.1 se muestra el diagrama de clase de *Updatable*, junto con sus atributos y métodos.

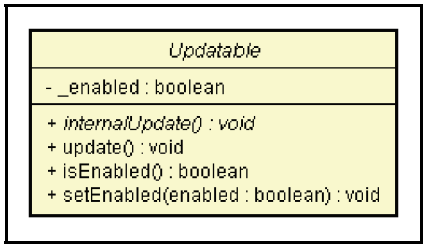



Figura 2.3: Diagrama de Clases,*Updatable*

Durante cada ciclo del *Main Loop* de nuestro motor, en la acción que corresponde a actualizar el estado interno del programa; el método `internalUpdate` de cada uno de los objetos que implementen la clase *Updatable* y que sean parte de la aplicación será ejecutado. De esta manera, es dentro de este método donde se debe implementar

la lógica propia y única de cada clase que herede de *Updatable*. La bandera `enabled` es la que indica si el contenido del método `InternalUpdate` debe ejecutarse o no, así que debe de tomarse en cuenta al momento de la implementación.

También tenemos el método `Update`, en el cual se deben definir las acciones que el desarrollador quiera proveer como funcionalidad extra. Es un método para separar la lógica propia del objeto y las acciones externas que el usuario de la clase desee agregarle para su ejecución en cada ciclo del *Main Loop*. `Update` debe ser llamado dentro del método `internalUpdate` para que sea ejecutado dentro del ciclo principal por lo que, al igual que la bandera `enabled`, debe tomarse en cuenta al momento de implementarse dicho `internalUpdate`.

Además de la clase abstracta *Updatable*, también se pueden crear otras clases para el manejo de la lógica común, como serían las máquinas de estado y métodos genéricos que se requieran ejecutar a los que denominaremos acciones. En la figura 2.5 se muestra el diagrama de clases de estos elementos.

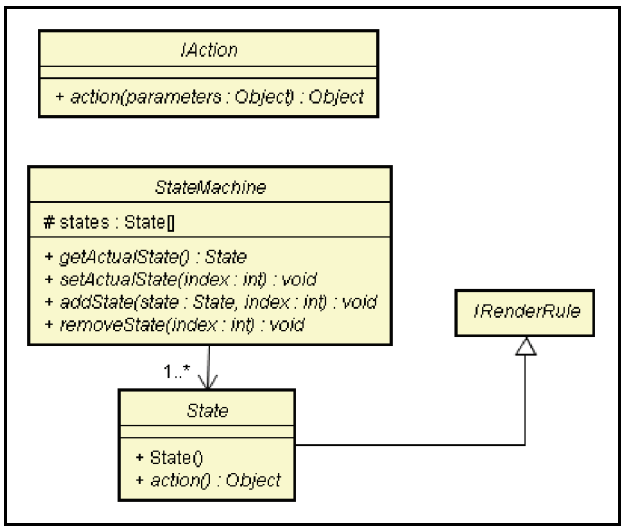



Figura 2.4: Diagrama de Clases, Paquete de Lógica

La máquina de estados, representada por la clase ***StateMachine***, estará formada por elementos que implementen y hereden de la clase ***State*** que representa a un estado de la aplicación. Cada vez que se necesite hacer un cambio de estado se hará a través del método `setActualState` que recibe como parámetro el identificador propio del estado; que por facilidad se sugiere que sea el índice del mismo dentro del

arreglo de estados que los contendrá en la clase *StateMachine*.

La clase *State* implementa a la interfaz ***IRenderRule*** ya que cada vez que la aplicación cambie de estado interno, muy probablemente las reglas para dibujarse también cambiarán y deben ser las propias del estado actual representado. Esta interfaz se estudia más adelante dentro del módulo de dibujo (Sección 2.4).

En cuanto a la interface ***IAction***, únicamente contiene un método que representará la acción a realizar. Esta clase se puede utilizar para mandar ejecutar una serie de acciones que algún objeto contenga y que estén implementadas por separado. Un ejemplo de esto se muestra más adelante cuando se describa la clase *Room* del apartado de módulos extra para juegos de rol (Ver sección 2.10). También la interfaz *IAction* debe proveer la posibilidad de ejecutarse en otro hilo o *thread*.

Por último tenemos a la clase abstracta ***LoadableContent*** (Ver figura **??**). Esta debe ser implementada por todas aquellas clases que requieran cargar elementos en memoria antes de ser utilizados y cuya carga requiere de tiempo y recursos que podrían considerarse extensos. La bandera `isLoaded` nos indica si el contenido ya esta en memoria y debe ser actualizada al momento de ejecutar el método `loadContent`, por lo que debe ser considerada al momento de su implementación.

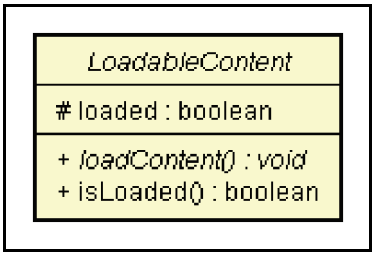



Figura 2.5: Diagrama de Clases, *LoadableContent*

## 2.4. Módulo de Dibujo

Como su nombre lo indica, este es el módulo encargado de dibujar en pantalla todos los elementos gráficos de la aplicación. En este módulo se implementan todos los algoritmos para el dibujo de gráficas en dos dimensiones, incluyendo recortes, transformaciones de ventana a puerto de vista, el manejo de capas, de texto, de imágenes, de *sprites*, de *tiles* y de dibujo de primitivas tales como líneas, puntos y polígonos con sus respectivos rellenos. En la figura 2.6 se muestra el diagrama de casos de uso para el diseño de éste módulo que incluyen todas las características mencionadas.

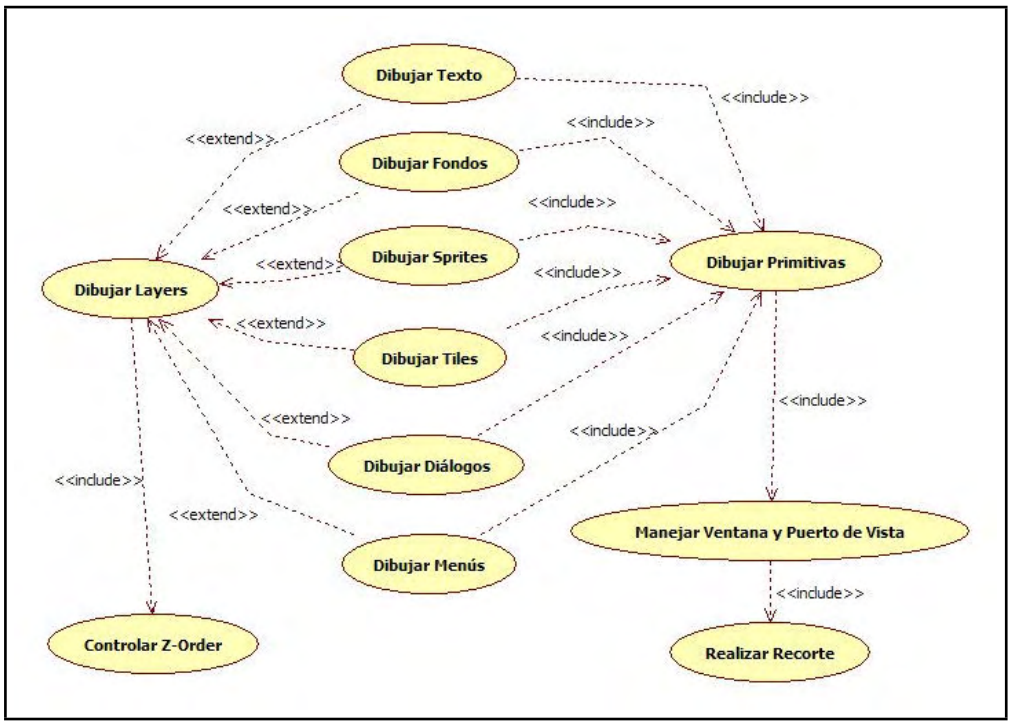



Figura 2.6: Diagrama de Casos de Uso, Módulo de Dibujo

Basándonos en la forma de dibujar primitivas en *J2ME*, se requerirá de una clase llamada ***Graphics*** para el dibujo de las primitivas más básicas como son: líneas, polígonos, rellenos, curvas, texto y archivos de imagen. Esta será la que esté en contacto directo con la memoria de video del dispositivo y será la encargada de escribir tanto en *buffers* secundarios, como en el de pantalla. También tendrá como función realizar los recortes y las transformaciones de ventana a puerto de vista.

Ahora bien, basándose en el paquete *Game* de la *J2ME* en su perfil *MIDP 2.0*, para el render de elementos más complejos se usarán las llamadas capas o *layers*. Un *layer* es un ente gráfico de la aplicación que puede ser trasladado y dibujado en el orden y posición que se le indique. En general, la capa engloba a un elemento gráfico que tiene su propia manera de dibujarse en pantalla y que implica el uso de varias primitivas de dibujo. Por lo anterior, un objeto *Graphics* siempre es utilizado por un *layer* para dibujarse a sí mismo.

A continuación en la figura 2.7 se muestra el diagrama de clase de ***Layer*** con sus principales métodos. Es una *clase abstracta* pues a partir de ella deben heredar todos los elementos gráficos que representen algún ente en particular.

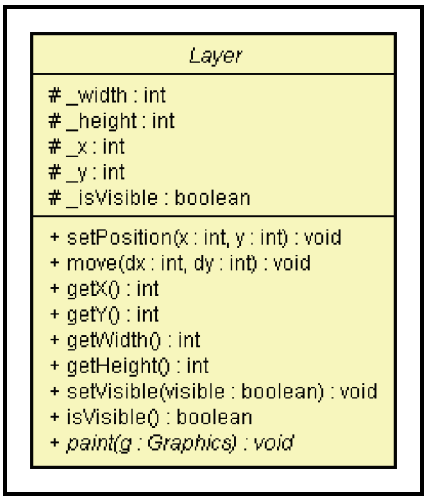



Figura 2.7: Diagrama de Clases, *Layer*

Para manejar una gran cantidad de *layers* crearemos una clase llamada ***LayerManager***, la cual también estará encargada de dibujarlos en el orden correcto. Esto implica que el *z-order* se manejará en esta clase. Además, el *LayerManager* tendrá su propia ventana y sistema coordenado, de manera que todas las capas dibujadas por él, estarán referenciadas con respecto al manejador y no a sus coordenadas absolutas. También, el *LayerManager* podrá tener su propia localización en pantalla, logrando así que todos los elementos gráficos que contiene se dibujen a partir de una posición determinada. En la figura 2.8 se puede observar el diagrama de clase del *LayerManager* con sus principales métodos.

Ahora bien, como se indicó en la sección 2.2 dentro del *Main Loop* deben

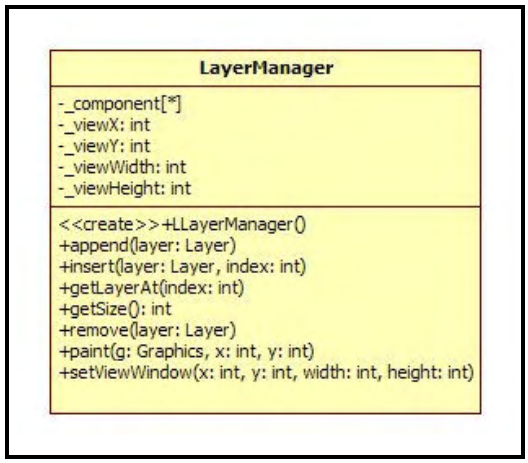



Figura 2.8: Diagrama de Clases, *LayerManager*

mandarse pintar todos los elementos gráficos que requieren aparecer en pantalla, los cuales obedecen a ciertas reglas de dibujo. Dichas reglas estarán definidas dentro de una clase que implemente a la interfaz llamada ***IRenderRule***, cuyo diagrama de clase se presenta en la figura 2.9.

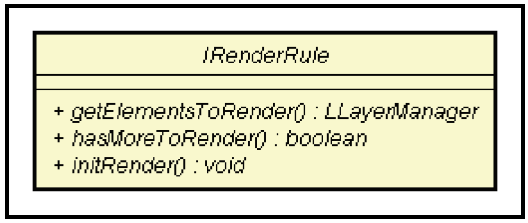



Figura 2.9: Diagrama de Clases, *IRenderRule*

*IRenderRule* pretende que se siga respetando el modelo de capas y por eso manda llamar al método `getElementsToRender` siempre y cuando el método `hasMoreToRender` regrese verdadero. Por lo anterior, al implementar un *IRenderRule*, se debe tener cuidado de que al terminar de proveer todos los elementos a dibujar, el método `hasMoreToRender` regrese falso; de lo contrario se podría entrar en un ciclo infinito que impediría que la aplicación saliera de la sección de render.

### 2.4.1. Elementos Gráficos Principales

Cómo ya se mencionó, a partir de la clase *Layer* se pretende crear las demás encargadas de dibujar todos los elementos gráficos necesarios. Los principales serían: el texto, los fondos y los *sprites*.

Para dibujar texto se tendría una clase ***Text***; para dibujar las imágenes de fondos se tendría la clase ***Background*** y para los *sprites* la clase ***Sprite***. En la figura 2.10 se muestra sus respectivos diagramas de clase.

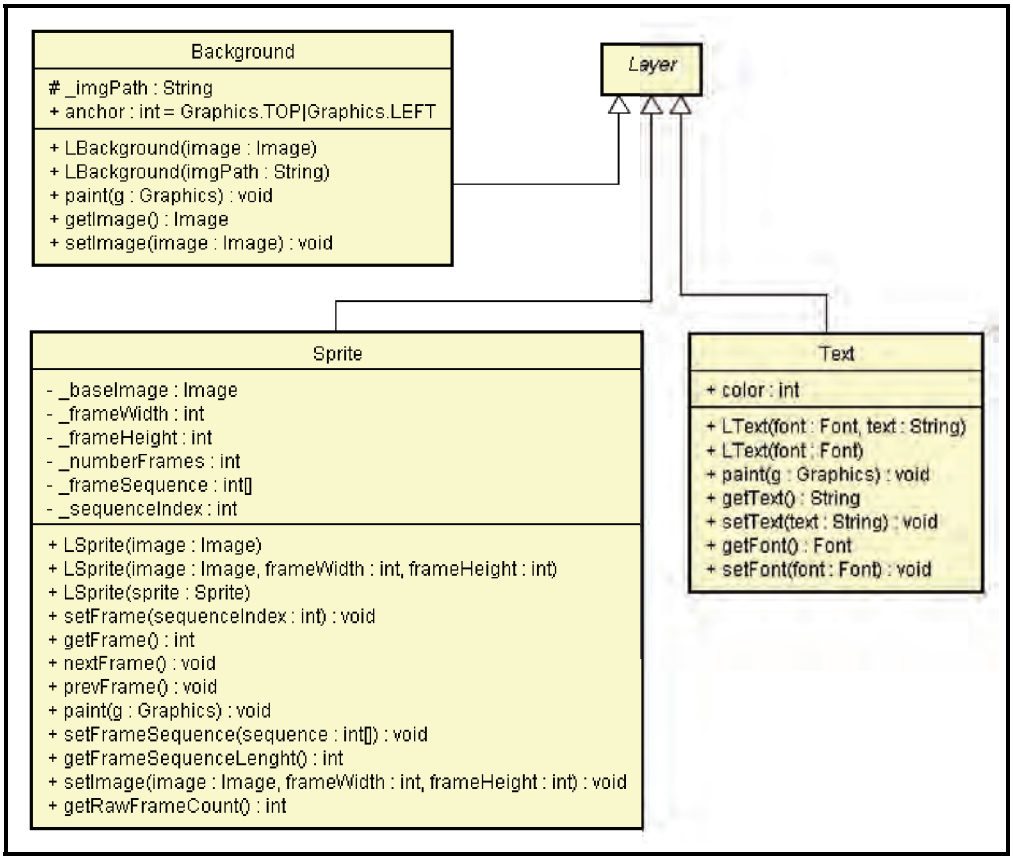



Figura 2.10: Diagrama de Clases, Elementos Gráficos Principales

Como se puede observar, la clase ***Sprite*** es la más compleja mostrada en la figura 2.10, ya que también contiene métodos para manejar su animación. Un objeto *Sprite* se creará a partir de un solo archivo de imagen que contendrá todos los cuadros de su animación, esto porque se requiere menos recursos para guardar una única

imagen que un conjunto de archivos de *bitmaps* separados. Todos los cuadros deben ser iguales en sus dimensiones, esto para facilitar la obtención de los mismos y usar menos cálculos al pintarlos. Al momento de cargarlos, a cada uno se le asignará un índice, el cual después se puede utilizar para crear la secuencia de animación del *Sprite*. El método `paint` se dedicará únicamente a pintar el cuadro de animación actual, el cual se irá cambiando siguiendo la secuencia predeterminada (usando `nextFrame`, para avanzar al siguiente y `prevFrame` para retroceder al anterior) o asignándole el índice de cuadro directamente (usando `setFrame`).

