UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

DESARROLLO DE VIDEOJUEGOS EN 2D CON LENGUAJE JAVA: 3 EXPERIENCIAS

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

OSCAR TONATIUH ZAMUDIO OCÁDIZ

DIRECTOR DE TESIS:
M. EN C. ALEJANDRO AGUILAR SIERRA

2009

1. Datos del alumno
   Zamudio
   Ocádiz
   Oscar Tonatiuh
   044 55 37 26 89 29
   Universidad Nacional Autónoma de México
   Licenciatura en Ciencias de la Computación
   099506818

2. Datos del tutor
   M. en C.
   Aguilar
   Sierra
   Alejandro

3. Datos del sinodal 1
   M. en C.
   Ibargûengoitia
   González
   María Guadalupe Elena

4. Datos del sinodal 2
   L. en C.C.
   Aldazosa
   Mariaca
   Mauricio

5. Datos del sinodal 3
   L. en C. C.
   Bautista
   Garcia Cano
   Gildardo

6. Datos del sinodal 4
   L en C. C.
   García
   Pasquel
   Jesus Adolfo

7. Datos del trabajo escrito
   Desarrollo de videojuegos en 2D con lenguaje Java
   3 experiencias
   101 p
   2009

A mi madre y mi padre
A mi tia Estela y mi tio Juan

Índice general

Introducción

Los videojuegos son, en la actualidad, un elemento común y corriente en la actividad humana. La internet, los teléfonos celulares y las consolas hacen que la mayoría de las personas estén expuestas al contacto con ellos. La industria del videojuego es en la actualidad un negocio millonario, con ganancias comparables a las de la industria cinematográfica. Los distintos actores económicos consideran los posibles usos que pueden dar a este tipo de software, y requieren de profesionales capacitados para elaborarlos.

Este documento presenta al lector 3 experiencias en el desarrollo de videojuegos utilizando el lenguaje de programación Java, en las cuales ha participado el autor:

1. Un clon del clásico *pong* cuyo objetivo era ser utilizado en terapia de rehabilitación en sobrevivientes a una apoplejía[1].

2. Un laberinto en 2D que sería utilizado para promover la lucha contra el tabaquismo[2], juego que lleva por título *¿¿¿Tú decides si fumas???* y cuyo sitio en Internet es `http://borges.dgsca.unam.mx/juegoTabaquismo`, elaborado para el DPID-SERUNAM-DGSCA[3].

3. Una segunda versión de *¿¿¿Tú decides si fumas???*, elaborada utilizando un motor de juego, la cual fue desarrollada con el objetivo de probar

[1]El juego fue desarrollado para el proyecto CONACYT *Computer Game Motivating Rehabilitation With Objective Measures Of Improvement In Motor Function*, a cargo del Dr. Ronald Leder

[2]El juego fue elaborado para el *Programa de Prevención del Tabaquismo en Mujeres Adolescentes* de la Facultad de Psicología de la UNAM, a cargo de la Mtra. Nazira Calleja Bello

[3]Departamento de Productos Interactivos para la Docencia de la Subdirección de Servicios Educativos en Red de la Dirección General de Computo Académico de la UNAM

las capacidades del motor de juego, con el objetivo de incorporarlo a futuros desarrollos del DPID-SERUNAM-DGSCA.

Respecto a cada videojuego se proporcionan detalles de diseño del software, los problemas más serios o de solución más difícil encontrados durante el desarrollo y las soluciones dadas a ellos. Con ello se proporciona al lector una serie de experiencias de primera mano en los problemas que conlleva el desarrollar este tipo de software en el lenguaje Java, el cual es utilizado en buena parte de los cursos de la carrera y en la actividad profesional. La experiencia que el autor ha adquirido al desarrollarlos será útil a otros programadores al familiarizarlos con las dificultades que enfrentarán al desarrollar videojuegos, y al advertirles sobre los problemas que distintos errores de los 3 desarrollos llevaron aparejados, atribuibles al lenguaje Java o al autor. Se trata de problemas reales encontrados por el autor al hacer uso de características, clases y funciones del lenguaje Java, y que reciben una atención marginal o nula en la documentación consultada al respecto, debido a que las aplicaciones que las utilizan no son lo suficientemente complejas para tener estos problemas, o a que estos problemas son particulares del desarrollo de videojuegos.

El desarrollo de *pong* comenzó en un seminario de la Facultad, como proyecto final elaborado en conjunto por la clase. Al finalizar el seminario el juego aun no contaba con toda la funcionalidad requerida para su empleo en terapia, por lo que el autor fue invitado a continuar participando en su desarrollo como becario. *pong* fue la primera experiencia del autor en el desarrollo de videojuegos. El juego no utiliza código de otros proyectos a excepción del incluido en el el Java Software Development Kit. La implementación sigue el patrón modelo-vista-controlador.

¿¿¿Tú decides si fumas??? fue desarrollado a lo largo del primer semestre de 2007 como parte de las obligaciones laborales del autor en la DGSCA. Personal de la DGSCA proveyó los elementos gráficos y las ideas respecto al juego y sus objetivos. El autor fue el único programador involucrado en el proyecto. La estructura del juego es más complicada que la de *pong*, puesto que los elementos con los que el jugador debe interactuar son mucho mas numerosos, lo que complica la detección de colisiones. Un hilo de ejecución aparte se encarga del movimiento del enemigo.

La segunda versión de *¿¿¿Tú decides si fumas???* fue desarrollada de Diciembre de 2008 a Febrero de 2009, utilizando un motor de juegos llamado

JIG[4]. Este desarrollo obedecía un doble propósito:

- Integrar a este documento una experiencia de primera mano en el uso de un motor de juego.
- Explorar las posibilidades de uso y adaptación de un motor de juego para otros proyectos del DPID-SERUNAM-DGSCA

El tiempo de desarrollo de esta segunda versión de *¿¿¿Tú decides si fumas???* fue sustancialmente menor al empleado al desarrollar la primera, dado que todos los elementos gráficos ya estaban elaborados, aun cuando en el mismo lapso el autor debió familiarizarse con la API de JIG.

El primer capítulo presenta una panorámica del desarrollo de juegos en Java, lenguaje utilizado para la elaboración de los juegos que son objeto de este documento. Se presenta una breve historia del uso de Java en el desarrollo de videojuegos y se exponen las características favorables y desfavorables de Java para su uso en el desarrollo de videojuegos que han sido consideradas en la literatura revisada sobre el tema.

El segundo capítulo familiariza al lector con los elementos de programación Java utilizados en el desarrollo de los 3 videojuegos:

1. Concurrencia.
2. Graficación, imágenes y animación.
3. Interacción con el usuario.
4. Sonido.
5. RMI.

El tercer capítulo ofrece una panorámica general de *Java Therapy* un sistema que ofrece terapia en linea a sobrevivientes de apoplejía. El desarrollo de *Pong* estuvo fuertemente influenciado por este sistema, así que reseñarlo cumple la doble función de:

1. Mostrar un ejemplo de un juego "serio" elaborado en Java, en este caso uno de la vasta cantidad de proyectos que conjugan videojuegos y terapia de rehabilitación de movimientos.

[4]Java Instructional Gaming

2. Mostrar el tipo de sistema en que *Pong* está basado y que intentaba mejorar.

El tercer capítulo trata sobre *Pong*, el primero de los juegos desarrollados. De forma análoga el cuarto capítulo reseña *¿¿¿Tú decides si fumas???* El quinto capítulo da cuenta del desarrollo de *¿¿¿Tú decides si fumas???* utilizando como base un motor de juego, con el objetivo de mostrar a los desarrolladores de videojuegos las ventajas y desventajas de este tipo de software. Finalmente, se dan las conclusiones del trabajo.

Resumen

Este documento describe 3 videojuegos en 2D desarrollados por el autor utilizando el lenguaje Java:

1. Un clon del clásico juego *pong*, desarrollado para ser utilizado en terapia de rehabilitación por personas con problemas motrices en las extremidades superiores. Múltiples parámetros del juego son configurables (p.ej. velocidad y tamaños de raquetas y pelota, dificultad del juego), puede grabar y reproducir los juegos jugados y tiene una modalidad para 2 jugadores a través de una LAN.

2. *¿¿¿Tú decides si fumas???* un laberinto utilizado para promover la lucha contra el tabaquismo entre niñas en un proyecto en el que colaboró el Departamento de Productos Interactivos para la Docencia de la Subdirección de Servicios en Red de la Dirección General de Servicios de Computo Académico de la UNAM. La jugadora debe recolectar una serie de "expedientes" evitando ser capturada por un policía que recorre el laberinto.

3. Otra versión de *¿¿¿Tú decides si fumas???*, esta vez realizada utilizando el *Java Instructional Gaming*(JIG), un motor de juego desarrollado en colaboración por Washington State University Vancouver y University of Puget Sound, con el objetivo de probar las características y posibilidades de uso de dicho motor de juego.

Respecto a cada videojuego se proporcionan detalles de diseño del software, los problemas más serios o de solución más difícil encontrados durante el desarrollo y las soluciones dadas a ellos. Se trata de problemas reales encontrados por el autor al hacer uso de características, clases y funciones del lenguaje Java, y que reciben una atención marginal o nula en la documentación consultada al respecto, debido a que las aplicaciones que las utilizan no son lo

suficientemente complejas para tener estos problemas, o a que estos problemas son particulares del desarrollo de videojuegos.

Capítulo 1

Java y el desarrollo de videojuegos

1.1. Una definición de videojuego

Si bien el concepto de videojuego es intuitivamente claro para la mayoría de las personas, no está fuera de lugar, en aras de la claridad, intentar una definición del mismo. Esta debe ser lo suficientemente amplia como para abarcar los numerosos y diversos videojuegos que existen en la actualidad. Una definición clara y a la mano es la que hace Marner:

> "Un videojuego es un programa de computadora cuyo objetivo principal es entretener al usuario.
>
> Esta definición tiene cierta ambigüedad pero es útil ya que caracteriza al videojuego típico.
>
> Desde el punto de vista del programa, un videojuego es una simulación interactiva en tiempo real, posiblemente distribuida. El programa crea un modelo de un mundo virtual, y este modelo es actualizado y visualizado en tiempo real según la dinámica del modelo y la interacción con el usuario o usuarios.
>
> El desarrollo de videojuegos no se considera un campo de las ciencias computacionales. Es, no obstante, una aplicación de las ciencias computacionales que hace uso de muy diversos campos, incluyendo graficación, audio, interacción humano-computadora, sistemas operativos, diseño de compiladores, algoritmos, redes y

sistemas distribuidos, ingeniería de software, diseño de software, inteligencia artificial y otras áreas"[1].

1.2. Justificación pedagógica de la utilidad del desarrollo de videojuegos

Algunas de las razones que hacen a los proyectos de desarrollo de videojuegos mecanismos ideales para el aprendizaje de Ciencias de la Computación son[2]:

- El desarrollo de videojuegos despierta interés entre los estudiantes, quienes, en su mayoría, juegan regularmente, lo cual los motiva en alto grado a trabajar en los proyectos. Dado que los estudiantes están familiarizados con diversos videojuegos, saben que tipo de funcionalidades son necesarias o útiles, y adoptan una postura crítica respecto a su propio trabajo.

- Los videojuegos incorporan un amplio espectro de técnicas de las ciencias computacionales y las matemáticas, de manera que su desarrollo puede constituir una buena motivación para aprender y aplicar técnicas que de otra manera no se tratarían en la currícula o cuyo uso resulta artificial o forzado, como diseño e implementación de despliegue gráfico interactivo en tiempo real, diseño eficiente de estructuras de datos, manejo eficiente de memoria, creación de interfaces de usuario intuitivas, uso de redes y concurrencia, acceso a grandes conjuntos de datos, manejo eficiente de los recursos del sistema, inteligencia artificial, etc. Puesto que todos estos elementos deben funcionar e integrarse adecuadamente, los estudiantes ven así una motivación práctica para aprender seriamente los fundamentos de las ciencias computacionales.

- Permiten al estudiante ejercitar su creatividad en una forma poco común en los demás proyectos de materias de la currícula de ciencias computacionales.

Todo ello sugiere que el relacionar la currícula con los videojuegos puede ser altamente benéfico para la formación de los estudiantes de Ciencias de la

[1] [Marner2002] pag. 7

[2] [Sweedyk2005] y [Wallace2006] proporcionan disertaciones más extensas al respecto.