### 2.4.2. Los *Tiles*

Como se vio en el capítulo 1 de esta tesis, los tiles son mosaicos que se unen para formar imágenes más complejas. Para manejarlos se tendrá una clase llamada ***TiledLayer*** cuyo diagrama de clases se muestra en la figura 2.11.

Como se puede observar, un *TiledLayer* necesita ser creado a partir de una imagen que contendrá todo el conjunto de *tiles* que se podrán utilizar. Todos éstos deben tener las mismas dimensiones entre sí. Al momento de cargarlos, a cada uno se le asignará un índice de identificación el cual se utilizará para crear la matriz de *tiles* que será dibujada, como se mostró en la figura 1.10.

La matriz de *tiles* que representa la imagen que se pretende dibujar, puede formarse asignando uno a uno cada índice de los mosaicos a una celda de la matriz, usando los métodos `fillCell`, `setSell` y `setAnimatedTile`. Si ya se conocen todos los índices de los *tiles* que conformarán la matriz, se podría utilizar el método `setTiles` que recibe como parámetro un arreglo. Por último, si se dispone de un archivo de imagen en donde cada píxel de color representa una celda de la matriz, se usaría el método `setTiles` cuyo parámetro es un *bitmap*.

En todos los casos, los índices negativos indican que se trata de un *tile* animado, es decir, que con el paso del tiempo, será substituido por otro para simular la animación. Estos son creados usando los métodos de `createAnimatedTile`, los cuales regresan el índice negativo del *tile* animado. Su representación está dada en la clase ***AnimatedLayer*** (Figura 2.11). Cada *tile* animado tendrá asociada una animación descrita por un arreglo con los índices de los *tiles* que representan sus cuadros de animación. Estos serán actualizados dentro del método `internalUpdate` de la clase *TiledLayer* en el tiempo indicado por el atributo `animatedTileVelocity`.

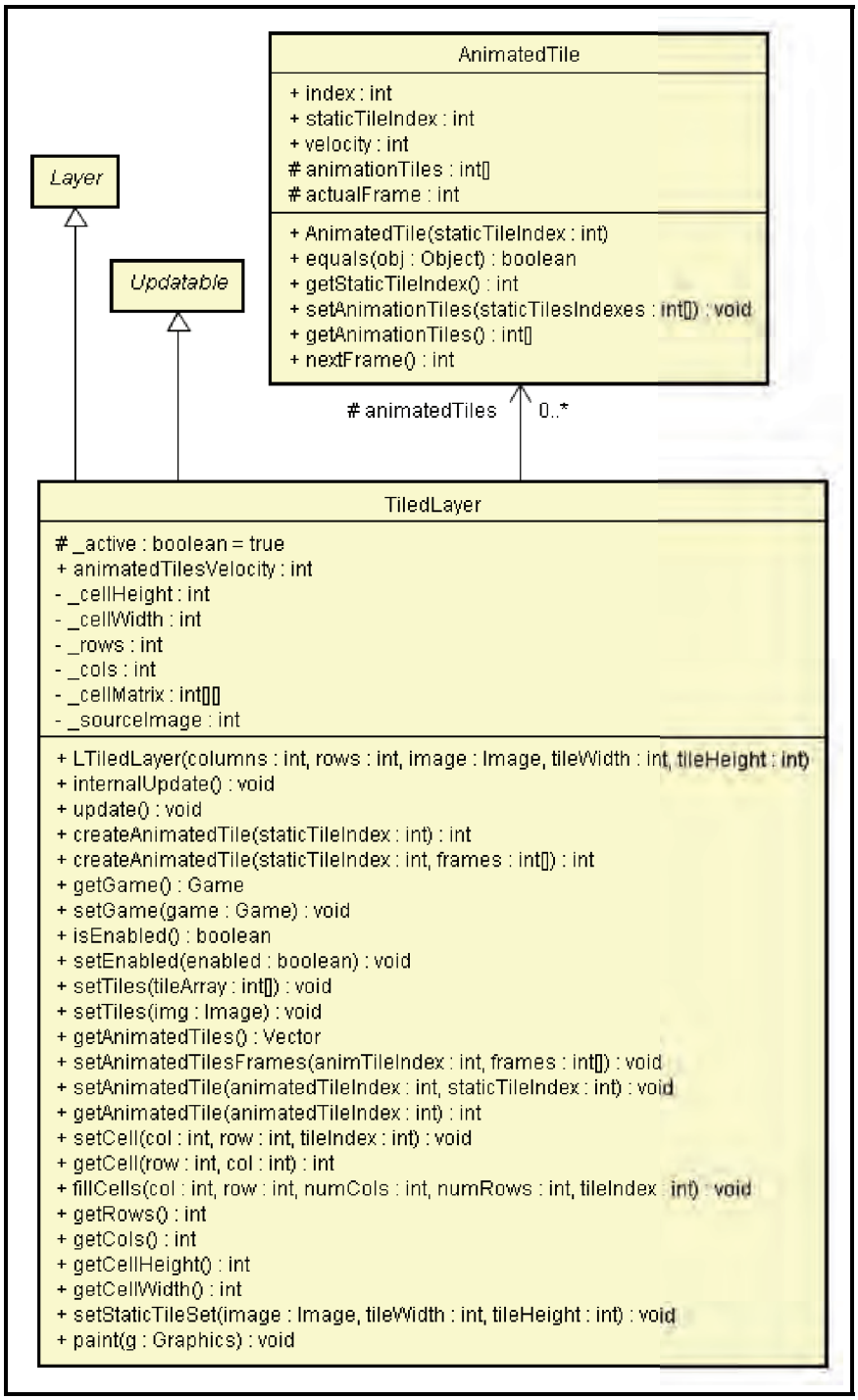



Figura 2.11: Diagrama de Clases, *Tiles*

Debido a que en cada ciclo del *Main Loop* el *TiledLayer* necesita actualizarse internamente, implementa la clase abstracta ***Updatable*** de la cual se habló en módulo de lógica del motor.

Además de las clases descritas anteriormente, dentro de los *tiles* se creó una interfaz llamada ***ITilable***, la cual describe a cualquier elemento que esté conformado por *tiles* y que no sea un *TiledLayer*. En la figura 2.12 se muestra el diagrama de clase de esta interfaz.

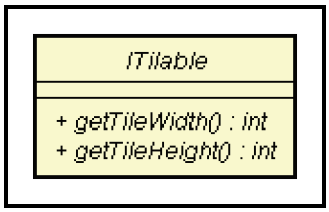



Figura 2.12: Diagrama de Clases, *ITilable*

### 2.4.3. Los Menús

Dentro de los elementos gráficos más utilizados, tenemos también a los *menús*. Estos están constituidos por un conjunto de opciones, las cuales pueden ser tanto *bitmaps* como texto. El usuario puede entonces seleccionar entre ellas usando los botones de acción que el desarrollador haya definido para tal propósito. Además, los menús pueden tener como fondo un *bitmap* o un color predefinido.

En la figura 2.13 se muestra el diagrama de clases que representan a los menús. La clase principal es la clase ***Menu***, la cual también implementa a *Updatable* ya que cada ciclo del *Main Loop* necesita actualizarse internamente. En la sección 2.3 se habla con mayor profundidad acerca de esta clase abstracta y su uso dentro del ciclo principal.

Todos los objetos *Menu* deben tener asociados un conjunto de instancias que extiendan la clase abstracta ***MenuOption*** con sus propias particularidades. Para el caso de este motor se crearon tres tipos: la formada únicamente por texto(`TextMenuOption`), la formada únicamente por un *bitmap*(***ImageMenuOption***) y la que esta formada por ambas cosas (***TextImgMenuOption***).

Los eventos que tiene un menú están definidos por la interfaz ***IMenuEvent***, la cual describe dos eventos principales:

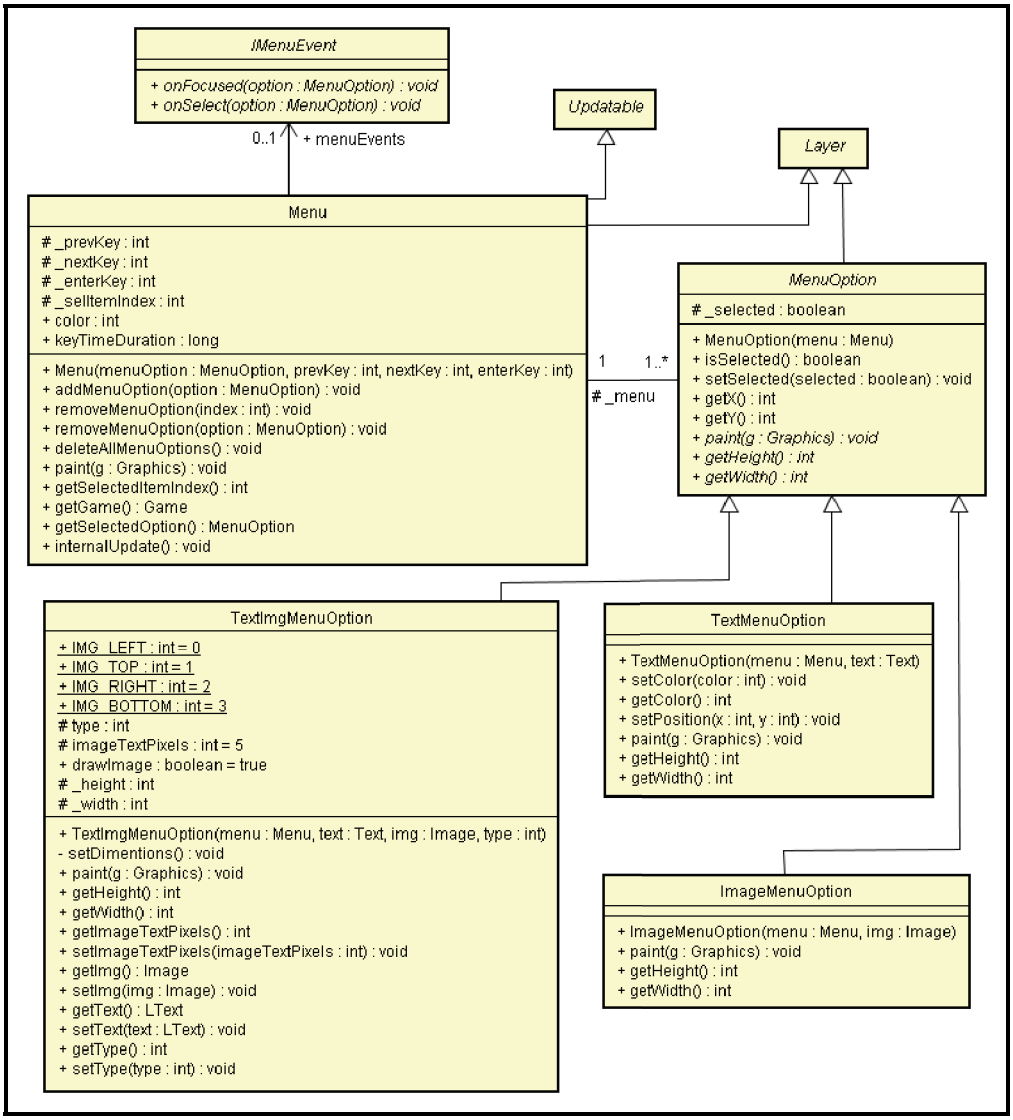



Figura 2.13: Diagrama de Clases, Menús

- `onFocused`, para cuando la opción está resaltada.
- `onSelect`, cuando la opción a sido seleccionada por el usuario.

Para usar los eventos es necesario implementar dicha interfaz y asignársela al atributo `menuEvents` de la clase *Menu*.

### 2.4.4. Los cuadros de texto o diálogos

Otro elemento gráfico importante son los cuadros de texto o diálogos. Estos consisten en rectángulos en cuyo interior se dibuja un texto predeterminado. Se les llamó también diálogos por el parecido que tienen con los cuadros de texto que presentan los videojuegos de rol (*RPG Role Playing Game*) para representar una plática entre personajes. En la figura 2.14 se muestra el diagrama de clases de este elemento gráfico.

La clase principal es la llamada ***Dialog***, la cual implementa tanto a *Layer* como a *Updatable* debido a que cada ciclo del *Main Loop* requiere actualizarse a sí misma (Ver sección 2.3).

Un diálogo despliega en pantalla únicamente el texto que puede dibujar sin sobrepasar el rectángulo que lo limita. De esta manera si el texto no puede ser dibujado en un solo recuadro, queda dividido en diferentes *vistas* a las que se va accediendo mediante los botones designados para dichas acciones. En cuanto al texto, tiene la propiedad de irse dibujando letra por letra a la velocidad definida por el atributo `velocity` de la clase *Dialog*.

El diálogo además, debe tener asignado un botón diferente para cada una sus acciones principales, las cuales son:

- Avanzar a la siguiente *vista*.
- Ir a la última *vista*.
- Cerrar el cuadro de diálogo.

Los eventos del cuadro de texto están definidos en la interfaz *`IDialogEvents`*, la cual debe implementarse y asignarse al atributo `dialogEvents` de la clase *Dialog*. Los principales eventos son:

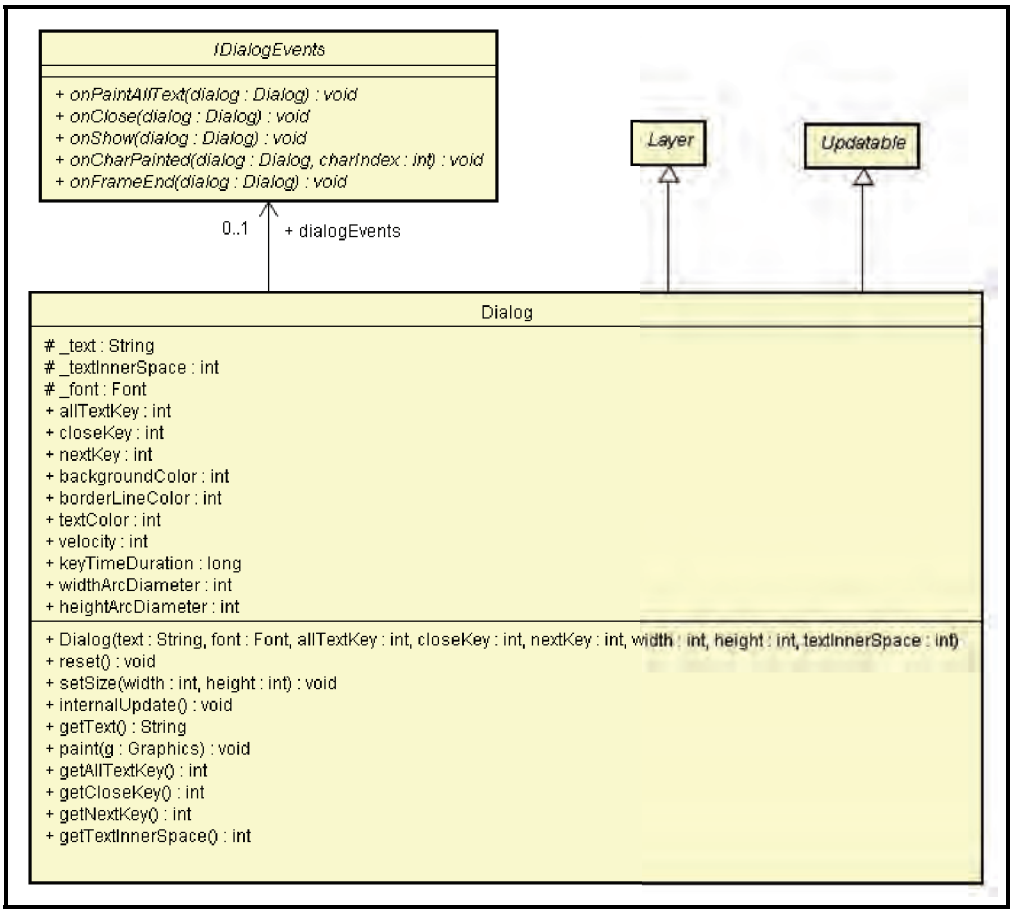



Figura 2.14: Diagrama de Clases, Diálogos

- `onPaintAllText`. Ocurre cuando el texto ha sido dibujado en su totalidad hasta su última vista.
- `onClose`. Ocurre cuando el diálogo es cerrado por el usuario.
- `onShow`. Ocurre cuando el diálogo es desplegado en pantalla después de haber estado oculto.
- `onCharPainted`. Ocurre cada vez que una letra es dibujada en pantalla.
- `onFrameEnd`. Ocurre cada vez que una *vista* es terminada de dibujar.

## 2.5. Módulo de Audio

Este es el módulo encargado de controlar cualquier procedimiento o acción que implique la reproducción de algún sonido dentro de la aplicación. Para su diseño, necesitan tenerse en consideración:

- **Las capacidades de reproducción del dispositivo**. Si se quiere reproducir cierto tipo de sonido, siempre será necesario saber si el dispositivo es capaz de hacerlo. Muchas veces, en el caso de los dispositivos móviles, se está tan limitado que solamente es posible reproducir cierta cantidad de tonos, por lo que sonidos más complejos y de mayor calidad como sería los contenidos en archivos *MIDI*, *wav* o *mp3* quedarían totalmente descartados. Por lo anterior, se debe proveer de algún método que evalúe las capacidades del dispositivo y nos permita tomar decisiones al momento de la reproducción. Así si alguno no puede reproducir archivos *MIDI* o *WAV*, se procederá a sustituirlos por algún equivalente en secuencia de tonos o algún otro formato factible de ejecución.
- **La memoria del dispositivo**. Ya que se revisó que el formato de audio sea reproducible hay que revisar si la memoria propia del dispositivo es la suficiente para almacenarlo. Un archivo de audio sin comprimir tipo *PCM* puede ocupar varios megabytes de memoria de los que muy probablemente un dispositivo móvil no dispone. Además, en muchas ocasiones, las aplicaciones que se instalan en dispositivos móviles tienen un límite de tamaño en memoria total que pueden usar, por lo que al elegir archivos de audio para nuestra aplicación, estos deben tener un tamaño adecuado a la necesidad.

- **La diversidad de formatos de audio**. Hoy en día existen muchas formas de representar audio digitalmente que van desde la más sencilla y costosa en cuanto a cantidad de memoria como es el *PCM* (*wav*, *aiff*), hasta las más modernas y comprimidas versiones como son los *mp3*, *mp4* y el audio *ogg*; pasando por los formatos *MIDI* y el más primitivo de todos: la secuencia de tonos. Por lo anterior, nuestro motor debe proporcionar la forma de reproducir el mayor número de formatos posibles dentro de las propias capacidades de cada dispositivo móvil.

Después de las anteriores consideraciones ya se puede proseguir a definir las principales funciones que nuestro motor debe proporcionar con respecto a la reproducción de audio. En general, las principales acciones que debe permitir ejecutar con el audio serían:

- **Inicialización**. En este paso se prepararía al dispositivo de audio para la reproducción del sonido.
- **Control de Reproducción**. Aquí se proporcionarían métodos para *detener*, *pausar*, *reiniciar* y *reproducir* el sonido; así como los métodos para modificarle su *volumen*.
- **Liberación de recursos**. Después que un sonido es reproducido y no se va usar más, es necesario proporcionar métodos para liberar la memoria que utiliza, así como los recursos de hardware que hayan sido reservados.

También, al momento de la reproducción hay que proporcionar las opciones para que el sonido se reproduzca tanto en el mismo hilo de ejecución, provocando que la aplicación se detenga hasta que el sonido termine; como en un hilo distinto para que se reproduzca mientras el programa continúa.

Ahora, en el caso de los dispositivos móviles, producir sonido puede que se reduzca a lo más primitivo que corresponde a generar audio a través de tonos a diferentes frecuencias. Como es sabido, el sonido es una onda con su propia frecuencia y magnitud que viaja a través de un medio ya sea sólido, líquido o gaseoso. Por eso una nota musical también puede ser representada por medio de su frecuencia y magnitud. Nuestro motor entonces, también debe proporcionar la opción de generar sonidos a partir de frecuencias, para solventar el caso en que las capacidades del dispositivo impidan reproducir audio más complicado.

En la figura 2.15 se muestra el diagrama de casos de uso que ilustra las consideraciones anteriores y que ilustra de una manera más concreta las acciones que se deben implementar.

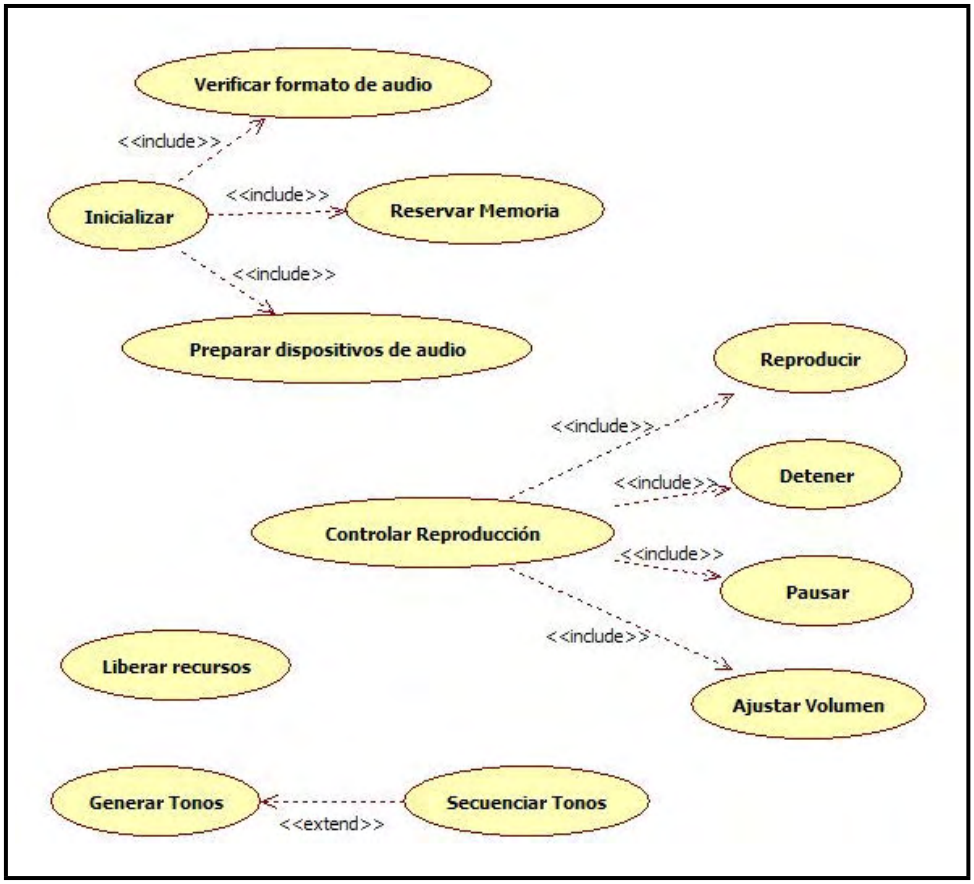



Figura 2.15: Casos de Uso, Módulo de Audio

El componente principal de éste módulo estará determinado por una clase *AudioPlayer*, encargada de inicializar, controlar la reproducción y liberar los recursos reservados de un archivo de audio. Cada sonido tendrá entonces su propio reproductor personal, de manera que en cualquier momento se pueda saber el estado del mismo. Estos estados en los que se puede encontrar el audio pueden ser los siguientes:

- Sin cargar (*Unloaded*) – estado cuando recién se crea la instancia de sonido.
- Inicializado (*Loaded*) – es cuando ya se hicieron todas las verificaciones necesa-

rias, se preparó el dispositivo de audio y se reservó la memoria necesaria para inicializar la reproducción.

- Reproduciendo (*Playing*)
- Liberado (*Disposed*) – todos los recursos que el audio utilizaba ya han sido liberados.

En la figura 2.16 se muestra el diagrama de clases que corresponde al *AudioPlayer*. Como se puede observar, tiene un método para cada una de las acciones principales y lo único que necesitaría para crearse sería la dirección o nombre del archivo de audio a reproducir.

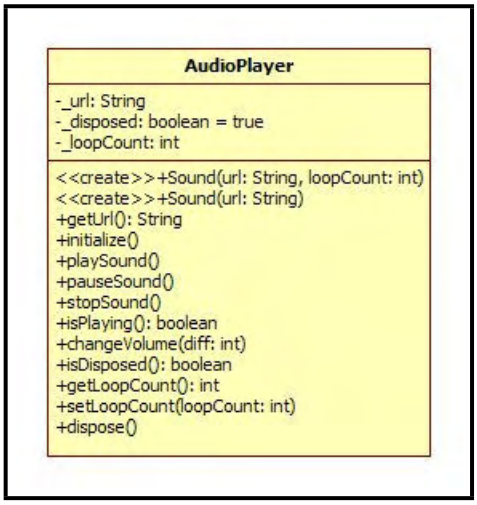



Figura 2.16: Diagrama de Clases, Módulo de Audio

## 2.6. Módulo de Colisiones

Este es el módulo encargado de implementar todos los algoritmos de colisiones que se explicaron en la sección 1.4.8. En el caso de nuestro motor, los algoritmos de rectángulo y de transparencia están implementados dentro de los métodos de la clase *Sprite* encargados de revisar la colisión (métodos `collidesWith`) en los que se recibe como parámetros un objeto tipo *Layer* y un *boolean* que indica si se va usar colisión a nivel de píxel (usando transparencia) o únicamente el basado en rectángulos.

Otra forma de encontrar colisiones es mediante mapas de colisión, los cuales pueden ser, tanto una imagen de bits, como una matriz de *booleanos*. Los mapas representados por *bitmaps*, consisten en imágenes a dos colores, uno de los cuales indica si existe colisión y el otro si no la hay. Dicha imagen tiene que tener el mismo tamaño en unidades fijas que el *Layer* al que quiere representar. Por ejemplo, si tenemos un *TiledLayer* de 25x25 *tiles*, nuestro mapa de colisión de dicho *TiledLayer* también debe medir 25x25 unidades. Además el *Layer* con el que se va verificar la colisión debe poder ser localizado dentro del mapa. Lo más sencillo es que tenga el mismo tamaño que la unidad de medida del *Layer* al que pertenece el mapa, pues así su posición se puede mapear uno a uno sin ninguna dificultad.

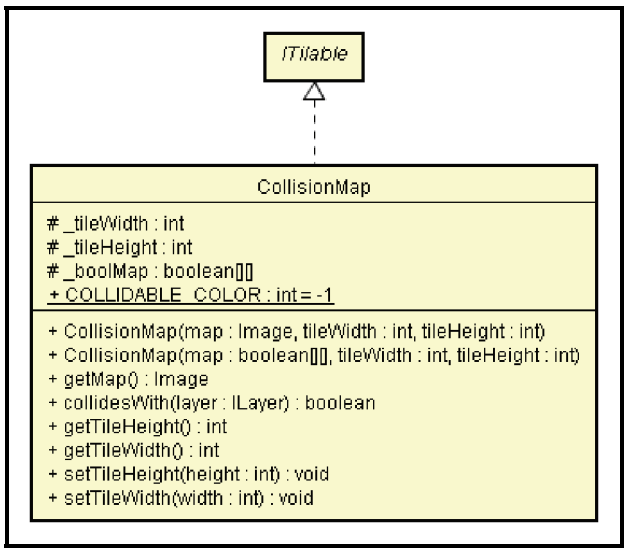



Figura 2.17: Diagrama de Clases, *CollitionMap*

La clase que representa a este mapa de colisión se llama ***CollitionMap*** y su diagrama de clase se muestra en la figura 2.17. Los atributos `tileWidth` y `tileHeight` representan el tamaño de cada una de las unidades en que está dividido dicho mapa, mientras que el método `collidesWidth` nos dice si el *Layer* proporcionado se encuentra en una posición de colisión. Como se puede observar, *CollitionMap* implementa a la interfaz *ITilable* debido a que esta dividido en pequeños mosaicos o unidades.

El mapa de colisión resulta muy útil en juegos que dibujan extensos mundos divididos en áreas formadas por conjuntos de *TiledLayers*, ya que estos están formados por unidades de tamaño fijo cuyas áreas de colisión pueden ser definidas muy fácil-

mente por estos mapas. Un ejemplo palpable se muestra más adelante en el módulo de funcionalidad extra dedicada a los juegos de rol. (Sección 2.10)

## 2.7. Módulo de interacción de usuario

Este es el módulo encargado de controlar los dispositivos de entrada de nuestro motor. Como se mencionó en la sección 1.4.7, en el caso de los dispositivos móviles, es común recibir entradas a través de teclado, pad de direcciones, botones y la pantalla táctil. Como cada dispositivo móvil es muy particular, se debe buscar dar soporte a los elementos que tienen en común la mayor parte de ellos.

En nuestro caso, buscaremos enfocarnos a los elementos que tienen en común todos los teléfonos celulares, como son el teclado numérico y su *pad* de direcciones.

Para percibir entonces el estado de todas estas teclas, el propio dispositivo debe proveernos un medio para leerlo, ya sea a través de una interrupción, una dirección de memoria, una serie de eventos o una estructura de datos bien definida. En el caso de *J2ME*, se proveen eventos para saber si una tecla ha sido presionada, liberada o dejada presionada (métodos `keyPressed`, `keyReleased` y `keyRepeated` de la clase *Canvas*). Además provee métodos para detectar también eventos en pantallas táctiles como son: `pointerDragged`, `pointerPressed` y `pointerReleased` para cuando el puntero es arrastrado a través de la pantalla, es presionado o liberado respectivamente.

En el caso de nuestro motor, revisar el estado de los dispositivos de entrada se realiza al iniciar cada ciclo del *Main Loop* a través de un método al que llamaremos `processKeys`. Este deberá ser implementado para cada aplicación atendiendo únicamente a las teclas y eventos del puntero que se vayan a utilizar; esto para evitar procesar datos innecesarios que robarían valiosos recursos de memoria y procesamiento.

Para poder obtener el estado de las teclas y después poderlo utilizar en otras partes del *Main Loop*, es necesario guardarlo en memoria. Se sugiere utilizar un arreglo que guarde el estado de cada tecla que nos interese y cuyos índices de acceso estén previamente definidos.

## 2.8. Módulo de Física

Cuando las aplicaciones gráficas requieren simular situaciones reales y en ciertos casos leyes físicas, suelen necesitar operaciones matemáticas más o menos complejas. El problema radica en que hay que considerar que los dispositivos móviles son, en la mayoría de las ocasiones, de pobre capacidad de proceso y que además, no siempre proporcionan soporte para operaciones matemáticas con números reales.

Las leyes físicas más comunes a incluir se refieren al cálculo de trayectorias, velocidades, gravedad, tiros parabólicos y más recientemente a manejo de partículas, dinámica de cuerpos deformables y energía.