Computación en particular, y de programadores en general, al punto de que en los últimos años distintas universidades estadounidenses y de otros países han añadido tópicos sobre videojuegos a sus currículas, cuando no cursos específicos[3].

1.3. Historia del uso de Java en el desarrollo de videojuegos

Antes de 1990 la mayoría de los videojuegos se desarrollaban en ensamblador, con el objetivo de maximizar su desempeño. Durante la década 1990-2000 el lenguaje C se volvió popular en la industria del videojuego, gracias a su elevada productividad y a las continuas mejoras en el hardware y los compiladores que mejoraron en forma constante su desempeño. El parteaguas en este viraje del ensamblador hacia C es *Doom*, desarrollado en 1993 por ID Software, el cual utiliza el lenguaje C para la mayoría del código del juego. Posteriormente se fue adoptando C++, cuya orientación a objetos mejoró la productividad, reusabilidad y mantenibilidad del código con poco impacto en el desempeño final del juego[4].

1.3.1. Primeros pasos: Applets

En 1996 fue lanzado Java 1.0. Siguieron las versiones 1.1 a comienzos de 1997, y 1.2 (Java 2) a finales de 1998. Durante este periodo ocurrió un verdadero *boom* en el uso de Java, principalmente debido a los *applet*s. Un *applet* es un tipo particular de programa Java que puede ser descargado a través de Internet y ejecutarse en navegadores de Internet compatibles con esta tecnología. Un applet puede empotrarse en una página HTML de manera similar a como se empotra una imagen. Cuando estas páginas son vistas en un navegador que cuenta con el plugin de Java, el código del applet es transferido al navegador y ejecutado por la máquina virtual de Java del navegador. Gracias al uso de los applets, Java fue promocionado como el lenguaje ideal para crear aplicaciones distribuibles por Internet, lo cual se vio reflejado en un número elevado de juegos desarrollados como applets durante el periodo 1996-98. Desafortunadamente las primeras versiones de Java eran muy lentas,

[3] Véase [Wallace2006]
[4] Cf. [Wang2007]

lo que se tradujo en experiencias de juego desilusionantes. Aunado a esto, muchos programadores ignoraron los retrasos que el usuario final experimenta al descargar de la red grandes cantidades de código, imágenes, audios, etc. Las restricciones de seguridad inherentes a los applets fueron otro problema. Todo lo cual dejó en los programadores la percepción generalizada de que Java era un lenguaje no apto para el desarrollo de videojuegos [5].

1.3.2. AWT y Swing

El primer mecanismo para la creación de GUI's en Java es el Abstract Window Toolkit(AWT). El AWT ofrece una serie de componentes(*widgets*) cuya apariencia y comportamiento (*look-and-feel*) es delegado al sistema operativo anfitrión[6], de suerte que los widgets del AWT se ven diferentes según el sistema operativo que los ejecute. Los widgets AWT no cuentan con componentes con funcionalidades más complejas comunes en los sistemas operativos modernos(p.ej. árboles y tablas), así como de capacidades gráficas avanzadas(p.ej. relleno con patrones y administración de colores).

Para solventar estas carencias, Sun comenzó en 1998 el desarrollo de Java Foundation Classes(JFC), que incluye 2 proyectos de graficación y desarrollo de GUI's: Swing y Java2D. Swing proporciona una serie de widgets cuyo look-and-feel no es delegado al sistema operativo anfitrión, de manera que sus componentes se dibujan igual en cualquier sistema operativo y pueden proporcionar funcionalidades más complejas(como los árboles y tablas de los que el AWT carecía). Java2D ofrece a los programadores una API para funciones gráficas avanzadas, manipulación gráfica de texto e impresión.

1.3.3. Java3D

Java3D es una API de alto nivel para la creación, dibujo y manipulación de escenas 3D con geometrías, materiales, luces, sonidos, etc. Cuenta con implementaciones en OpenGL y DirectX. Java3D es una API basada en gráfico de escena(scene graph): el programador organiza la escena como un gráfico y deja los detalles del dibujo de la misma en manos de la biblioteca. No es parte del JDK6, pero se espera que sea incluido en futuras versiones del mismo.

---

[5] Cf. [Davison2003]

[6] Los componentes cuyo look-and-feel es delegado al sistema operativo anfitrión son designados *pesados(heavyweight)* en Java.

Su desarrollo inicial se realizó de forma conjunta por Intel, Silicon Graphics, Apple y Sun. La primera versión apareció en 1998. Desde mediados de 2003 hasta el verano de 2004 su desarrollo fue descontinuado, fecha en que fue convertida en un "community source project". En buena medida debido a la pausa en su desarrollo en el periodo 2003-4 existen numerosos proyectos para graficación 3D en Java alternativos.[7]

[Marner2002] cita entre sus ventajas la reducción de costos, portabilidad entre DirectX y OpenGL y soporte a dispositivos especializados de despliegue 3D. Entre sus desventajas se encuentra su bajo desempeño y su falta de soporte a las características más avanzadas y especializadas del hardware 3D, debidas al hecho de que Java3D esta diseñada pensando en el común denominador de funcionalidades 3D del hardware y controladores.[8]

1.3.4. Motores de juego

Un motor de juego es un sistema de software diseñado para la creación y desarrollo de videojuegos. Proveen funcionalidades necesarias a los juegos, tales como graficación en 2D/3D, motores físicos, detección de colisiones, sonido, guión, animación, inteligencia artificial, red, manejo de memoria, multihilos y gráfico de escena. Fomentan la reutilización de funcionalidades al desarrollar distintos videojuegos y, por tanto, hacen menos oneroso el desarrollo. Su aparición y uso sistematiza el desarrollo de videojuegos, y es, por tanto, una señal clara de plena madurez del desarrollo de juegos en Java. El artículo de la wikipedia[9] cita 3 motores de juego (jMonkeyEngine, Jogre, Lightweight Java Game Library), proyectos que datan de 2003, 2004 y 2002, respectivamente. Los comienzos de JIG datan de 2004.

1.3.5. Java en dispositivos móviles

En la actualidad existen varias tecnologías Java enfocadas a dispositivos móviles. La plataforma Java para móviles se llama *Java 2 Plataform Micro Edition* (J2ME). Dicho término designa a todas las tecnologías Java para dispositivos móviles. Con más de 250 millones de dispositivos[10] que la implementan, J2ME es la plataforma dominante en el desarrollo de software

[7] [Java3D] proporciona una lista de algunos de ellos.

[8] [Marner2002] pag. 80

[9] `http://en.wikipedia.org/wiki/List_of_game_engines`

[10] [Heiss2004]

para dispositivos móviles. Este dominio de Java en el desarrollo de aplicaciones para móviles, principalmente teléfonos celulares, comenzó con MIDP 1.0 (Mobile Information Device Profile) hacia el año 2000. MIDP 1.0 brinda funcionalidades restringidas para videojuegos: cada fabricante tiene extensiones propietarias e incompatibles entre si al estándar MIDP, las funciones de red están restringidas a HTTP (único protocolo requerido por el estándar), y diversas características especificas de cada dispositivo no son accesibles al programador (p.ej. las interfaces Bluetooth e infrarroja). Por ello la segunda versión del perfil, MIDP 2.0, provee numerosas mejorías, como son funciones específicas para el desarrollo de videojuegos (una API especializada en ello, *javax.microedition.lcdui.game*), funciones de sonido y red mejoradas, y mejoras en el despliegue gráfico. Desde MIDP 1.0 la proliferación de Java en el campo de los móviles fue gigantesca, y se desarrollaron cientos de juegos (comerciales y no comerciales, libres y propietarios) utilizando J2ME. Algunos títulos comerciales que utilizan dicha plataforma son[11]: *Splinter Cell: Pandora Tomorrow*, *XIII*, *Prince of Persia*, *Tony Hawk's Pro Skater 3D edition*, *Samurai Romanesque*, *Pursuit Squad,3D Golf*, *Ancient Empires*, *Poker Solitarire* y *Shade*.

1.4. Prejuicios y problemas del uso de Java en el desarrollo de videojuegos

Entre los programadores existen numerosos prejuicios respecto al uso de Java en el desarrollo de videojuegos:

La imposibilidad de uso en juegos de alto nivel Como ya se mencionó, Java ganó en sus comienzos la reputación de ser un lenguaje no apto para el desarrollo de videojuegos.[12] Sin embargo los desarrolladores se han dado cuenta con el transcurso del tiempo de que esta afirmación es, en términos generales, falsa. La mejor prueba de ello es que existen numerosos juegos de alto y bajo perfil que están siendo desarrollados

[11] [Heiss2004] y [Barbagallo2004] pp. 12-15.

[12] Un ejemplo ilustrativo es el comentario de Robert Huebner, desarrollador de "Vampire: the Masquerade – Redemption": "[...]yo conocía muy poco acerca de Java, y lo consideraba únicamente como un lenguaje para hacer bailar íconos en una página web y cosas así" [Huebner2000](traducción mía).

en Java. Véase más adelante la lista de juegos comerciales y los sitios que ofrecen juegos elaborados en Java.

Los problemas de Swing y Java2D El problema con Swing y Java2D es su pobre desempeño respecto al AWT. Con el fin de hacer frente a este problema, Sun comenzó a utilizar características especializadas del hardware mediante DirectX(para Java 1.4) y OpenGL(a partir de la v.1.5 del lenguaje, pero que cristalizaría en una versión realmente utilizable solo hasta la v.1.6). Esto mejoró notablemente el desempeño, pero acarreó otros problemas: para beneficiarse de estas mejoras los usuarios de applets tienen que acceder directamente el panel de control de Java, tarea poco conveniente y que requiere de permisos para modificar propiedades de todo el sistema. Además estas modificaciones alteran a toda la JVM, y no solo a la aplicación que requiere estas mejoras de desempeño[13]

Para solucionar el problema del pobre desempeño de Swing se han intentado distintos tipos de soluciones:

- La creación de conjuntos de widgets alternativos a AWT/Swing con mejor rendimiento, como el Standard Widget Toolkit de IBM. Si bien estos conjuntos de widgets pueden solucionar algunos problemas de rendimiento, no utilizan a toda su capacidad las características especializadas del hardware gráfico, por lo que no resultan eficientes para el desarrollo de juegos de alto nivel [14]
- La creación de API's que incorporan DirectX u OpenGL al diseño de GUI's en Java, como DirectJ, GL4Java(OpenGL for Java), JOGL(Java binding for OpenGL API) y LWJGL(Lightweight Java Game Library)[15]. Este tipo de soluciones tiene, a su vez, sus propios problemas:
  - Carecen de sus propios widgets, de suerte que para su uso real en videojuegos los programadores se ven obligados a recurrir a conjuntos de widgets nativos(p.ej. Win32) con lo cual el desempeño final del sistema se ve reducido al de los widgets nativos en el mejor de los casos. Además el hecho de que los

[13] Véase [Wang2007] pag.729
[14] [Wang2007] pag.729
[15] [Wang2007] pag.729

conjuntos de widgets nativos controlan sus propios tiempos de pintado y procesamiento trae consigo problemas difíciles de resolver, como el "parpadeo"(flickering).

- La mayoría de estos vínculos a API's nativas están disponibles únicamente para Java 2 y posteriores. Desafortunadamente la implementación de Microsoft(la MSV - Microsoft Virtual Machine) es Java v.1.1.4. Dado su extenso número de usuarios[16], el resultado final es que estas API's no son accesibles a todos los usuarios de Java.