Es común que en los dispositivos móviles, no se proporcionan métodos para calcular *senos* y *cosenos* por lo que será necesario implementarlos. Se sugiere crear un arreglo que contenga en memoria precalculados el valor de los senos y cosenos entre los ángulos de 0 y 90, ya que el valor para el resto de los ángulos se puede calcular a partir de estos. Otro problema que nos encontramos en el manejo de los senos y cosenos es que estos elementos presentan valores entre 0 y 1 y muchas veces, no existe soporte para números no enteros. Para solucionar este problema, se ha optado por tomar en el precálculo, el valor de las variables multiplicado por 10,000. Así siempre se trabajará siempre con enteros con una precisión de 4 decimales. (Aunque al final de los cálculos, siempre habrá que dividir entre 10,000).

A parte de esto, se pueden proporcionar métodos para calcular posiciones en `X` y `Y` dadas la velocidad y el ángulo de disparo; métodos para calcular nuevas trayectorias al momento de colisionar contra algún elemento; métodos para calcular velocidades de caída libre; métodos para calcular la trayectoria de brincos y rebotes, etc.

Como se mencionó al inicio de este capítulo, por cuestiones de tiempo, el diseño e implementación de este módulo se deja abierto para futuras aportaciones a este trabajo de tesis.

## 2.9. Módulo de comunicación vía red

Tal y como se mencionó en la sección 1.4.12, las aplicaciones que permitan comunicarse con otros dispositivos han crecido en popularidad en nuestros días; por lo

que proporcionar opciones para dicha interacción, forma ya una parte importante de muchos motores gráficos del mercado.

La implementación básica partiría de poder enviar datos a una dirección y/o puerto predeterminados, creando una conexión entre ambos dispositivos. Para lograr esto se pueden implementar *sockets*, *streams* de datos y protocolos más avanzados como *FTP* (*File Transfer Protocol*), *HTTP* (*HyperText Transfer Protocol*), *SOAP* (*Simple Object Access Protocol*), *SMTP* (Simple Mail Transfer Protocol) o hasta protocolos propios que agilicen la comunicación de nuestras aplicaciones.

En el caso de los dispositivos móviles, cada uno tendrá implementadas sus propias formas de comunicarse con otros según sus posibilidades particulares. Esto también debe de tomarse en cuenta al momento de crear la aplicación, ya que canales de comunicación lentos, generan latencias muy grandes con las que hay que poder lidiar para prevenir posibles errores.

Además, también hay que considerar el número de dispositivos a los que se puede estar conectado al mismo tiempo, ya que esto presenta una limitación de diseño que será propia de cada dispositivo.

Los rangos de conexión también deben ser considerados, ya que redes tipo *wi-fi* y *bluetooth* son de apenas unos cuantos metros, mientras que las conexiones a red celular ya son consideradas de acceso remoto. Esto sirve también para definir si nuestra aplicación que comunica varios móviles correrá en una red local o servirá para conectarse con otros en acceso remoto.

Estas y otras consideraciones deben tomarse en cuenta para el diseño y la implementación de este módulo, el cual se deja abierto para futuros trabajos de ampliación de esta tesis.

## 2.10. Módulos de Funcionalidad Extra

Este es el módulo que incluye todos aquellos paquetes y clases que amplían la funcionalidad de nuestro motor y que está relacionado con aplicaciones específicas. Aquí entrarían todas las clases dedicadas, por ejemplo, a pintar gráficas de barras, pastel; paquetes especiales para cierto tipo de juegos, inteligencia artificial, etc.

Por lo anterior y debido a que una parte de las pruebas de este motor consistió en la creación de un demo de videojuego de rol o *RPG* por sus siglas en

inglés (*Role Playing Game*, ver sección 4.2), clases que pueden ser utilizadas en estos juegos fueron anexadas al *engine* descrito en esta tesis. El diagrama de las clases que forman parte de este módulo se puede observar en la figura 2.18.

La clase más importante es la que describe la unidad principal dentro de un juego tipo *rpg*: la clase ***Room*** (*Cuarto*). Este está formado por tres *TiledLayer* que describen el piso, las paredes y el techo del mismo. Además, en cada cuarto pueden haber personajes con los que el jugador puede interactuar pero a los que no puede manejar (*NPC* o *Non Player Characters*). Debido a que un cuarto esta formado por múltiples componentes que requieren tiempo para cargarse y que además requieren de recursos de memoria, *Room* implementa la clase *LoadableContent* para que sólo en el momento en que se va a utilizar, se carguen en memoria los *TiledLayer*, los *NPC* y todos los demás componentes que lo conforman. También, *Room* implementa *Updatable* pues durante cada ciclo del *Main Loop*, necesita actualizar todos sus componentes. Además, tiene un vector con todas las acciones que un jugador puede realizar dentro del cuarto, las cuales pueden ser: activar una palanca, hablar con un NPC, subir o bajar una escalera, abrir una puerta, abrir un cofre, salir del cuarto, etc. Estas acciones implementan la interfaz IAction descrita en la sección 2.3. La clase *Room* consta de una instancia de CollisionMap (Ver sección 2.6) que representa todos los lugares por los que el jugador puede pasar sin chocar al moverse dentro del cuarto. Por último, *Room* implementa a la interfaz *ITilable* debido a que, al estar formado por un conjunto de *TiledLayers*, también esta dividido en pequeñas unidades o mosaicos.

Ahora bien, un conjunto de cuartos forma la llamada área, la cual está representada en la clase ***Area***. Esta también implementa a *LoadableContent* porque, al estar formada de cuartos (*Rooms*), cuenta con la opción de cargarlos a todos en cualquier momento que se requiera.

También contamos con una clase que representa a los personajes del juego: la clase ***Character***. Esta extiende a la clase *Sprite*, ya que cada personaje puede ser representado por un elemento gráfico con animación propia. Dicho personaje tendrá animaciones para moverse hacia el norte, sur, este y oeste con un número de cuadros fijo para todas ellas.

La clase para personajes que el jugador no puede manejar, llamada ***NPC***, está conformada por un objeto *Character* que representa al personaje. Implementa a *LoadableContent* pues su contenido necesita ser cargado solamente cuando sea necesario, esto es, cuando el cuarto al que pertenece también es cargado.

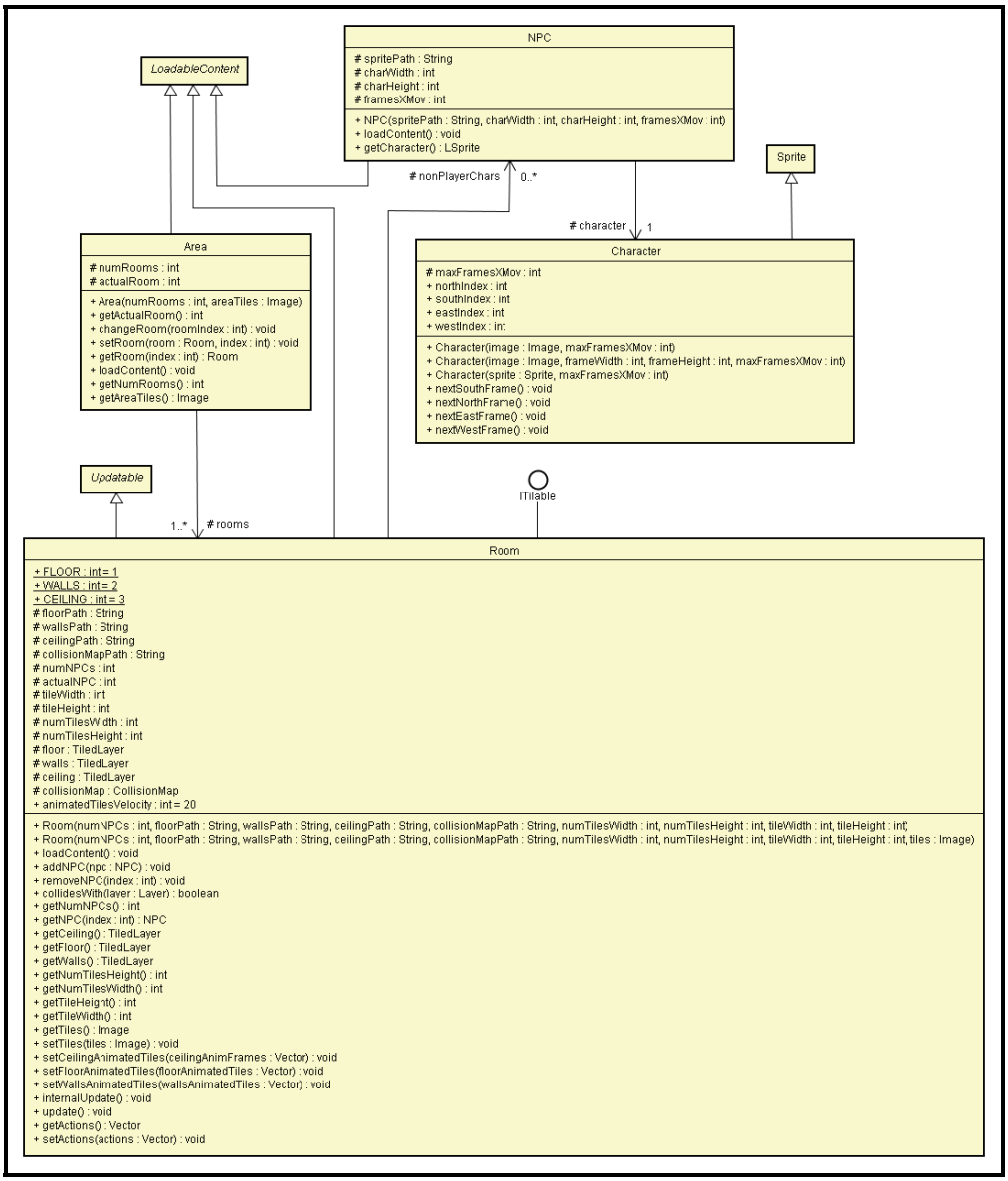



Figura 2.18: Diagrama de Clases, Paquete *RPG*

# Capítulo 3

# Implementación de la Arquitectura

En el capítulo 2 se describió toda la funcionalidad básica que un motor debe considerar al momento de su implementación. Ahora, en este presente capítulo, se describirá el *engine* desarrollado en este trabajo de tesis con todos sus módulos, interacciones y particularidades enfocadas totalmente en la plataforma *J2ME* y en especial para teléfonos celulares.

## 3.1. *Kansik*, un motor para la enseñanza

En la actualidad, como se describió en el capítulo 1, existen muchos tipos de motores gráficos comerciales, tanto de código abierto como propietarios que implementan toda la funcionalidad y los módulos descritos en el capítulo 2. Desgraciadamente la mayoría de los que hay para dispositivos móviles son costosos o de documentación

incompleta. Por lo anterior, el motor que se presenta en esta tesis busca implementar toda la funcionalidad básica y al mismo tiempo, sentar una base para futuros trabajos en la facultad de Ingeniería, proporcionándole herramientas propias para desarrollo de este tipo de aplicaciones en dispositivos móviles.

Estas fueron algunas de las razones consideradas para el desarrollo del Motor **Kansik** o *Kansik Engine*. *Kansik* es una palabra maya que significa enseñar y debido a que este trabajo está dirigido tanto a estudiantes como profesores de la facultad, se consideró como un nombre conveniente, dándole la connotación de ser *un motor para la enseñanza*.

*Kansik* fue implementado en su totalidad bajo la plataforma *J2ME*, principalmente, por la gran cantidad de dispositivos móviles con la que es compatible, además de que es de uso libre, fácil de acceder, tiene gran cantidad de documentación disponible sobre la misma y existen múltiples herramientas de uso libre para la implementación de aplicaciones. Tal es el caso de las *SDK's* de los fabricantes de celulares, los *plugins* del *IDE* de desarrollo *Eclipse* como *EclipseME* y la propia *Wireless Toolkit* de *Sun*; todas estas últimas utilizadas para el desarrollo del software presentado en esta tesis.

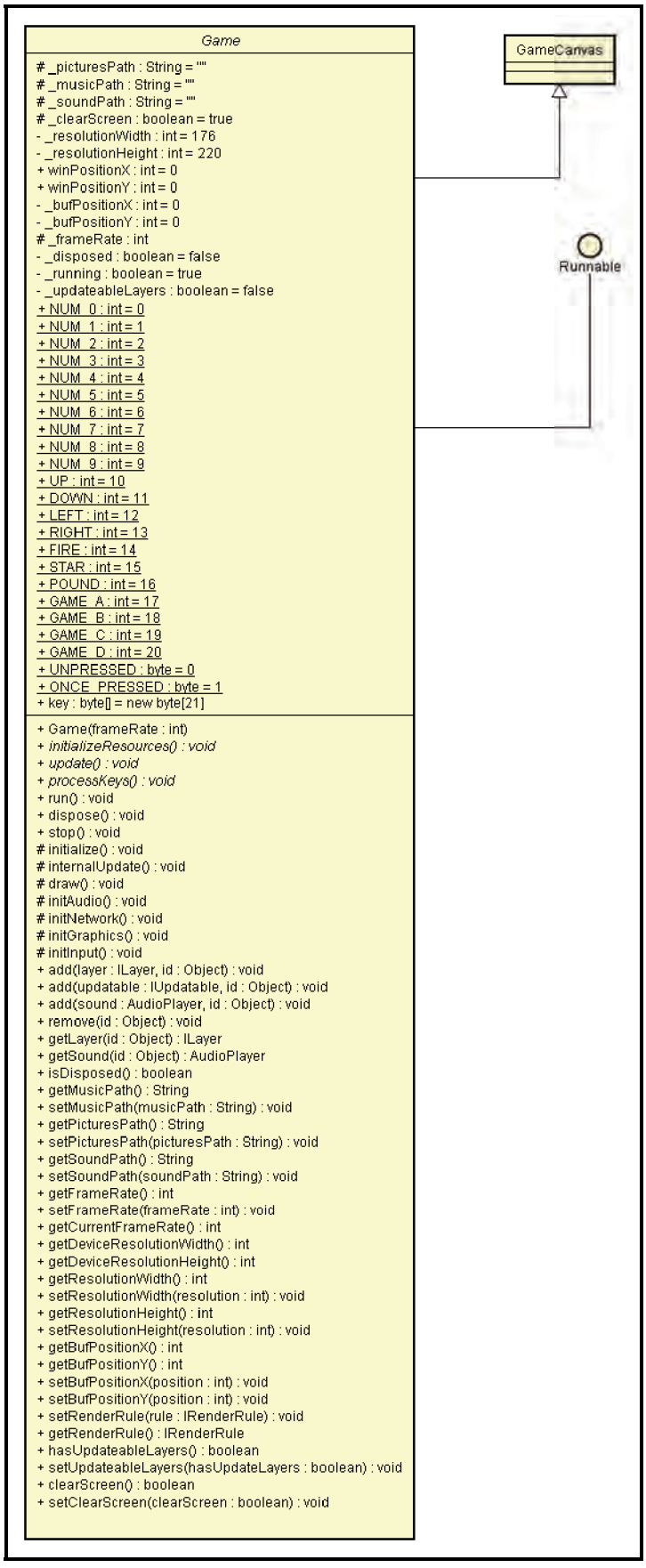
Es así que *Kansik* implementa en J2ME todos los módulos descritos con algunas variaciones, debidas a las particularidades del lenguaje Java, como sería que la herencia no puede ser múltiple, por lo que varias clases abstractas descritas en el capítulo 2 se tuvieron que crear como interfaces.

A continuación se describirán las particularidades de cada módulo al momento de implementarlos en *J2ME*, los problemas presentados y las soluciones a las que se tuvo que llegar.

## 3.2. La clase *Game*

Debido a que de las aplicaciones que más utilizan los motores gráficos son los videojuegos, al momento de crear la clase principal, se le asignó el nombre de ***Game***, (*Juego*). Esta clase es la que implementa al *Main Loop*, por lo que es el corazón de todas las aplicaciones que vayan a utilizar *Kansik*. En la figura 3.1 se muestra el diagrama de clases de la misma.

Como se puede observar *Game* es una clase abstracta que hereda de *Game-*


Game
# _picturesPath : String = ""
# _musicPath : String = ""
# _soundPath : String = ""
# _clearScreen : boolean = true
- _resolutionWidth : int = 176
- _resolutionHeight : int = 220
+ winPositionX : int = 0
+ winPositionY : int = 0
- _bufPositionX : int = 0
- _bufPositionY : int = 0
# _frameRate : int
- _disposed : boolean = false
- _running : boolean = true
- _updateableLayers : boolean = false
+ NUM_0 : int = 0
+ NUM_1 : int = 1
+ NUM_2 : int = 2
+ NUM_3 : int = 3
+ NUM_4 : int = 4
+ NUM_5 : int = 5
+ NUM_6 : int = 6
+ NUM_7 : int = 7
+ NUM_8 : int = 8
+ NUM_9 : int = 9
+ UP : int = 10
+ DOWN : int = 11
+ LEFT : int = 12
+ RIGHT : int = 13
+ FIRE : int = 14
+ STAR : int = 15
+ POUND : int = 16
+ GAME_A : int = 17
+ GAME_B : int = 18
+ GAME_C : int = 19
+ GAME_D : int = 20
+ UNPRESSED : byte = 0
+ ONCE_PRESSED : byte = 1
+ key : byte[] = new byte[21]
+ Game(frameRate : int)
+ initializeResources() : void
+ update() : void
+ processKeys() : void
+ run() : void
+ dispose() : void
+ stop() : void
# initialize() : void
# internalUpdate() : void
# draw() : void
# initAudio() : void
# initNetwork() : void
# initGraphics() : void
# initInput() : void
+ add(layer : ILayer, id : Object) : void
+ add(updatable : IUpdatable, id : Object) : void
+ add(sound : AudioPlayer, id : Object) : void
+ remove(id : Object) : void
+ getLayer(id : Object) : ILayer
+ getSound(id : Object) : AudioPlayer
+ isDisposed() : boolean
+ getMusicPath() : String
+ setMusicPath(musicPath : String) : void
+ getPicturesPath() : String
+ setPicturesPath(picturesPath : String) : void
+ getSoundPath() : String
+ setSoundPath(soundPath : String) : void
+ getFrameRate() : int
+ setFrameRate(frameRate : int) : void
+ getCurrentFrameRate() : int
+ getDeviceResolutionWidth() : int
+ getDeviceResolutionHeight() : int
+ getResolutionWidth() : int
+ setResolutionWidth(resolution : int) : void
+ getResolutionHeight() : int
+ setResolutionHeight(resolution : int) : void
+ getBufPositionX() : int
+ getBufPositionY() : int
+ setBufPositionX(position : int) : void
+ setBufPositionY(position : int) : void
+ setRenderRule(rule : IRenderRule) : void
+ getRenderRule() : IRenderRule
+ hasUpdateableLayers() : boolean
+ setUpdateableLayers(hasUpdateLayers : boolean) : void
+ clearScreen() : boolean
+ setClearScreen(clearScreen : boolean) : void
GameCanvas
Runnable

*Canvas*, una clase provista por *J2ME* cuya principal especialidad es proporcionar un *buffer* para dibujar fuera de pantalla (*off-screen*) con el que se logra evitar "parpadeos"al momento de dibujar la aplicación. Además proporciona métodos específicos para leer los estados del teclado más sencillos que si usáramos directamente los de su clase padre *Canvas*, es decir, en lugar de usar `keyPressed`, `keyReleased` y `keyRepeated`, usa únicamente `getKeyStates`.

*Game* también implementa la interfaz *Runnable* pues, de ser necesario, podría hacerse correr toda la aplicación gráfica en otro hilo de ejecución distinto al del *MIDlet* que la contiene. Dentro del método *Run* es en donde se encuentra implementado el *Main Loop* con todos sus métodos principales los cuales son: `initialize` e `initializeResources` que van antes de que inicie el ciclo en sí; y `processKeys`, `internalUpdate` y `draw` que ya son parte del ciclo principal. A continuación se describen cada uno de estos métodos:

***Método `initialize`.*** Es el encargado de mandar llamar a todos los métodos de inicialización propios del motor y de su arquitectura. Estos son: `initAudio` (encargado de inicializar el hardware de audio), `initGraphics` (encargado de inicializar el *buffer* secundario junto con el hardware de video), `initNetwork` (encargado de inicializar los dispositivos de red) e `initInput` (encargado de los dispositivos de entrada). Estos últimos son métodos protegidos de *Game* que la implementación debe sobrescribir para el caso particular del dispositivo donde vaya a correr la aplicación, por lo que en su mayoría están vacíos.

***Método `initializeResources` .*** Es un método abstracto para que la aplicación que extienda a *Game* implemente. En ella, se deben inicializar todos los recursos propios de la aplicación y que no tengan que ver con los métodos descritos en el punto anterior. Aquí podrían cargarse todos los *layers*, imágenes, archivos, etc. que la aplicación vaya a utilizar a lo largo de toda su ejecución.

***Método `processKeys`.*** Es un método abstracto que cada aplicación que extienda a *Game* debe implementar a razón de las teclas que vaya a utilizar a lo largo de toda la aplicación. Es aquí donde debe registrarse dentro del arreglo de *bytes* `key` si la tecla ha sido o no presionada. La manera de guardar esto es mediante la constante *UNPRESSED*, para cuando la tecla no ha sido presionada, o un número mayor a cero que indica las veces que la tecla se ha mantenido presionada desde la última vez que se usó. Este arreglo sirve entonces para que desde cualquier

otra parte de la aplicación se tenga acceso al estado actual de las teclas leyendo de dicho arreglo. Además, tal y como se puede observar en el diagrama de clase (Figura 3.1) se han definido constantes que se refieren a las principales teclas que están definidas en *J2ME* como las estándar en los dispositivos móviles. Estas constantes se deben usar como índices, tanto como para leer del arreglo `key`, como para actualizar en el mismo el estado de las teclas.

***Método `internalUpdate`.*** Este es el método encargado de mandar llamar a todos los `internalUpdate` de cada uno de los elementos que hayan sido agregados a la clase *Game* usando los métodos `add`; pero sólo se ejecutará de esta manera cuando la bandera `_updateableLayers` sea verdadera. Esto es porque si ninguno de los elementos necesita actualizarse en realidad, sería una pérdida valiosa de tiempo de ejecución el recorrer todo el arreglo de elementos y mandar llamar a estos métodos uno por uno.

Por otra parte, `internalUpdate` siempre manda llamar al método abstracto `update`, sin importar el valor de `_updateableLayers`. Este método `update` debe ser implementado por la clase que herede de *Game* pues es donde estará la lógica propia de la aplicación.

***Método `draw`.*** Este método es el encargado de dibujar en el *buffer* secundario u *off-screen* todos los elementos gráficos de la aplicación. Para esto se auxilia de la interfaz *IRenderRule* descrita en la sección 2.4 de este trabajo. Todas las aplicaciones que pretendan utilizar la clase *Game*, pueden implementar al menos un *IRenderRule* y asignarlo mediante el método `setRenderRule`. Esto permite que las reglas de dibujo puedan quedar independientes y específicas de cada aplicación. En el caso de que un *renderRule* no sea asignado, `draw` mandará llamar todos los métodos `paint` de cada uno de los *layers* agregados mediante los métodos `add`. También dentro de `draw`, siempre se borra la pantalla si la bandera `_clearScreen` es verdadera, de lo contrario, el *buffer* de dibujo se mantendrá igual al anterior.

Al concluir el método `draw`, se manda llamar a `flushGraphics`, un método de *GameCanvas* que transmite al *buffer* principal de video toda la información dibujada en el *buffer* secundario del método `draw`. Después se dará paso a la rectificación del tiempo de ejecución para lograr el *framerate* deseado y así concluir con todos los pasos del ciclo principal.

Aparte de la implementación del *Main Loop*, dentro de la clase *Game* se define la resolución a la que la aplicación se ejecutará, así como su *framerate*, las direcciones de todos los archivos de recursos (audio, imágenes, texto), los sonidos que se usarán y los respectivos métodos para detener la ejecución cuando sea requerido.

## 3.3. Implementación del Módulo de Lógica

Tal y como se describieron en la sección 2.3 de este trabajo, las clases del módulo de lógica fueron implementadas usando *J2ME* pero surgieron algunos cambios para acomodarse al lenguaje.

En primer lugar la clase *Updatable* fue convertida en la interfaz ***IUpdatable***. Esto porque muchos elementos de *Kansik* necesitaban heredar de varias clases además de ésta y el lenguaje *Java* no permite la herencia múltiple. Además se le agregó un parámetro más: una referencia a la clase *Game* al que fue añadido el elemento actualizable. Esto fue porque es común que dentro de los métodos `update` e `internalUpdate` que implementarán las clases que usen esta interfaz, es necesario acceder a variables propias de *Game*, como sería el estado de las teclas, la dirección de los recursos de audio o imágenes de la aplicación o algún otro elemento gráfico que se necesite utilizar.

En el caso de las clases de este módulo ubicadas en el paquete *logic*, también sufrieron algunos cambios. La clase abstracta *StateMachine*, se convirtió en un interfaz llamada ***IStateMachine*** y a la clase abstracta *State* se le agregó una referencia a la clase *Game*. Esto apunta a la misma razón por la que la interfaz *IUpdatable* necesita una referencia a *Game*: conocer la información de las variables que serán luego útiles dentro del método `action` que describe *State*.

Además, para proveer la posibilidad de que la interfaz *IAction* pudiera ejecutarse en otro hilo, se le agregó que extendiera a la interfaz *Runnable* de Java; logrando así que, de ser necesario, se implemente la acción dentro del método `run` para después ser llamado como un *thread* aparte.

Por último, la clase abstracta *LoadableContent* también se convirtió en una interfaz llamada ***ILoadableContent*** atendiendo a la misma situación de la no herencia múltiple en Java.

## 3.4. Implementación del Módulo de Dibujo

Como ya se mencionó en la sección 2.4, el diseño del módulo de dibujo está basado en la forma como *J2ME* en su versión *MIDP* 2.0 sugiere que se realice. La idea era utilizar la clase abstracta *javax.microedition.lcdui.game.Layer* como padre de todas las clases *Layer* que serían definidas para el motor y usar *javax.microedition.lcdui.game.LayerManager* como el manejador de las mismas, pero esto no fue posible. La causa fue que el constructor de la clase *Layer*, está definido como tipo *package*, es decir, sólo las clases que pertenecen al mismo paquete pueden acceder a él. Por eso, el querer definir nuevos *Layers* resultó imposible ya que no se puede meter nuevas clases al paquete *javax.microedition.lcdui.game*, y desde fuera no se podía tener acceso al constructor de *Layer*, pues se ocasionaba un error de compilación.

Por lo anterior, fue necesario implementar nuevamente la clase abstracta *Layer*, sólo que ahora en forma de interfaz con el nombre de ***ILayer***. También se tuvo que implementar nuevamente la clase *LayerManager*, a la que se nombró ***LLayerManager***. Además para que todas las clases que heredaban del *Layer* de J2ME fueran compatibles con la nueva *LLayerManager*, fue necesario rescribir un poco las clases *javax.microedition.lcdui.game.Sprite* y *javax.microedition.lcdui.game.TiledLayer*, de manera que implementaran la interfaz *ILayer*, al igual que todas las demás clases definidas en este módulo de dibujo (Ver figura 3.2). Por eso, las clases cambiaron de nombre a ***LSprite*** y ***LTiledLayer***. En el caso de los *Layers* propios de *Kansik*, se respetó la convención de nombres y se terminaron llamando: ***LText*** y ***LBackground***.

En cuanto a las demás clases, todas quedaron prácticamente iguales, sólo ajustándoles los detalles que implicaron el cambio en las interfaces que implementan como *IUpdatable*, en los que se tuvo que agregar el atributo que corresponde a la referencia de la clase *Game*.

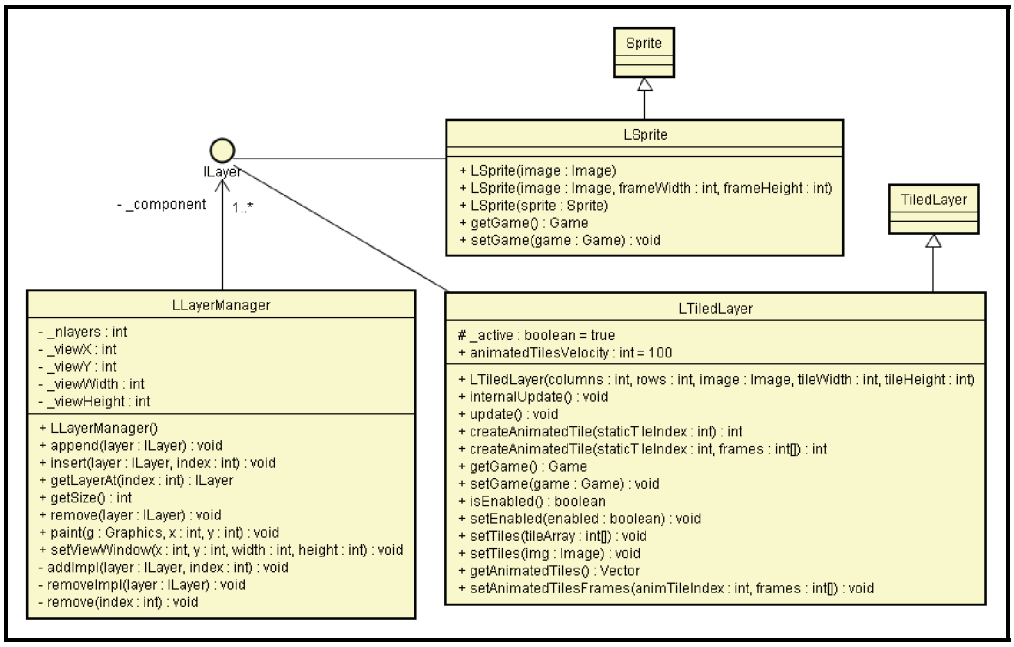



Figura 3.2: Diagrama de Clases, *ILayers*

## 3.5. Implementación del Módulo de Audio

El módulo de audio fue implementado usando las clases provistas por *J2ME* para este propósito, en especial las interfaces *Player* y *VolumeControl*. Estas están implementadas por cada fabricante de dispositivos móviles, por lo que es particular de cada aparato. Para obtenerlas se utiliza, en el caso de la interfaz *Player*, la clase *javax.microedition.media.Manager* en su método `createPlayer`. En el caso de la interfaz *VolumeControl*, se usa el método `getControl` de la clase *Player* regresada por *Manager*. Todas estas interfaces se usan para implementar la clase ***AudioPlayer*** descrita en la sección 2.5 de este trabajo.

Ahora bien, para lograr que el sonido se pudiera reproducir en otro hilo de ejecución, *AudioPlayer* implementa a la interfaz *Runnable*, dentro de la cual, en su método `run`, se manda reproducir el sonido que se haya solicitado.

Además, para revisar que el dispositivo sea capaz de reproducir el audio deseado, se debe hacer uso del método `getSupportedContentTypes` de la clase *javax.microedition.media.Manager*, la cual regresa una lista de todos los *content types* que puede manejar la implementación de *J2ME* del dispositivo en cuestión.

Por último, para la reproducción de tonos, se usarán las clases *Manager*, con su método `playTone` y la interfaz *ToneControl*, cuya implementación es propia de cada dispositivo y se obtiene a través el método `getControl` de la implementación de *Player* del aparato.

## 3.6. Implementación del Módulo de Colisiones

Tal y como se mencionó en la sección 2.6 de este trabajo, el módulo de colisiones se quedó dividido en los métodos implementados por la clase *Sprite* llamados `collidesWith`, los cuales pueden recibir como parámetros: otro *Sprite*, un *TiledLayer* o una imagen; y la clase *CollitionMap* la cual fue implementada en el paquete *collision* tal y como se concibió en el diagrama 2.17.