La lentitud respecto a C/C++ Los programas hechos en Java son mas lentos que los programas hechos en C o C++. Esto es innegable, y se debe a la naturaleza interpretada del lenguaje, que no permite el obtener código tan eficiente como un compilador, y a su ejecución en una máquina virtual. Sin embargo las diferencias de velocidad entre Java y C/C++ han disminuido drásticamente, desde ser 20-40 veces más lento [17] hasta 1.2-1.5[18]. Davison cita ejemplos de benchmarks en los que el desempeño de código Java resultó incluso superior a C++[19], pero estos resultados dependen de los compiladores y JVMs involucrados. Los "puntos débiles" de Java en cuanto a velocidad se refiere son:

- El dibujo de la interfaz gráfica con Swing, que añade una "capa" extra de procesamiento entre el sistema de ventanas del sistema operativo anfitrión y el programa. Por ello muchos videojuegos elaborados en Java utilizan únicamente AWT. Dadas las modestas necesidades de un videojuego en términos de GUI, esto normalmente no representa un problema.
- El recolector de basura, el cual es ejecutado automáticamente por la JVM, pudiendo causar pequeños retrasos indeseados que son bastante notables en un videojuego. Este inconveniente puede solucionarse si el programador tiene un buen estilo de programar,

[16]El porcentaje de usuarios utilizando la MSV sería cercano al 30 %. Según datos de Gray, A. *GC usage statistics*; http://andrew-gray.com/dist/stats.shtml, citado por [Wang2007]

[17]En las versiones del lenguaje Java previas a 1.2

[18]En Java 4, la versión con la que Marner realizó sus *benchmarks* ([Marner2002] pags. 33 y 35). Probablemente estos números serán aún menores actualmente.

[19][Davison2003] pag.6

evitando la creación de muchos objetos temporales, y sugiriendo a la JVM llamar al recolector en los momentos en que su ejecución no cause retrasos notables al usuario.

El consumo elevado de memoria Java consume una gran cantidad de memoria RAM. Esto se debe a 2 razones:

1. La JVM y sus bibliotecas básicas deben ser ubicadas en RAM, lo que consume unos 5-10 Mb.
2. Los objetos Java se almacenan en el heap y no en el stack.

Si bien esto es un consumo de memoria elevado, no representa un problema serio dados los tamaños más comunes de RAM actuales. Estos niveles de consumo de memoria eran un problema para la penúltima generación de consolas (PS2 con sus 32 Mb de RAM, XBox con sus 64 Mb de RAM), no así para la última generación (XBox 360 con 512 MB, PS3 con 256 MB RAM).

1.5. Ventajas del uso de Java en el desarrollo de videojuegos

Productividad elevada Utilizando Java los programadores son mucho mas productivos que con C/C++. Entre las razones para ello cabe citar[20]:

- La sintaxis de Java es mucho mas simple que la de C++.
- Los programas escritos en Java son menos propensos a errores y son mas fáciles de depurar que los escritos en C/C++.
- Las bibliotecas estándares de Java son mucho más amplias que las de C++.

Confiabilidad y estabilidad Java ha sido usado a lo largo de mas de una década por millones de usuarios, y es considerado un lenguaje sumamente estable y confiable. Sus compiladores tienen muchos menos errores que la mayoría de los compiladores de C/C++.

[20]Para una discusión más amplia al respecto véase el capítulo 5 de [Marner2002]

Documentación adecuada Java tiene una documentación muy amplia que incluye tutores, manuales, especificaciones, foros de discusión y consulta, artículos y libros. La mayoría de estos recursos son proporcionados sin costo por Sun vía Internet, o bien a través de comunidades y sitios en linea también gratuitos.

Amplio soporte Sun lanza una nueva versión del JDK aproximadamente cada 18 meses, añadiendo nuevas características al lenguaje y solucionando problemas del mismo. Además de Sun muchas otras grandes corporaciones – como IBM – lanzan mes con mes múltiples productos relacionados con Java. El programador interesado puede encontrar en Internet muchos lugares donde los programadores comparten experiencias y recursos de Java.

1.6. Juegos elaborados con Java

[Davison2003] menciona varios videojuegos comerciales desarrollados a partir de 1997 que utilizan Java[21]:

- Vampire – the Masquerade: Redemption. Utiliza Java como lenguaje de scripting.
- Runescape. Construido totalmente en Java.
- Law and Order. Construido totalmente en Java.
- Puzzle Pirates. Construido totalmente en Java.
- Chrome. Utiliza Java como lenguaje de scripting.
- Who wants to be a millionaire. Utiliza Java para implementar la lógica de juego.
- You don't know Jack. Utiliza Java para implementar la lógica de juego.
- Majestic. Utiliza Java para implementar la lógica de juego.
- Tom Clancy's Politika. Construido utilizando una mezcla de Java y C++.

[21] [Davison2003] pag.23, [Heiss2004]

- Tom Clancy's ruthless.com. Construido utilizando una mezcla de Java y C++.
- Shadow Watch. Construido utilizando una mezcla de Java y C++.
- IL-2 Sturmovik. Construido utilizando una mezcla de Java y C++.
- Star Wars Galaxies. Utiliza Java para implementar la Lógica del juego.

Así mismo proporciona una lista de sitios web que ofrecen juegos freeware/shareware hechos en java:[22]

- ArcadePod.com, http://www.arcadepod.com/java/
- Java 4 Fun, http://www.java4fun.com/java.html
- Java Game Park, http://javagamepark.com
- Java Games Central, http://www.mnsi.net/ rkerr/
- jars.com, http://www.jars.com/games
- Java Shareware, http://www.javashareware.com

1.6.1. Dos ejemplos

A continuación describiremos con mas detalle dos de los juegos arriba citados con el objetivo de mostrar casos específicos de los prejuicios al respecto, problemas, ventajas y alcances del uso de Java en videojuegos

Vampire – the Masquerade: Redemption

Juego de rol lanzado el 7 de Junio de 2000 por Activision. El juego trata de las aventuras de un cruzado francés, Christof Romuald, a través de un periodo de 800 años por las ciudades de Praga y Viena en las "edades oscuras" y las modernas Londres y Nueva York.

El juego utiliza Java como lenguaje de scripting. El equipo de desarrolladores del juego había previamente creado su propio lenguaje de scripting para un proyecto anterior, y decidieron adoptar Java para no tener que desarrollar nuevamente por cuenta propia el lenguaje de scripting, implementar

[22] [Davison2003] pag. 5

su interprete y empotrarlo en el juego. El principal desarrollador del juego, Robert Huebner, comenta que él mismo tenía serios prejuicios respecto al uso de Java[23]. La experiencia usando Java fue muy exitosa: luego de un curso rápido de Java, los desarrolladores consiguieron, en el curso de unas cuantas semanas, convertir Java en el lenguaje de scripting del motor de 3D-RPG[24]. Posteriormente entrenaron a sus diseñadores en el uso de Java como lenguaje de scripting, y ellos se encargaron de crear los cientos de pequeños guiones que, eventualmente, guían la historia del juego.

El prejuicio respecto al consumo elevado de recursos por Java resultó falso: Java funcionó sin problemas a lo largo de todo el desarrollo, y jamás se convirtió en una perdida considerable de tiempo de procesador o de memoria. En palabras de Robert Huebner:

> "La apuesta rindió frutos. Ahorramos meses de tiempo de programación que de otra forma hubiésemos destinado a crear un ambiente para scripting, y el resultado final fue un sistema significativamente más eficiente y robusto que lo que nosotros hubiésemos podido crear[25]".

Runescape

Runescape[26] es un MMORPG[27] elaborado en Java por Jagex[28]. La interfaz para los clientes consiste en un applet Java, por lo que se puede jugar desde cualquier navegador de Internet que utilice el plugin de Java. Utiliza gráficos en 3D. Con un total de 8.5 millones de cuentas activas, posee el récord Guiness para el MMORPG más popular.

El juego tiene lugar en el mundo de Gielinor, de tema fantástico, dividido en reinos, regiones y ciudades. Los jugadores se desplazan por Gielinor a pie, mediante el uso de hechizos y objetos mágicos, o por medios de transporte mecánicos. Cada región tiene distintos tipos de monstruos, recursos y misiones. Los jugadores aparecen en la pantalla como avatares configurables. El juego no sigue una historia lineal: cada jugador puede escoger sus propias

[23] Véase la nota respecto a los comentarios de Huebner en la página 6.

[24] 3-Dimension Role Playing Game, motor de juego de interpretación de papeles en 3D

[25] Traducido de [Huebner2000]

[26] `http://www.runescape.com`

[27] Massively Multiplayer On-line Role Playing Game, juego multijugador de interpretación de papeles (rol) en linea

[28] Java Games Experts

metas. Los jugadores pueden pelear con monstruos NPC[29], completar misiones o aumentar su experiencia en distintas habilidades. Los jugadores interactúan entre si al comerciar, conversar o participar en minijuegos (combativos o cooperativos) y actividades. El juego nunca termina: nuevas misiones son agregadas al juego cada pocas semanas.

El motor del juego esta totalmente escrito en Java, y es controlado por los desarrolladores de Jagex a través de un lenguaje de scripting de propósito específico, *RuneScript*.

Si bien los gráficos de Runescape no son impresionantes, el juego es muy completo y entretenido, y demuestra que Java es más que adecuado para desarrollar juegos, al dar vida a este basto mundo que proporciona entretenimiento a miles de personas.

[29]Non-player characters, personajes controlados por el sistema

Capítulo 2

Antecedentes de programación para videojuegos en lenguaje Java

Este capítulo proporciona nociones de programación fundamentales para el desarrollo de videojuegos utilizando el lenguaje Java.

2.1. Concurrencia e hilos de ejecución

En programación concurrente existen, básicamente, 2 unidades de ejecución: procesos e hilos (*threads*). En el lenguaje Java la programación concurrente se relaciona normalmente con hilos.

Una computadora tiene, normalmente, muchos procesos e hilos activos. Esto sucede aun en en sistemas con un solo núcleo en el procesador, y que, por tanto, tienen siempre un único hilo en ejecución en un momento cualquiera. El tiempo del procesador es compartido, en estos sistemas, gracias a una característica del sistema operativo, denominada en inglés *time slicing*: un proceso es ejecutado por un pequeño lapso de tiempo (*time slice*). Cuando este lapso, determinado por el sistema operativo, termina, el sistema operativo toma control del procesador, y el proceso deja de ser ejecutado. El sistema operativo selecciona a otro proceso para ser ejecutado, y el ciclo se repite. Los lapsos de tiempo en los que se ejecuta un proceso antes de ser desplazado por el sistema operativo son tan pequeños que, para el usuario, el sistema aparenta estar ejecutando múltiples procesos a la vez.

En la actualidad es común que un sistema de computo tenga muchos procesadores, o procesadores con múltiples núcleos de ejecución. Estos sistemas en realidad ejecutan en forma concurrente los procesos e hilos, pero la concurrencia es posible aun en sistemas con un único procesador o núcleo de ejecución.

2.1.1. Procesos

Un proceso es un ambiente de ejecución autocontenido. Normalmente, un proceso tiene asociado un conjunto (completo y privado) de recursos de tiempo de ejecución, particularmente espacio de memoria. Normalmente se considera el término "proceso" como sinónimo de programa o aplicación. Sin embargo, lo que el usuario ve como una aplicación individual puede tratarse en realidad de un conjunto de procesos cooperando. Para permitir la comunicación entre procesos, la mayoría de los sistemas operativos proveen recursos de comunicación entre procesos (IPC, siglas en inglés de Inter Process Communication), como *sockets* y *pipes*. IPC es usado también para comunicar procesos que se hallan en distintos sistemas.

La mayoría de las implementaciones de la JVM se ejecutan como un proceso único. Una aplicación Java puede crear procesos adicionales utilizando un objeto *ProcessBuilder*, pero este no es el tipo de concurrencia que normalmente se utiliza en un videojuego.

2.1.2. Hilos

Los hilos o *threads* son conocidos también como procesos ligeros. Tanto procesos como hilos proveen un ambiente de ejecución, pero el crear un hilo requiere menos recursos que el crear un proceso.

Los hilos existen dentro de un proceso. Cada proceso tiene el menos un hilo de ejecución. Los hilos comparten los recursos del proceso al que pertenecen, incluyendo memoria y archivos abiertos. Esto estimula el uso eficiente de los recursos, pero da origen a potenciales problemas de comunicación.

La ejecución de múltiples *threads* es una característica esencial de la plataforma Java. Cada aplicación tiene al menos un hilo de ejecución - en realidad, varios, si se considera a los hilos del sistema que realizan tareas como la administración de memoria y señales. Pero, desde el punto de vista del programador, todo inicia con un único hilo de ejecución, el hilo principal

o *main thread*. A partir de allí el programador puede crear hilos de ejecución adicionales.