La implementación de colisiones se puede hacer de varias maneras, tal y como se mencionó en la sección 1.4.8 siendo la más complicada la que se hace a nivel de píxel. Se investigaron formas de implementarla llegando a dos tipos de solución: la primera basada en máscaras y la segunda basada en un color de transparencia. *J2ME* usa la última técnica para dicho propósito, revisando píxel por píxel si uno de color opaco (no de transparencia) se traslapa con otro también de color opaco del *Layer* con el que se quiere revisar la colisión.

## 3.7. Implementación del Módulo de Interacción de Usuario

Prácticamente toda la implementación del módulo de interacción de usuario está dada por las clases *Canvas* y *GameCanvas* de *J2ME* dentro de sus eventos `keyPressed`, `keyReleased`, `keyRepeated`, `pointerDragged`, `pointerPressed`, `pointerReleased` y el método `getKeyStates`. Este último es el que generalmente se usa en *Kansik* dentro del método `processKeys` de la clase *Game*.

Dentro de `processKeys` se revisa el estado de las teclas que nos interesan usando el método `getKeyStates`, el cual regresa un entero que se compara con las constantes definidas en *GameCanvas* indicadoras del estado de las teclas (las variables estáticas ***_PRESSED*). Acto seguido, se actualiza el arreglo definido en Game en el índice de la tecla correspondientes ya sea con el estado UNPRESSED cuando la tecla

no esta presionada, o sumándole una unidad cuando la tecla está presionada. De esta manera, se tendrá en dicho arreglo el número de veces que la tecla se ha mantenido presionada desde la última vez que se usó. Ya en los casos particulares de la sección 4 de este trabajo se profundiza más en la forma como cada aplicación podría utilizar este arreglo para sus propios beneficios.

## 3.8. Implementación del Módulo de Funcionalidad Extra

En este módulo, tal y como se explicó en la sección 2.10, se implementaron clases para el manejo de juegos de rol (*RPG*). Todas siguieron la especificación del diagrama de clases de la figura 2.18 y no sufrieron mayores cambios al momento de implementarse en *J2ME*.

# Capítulo 4

# Módulos de Prueba

En los capítulos anteriores se definieron todas las funcionalidades de *Kansik* y la forma como dicho motor fue implementado en *J2ME* dentro de sus propias particularidades. Ahora, durante el presente capítulo, se explicarán las diferentes aplicaciones creadas con el propósito de probar todos los módulos desarrollados y su desempeño en un dispositivo móvil real.

Las aplicaciones implementadas tienen que ver directamente con el desarrollo de videojuegos ya que, como ya se había mencionado en la sección 1.3.2, son de las aplicaciones que más utilizan este tipo de motores y sin lugar a duda, ponen a prueba todos los elementos implementados en *Kansik*. Con esto en mente se crearon dos demos de juegos, uno de tipo puzzle (*PuyoPuyo*) y otro de tipo *RPG* (*MicroFinalFantasy*). En conjunto, ambos juegos utilizan todas las interfaces y clases de *Kansik*.

Además para ambos juegos se realizaron pruebas de estrés relacionadas directamente con su *framerate*, tanto en el emulador de varios tipos de celulares (emuladores de *J2ME*, de *Sony Ericsson* y de *Nokia*), como en celulares reales: el **Nokia 5300** y el **Sony Ericsson S710**.

## 4.1. Juego tipo *puzzle*

Los videojuegos tipos *puzzle* son aquellos que se basan en "acomodar"fichas, globos, cajas, cápsulas, gotas, etc. generadas al azar, según su forma, color o figura para formar algún "patrón"que dará puntuaciones al jugador y en algunas ocasiones, castigos al oponente. Normalmente terminan cuando ya no se puede crear ningún *patrón* nuevo, por lo que en general son cortos, pero siempre diferentes cuando se les vuelven a jugar. Ejemplos de este tipo de juegos son: *Tetris*, *Puyo Puyo*, *Luminies*, *Puzzle Fighter* y *Meteos*.

El **Puyo Puyo** es un juego cuyo objetivo es juntar cuatro elementos llamados *puyos* que sean del mismo color, creando cadenas que darán puntuación o llenarán de basura la pantalla del oponente venciéndolo. Los *puyos* caen de la parte superior en un par, el cual se puede mover hacia los lados y girar para acomodarlo a la necesidad del jugador. El par cae hasta que uno de los *puyos* queda sobre otro o en el fondo de la pantalla, siguiendo las leyes de la gravedad. Es entonces cuando el par se rompe y los *puyos* por separado siguen cayendo (ya no es posible moverlos) hasta que caen sobre otro *puyo* o en el fondo de la pantalla.

Cada vez que se hagan cadenas, los puyos involucrados desaparecerán de la pantalla y los restantes seguirán los efectos de la gravedad para reacomodarse. De esta manera se pueden crear combos de una cadena hasta varias en un solo movimiento. De esto y la cantidad de *puyos* borrados, depende la cantidad de basura que se le enviará al oponente y la totalidad de puntos a favor generados, por lo que entre más cadenas y *puyos*, más castigo y/o puntuación propia se acumulará.

En el caso particular de la aplicación desarrollada como prueba, se implementó un *Puyo Puyo* que siguiera todas las reglas establecidas anteriormente pero con la limitación a un solo jugador, por lo que la parte de generar basura al contrario no se desarrolló y el juego termina cuando los *puyos* llenan en su totalidad la pantalla del jugador. En la figura 4.1 se muestra una captura de pantalla del juego en ejecución.

El juego presenta un menú de inicio con el fin de probar dicha clase y comandos para terminar su ejecución en cualquier momento. Se controla con el pad de direcciones del celular y el botón de acción del mismo para girar el *puyo*, con los que se probó el módulo de interacción con el usuario.

Para implementar la lógica e inteligencia artificial del juego, se desarrolló una máquina de estados que implementó la interfaz *IStateMachine* la cual estuvo formada


Figura 4.1: Pantalla del *Puyo Puyo*

por seis estados diferentes:

- **Generar *puyo***: el estado donde se genera el par de *puyos* que va a caer. Es el estado inicial de la máquina de estados.
- **Colocar *puyo***: es el estado donde al par de *puyos* se le coloca en su posición inicial.
- **Jugar**: es el estado donde el par de *puyos* va cayendo hasta que se detiene por completo. Es aquí donde la interacción del usuario se prueba, pues sólo durante este estado se puede mover al par de *puyos* y girarlos.
- **Buscar cadena**: es el estado donde se implementó un algoritmo de búsqueda para localizar las cadenas que se podrían haber formado. Es en este estado donde la inteligencia artificial del juego se hace visible.
- **Borrar cadena**: es aquí donde las cadenas detectadas en el estado anterior se borran de la pantalla. En este estado, además se anima el borrado de los *puyos* y su reacomodo por los efectos de la *gravedad*. También aquí se genera el debido puntaje derivado del borrado de cadenas.
- **Juego terminado**: se activa cuando la pantalla ha sido llenada por completo de *puyos* y resulta imposible realizar alguna cadena más. Es el estado final de la máquina de estados.

Cada *puyo* que va cayendo tiene una animación propia, lo mismo que los *puyos* al ser borrados en pantalla. También cuando un *puyo* alcanza su posición final, un sonido es reproducido; probando así el módulo de audio de *Kansik*.

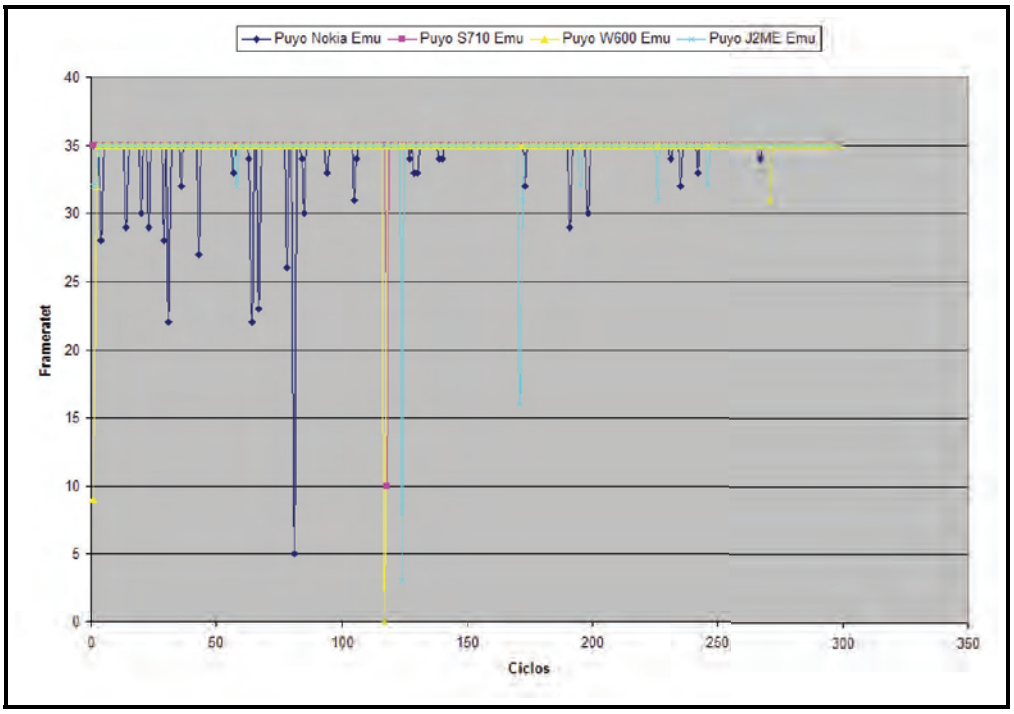
Los cuatro diferentes colores de *puyos* están representados con un *LSprite*, por lo que el funcionamiento de dicha clase fue verificado. Ahora bien, el *framerate* de la aplicación se buscó mostrarlo en pantalla usando la clase *LText* por lo que también el funcionamiento de dicha clase fue verificado.

En cuanto a la prueba de estrés realizada, se probó la aplicación sobre los siguientes emuladores para celulares: *J2ME DefaultColorPhone*, *Sony-Ericsson S710*, *Sony-Ericsson W600* y *Nokia 5300* obteniendo la gráfica de la figura 4.2(a) para cuando se fijó el *framerate* en 35 y la gráfica de la figura 4.2(b) para cuando se dejó libre el *framerate*.

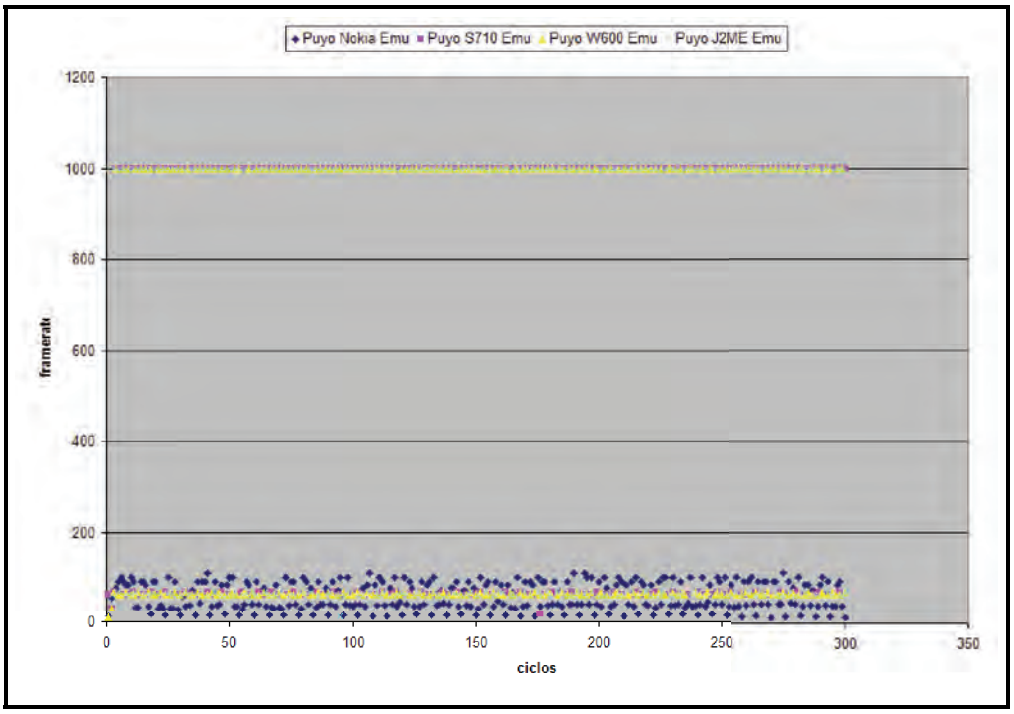
Como se puede ver en las gráficas de la figura 4.2, cuando se fijó el *framerate* en 35, prácticamente se mantuvo constante en todos los emuladores, teniendo mayor variación en el de Nokia, quien mantuvo un *framerate* más parecido al real, lo que nos hace pensar que, además, dicho emulador es mejor. Esto quedó demostrado al observar el comportamiento de la aplicación cuando se dejó un *framerate* libre, pues los demás emuladores presentaron *framerates* poco reales en celulares, pero sí posibles en computadoras de escritorio. De hecho, los casos con un *framerate* de 1000 se refieren a tiempos de *Main Loop* tan cortos que el temporizador no los pudo detectar, por lo que dicho *framerate* fue puesto deliberadamente como máximo dentro del programa.

También se hicieron las mismas pruebas sobre celulares reales de los modelos *Nokia 5300* y *Sony-Ericsson S710* obteniendo la gráfica de la figura 4.3. Debido a que usando *framerate* fijo y libre dieron resultados muy parecidos, sólo se incluyó el del *framerate* libre en la figura. Al observar la gráfica, se puede concluir que el máximo de cuadros que pudo dar el celular *Nokia* está prácticamente en 35 cuadros, con un promedio de 34, mientras que en el *S710* de *Sony* el *framerate* baja a un máximo de 20 cuadros. Después de varias pruebas, se pudo concluir que para dichos celulares, la velocidad de refresco de la pantalla es de 35 y 20 cuadros por segundo respectivamente, ya que tanto usando *framerate* libre como fijo, no se obtuvo una frecuencia mayor, aún limitando al máximo el número de elementos dibujados en pantalla.

Para obtener los datos anteriores se utilizaron las herramientas de desarrollo proporcionadas por las compañías manufacturadoras de los dispositivos sobre los que se hicieron las pruebas, esto es la *Sony Ericsson SDK 2.2.3*, la *Nokia S40 SDK 3rd Edition*




(a) *Framerate* limitado




(b) *Framerate* libre

Figura 4.2: *Framerates* del *Puyo* en Emuladores

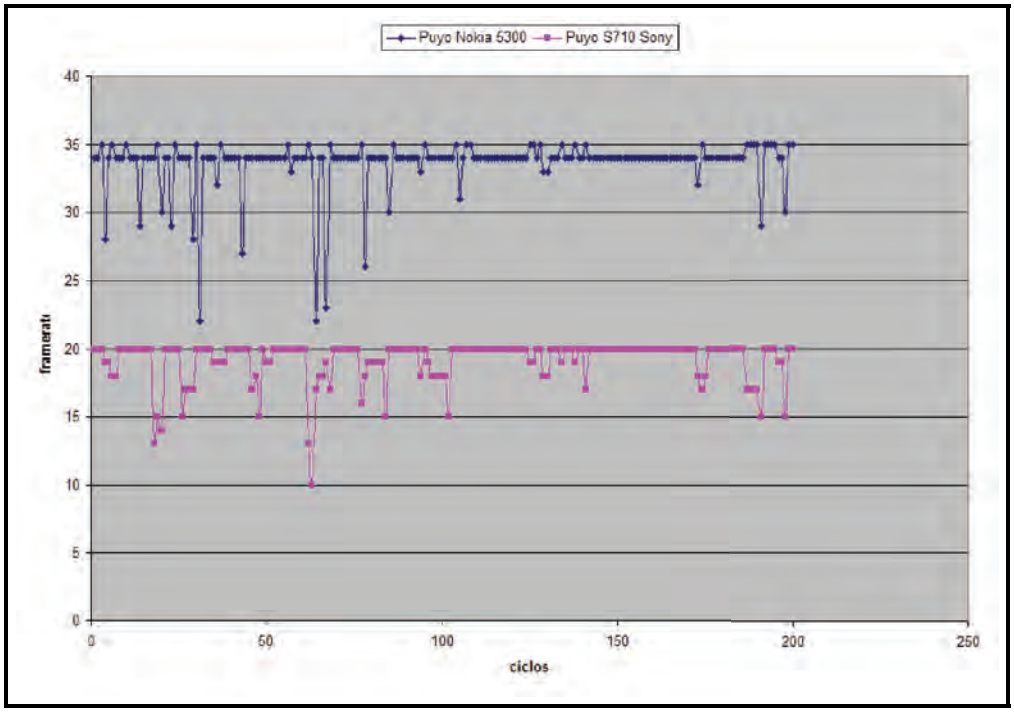



Figura 4.3: *Framerates* del *Puyo* celulares reales

y la *J2ME Wireless Toolkit*.