Para crear e iniciar un nuevo thread en Java, simplemente cree un ejemplar de la clase *Thread* y llame el método *start*:

```
Thread myThread = new Thread();
myThread.start();
```

Este código no realiza ninguna acción útil: la JVM crea el nuevo hilo, y llama el método *start* del mismo, el cual hace que el nuevo hilo se ejecute (método *run*). Puesto que, por omisión, el método *run* no hace nada, el hilo muere. Para evitar esto, hay 3 mecanismos básicos para dar al método *run* una labor:

- Extender la clase *Thread*.
- Implementar la interfaz *Runnable*.
- Usar clases anónimas.

Extender la clase *Thread*

Esta es una forma rápida de dar a un hilo una tarea: simplemente se extiende la clase *Thread* y se sobrescribe el método *run*:

```
public class MyThread extends Thread {

    public void run() {
        System.out.println("En esta parte realizamos alguna
            tarea útil.");
    }
}
```

Y, posteriormente, creamos e iniciamos el hilo:

```
Thread myThread = new MyThread();
myThread.start();
```

Implementar la interfaz *Runnable*

La mayoría de las veces resulta poco práctico crear una clase simplemente para correr un nuevo hilo. Por ejemplo, puede ser necesario que una clase extienda a otra y que además deba ejecutarse en su propio thread. Para estos casos, la solución es implementar la interfaz *Runnable*:

```
public class MyClass extends SomeOtherClass implements Runnable {

    public MyClass() {
        Thread thread = new Thread(this);
        thread.start();
    }

    public void run() {
        System.out.println("En esta parte realizamos alguna
            tarea útil.");
    }
}
```

En este ejemplo, el objeto de tipo *MyClass* crea e inicia un nuevo hilo en su constructor. La clase *Thread* toma, en su constructor, un *Runnable* como parámetro, y ese *Runnable* es ejecutado una vez que el hilo es iniciado (método *start*).

Usar clases anónimas

Cuando ninguna de las anteriores opciones es viable, se pueden usar clases anónimas para iniciar un nuevo hilo:

```
new Thread() {
    public void run() {
        System.out.println("Realizar una tarea útil aquí.");
    }
}.start();
```

Esta forma de código se puede volver ilegible con facilidad, por lo que debe ser usada únicamente para tareas muy sencillas.

Esperando a que un hilo finalice

Para hacer que la ejecución del hilo actual se detenga hasta que otro hilo termine de ejecutarse, use el método *join*:

```
myThread.join();
```

Dormir un hilo

En ocasiones se puede necesitar que un hilo se detenga por un periodo, lo cual puede lograrse utilizando el método estático *sleep*:

```
Thread.sleep(1000);
```

Esto causa que el hilo de ejecución actual duerma por 1000 milisegundos. Un hilo dormido no consume ciclos de ejecución del CPU.

2.1.3. Sincronización

Una vez que existe más de un hilo de ejecución y estos hilos acceden en forma concurrente a los mismos objetos y variables, se presentan problemas de sincronización. Ilustraremos la situación con un ejemplo. Supongamos que un juego utiliza el siguiente código:

```
public class Maze {

    private int playerX;
    private int playerY;

    public boolean isAtExit() {
        return (playerX == 0 && playerY == 0);
    }

    public void setPosition(int x, int y) {
        playerX = x;
        playerY = y;
    }
```

E imagine el siguiente escenario de ejecución de 2 hilos, A y B:

1. Inicialmente, las variables *playerX* y *playerY* valen ambas 0.

2. El hilo A hace la llamada *setPosition(0,1)*.

3. La linea *playerX=x* es ejecutada, de forma que *playerX* vale *0*.

4. El hilo A deja de ejecutarse pues su *time slice* se ha agotado, y el hilo B comienza a ejecutarse.

5. El hilo B hace la llamada *isAtExit()*.

6. Puesto que *playerX* y *playerY* valen ambas 0, el método devuelve el valor *true*, lo cual es erróneo.

Este comportamiento erróneo se debe a que los métodos *isAtExit* y *setPosition* no deben ser ejecutados en forma simultanea. Esto se puede lograr sincronizándolos, mediante el uso de la palabra reservada *synchonized*:

```
public class Maze {

    private int playerX;
    private int playerY;

    public synchronized boolean isAtExit() {
        return (playerX == 0 && playerY == 0);
    }

    public synchronized void setPosition(int x, int y) {
        playerX = x;
        playerY = y;
    }
}
```

Cuando la JVM ejecuta un método *synchronized*, se adquiere un cerrojo (*lock*) sobre el objeto. Cada objeto posee únicamente un cerrojo, y este no es abierto hasta que la ejecución del método termina, ya sea normalmente o debido a una excepción. De forma que si un método *synchronized* tiene el cerrojo, ningún otro método *synchronized* puede ejecutarse hasta que el cerrojo sea abierto.

Además de la sincronización de métodos, existe la sincronización de objetos, que permite utilizar a cualquier objeto como cerrojo. En realidad la

sincronización de métodos es un caso particular de la sincronización de objetos, en la cual se utiliza como cerrojo el ejemplar *this*. Un ejemplo de sincronización por objeto equivalente al anterior método *setPosition* es:

```
public void setPosition(int x, int y) {
    synchronized(this) {
        playerX = x;
        playerY = y;
    }
}
```

La sincronización por objetos es útil cuando se requiere de más de un cerrojo, cuando se requiere el cerrojo de un objeto que no es *this* o cuando no es necesario sincronizar un método en su totalidad.

Cualquier objeto puede ser usado como cerrojo, únicamente no es permitido utilizar como cerrojos a valores primitivos.

Si bien es necesario sincronizar los accesos de 2 o más hilos a variables u objetos compartidos, no se debe sincronizar más código del necesario, pues esto crea latencias innecesarias a los hilos involucrados. Veamos algunos casos:

- Cuando no se requiere sincronizar todo un método, sino únicamente secciones críticas del mismo. Un ejemplo de como lograr esto es:

  ```
  public void myMethod() {
    synchronized(this) {
        // código crítico que requiere sincronización
    }
    // código que no requiere sincronización
  }
  ```

- No es necesario sincronizar métodos que únicamente utilizan variables locales, pues estas son almacenadas en el stack, el cual es independiente para cada hilo de ejecución. Un ejemplo de este tipo de código es:

  ```
  public int square(int n) {
      int s = n * n;
      return s;
  }
  ```

2.1.4. Deadlock

Ocurre un *deadlock* cuando 2 o más hilos se "atascan" por estar esperando unos a otros. P.ej:

1. El hilo A adquiere el cerrojo 1.
2. El hilo B adquiere el cerrojo 2.
3. El hilo A espera a que el cerrojo 2 sea abierto.
4. El hilo B espera a que el cerrojo 1 sea abierto.

Cada hilo está esperando un cerrojo que el otro hilo posee, de forma que están "atascados" y ocurre *deadlock*. Esta es una situación que se puede dar en cualquier momento cuando más de un hilo de ejecución está adquiriendo cerrojos.

2.1.5. Comunicación entre hilos: *wait* y *notify*

El método *wait* puede ser utilizado dentro de bloques *synchronized*. Causa que el hilo actual abra el cerrojo y espere a que otro hilo lo notifique, mediante una llamada al método *notify*.

El método *notify* también puede ser utilizado dentro de un bloque *synchronized*. Este método notifica a un hilo que esté esperando en el cerrojo actual. Si existe más de un hilo esperando, uno es escogido al azar. En el siguiente ejemplo:

```
// Thread A
public synchronized void waitForMessage() {
    try {
        wait();
    }
    catch (InterruptedException ex) { }
}

// Thread B
public synchronized void setMessage(String message) {
    ...
    notify();
}
```

Una vez que el hilo B llama al método *notify* y sale del método *setMessage* (abriendo el cerrojo), el método A adquiere el cerrojo y puede finalizar la ejecución del código *waitForMessage*.

Es posible esperar únicamente por un periodo definido, indicándolo como parámetro al método *wait*:

```
wait(100);
```

Si el hilo no es notificado, la anterior llamada es equivalente a poner a dormir al hilo por 1000 milisegundos. No existe manera de saber si un hilo salió de la espera porque fue notificado o porque su tiempo de espera se agotó.

El método *notifyAll* notifica a todos los hilos esperando por un cerrojo, en lugar de notificar a un solo hilo.

2.1.6. Hilos de ejecución en Java y el manejo de eventos

En toda aplicación gráfica de Java existen al menos 2 hilos de ejecución: el de la aplicación del usuario, y el hilo de despacho de eventos del AWT, que se encarga de manejar la interacción con el usuario: clics y movimientos del ratón, eventos del teclado, redimensionamiento de ventanas, etc.

2.2. Gráficos y animación

Existen 3 formas de crear juegos basados en gráficos en Java:

Applets Un applet es un tipo de aplicación Java que puede ejecutarse en un navegador de Internet. El principal beneficio que se obtiene al programar applets es que el usuario no requiere instalar ningún software, salvo el plugin de Java para su navegador. La desventaja es que los jugadores deberán estar en linea y ejecutando el navegador para poder utilizar el juego, además de que los applets tienen muchas restricciones de seguridad para evitar que código malicioso distribuido por Internet pueda causar daño a los usuarios. P.ej. los applets no pueden escribir al disco duro, y pueden establecer conexiones de red únicamente al servidor que las alberga.

Juegos que se ejecutan dentro de una ventana Este tipo de juegos no padecen las restricciones de seguridad que tienen los applets. Tienen

los mismos elementos que cualquier otra aplicación en un sistema de ventanas: títulos, botones para cerrar, maximizar, minimizar, etc. Estos elementos pueden resultar distractores cuando se desea que el jugador tenga una experiencia inmersiva total en el juego.

Juegos en pantalla completa Los juegos de pantalla completa eliminan todos los elementos de escritorio de la aplicación, tales como barras de título y desplazamiento, botones para cerrar, maximizar, minimizar, etc., dando completo control al programador sobre la presentación visual del juego.

2.2.1. El modo de pantalla completa

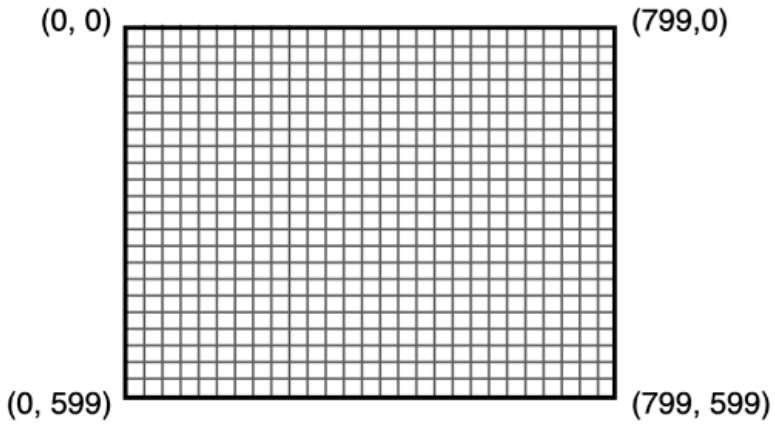
En la versión 1.4 de Java se introdujo el modo de pantalla completa (full mode exclusive mode). Dicho modo de pantalla suspende el ambiente de ventanas y permite el acceso eficiente al hardware gráfico, además de permitir el uso de técnicas avanzadas de dibujo en pantalla, como el double buffering y el page flipping, y brinda control a las aplicaciones sobre la resolución de la pantalla y profundidad de color. El objetivo del modo de pantalla completa es permitir un desempeño óptimo a aplicaciones que hacen uso intensivo de los recursos gráficos del sistema, como los videojuegos.

Resolución de pantalla

El hardware para despliegue gráfico en una PC común consta de la tarjeta de video y el monitor. La tarjeta de video almacena en su memoria todo lo que es desplegado en pantalla, y provee distintas funciones para su modificación. La tarjeta de video envía esta información al monitor para su despliegue en pantalla. El monitor simplemente muestra la información que recibe de la tarjeta de video. La pantalla del monitor se divide en una cuadrícula de pixeles (palabra derivada del inglés *picture element*), que son puntos de luz individuales. El número de pixeles horizontales y verticales de los que consta un monitor se conoce como su *resolución de pantalla*. Los pixeles forman un sistema coordenado cuyo origen se encuentra en la esquina superior izquierda de la pantalla, de forma que un pixel individual puede ser expresado a través de sus coordenadas (x,y) (Véase la figura 2.1).

Las resoluciones que se pueden elegir para el modo de pantalla completa dependen de las características del hardware involucrado. Resoluciones

Figura 2.1: *Una pantalla de 800x600. Tomada de [Brackeen2003]*



típicas son 640x480, 800x600, 1024x768 y 1280x1024. Los monitores y televisores más comunes tienen una proporción de tamaño de pantalla (*display size ratio*) de 4:3, que significa que la altura del dispositivo es tres cuartos de la anchura. Dispositivos más modernos usan pantallas más anchas, con proporciones de 3:2 o 16:10.

Los monitores pueden usar casi cualquier resolución debido a que su medio físico de despliegue en pantalla es un tubo de rayos catódicos que dibuja los pixeles en pantalla con un rayo de electrones. En el caso de las pantallas LCD (liquid cristal display) utilizadas en laptops y PCs modernas, el medio físico de despliegue es un transistor asociado a cada pixel, de manera que estos dispositivos solo pueden trabajar en una resolución nativa que expresa el número real de transistores asociados con los pixeles. Otras resoluciones pueden ser usadas en estos dispositivos, pero normalmente esto acarreará problemas: la pantalla se verá borrosa, o *pixeleada*.

Color y profundidad en bits

Los monitores combinan los colores rojo, verde y azul para crear cualquier color, a través de un modelo aditivo en el que la suma de todos los colores da por resultado el color blanco. El número de colores que estos dispositivos pueden desplegar depende de la profundidad en bits del modo de pantalla. Las profundidades más comunes son 8,15,16, y 24. Si un monitor tiene una

profundidad de bits n entonces puede desplegar 2^n colores.

Tasa de refrescamiento

En un monitor, un pixel se desvanece unos cuantos segundos luego de haber sido dibujado. Para mantener ante el ojo humano la ilusión de una imagen "solida", el monitor redibuja la pantalla continuamente. La frecuencia con que el monitor hace este redibujado se conoce como tasa de refrescamiento, en inglés *refresh rate*, y se expresa en hertz. Tasas entre 75 y 85Hz son suficientes para el ojo humano.

Cambiando a pantalla completa en Java

Para realizara este cambio se requieren objetos de varias clases:

Window Los objetos *Window* son abstracciones de lo que se despliega en la pantalla a través de una ventana. En la mayoría de los casos se utiliza un *JFrame*, que hereda de *Window* pero con las características *lightweight* de Swing.

DisplayMode Especifica la resolución, profundidad en bits y tasa de refrescamiento a los que será cambiada la pantalla.

GraphicsDevice Este objeto es el que permite el cambio en el modo de pantalla, y permite inspeccionar las propiedades del dispositivo gráfico (monitor o LCD). Se adquieren del *GraphicsEnvironment*.

Veamos un ejemplo de como entrar en modo de pantalla completa:

```
JFrame window = new JFrame();
DisplayMode displayMode = new DisplayMode(800, 600, 16, 75);

// Obtenemos el objeto GraphicsDevice
GraphicsEnvironment environment =
    GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice device = environment.getDefaultScreenDevice();

// Usamos el JFrame como la ventana en pantalla completa
device.setFullScreenWindow(window);
```

```
// Cambiamos de modo de pantalla
device.setDisplayMode(displayMode);
```

Si posteriormente deseamos abandonar el modo pantalla completa, hacemos que la ventana en dicho modo sea *null*:

```
device.setFullScreenWindow(null);
```

A este código le falta manejar la excepción *IllegalArgumentException*, que es lanzada en sistemas que no permiten el cambio de *DisplayMode*. Además, la ventana continua mostrando la barra de título y botones, para hacer que estos no sean visibles el código real debería llamar:

```
window.setUndecorated(true);
```

2.2.2. Dibujo activo

Para realizar la animación de cualquier elemento de un juego, se requiere de un mecanismo que actualice la pantalla continuamente y en forma eficiente. Una aplicación AWT/Swing normalmente realiza esta tarea llamando al método *repaint*, el cual informa al hilo de manejo de eventos del AWT que se requiere que la pantalla sea nuevamente dibujada – mediante el método *paint* del objeto AWT/Swing correspondiente –, pero esto puede traer aparejadas latencias debidas a que el hilo de manejo de eventos del AWT puede estar ocupando realizando otras tareas, p.ej. atendiendo de la entrada del usuario a través del teclado o ratón.

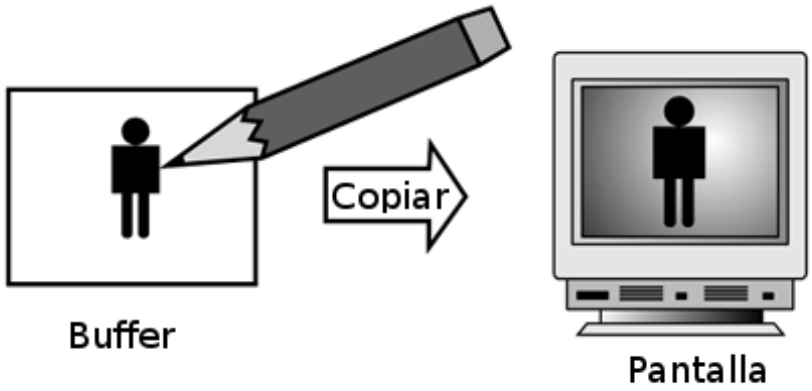
La solución a estos problemas es el dibujo activo (*active rendering* en inglés), término con el que se designa el dibujar directamente en pantalla en el hilo de ejecución principal de la aplicación, de forma que el programador tiene control de cuando exactamente se realiza el dibujo en pantalla. Para ello se usa el método *getGraphics* de la clase *Component* y sus subclases:

```
Graphics g = screen.getFullScreenWindow().getGraphics();
draw(g);
g.dispose();
```

2.2.3. Efectos indeseados del dibujo activo: *flickering* y *tearing*

Al dibujar activamente en pantalla, hay pequeños intervalos en que algunas imágenes y dibujos aun no aparecen en pantalla, sino que son dibujados

Figura 2.2: *Utilizando double buffering. Adaptada de [Brackeen2003]*



unos instantes después conforme el dibujo activo se ejecuta. Esto sucede tan rápido que el ojo humano solo percibe un "parpadeo" de la imagen, efecto conocido en inglés como flickering.

Double buffering

La solución a este problema es el *double buffering*. Un buffer es, simplemente, un área de memoria gráfica no desplegada en pantalla, utilizada para dibujar. Cuando se usa double buffering, en lugar de dibujar directamente en pantalla se dibuja en el buffer. Al finalizar el dibujado sobre el buffer, este es copiado en su totalidad a la pantalla, de forma que esta última se actualiza únicamente una vez, con lo que desaparece el efecto de flickering. Véase la figura 2.2

El buffer puede ser una imagen Java cualquiera. Para obtener uno puede usarse el método *createImage* de la clase *Component*. P.ej. en un applet que requiera usar double buffering y no esté realizando dibujo activo, se podría sobrescribir el método *update* para que utilizara un buffer y llamara al método *paint* utilizando el contexto gráfico del buffer:

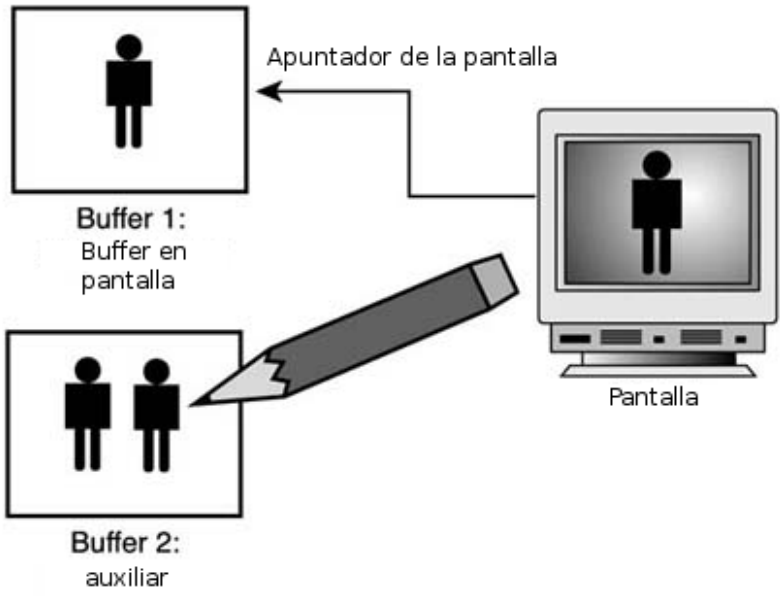
```
private Image doubleBuffer;
...
public void update(Graphics g) {
    Dimension size = getSize();
    if (doubleBuffer == null ||
```

```
        doubleBuffer.getWidth(this) != size.width ||
        doubleBuffer.getHeight(this) != size.height)
    {
        doubleBuffer = createImage(size.width, size.height);
    }


    if (doubleBuffer != null) {
        // Se dibuja en el buffer
        Graphics g2 = doubleBuffer.getGraphics();
        paint(g2);
        g2.dispose();

        // y se copia este a la pantalla
        g.drawImage(doubleBuffer, 0, 0, null);
    }
    else {
        // si no se puede usar el buffer,
        // simplemente se dibuja directamente
        // a la pantalla
        paint(g);
    }
}

public void paint(Graphics g) {
    // el dibujo de la pantalla se realiza aquí
    ...
}
```

Page flipping

Un problema con el double buffering es que requiere que todo el contenido del buffer sea copiado a la pantalla. La solución a esto es el *page flipping* (literalmente "cambio de página"). Véase la figura 2.3. En page flipping se utilizan dos buffers, uno es desplegado en la pantalla y el otro se utiliza como auxiliar para realizar el dibujo. El apuntador de la pantalla apunta al buffer que está siendo desplegado, y puede ser cambiado en la mayoría de los sistemas operativos modernos. Cuando se ha terminado de dibujar sobre

Figura 2.3: *Page flipping. Adaptada de [Brackeen2003]*



el buffer secundario, el apuntador de la pantalla se modifica, con lo que el buffer auxiliar se vuelve el buffer en pantalla de forma inmediata, y el anterior buffer en pantalla se vuelve el nuevo buffer secundario. Puesto que cambiar un apuntador es mucho más eficiente que copiar todo el contenido del buffer, esta técnica es más rápida que el doble buffering.

Tearing

Otro posible efecto desagradable a la vista es el tearing, que ocurre cuando la actualización del apuntador en page flipping, o la copia del buffer a la pantalla en double buffering, ocurre en forma simultanea a un refrescamiento del monitor. Es un efecto raro y que ocurre con mucha rapidez, y que parece un rasguño o ruptura de la imagen, de allí su nombre en inglés. Véase la figura 2.4. Para evitar el tearing, el cambio o copia del buffer secundario debe realizarse justo antes de que el monitor vaya a ser refrescado. Java se encarga de esta tarea mediante la clase *BufferStrategy*.

Figura 2.4: *Tearing. El buffer antiguo (arriba) aun es visible cuando el nuevo buffer (abajo) aparece por primera vez en pantalla. Adaptada de [Brackeen2003]*

La clase *BufferStrategy*

Esta clase se encarga de realizar el double buffering y page flipping. Primero intenta realizar page flipping, recurre a double buffering si no es posible utilizar page flipping. Se encarga además de que la copia del buffer a la pantalla o el cambio de apuntador se realicen en el momento adecuado.

Los objetos *Canvas* y *Window* pueden tener un *BufferStrategy*. Para ello se crea un *BufferStrategy* con el método *createBufferStrategy*, que recibe como parámetro el número de buffers que deben utilizarse – al menos 2 para que el double buffering o page flipping funcionen:

```
frame.createBufferStrategy(2);
```

Una vez se ha creado un *BufferStrategy*, se puede obtener una referencia a él mediante el método *getBufferStrategy*, y obtener una referencia a su contexto gráfico con el método *getDrawGraphics*. Una vez que se ha concluido el dibujo en el buffer, el método *show* muestra su contenido en pantalla, usando double buffering o page flipping:

```
BufferStrategy strategy = frame.getBufferStrategy();
Graphics g = strategy.getDrawGraphics();
draw(g);
g.dispose();
strategy.show();
```

2.2.4. Imágenes

Distintos tipos de transparencia

Opaca Cada pixel de la imagen es visible.