## 4.2. Juego tipo RPG

Los *Role Playing Games* (*RPG*) o en español *juegos de Rol* son aquellos en los que se toma el papel de algún personaje que va descubriendo su misión y al mismo tiempo va aumentando sus poderes por la *experiencia* que va recibiendo de cada combate al que se enfrenta. También se caracterizan por la gran cantidad de diálogos y la *profundidad* de sus historias. A diferencia de los juegos de plataforma o de disparos, se busca que la forma de jugarlo sea basada más en estrategia que en precisión. Sus escenarios tienden a estar divididos en zonas. De los ejemplos más representativos tenemos a los *Final Fantasy* y los *Dragon Quest*.

En el caso de la prueba realizada, se buscó elaborar una zona o área de algún *RPG* que implementara los elementos básicos de un juego de esta naturaleza, es decir, que tuviera animaciones, personajes, diálogos y acciones. En la figura 4.4 se muestra una captura de pantalla del *demo* de juego tipo *RPG* desarrollado.

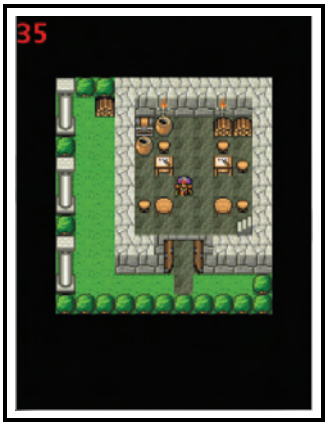



Figura 4.4: Pantalla del *RPG*

Así, nuestra aplicación de prueba esta formada por un área de dos cuartos comunicados entre sí mediante una escalera. También uno de los cuartos tiene un *switch* que, al intentar accionarlo, despliega un cuadro de texto, el cual corresponde a una instancia de la clase *Dialog*. Ambas acciones, tanto la de cambiarse de cuarto como la de accionar el *switch*, corresponden a instancias de la interfaz *IAction* agregadas a

la clase *Room* para formar parte de su conjunto de acciones.

Para la interacción del usuario, se usó el *pad* de direcciones para mover al personaje principal, el cual es una instancia de la clase *Character*; además del botón de acción predeterminado del celular para seleccionar en el menú y tratar de activar el *switch* de la pared del primer cuarto.

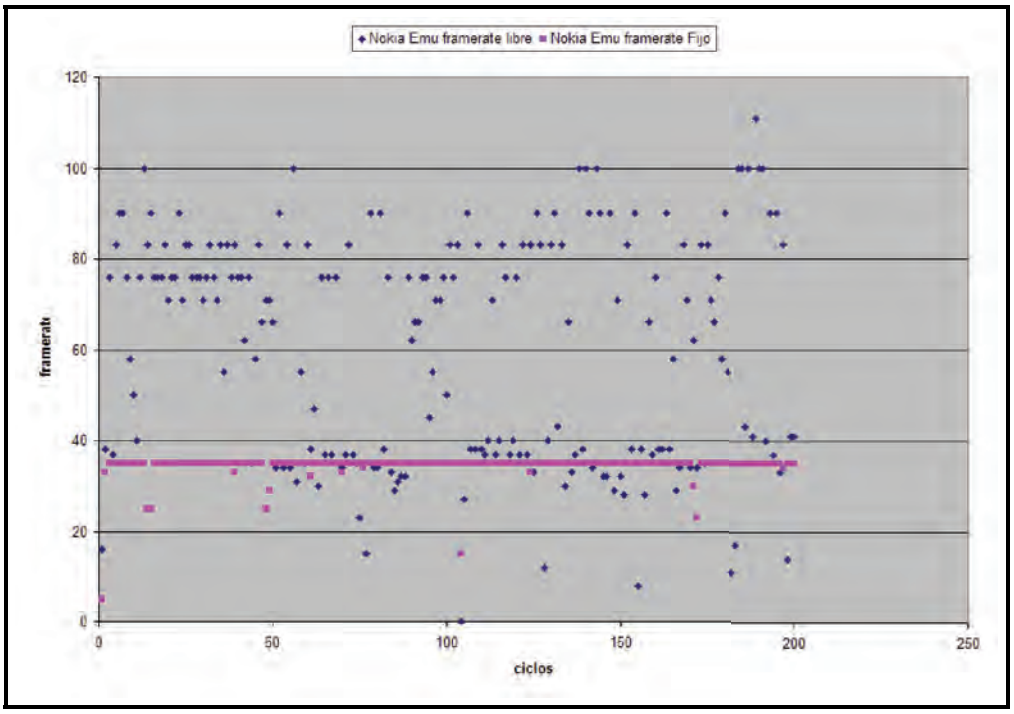
Los cuartos son instancias de la clase *Room* y estas están contenidas dentro una clase *Area*. Cada cuarto es cargado una sola vez en el momento en que va ser mostrado por la primera vez usando su método `loadContent` de la interfaz *ILoadableContent* que implementa *Room*. También cada cuarto fue agregado a la clase *Game* mediante su método `add`, para que fueran actualizados automáticamente dentro del método `internalUpdate` de *Game*, ya que *Room* también implementa la interfaz *IUpdatable* pues, dentro de su propio `internalUpdate`, se manejan los *tiles* animados que el cuarto pudiera tener.

Las acciones agregadas a cada cuarto son activadas desde el método `update` de la clase principal de la aplicación, ya que necesitan parámetros propios del *Main Loop* actual, además de condiciones especiales para ejecutarse.

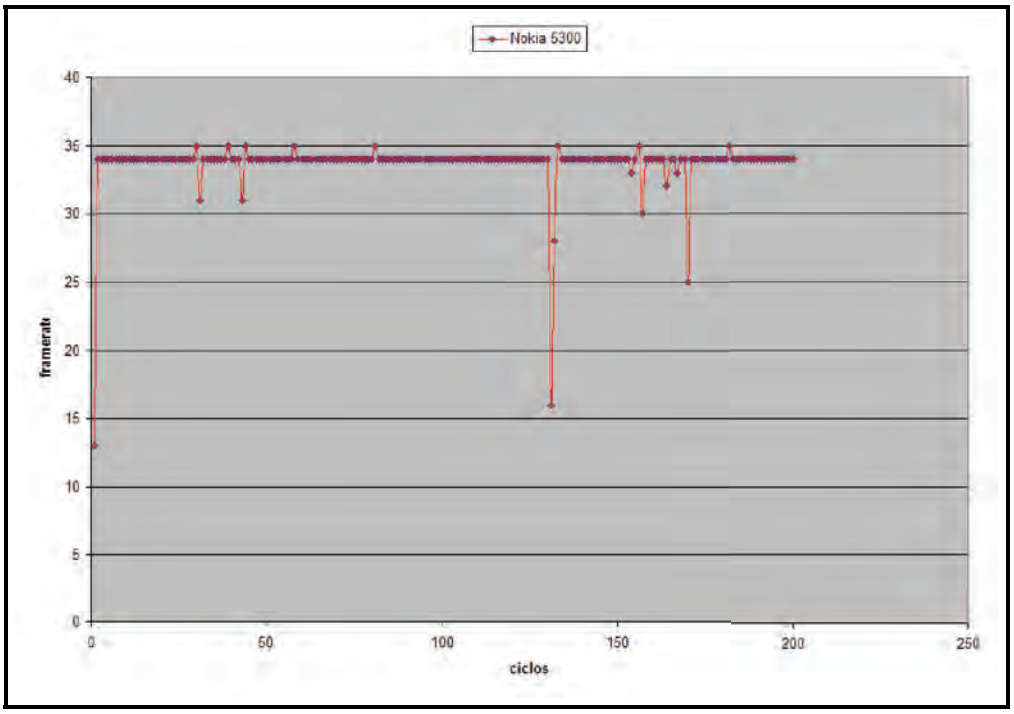
Debido a la cantidad de información que es necesario configurar para cada clase *Room* se sugeriría que para la implementación de un juego completo de esta naturaleza, se crearan herramientas para cargar de archivos toda la configuración de cada cuarto y zona y así fuera más rápida la creación de niveles, evitando también el dejarlos fijos en código.

En cuanto a las pruebas de estrés, también para esta aplicación se realizaron las mismas que al juego de *Puyo Puyo*, pero se presentó un problema: al correrlo en los emuladores de *Sony-Ericsson*, se provocaba en los mismos un error general que causaba su terminación inmediata, cosa que no ocurría en el emulador de *Nokia*, del cual se presentan los resultados obtenidos en la figura 4.5.

Como se puede observar se obtuvo un comportamiento muy semejante al del juego *Puyo Puyo*. En la figura 4.5(a) se muestra el comportamiento del *framerate* del *RPG* en el emulador de *Nokia* en el que al fijarlo, se mantuvo prácticamente constante en 35 cuadros; mientras que al dejarlo libre, estuvo variando más sin mostrar una tendencia clara. Ya en el celular real, el comportamiento mostró un *framerate* casi fijo en 34 cuadros, tal y como se muestra en la figura 4.5(b).




(a) *Framerate* Emulador de *Nokia*




(b) *Framerate* Nokia real

Figura 4.5: *Framerates* del *RPG*

# Conclusiones y trabajo futuro

A lo largo de este trabajo se investigaron los diferentes motores o *API* disponibles en el mercado para el desarrollo de aplicaciones interactivas y de eventos concurrentes que dibujan en tiempo real, para dispositivos móviles. Se estudió la funcionalidad básica para después construir un motor propio, sin dependencias externas y de acceso público. Así se implementó *Kansik*, un motor desarrollado totalmente en *J2ME* el cual abarcó la mayor parte de la funcionalidad que los motores comerciales o de código cerrado presentan, por lo que se puede decir que se cumplió con el objetivo principal de esta tesis.

También se buscaba que el motor tuviera un desempeño aceptable al ejecutarse en *hardware* real, lo cual fue comprobado en el capítulo 4 durante las pruebas tanto del juego *Puyo Puyo* como el demo de *RPG*, al presentarse un *framerate* bastante aceptable y muy cercano al máximo que el propio dispositivo era capaz de proporcionar. De lo anterior se puede concluir que el motor *Kansik* pudo proporcionar la funcionalidad y la velocidad necesaria para correr la parte central de una aplicación gráfica en un dispositivo móvil real, sin comprometer su desempeño.

Además el diseño presentado en el capítulo 2 es aplicable a cualquier lenguaje orientado a objetos como *C#* o *C++*, por lo que se podría tomar como base para el desarrollo de otros motores que abarcaran diferentes dispositivos, no sólo limitándolos a móviles, sino también a consolas y computadoras personales. De hecho *XNA*,

el *API* de *Microsoft* para desarrollar videojuegos, presenta una estructura semejante a la de *Kansik*, pero ya enfocada a desarrollo de aplicaciones en 3D para la consola *XBox360* y la *PC*.

Aún así todavía queda trabajo por hacer. Si bien *Kansik* abarcó la mayoría de la funcionalidad presentada por otros motores comerciales, todavía quedaron módulos que son necesarios de diseñar e implementar. El módulo de comunicación vía red y el módulo de física se consideraron pero no se desarrollaron.

Además, para la facilidad de implementación y configuración de aplicaciones, hizo falta el desarrollo de clases que permitieran la lectura de archivos de texto o binarios, de manera que sobre ellos se describieran las características de todos los elementos de la aplicación y evitar así que estos quedaran fijos en código, dándole mayor versatilidad a las aplicaciones desarrolladas sobre *Kansik*.

Por último, hubiera sido deseable contar con pruebas en una variedad más amplia de teléfonos celulares y *PDA's*, para así tener la certidumbre de que el campo de aplicación de *Kansik* se extiende, efectivamente, a una amplia gama de dispositivos móviles, puesto que las pruebas en emulador difieren de las pruebas sobre el *hardware* real.

# Apéndice A

# Ejemplo de uso de Kansik

En este apéndice se explica a grandes rasgos, una aplicación ejemplo de la utilización de *Kansik*. Esta consiste en un *MIDLet* que dibuja en pantalla un *sprite* que camina hacia los lados al presionar el pad de direcciones del dispositivo móvil ya sea a la derecha o a la izquierda.

Antes de comenzar a codificar cualquier aplicación que use *Kansik*, es necesario incluir en el *classpath* de la misma el archivo `kansik.jar`, para evitar errores al momento de compilación. Todos los recursos que se vayan a utilizar, tanto las imágenes, como los archivos de audio, deben colocarse bajo la carpeta `/res` a partir de la raíz del proyecto.

La aplicación está formada por tres partes:

- El *MIDLet*, llamado `MiniGameMidlet`.
- La clase `MiniGame` que hereda de `engine2D.Game`, la clase principal de *Kansik*.
- La clase `MegaMan` que hereda de `LSprite`.

## A.1. La clase MiniGameMidlet

El *MIDLet* es una clase que hereda de `javax.microedition.midlet.MIDlet`, la cual es necesaria para que la aplicación sea ejecutada dentro del dispositivo móvil. En el caso particular de este ejemplo, está conformada sólo por dos atributos privados que corresponden a una instancia de la clase `javax.microedition.lcdui.Display` y otra de `engine2D.Game` tal y como se muestra a continuación:

```
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

public class MiniGameMidlet extends MIDlet implements CommandListener {

    private Display display = null;
    private MiniGame game = null;

    public MiniGameMidlet() {
        super();
    }

    ...
}
```

Ambos atributos son inicializados en el método `startApp()` de la clase `MIDlet`, el cual es llamado inmediatamente que la aplicación es ejecutada por la primera vez. El atributo `display`, corresponde al objeto encargado manejar los dispositivos de entrada y salida de la aplicación, como sería la pantalla y el teclado, y al ser único para cada `MIDlet`, debe ser solicitado mediante el método estático `getDisplay` de la clase `Display`. Una vez obtenido se le asigna como pantalla desplegable principal al atributo `game` ya inicializado. De esta manera el objeto `game`, tendrá el control de toda la aplicación. A continuación se muestra la implementación del método `startApp()`:

```
protected void startApp() throws MIDletStateChangeException {
   display = Display.getDisplay(this); //Obtenemos el display

   game = new MiniGame(35); //inicializamos la clase MiniGame
   game.addCommand(new Command("Quit", Command.EXIT, 1)); //Agregamos el comando de salida
   game.setCommandListener(this); //Establecemos como manejador de comandos al MIDLet

   display.setCurrent(game); // Establecemos como display principal a la clase MiniGame
```

```
    game.run(); // Inicializamos el MainLoop de MiniGame
}
```

Como se puede observar, al objeto `game` también se le agregó un comando para salir de la aplicación en cualquier momento y su manejador de evento es el propio *MIDLet* (de ahí que la clase `MiniGameMidlet` implemente a `CommmandListener`). Al final de `startApp()` se manda ejecutar el método `run` de `game` para que la aplicación que usa *Kansik* sea ejecutada.

Por último, dentro de `MiniGameMidlet`, se agregó el código para implementar `CommandListener`, dentro del cual se le indica a la aplicación que es necesario que se cierre al momento de recibir el comando de salida.

```
public void commandAction(Command c, Displayable d) {
    // If we get an EXIT command we destroy the application
    if(c.getCommandType() == Command.EXIT) {
        game.stop();
        game.dispose();
    notifyDestroyed();
    }
}
```

## A.2. La clase MiniGame

`MiniGame` es una clase que hereda de `engine2D.Game`, la clase principal de *Kansik*, donde los principales métodos que se implementan son `initializeResources()`, `processKeys()` y `update()`. Contiene sólo un atributo que corresponde a un objeto de la clase `LText`, el cual se usa para dibujar en pantalla el *framerate* de la aplicación. A continuación se muestra el cuerpo principal de la clase:

```
import javax.microedition.lcdui.Font;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.game.GameCanvas;
import audio.AudioPlayer;
import engine2D.Game;
import engine2D.drawing.ILayer;
import engine2D.drawing.LText;
import Megaman;

public class MiniGame extends Game {
    private LText frame;

    public MiniGameTest(int frameRate) {
```

```
        super(frameRate);

        setPicturesPath("/img/");
        setSoundPath("/audio/");
        setUpdateableLayers(true);
    }

    ...
}
```

En el constructor se configura la dirección dentro de la carpeta `/res` donde estarán respectivamente los archivos de audio y los de imagen que se usarán. Además se establece que los *layers* añadidos tendrán que ser actualizados automáticamente por *Kansik*.

A continuación se muestra la implementación del método `initializeResources()`:

```
public void initializeResources() {
   try {

      //Inicializar el Sprite
    Image fichaImg = Image.createImage(getPicturesPath()+"sprite/megaprueba.png");
    Megaman megaman = new Megaman(fichaImg,25,29);
    megaman.move(30, 30);

    add((ILayer)megaman, "megaman");

    //Declaración del sonido
      AudioPlayer background = new
      AudioPlayer(getSoundPath()+"sound90.wav");
      background.initialize();
      Thread audio = new Thread(background);
      audio.start();

      //Declaracion del texto
      frame = new LText(Font.getDefaultFont());
      frame.color = 0xff0000;
      frame.setPosition(0,0);

      add((ILayer)frame, "frame");

   } catch (Exception ex) {
      System.out.println("Error initializing resources: "+ex.toString());
   }
}
```

En este método, se inicializan todos los recursos que se utilizarán durante la ejecución del programa. Esto incluye: cargar la imagen para después inicializar el *Sprite* que se dibujará, la creación del texto que desplegará el *framerate* y por último

cargar el archivo de sonido que se ejecutará como música de fondo.

Después tenemos la implementación del método `processKeys()`, el cual es el encargado de procesar únicamente las teclas que nuestra aplicación utilizará. A continuación se muestra su código:

```
public void processKeys() {

   int keys = getKeyStates();

   if((keys & GameCanvas.LEFT_PRESSED) != 0)
      key[LEFT] += 1;
   else
      key[LEFT] = UNPRESSED;

   if((keys & GameCanvas.RIGHT_PRESSED) != 0)
      key[RIGHT] += 1;
   else
      key[RIGHT] = UNPRESSED;
}
```

La clase `Game` tiene un arreglo donde se van acumulando las veces que una tecla ha sido presionada, de ahí que cuando es detectada, el método `processKeys()` aumenta en uno la cuenta para dicha tecla. En el caso contrario la pone en estado de *UNPRESSED*, el cual equivale a un cero en las veces que se ha presionado. Más adelante, en la clase `Megaman`, se verá como se usa la información recopilada por `processKeys()` en la aplicación.

Por último tenemos la implementación del método `update()`, que en nuestro caso, sólo es responsable de actualizar la información de *framerate* a desplegar por el texto `frame`:

```
public void update() {
   frame.setText(Integer.toString(this.getCurrentFrameRate()));
}
```

## A.3. La clase Megaman

La clase `Megaman` fue creada para proporcionar la posibilidad de que el *sprite* que representa, se pueda actualizar a sí mismo durante cada ciclo con su lógica propia. Es por eso que hereda de `LSprite`, la clase de *Kansik* para manejar *sprites*, e implementa `IUpdatable`, la interfaz de *Kansik* que permite que un objeto sea actualizado durante cada ciclo del *Main Loop*. A continuación se muestra el cuerpo principal de dicha clase:

```
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.game.Sprite;
import engine2D.Game;
import engine2D.IUpdatable;
import engine2D.drawing.LSprite;

public class Megaman extends LSprite implements IUpdatable {
    int frameDelay = 0;
    private boolean _enabled = true;
    private Game _game;

    public Megaman(Image image, int frameWidth, int frameHeight) {
        super(image, frameWidth, frameHeight);
    }

    public Megaman(Image image) {
        super(image);
    }

    public Megaman(Sprite sprite) {
        super(sprite);
    }

    public void update() {

    }

    public Game getGame() {
        return _game;
    }

    public void setGame(Game game) {
        _game = game;
    }

    public boolean isEnabled() {
        return _enabled;
    }

    public void setEnabled(boolean enabled) {
        _enabled = enabled;

    }

    public void internalUpdate() {

        ...
    }
}
```

El método más importante de esta clase es `internalUpdate()`, ya que dentro de él es donde se ejecuta toda la lógica propia del *sprite* en cuestión, es decir, en este método es donde se lee si las teclas a la derecha o a la izquierda han sido presionadas y se procede a mover y animar al *sprite*. A continuación se muestra la implementación de dicho método:

```
public void internalUpdate() {
   //Verifica que el sprite este activado
   if(!_enabled) return;

   //revisa el estado de las teclas
   if(_game.key[Game.LEFT]>=Game.ONCE_PRESSED) {
      if(frameDelay>=2) {
         nextFrame();
         if(getFrame()==0) nextFrame();
         frameDelay=0;
      } else {
         frameDelay++;
      }
      move(-5,0);
   } else if(_game.key[Game.RIGHT]>=Game.ONCE_PRESSED) {
      if(frameDelay>=2) {
         nextFrame();
         if(getFrame()==0) nextFrame();
         frameDelay=0;
      } else {
         frameDelay++;
      }
      move(5,0);
   } else {
      setFrame(0);
   }
   update();
}
```

Como se puede observar, primero se revisa si el *layer* está habilitado ya que si no lo está, el contenido de `internalUpdate()` no debe de ejecutarse. Después, se procede a revisar el estado de las teclas, el cual fue actualizado por `processKeys()` durante el ciclo principal. La variable `frameDelay` se usa para evitar que cada vez que se presione una tecla de movimiento el *sprite* cambie al siguiente cuadro, lo cual provocaría que la animación se viera demasiado rápida y poco fluida. Al final se manda llamar al método `update()`, ya que ,en caso de que alguien herede de esta clase, `internalUpdate()` siempre mande ejecutar al `update()` de la clase hijo.

Al final la aplicación funcionaría así:

1. Se inicializan todos los recursos a utilizar.

2. Se inicia el *Main Loop* en donde:

   *a*) Se manda llamar a `processKeys()`.

   *b*) Se mandan llamar todos los `internalUpdate()` de los *layers* agregados a `MiniGame`.

   *c*) Se manda llamar al método `update()` de `MiniGame`.

   *d*) Se manda llamar a todos los métodos `paint` de los *layers* agregados a `MiniGame`.

3. Se reinicia el ciclo.

Al momento de ejecutar `processKeys()` se actualiza el estado de las teclas, posteriormente dentro del `internalUpdate()` de la clase `Megaman`, se revisa si hay que animar y mover al *sprite*; después en el método `update()` se actualiza el *framerate* actual, para por último llamar al método de dibujo de ambos *layers* y que estos se muestren en pantalla.

Para mayor referencia, consultar el **javaDoc** de *Kansik* proporcionado en el disco anexo a esta tesis.

# Apéndice B

# Instalación de Aplicaciones J2ME

En este apéndice se explica la forma de instalar una aplicación *J2ME* en algún dispositivo móvil, partiendo desde la generación del archivo `.jar` y el `.jad`; hasta su ejecución en el teléfono celular.

Los requerimientos para lograrlo son: el *Wireless Toolkit* de *Sun* (disponible en la página oficial de *J2ME*), el software propio del celular para conectarse a una computadora y, si se desea, un programa ofuscador para reducir el tamaño del archivo `.jar` final. Para este ejemplo se usará el software para conectar celulares *Nokia* a la computadora llamado *Nokia PC Suite*, el cual venía incluido con la compra del celular junto con un cable *USB*. El ofuscador tampoco será utilizado en este ejemplo.

En primer lugar, donde se encuentra instalado el *Wireless Toolkit*, hay que copiar la carpeta que contenga nuestro programa en *J2ME* bajo la carpeta `app`. De esta manera el código quedará visible para que el *Wireless Toolkit* lo pueda manipular. Nuestro proyecto debe tener la siguiente estructura de carpetas:

- `/res` – estarán los archivos de imagen, texto o audio usados por la aplicación.
- `/lib` – estarán los archivos .jar que use la aplicación.
- `/scr` – estarán los archivos fuente de la aplicación.

A continuación será necesario ejecutar la aplicación llamada *KToolbar* localizada en: Inicio-¿Programas-¿J2ME Wireless Toolkit-¿KToolbar, la cual nos desplegará la siguiente pantalla:

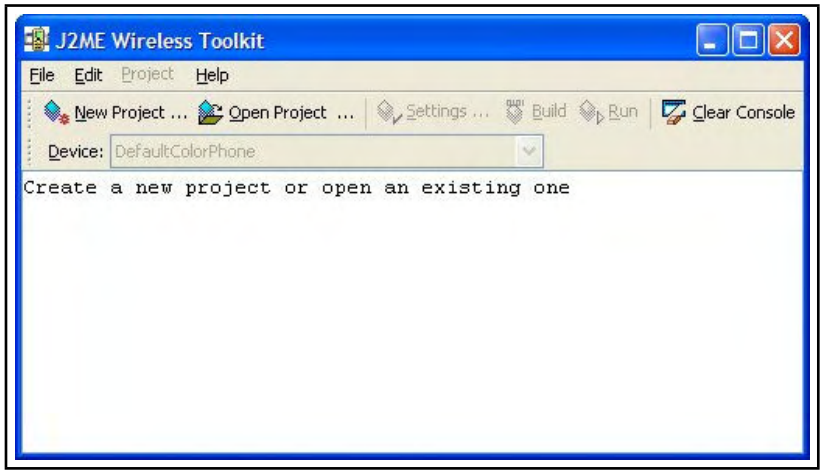



Figura B.1: *KToolBar*

Ahí hay que seleccionar la opción **NewProyect**, donde solicitará un nombre para dicho proyecto y el nombre del *MIDLet* de nuestra aplicación. Se mostrará entonces una ventana de configuración de la aplicación en la que hay que seleccionar el tipo de configuración de nuestro dispositivo (*CLDC* 1.0 o 1.1) junto con los paquetes opcionales que la aplicación utilizará. Los demás parámetros pueden quedarse sin modificar.

En este momento el *Wireless Toolkit* ya puede compilar la aplicación presionando el botón **Build**, marcando cualquier error que la aplicación pudiera tener. Esta compilación incluye la preverificación necesaria para que el programa pueda ejecutarse en el dispositivo móvil. Después de compilarse, la aplicación puede ejecutarse en el emulador que el *Wireless Toolkit* proporciona, presionando el botón **Run**.

Si no hay errores se puede proseguir a generar el `.jar` y `.jad` de la aplicación. Esto se hace ejecutando el comando localizado en Project-¿Package-¿Create

Package. En este momento se generarán los archivos `NombreDelProyecto.jar` y `NombreDelProyecto.jad` que corresponden a la aplicación y estarán localizados bajo la carpeta del proyecto dentro de la carpeta `bin`. Estos dos archivos son los que se deben de subir a algún portal *WAP*[1] para bajarlos al celular o ser transferidos mediante otro medio desde la computadora al dispositivo móvil.

Para instalar la aplicación en el celular, en el caso del teléfono *Nokia 5300*, se conecta el mismo al la computadora mediante el cable *USB*. El celular solicitará el modo en que debe conectarse y debe seleccionarse el modo *Nokia*. Si el software *Nokia PC Suite* está correctamente instalado, inmediatamente se ejecutará.

En esta aplicación hay que seleccionar el ícono de **Instalar Aplicaciones**. Aquí sólo hay que buscar la carpeta donde esté el `.jar` y `.jad` de la aplicación y presionar la flecha que lo copia al celular. En ese momento la aplicación quedará instalada en el teléfono.

[1] Protocolo de aplicaciones inalámbricas o *Wireless Application Protocol* por sus siglas en inglés

# Bibliografía


- Manuel J. Prieto, *Desarrollo de juegos con J2ME*, Alfaomega, 1a Edición, Abril 2005.
- Morrison Michael, *Beginning Mobile Phone Game Programing*, Sams, 1a Edición, 2004.
- Digipen Institute of Technology, *MSDN/DigiPen Video Game Development Webcast*, 2005. Página oficial: `http://www.digipen.edu/main/Webcast/Introduction\_to\_2-D\_Video\_Game\_Development`. Consultada el 4 Junio 2008, a las 18:50.
- Foley, van Dam, Feiner, Huges, *Computer Graphics, Principles and Practice*, Addison Wesley, 2a Edición, 1996.
- Hearn Donald, Baker Pauline, *Gráficas por Computadora*, Prentice Hall, 2a Edición, 1995
- Página oficial de J2ME: `http://java.sun.com/javame/downloads/index.jsp` Consultada el 4 Junio 2008, a las 18:51.
- Tutorial de dibujo 2D en J2ME: `http://developers.sun.com/mobility/midp/articles/s2dvg/index.html`. Consultada el 4 Junio 2008, a las 18:52.
- Sitio oficial de desarrolladores de *Sony-Ericsson*: `http://developer.sonyericsson.com/site/global/home/p\_home.jsp`. Consultada el 4 Junio 2008, a las 18:55.
- Sitio oficial de desarrolladores de *Nokia*: `http://www.forum.nokia.com/index.html`. Consultada el 4 Junio 2008, a las 18:55.
- Página oficial de *Brew*: `http://brew.qualcomm.com/brew/en/`. Consultada el 4 Junio 2008, a las 18:57.

- Usos de la Tecnología *Brew*: http://www.developer.com/ws/brew/article.php/1454711. Consultada el 4 Junio 2008, a las 18:57.
- Christophe Quarre, *Trends and Standards for 3D Graphics for Handsets*, State Key Lab of CAD&CG, Zhejiang University, Game Developers Conference 2005.
- Eberly, David H., *3D Game Engine Design*,Morgan Kaufmann Publishers, 1a Edición, 2001
- Guía para motores existentes: http://www.gamemiddleware.org/. Consultada el 4 Junio 2008, a las 18:58.
- Torque Game Engines: http://www.garagegames.com/. Consultada el 4 Junio 2008, a las 18:58.
- CRM32Pro Engine: http://www.megastormsystems.com/sdk/crm32pro.htm. Consultada el 4 Junio 2008, a las 18:59.
- FIFEngine: http://www.fifengine.de/. Consultada el 4 Junio 2008, a las 18:59.
- Herocraft Mobile Dragon Engine: http://www.hitech.herocraft.com/technology.htm. Consultada el 4 Junio 2008, a las 18:52.
- Ideaworks3D Airplay Engine: http://www3.ideaworks3d.com/products2.htm. Consultada el 4 Junio 2008, a las 18:54.
- In-fusio EGE Engine: http://developer.in-fusio.com/. Consultada el 4 Junio 2008, a las 18:57.
- Situación actual de la industria de los videojuegos: http://www.next-gen.biz/index.php?option=com_content&task=view&id=8750&Itemid=2. Consultada el 4 Junio 2008, a las 18:53.
- Descripción del juego. *Puyo Puyo*: http://en.wikipedia.org/wiki/Puyo_Puyo. Consultada el 4 Junio 2008, a las 18:57.
- Historia de los juegos en dispositivos móviles: http://www.next-gen.biz/index.php?option=com_content&task=view&id=2090&Itemid=2&limit=1&limitstart=0. Consultada el 4 Junio 2008, a las 18:58.