Transparente Cada pixel de la imagen es visible o por completo transparente.

Translúcida Cada pixel puede ser parcialmente transparente, con lo que se pueden crear efectos tipo "fantasma", en los que se puede ver a través de la imagen.

Formatos

Java soporta al menos tres formatos de imágenes:

GIF Imágenes que pueden ser trasparentes u opacas, y tienen 8 bits o menos de color. El formato tiene una buena compresión en archivos con poca variabilidad de color, pero su funcionalidad ha sido vuelta obsoleta por el formato PNG.

PNG Pueden tener cualquiera de los tres tipos de transparencia, y cualquier profundidad de color hasta 24 bits. Su compresión para archivos de 8 bits de color es semejante a la del formato GIF.

JPEG Para imágenes opacas con 24 bits de color. Tiene una buena compresión para imágenes fotográficas, pero es una compresión con perdida, de forma que la imagen no es una replica exacta de la fuente.

Imágenes en Java

Para obtener imágenes en Java, utilice el método *getImage* de la clase *Toolkit*:

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image image = toolkit.getImage(fileName);
```

La imagen es cargada en la memoria gráfica mediante un hilo de ejecución distinto, de suerte que si se intenta mostrar la imagen en pantalla antes de que la operación sea completada, solo partes de la imagen serán visibles. La solución a este problema es utilizar la clase *ImageIcon*, que carga la imagen

utilizando el *Toolkit* y espera a que la operación termine antes de retornar de la llamada al método *getImage*:

```
ImageIcon icon = new ImageIcon(fileName);
Image image = icon.getImage();
```

2.2.5. El ciclo principal del juego

Es la parte central del videojuego, se encarga de actualizar la pantalla. Para ello se siguen los siguientes pasos:

1. Actualizar las animaciones.
2. Dibujar la pantalla.
3. Dormir por un cierto periodo (opcional).
4. Comenzar el ciclo de nuevo.

En código, esto se vería así:

```
while (true) {

    // Actualizar las animaciones
    updateAnimations();

    // Dibujar la pantalla
    Graphics g = screen.getFullScreenWindow().getGraphics();
    draw(g);
    g.dispose();

    // Dormir
    try {
        Thread.sleep(20);
    }
    catch (InterruptedException ex) { }
}
```

En un juego real la condición para repetir el ciclo while dependería del estado del juego.

La llamada al método *sleep* es necesaria para que el ciclo principal del juego no consuma todo el tiempo del procesador. Esto permite que otros hilos puedan ejecutarse, particularmente el hilo de manejo de eventos del AWT y el recolector de basura de la JVM.

El principal problema de este ciclo principal del juego es que su frecuencia depende del hardware que lo ejecuta: la frecuencia es mayor en hardware veloz, y menor en hardware más lento. Además, el periodo que el ciclo duerme es arbitrario, y las mismas variaciones en la velocidad del hardware subyacente pueden hacer que este periodo de sueño sea muy amplio o muy pequeño. Lo que se requiere es que la frecuencia del ciclo sea independiente de la velocidad del hardware que lo ejecuta, y que el periodo de sueño se adapte dinámicamente a estos cambios en la velocidad del hardware. Para lograr esto se requiere código como el presentado en [Davison2005], en el capítulo 2, en la sección "Sleeping better":

```
private static final int NO_DELAYS_PER_YIELD = 16;
  /* Number of frames with a delay of 0 ms before the
     animation thread yields to other running threads. */


  public void run( )
  /* Repeatedly update, render, sleep so loop takes close
     to period nsecs. Sleep inaccuracies are handled.
     The timing calculation use the Java 3D timer.
  */
  {
    long beforeTime, afterTime, timeDiff, sleepTime;
    long overSleepTime = 0L;
    int noDelays = 0;

    beforeTime = J3DTimer.getValue( );

    running = true;
    while(running) {
      gameUpdate( );
      gameRender( );
      paintScreen( );
```

```
      afterTime = J3DTimer.getValue( );
      timeDiff = afterTime - beforeTime;
      sleepTime = (period - timeDiff) - overSleepTime;

      if (sleepTime > 0) {   // some time left in this cycle
        try {
          Thread.sleep(sleepTime/1000000L);  // nano -> ms
        }
        catch(InterruptedException ex){}
        overSleepTime =
              (J3DTimer.getValue( ) - afterTime) - sleepTime;
      }
      else {    // sleepTime <= 0; frame took longer than the period
      overSleepTime = 0L;

        if (++noDelays >= NO_DELAYS_PER_YIELD) {
          Thread.yield( );   // give another thread a chance to run
          noDelays = 0;
        }
      }
      beforeTime = J3DTimer.getValue( );
    }

    System.exit(0);
  } // end of run( )
```

Algunos puntos a destacar de este código son:

1. Utiliza *J3DTimer*, un timer de Java3D, en vez de *System.currentTimeMillis*. Este timer tiene una precisión de nanosegundos, por lo que se realizan las conversiones adecuadas para que las llamadas al método *sleep* reciban milisegundos como parámetro.

2. La variable overSleepTime calcula el tiempo "extra" que el hilo durmió en la última iteración, y reduce en esa medida el monto de sueño que el ciclo tendrá en la siguiente iteración.

3. Si *sleepTime* es negativo, el ciclo no tendrá un periodo de sueño entre esta iteración y la siguiente. Cuando se completan *NO_DELAYS_PER_YIELD*

iteraciones sin dormir, el ciclo no dormirá pero llamará a *yield* para permitir que otros hilos se ejecuten.

Estas técnicas para ajustar dinámicamente el periodo de sueño eran desconocidas por el autor al implementar los juegos que son objeto de este documento. El lector interesado encontrará en [Davison2005] una discusión más amplia al respecto, así como implementaciones alternativas del ciclo principal de juego.

2.2.6. Sprites y animaciones

La animación de imágenes se puede realizar mostrando una secuencia de imágenes conforme avanza el tiempo, como hacen las caricaturas. Este concepto es mejorado en los videojuegos para llegar a un sprite (literalmente "duendecillo"). Un sprite es una animación capaz de desplazarse por la pantalla, es decir, un sprite encapsula en si su propia animación y movimiento.

2.3. Interacción con el usuario

Como ya se mencionó, existe un hilo de ejecución que administra, en primera instancia, toda la interacción con el usuario: el despachador de eventos del AWT. Este hilo recibe eventos tales como movimiento del ratón, clics en los botones del mismo, presión de teclas en el teclado y otros. Cuando uno de estos eventos ocurre, el despachador verifica qué objetos Java están registrados para recibir estos eventos, y notifica a dichos objetos de que ocurrió el evento para que estos realicen el procesamiento pertinente. Con cada tipo de evento (teclado, movimiento del ratón, botones del ratón, etc.) está asociada una interfaz *Listener* y un evento *Event*. Para poder recibir notificaciones cuando un evento sucede, un objeto debe implementar la interfaz asociada. Por ejemplo, para eventos del teclado la interfaz y evento asociados son *KeyEvent* y *KeyListener*. Veamos como es manejada esta interacción del usuario:

1. El usuario presiona una tecla.
2. El sistema operativo notifica al JRE que el usuario ha presionado una tecla.
3. El JRE coloca un *KeyEvent* en la cola de procesamiento del hilo despachador del AWT.

4. El hilo despachador de eventos del AWT verifica si existen objetos esperando ser notificados del evento *KeyEvent*.

5. Los *KeyListener* registrados para recibir notificaciones del evento son notificados, reciben el *KeyEvent*, y realizan el procesamiento pertinente.

Este tipo de comportamiento ocurre para cada evento. Dada su importancia para el desarrollo de videojuegos, centraremos nuestra atención en eventos del teclado y del ratón.

2.3.1. Interacción con el teclado

Para responder a la interacción con el teclado se requiere crear un *KeyListener* y registrarlo para que reciba los eventos asociados. Para registrar el *KeyListener* se debe llamar el método *addKeyListener* en el componente cuyos eventos se desea manejar. En el caso de videojuegos, este componente será, normalmente, la ventana del juego:

```
Window window = screen.getFullScreenWindow();
window.addKeyListener(keyListener);
```

La interfaz *KeyListener* define tres métodos: *keyPressed*, *keyReleased* y *keyTyped*. El evento "typed" ocurre en primer lugar cuando una tecla es presionada, y se repite subsecuentemente basado en la tasa de repetición del teclado mientras la tecla se mantenga presionada. Cada uno de los tres métodos recibe como parámetro un *KeyEvent*, el cual permite averiguar que tecla fue presionada mediante un *vitual key code*, el cual es una abstracción asociada con una tecla en específico (es distinto de un carácter: los caracteres "q" y "Q" son el mismo *virtual key code*). Los *virtual key code*s están definidos como constantes estáticas de la clase *KeyEvent* de la forma *VK_XXX*, p.ej. *VK_Q*.

2.3.2. Interacción con el ratón

El ratón responde a desplazamientos, clics en sus botones y, posiblemente, el movimiento de su rueda de desplazamiento (*scroll wheel*). A cada una de estas interacciones se asocian un evento Java. Los botones del ratón se comportan como los botones del teclado, salvo que no existe la repetición automática del evento si los botones se mantienen presionados. Al movimiento

del ratón se asocian sus coordenadas (x,y), y al movimiento de la scroll wheel se asocia el desplazamiento vertical de la rueda. Cada evento tiene su propio listener: *MouseListener*, *MouseMotionListener* y *MouseWheelListener*. El evento asociado con las tres interfaces es *MouseEvent*.

La interfaz *MouseListener* define los eventos *mouseClicked*, *mouseEntered*, *mouseExited*, *mousePressed* y *mouseReleased*. *mousePressed* y *mouseReleased* son llamados cuando un botón es presionado y soltado, respectivamente. Un clic ocurre cuando un botón es presionado y liberado en sucesión, y es manejado por el método *mouseClicked*. Finalmente, *mouseEntered* y *mouseExited* ocurren cuando el apuntador entra y sale del área del componente asociado.

La interfaz *MouseMotionListener* responde a dos tipos de eventos: movimiento regular y movimiento de arrastre (el movimiento ocurre mientras se presiona un botón). Son manejados respectivamente por los métodos *mouseDragged* y *mouseMoved*.

La interfaz *MouseWheelListener* utiliza una subclase de *MouseEvent* llamada *MouseWheelEvent*. El único método de la interfaz es *mouseWheelMoved*. El método *getWheelRotation* de *MouseWheelEvent* proporciona el desplazamiento de la rueda. El desplazamiento hacia arriba es negativo.

2.4. Sonido

Para obtener sonidos en Java se utiliza la clase *AudioSystem*, la cual provee distintos métodos *getAudioInputStream* que permiten obtener sonidos de distintas fuentes, como el sistema de archivos o la internet. El método regresa un *AudioInputStream*, el cual permite leer las muestras de sonido sin lidiar con detalles específicos del formato del sonido, como los encabezados. Es posible obtener información específica del formato del sonido, como la tasa de muestreo, el número de canales y el tamaño del frame, mediante el método *getFormat*:

```
File file = new File("sound.wav");
AudioInputStream stream = AudioSystem.getAudioInputStream(file);
AudioFormat format = stream.getFormat();
```

Una vez se tiene el stream de audio y el formato del mismo, este debe ser alimentado a un objeto que implemente la interfaz *Line*. Un *Line* sirve para enviar y recibir audio al sistema de sonido. Con ellos es posible, p.ej. enviar

un sonido a las bocinas, o recibir sonido desde un micrófono. La interfaz *Line* tiene muchas subinterfaces, la más usada es *SourceDataLine*, la cual permite enviar sonidos a la tarjeta de sonido. Las lineas son creadas con el método *getLine* de la clase *AudioSystem*. Este método requiere como parámetro un objeto de tipo *Line.Info*. En los programas desarrollados en este documento se utiliza a *DataLine.Info*, una subclase de *Line.Info*, ya que dicha clase provee información respecto al formato del sonido. La manera más sencilla de utilizar una linea es usando una subclase de *Line* llamada *Clip*, la cual se encarga de cargar el sonido en memoria a partir de un *AudioInputStream* y de alimentarlo a la tarjeta de sonido:

```
// Se especifica el tipo de linea a crear
// a partir del formato del audio
DataLine.Info info = new DataLine.Info(Clip.class, format);
// Creamos la linea
Clip clip = (Clip)AudioSystem.getLine(info);
// Se carga el sonido en memoria
clip.open(stream);
// Se reproduce el sonido
clip.start();
```

Existen dos desventajas en esta forma de ejecutar sonidos:

1. Las clases del paquete *javax.sound* tienen una limitación respecto al número total de *Line*s que pueden estar abiertas en un momento determinado, normalmente 32, lo que impone un límite al número de sonidos que se pueden reproducir.

2. Si bien los *Clip*s pueden reproducirse simultáneamente, un mismo clip puede reproducir un sonido único sonido a la vez, de forma que si se requiere reproducir el mismo sonido en rápida sucesión se requiere un *Clip* por cada uno de estos sonidos.

Para evitar estas limitaciones, se utiliza la clase *SoundManager* del paquete *com.brackeen.gamebook*. *SoundManager* crea un pool de *Line*s a partir de un formato de audio. La clase *SoundManager* permite obtener objetos *Sound* a partir de archivos, los cuales almacenan las muestras de audio de un sonido en memoria. *SoundManager* permite ejecutar estos *Sound*s en forma concurrente, al enviar en distintos hilos de ejecución las muestras de audio de los *Sound*s a las *Line*s en su pool.

2.5. Comunicación a través de la red: RMI

RMI[1] permite a un objeto que se está ejecutando en una JVM invocar métodos en un objeto que se ejecuta en otra JVM. Para ello las aplicaciones que utilizan RMI requieren:

1. Obtener referencias a objetos remotos.
2. Comunicarse con los objetos remotos
3. Cargar a las máquinas remotas las definiciones de los objetos remotos, las cuales residen, posiblemente, en otras JVMs.

Todas estas tareas son realizadas por RMI.Las tareas del programador para invocar métodos en un objeto remoto son:

Crea una interfaz heredera de *Remote* Esta define los métodos accesibles en un objeto remoto:

```
public interface Compute extends Remote {
    void executeTask() throws RemoteException;
}
```

Implementar la interfaz definida Definiendo el comportamiento real de los métodos definidos en la interfaz:

```
public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public void executeTask() {
        //Aqui se realizan las tareas en el objeto remoto
    }
}
```

[1]Remote Method Invocation

Registrar los objetos que implementan la interfaz para su acceso remoto La forma más común es utilizar el rmiregistry para registrar objetos Java como disponibles a otras aplicaciones, con código como el siguiente:

```
Compute engine = new ComputeEngine();
Compute stub =
    (Compute) UnicastRemoteObject.exportObject(
                      engine, 0);
Registry registry = LocateRegistry.getRegistry();
registry.rebind(name, stub);
```

Obtener una referencia al objeto remoto En el lado del cliente, se obtiene una referencia al objeto remoto con código como el siguiente:

```
Registry registry = LocateRegistry.getRegistry(
                          "www.my-remote-host-name.com");
Compute comp = (Compute) registry.lookup(name);
```

Se llaman en el lado del cliente los métodos definidos en la interfaz remota Estas llamadas son iguales a cualquier otra invocación de un método, y lanzan *RemoteException*s:

```
  try {
    Compute comp = (Compute) registry.lookup(name);
    comp.executeTask();
} catch (RemoteException e) {
    e.printStackTrace();
}
```

El rmiregistry es el programa encargado de registrar a los objetos remotos y de resolver las peticiones de referencias a ellos, y se incluye en el JDK.

Capítulo 3

Un ejemplo del uso de videojuegos en rehabilitación: Java Therapy

Cerca de 80 % de los sobrevivientes de un ataque apopléjico sufren de perdida de capacidades motrices en brazos y manos. Distintos estudios muestran que terapias consistentes en la repetición intensiva y guiada de movimientos con la extremidad dañada ayudan a la recuperación de estos pacientes. Java Therapy[1] es un sistema diseñado para proporcionar terapia mediante repetición de movimiento a pacientes de apoplejía. Los usuarios del sistema, previa autenticación, reciben un programa de actividades personalizado y retroalimentación en cuanto a su progreso. Utilizando el sistema, un terapeuta puede supervisar a los pacientes en forma remota y brindar retroalimentación. Gracias a que Java Therapy utiliza dispositivos de entrada fabricados masivamente, la infraestructura de Internet y applets Java, el sistema es económico, accesible y adaptable.

3.1. Dispositivos de entrada

Java Therapy puede ser controlado por diversos dispositivos de entrada como ratones y joysticks. Mediante el uso de un joystick con retroalimentación de fuerza, un soporte para el brazo y tablillas ortopédicas para

[1] `http://www.javatherapy.com`

que pacientes que carecen de sujeción manual puedan asir el joystick, se brinda al usuario un sistema que mantiene una postura de brazo similar para cada sesión, proporciona un espacio de trabajo confortable de aproximadamente 10x10 cm., y es de bajo costo.

3.2. La interfaz de usuario

Las interfaces del sistema están diseñadas pensando en que el usuario navegará por ellas utilizando el dispositivo de entrada con propósitos terapéuticos, de forma que todas las zonas "seleccionables" son de gran tamaño. Para usuarios con mayores problemas de movimiento el teclado puede ser configurado utilizando las opciones de accesibilidad de Microsoft de manera que los elementos de la interfaz respondan a teclas accionadas con el brazo no afectado. De esta manera también es posible navegar la interfaz utilizando la tecla *tab* y las flechas del teclado.

Los componentes de la interfaz de usuario son 4:

3.2.1. Pruebas

Las pruebas evalúan capacidades de movimiento específicas del usuario del sistema:

Velocidad En esta prueba el usuario debe mover el apuntador hasta una distancia fija a un blanco. El blanco cambia de posición cada que el usuario logra mover el apuntador hasta él, o cada 6 segundos, lo que ocurra primero. La prueba consta de 16 blancos aleatorios. La trayectoria del apuntador es guardada por el sistema para ser graficada posteriormente si se requiere.

Coordinación El usuario debe seguir con el apuntador el trazo de un círculo dibujado en pantalla. El sistema mide los errores en el trazado.

Velocidad de digitación El usuario debe hacer clic con el ratón tantas veces como pueda en 10 segs.

Fuerza Este test solo es posible si se utiliza como dispositivo de entrada el joystick con retroalimentación de fuerza. El usuario debe intentar mantener el joystick inmóvil mientras este aplica una fuerza sinusoidal.

La distancia de desplazamiento del joystick es guardada como la puntuación de esta prueba.

Las pruebas fueron los únicos elementos de Java Therapy que el autor pudo observar en funcionamiento.

3.2.2. Juegos

Java Therapy incluye un clon del clásico *Breakout*. En este juego el usuario controla una pala que se mueve a lo largo de uno de los bordes de la pantalla, con la que hace rebotar una pelota. En el otro extremo de la pantalla se encuentran una serie de "ladrillos" que son destruidos al rebotar la pelota en ellos. La puntuación del juego es el número de ladrillos destruidos. Si se utiliza el joystick retroalimentado, el sistema puede ayudar a los jugadores con mayores problemas motrices al predecir la trayectoria de la pelota y "atrayendo" al joystick hacia dicha posición mediante un resorte virtual. Si el jugador tiene mejor capacidad de movimiento, el sistema puede oponerse al movimiento para mejorar la terapia.

La distinción entre pruebas y juegos es enteramente arbitraria: ambas actividades son cuantificadas y ambas suponen una actividad terapéutica si son realizadas periódicamente, si bien se supone que los "juegos" resultan más atractivos al usuario.

3.2.3. Gráficos e interfaces de uso y progreso

El sistema cuenta con 3 tipos de interfaces que permiten al usuario dar seguimiento a su terapia:

Lista de actividades Es la interfaz que aparece al autenticarse el usuario ante el sistema. Muestra el uso real semanal del sistema contra el uso deseado. Este último puede ser establecido por el usuario o por un terapeuta. Para facilitar la navegación las actividades en la lista son hipervínculos.

Reporte de actividad concluida Muestra el puntaje alcanzado en la actividad(prueba o juego) que acaba de concluirse comparado con:

- El promedio general en la actividad.
- El puntaje obtenido la última vez que se realizó la actividad.

- Un puntaje meta determinable por el usuario o terapeuta.

Progreso general Muestra gráficamente la historia de los puntajes obtenidos como función del tiempo o del número de veces que se ha ejecutado la actividad, así como el puntaje meta determinado por el usuario o terapeuta.

3.2.4. Página del terapeuta

Permite:

- Crear nuevas cuentas de usuario.
- Crear perfiles de uso(listas de actividades) para los usuarios.
- Dar seguimiento al uso y progreso de los distintos usuarios del sistema.

3.3. Tecnologías utilizadas en el sistema

Los componentes del sistema fueron elaborados, a grandes rasgos, usando 3 distintas tecnologías:

Java Las pruebas, tests e interfaces para el seguimiento de uso de y progreso en el sistema son applets Java

ActiveX/Javascript Un controlador ActiveX es utilizado para el control del joystick retroalimentado. Este controlador recibe ordenes del navegador, quien a su vez las recibe, vía Javascript, de los distintos applets.

ASP's y Perl Páginas ASP son utilizadas para recolectar datos en el servidor. Estas son escritas a archivos(trayectoria del cursor en las pruebas) o a una base de datos de MS-Access(cuenta de usuario, lista de actividades, uso del sistema y puntajes).

Capítulo 4

Primer juego: Pong

Durante su antepenúltimo semestre en la licenciatura en Ciencias de la Computación, el autor cursó el seminario "Diseño y programación Orientados a Objetos" con el M. en C. Alejandro Aguilar Sierra. El proyecto final del seminario era la elaboración en conjunto por la clase (6 o 7 personas) de un videojuego sencillo (un clon del clásico *pong*) que sería utilizado en un proyecto en el que el Mtro. Aguilar participaba, dirigido por el Dr. Ronald Leder y cuyo objetivo era el desarrollo de videojuegos para su utilización en la rehabilitación de personas con discapacidades motoras. El desarrollo del juego utilizó como base un applet cuyo código fuente se obtuvo de Internet[1]. Al finalizar el seminario la clase había rediseñado la estructura del videojuego completamente - uno de los objetivos fundamentales del seminario dada su temática - e implementado algunas de las características requeridas para el uso del juego en terapia(sensibilidad configurable, velocidad de juego configurable, etc.), pero aun faltaban otras funcionalidades por implementar.

Luego de la conclusión del seminario, el autor continuó trabajando como becario en el proyecto. Por motivos ajenos al tema de esta tesis, el desarrollo del juego se prolongó por un periodo de más de 2 años, y el juego jamás llegó a ser utilizado en terapia.

[1]Desafortunadamente ninguno de los involucrados en el proyecto tiene la URL del sitio del que provenía. El juego original no proveía ninguna de las características configurables mencionadas en la siguiente sección. En él, la raqueta del jugador se desplazaba horizontalmente por el borde inferior de la pantalla, y había ladrillos entre la raqueta y el extremo contrario de la pantalla, los cuales iban desapareciendo al ser impactados por la pelota. La puntuación del jugador era proporcional al número de ladrillos destruidos. No se conservó ningún elemento gráfico del juego original, se eliminaron los ladrillos y se modifico el movimiento y posición de las raquetas, que se desplazan ahora verticalmente.

4.1. Descripción

Al igual que en el "pong" original, los jugadores controlan una raqueta que se mueve a lo largo de una de las orillas de la pantalla, y deben impactar con ella a una pelota que rebota de un extremo de la pantalla a otro, a la manera del juego de tenis de mesa. El jugador recibe puntos cada vez que golpea la pelota, y pierde un turno cada que falla. El jugador cuenta con un número configurable de turnos (3 por omisión) en cada juego.

El clon de "pong" que este capítulo reseña tenía por objetivo ser utilizado en terapia de rehabilitación para personas con discapacidades motrices en los brazos, en forma similar al sistema descrito en [Reinkensmeyer2002]. El sistema nunca llegó a probarse con los dispositivos de entrada para terapia, por lo que los mecanismos de control del juego utilizados son únicamente el teclado y el ratón.

Otras características del juego son:

- Velocidad de la pelota configurable.
- Sensibilidad de la raqueta configurable.
- Tamaño de la raqueta configurable.
- Tamaño de la pelota configurable.
- Lectura/escritura de las anteriores propiedades a/desde un archivo *properties* de Java.
- Tamaño del juego(resolución de pantalla) configurable.
- Posibilidad de jugar contra un 2º jugador en otra PC cuya dirección IP sea conocida. Con este fin el juego puede desplegar la dirección IP de la PC en la que está siendo utilizado.
- Cada juego es grabado a un archivo de texto plano para su posterior reproducción.

4.2. Diseño del juego

La intención al desarrollar Pong era que se apegara estrictamente al patrón modelo-vista-controlador (MVC). Sin embargo, hay algunas caracteris-

tícas en que se aparta del mismo. A continuación se define MVC, y, posteriormente, se da cuenta de los detalles en los que el juego se aparta de dicho patrón de diseño.

4.2.1. Modelo-Vista-Controlador (MVC)

MVC es un patrón de diseño utilizado en Ingeniería de Software. El patrón aisla:

- La lógica de negocio (modelo).
- La interfaz de usuario (vista).
- Las interacciones entre ambas en respuesta a las acciones del usuario (controlador).

De lo que resulta una aplicación donde es sencillo modificar la apariencia visual, las reglas de negocio subyacentes y las interacciones entre ambas sin que un cambio en ninguna de ellas afecte a las otras. Las raices de este patrón se remontan a Smalltalk, lenguaje en el que era utilizado para relacionar las tareas de entrada, procesamiento y salida con el modelo de interacción gráfica con el usuario. Véase el diagrama en la página 48

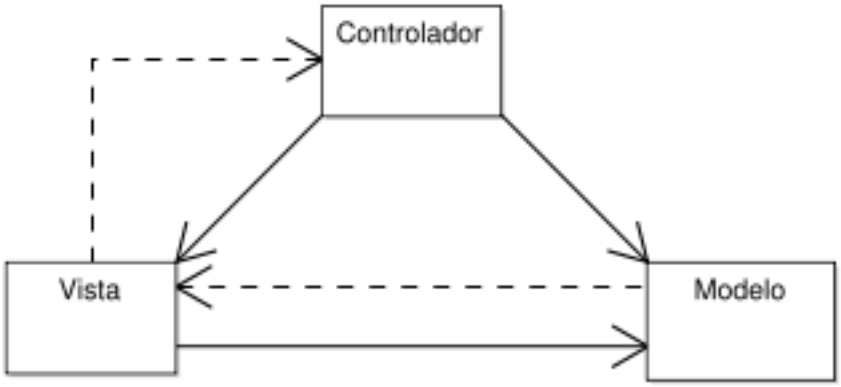
Los tres elementos que forman el patrón son:

Modelo Representa los datos del dominio específico y las reglas que controlan el acceso y modificación de los mismo.El patrón MVC no menciona específicamente la capa de persistencia de datos, pues se supone que actúa a un nivel más bajo que el modelo o es encapsulada por el.

Vista Presenta el modelo al usuario, en una forma adecuada para la interacción, típicamente mediante una interfaz de usuario. Accede a los datos de negocio a través del modelo y determina como debe ser presentada esta información al usuario. Pueden existir, con propósitos distintos, múltiples vistas para un solo modelo.

Controlador Procesa y responde a eventos, típicamente interacciones del usuario con la vista mediante el uso del ratón, teclado u otros dispositivos, posiblemente invocando cambios en el modelo. Expresado en términos más generales se puede decir que el controlador administra la comunicación al modelo de las acciones del usuario. El modelo convierte

Figura 4.1: *Diagrama del patrón MVC. Las lineas solidas indican una relación directa (una referencia en lenguaje Java), las punteadas una indirecta, p.ej. mediante el patrón observador.*



las interacciones del usuario con la vista en acciones a ser realizadas por el modelo.

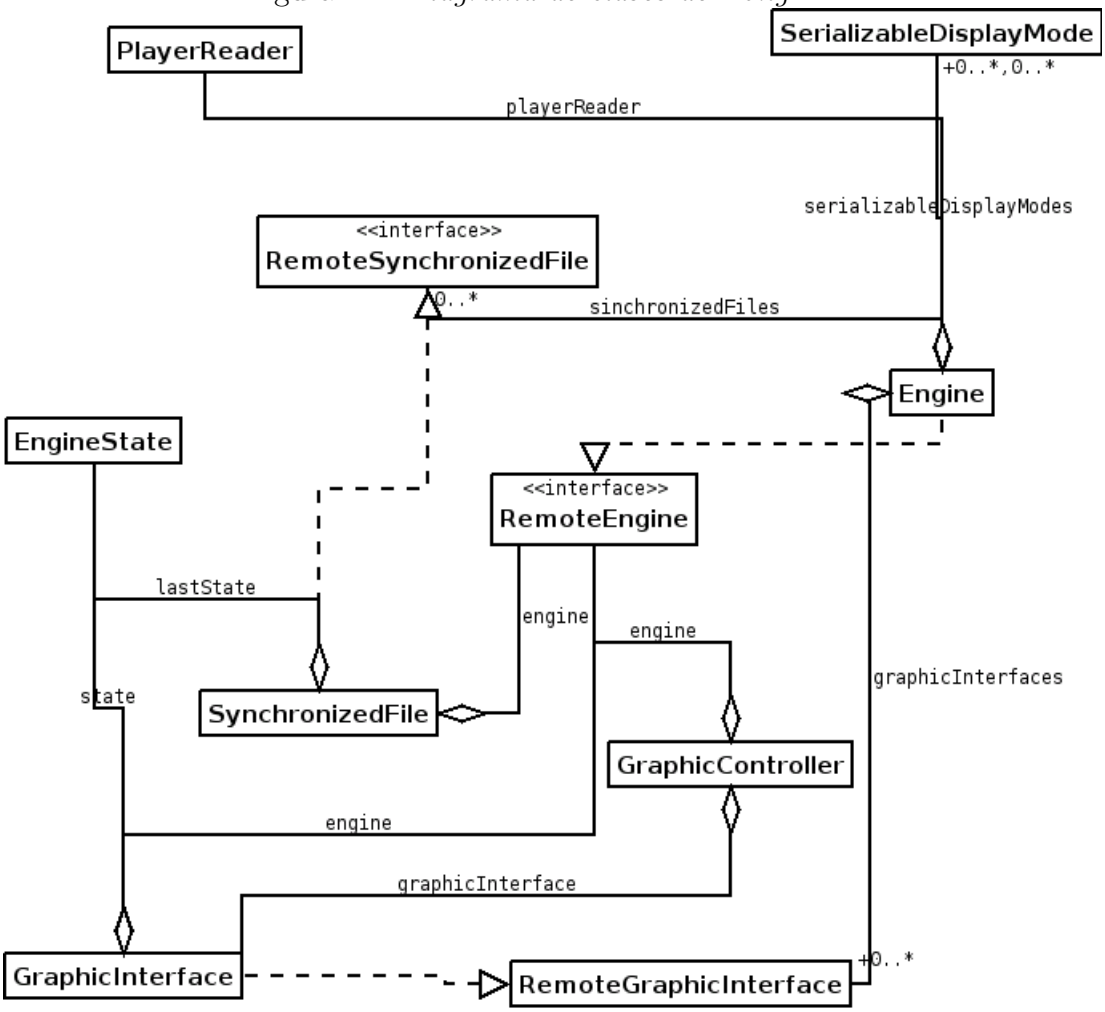
4.2.2. Divergencias respecto a MVC

1. El modelo(*Engine*) y la interfaz gráfica(*GraphicInterface*) implementan interfaces que a su vez heredan de *java.rmi.Remote*, para que puedan ser utilizados en forma remota. De forma que la referencia al modelo en la vista no utiliza directamente el tipo *Engine*, sino la interfaz remota *RemoteEngine*, como puede apreciarse en la figura 4.2.

2. Existe una relación directa del modelo a la vista (clases *Engine* y *RemoteGraphicInterface* en la figura 4.2) ya que utilizar una relación indirecta, p.ej. mediante el patrón "Observador"(clase *java.util.Observable* e interfaz *java.util.Observer*) hubiera resultado en relaciones más complejas entre ambas clases(cuando no imposibles o poco claras dada la necesidad de usar interfaces herederas de *java.rmi.Remote* como el tipo de los objetos remotos). Si bien esta relación existe, solo es utilizada para:

   a) Solicitar a la interfaz que se redibuje en respuesta a un cambio en el modelo.

Figura 4.2: *Diagrama de clases de Pong.*



b) Obtener información respecto a las resoluciones de pantalla posible y/o tamaño de la ventana, es decir, del área de juego, y negociar esta resolución común cuando se trata de la modalidad 2 jugadores.

De tal manera que esta relación directa entre el modelo y la vista no violenta dramáticamente la independencia entre ambas fomentada por MVC, y su relación queda bien documentada en la interfaz *RemoteGraphicInterface*.

3. La actualización de la interfaz gráfica es posible mediante la transmisión, vía *RMI*, de las principales variables de estado de *Engine* a través del método *getState* de la interfaz *RemoteEngine*, el cual regresa un objeto de tipo *EngineState*. Esta última clase existe únicamente con este fin, y "resume" a *Engine* en una versión serializable muy reducida para su rápida transmisión vía rmi.

4.2.3. El modo pantalla completa

De los tres juegos desarrollados, *pong* es el único puede ejecutarse en modo pantalla completa. Si bien el uso del modo pantalla completa es conceptualmente simple, su uso apropiado impone modificaciones respecto al esquema orientado a eventos utilizado en una simple aplicación Swing, p.ej. el uso de ciclos de dibujo en pantalla(rendering loop), el verificar la disponibilidad del modo pantalla completa y decidir cual es la respuesta en aquellos casos en los que no está disponible, etc.. Esta programación extra es sencilla, pero se aparta ya de una aplicación Swing clásica, y lleva emparejados otros problemas que no se reseñan específicamente en el "Java Tutorial", de los que se habla más adelante.

4.2.4. Sonido

La versión del juego elaborada durante el seminario ya contaba con sonido. El problema con esta versión era que si el mismo sonido requería ser reproducido simultáneamente, el primero de los sonidos era detenido debido a los problemas de simultaneidad de reproducción de *javax.applet.AudioClip* que se mencionan en la página 37. La solución a este problema fue incorporar al juego la arquitectura para reproducción de sonidos de [Brackeen2003], como

se explicó en la sección 2.4. Los sonidos del juego se obtuvieron de Internet, y no se cuenta con las fuentes de dichos sonidos.

4.2.5. El juego por red

El juego utiliza *RMI* para la comunicación entre el modelo (servidor), por una parte, y la vista y el controlador por otra (cliente). En una primera implementación de este modelo distribuido del juego, se añadieron a la interfaz *RemoteEngine* todos aquellos métodos de la clase *Engine* a los que la vista y el controlador requerían acceso, y a la interfaz *RemoteGraphicInterface* todos los métodos de *GraphicInterface* a los que requería acceso la *Engine*. Puesto que el dibujo activo de pantalla requiere que estas llamadas remotas se hagan varias veces por segundo, el resultado era que existían latencias notables. Para eliminar este problema fue necesario "resumir" la información de *Engine* necesaria en la clase auxiliar *EngineState*, la cual proporciona a la interfaz gráfica toda la información que requiere para hacer el dibujo activo de pantalla.

Puesto que las pruebas del juego remoto se hicieron con equipos que se encontraban conectados en el mismo segmento de red, jamás se llegaron a experimentar latencias por el tráfico en la LAN. Esto simplificó en gran medida el código del juego, pero, con toda probabilidad, hará que el juego deje de funcionar al utilizarse en una red con latencias debidas al tráfico en la misma. Puesto que la finalidad de la modalidad 2 jugadores era proveer al segundo jugador de un medio de control físico del juego, estando, idealmente, ambos jugadores en el mismo espacio físico (lo cual no pudo lograrse con un segundo ratón), el juego no fue modificado para que lidiara con estos problemas debidos a latencias en la red.

4.3. Problemas del desarrollo

4.3.1. La integración de widgets de Swing

El problema más acusado al trabajar en el modo de pantalla completa es la integración de widgets de Swing y el AWT en el juego. En particular mencionaré la ausencia de la barra de menús en una ventana en modo pantalla completa. La ausencia de la barra me llevó a intentar utilizar menús emergentes (pop-up menús), los cuales jamás funcionaron adecuadamente debido

al dibujo activo de la pantalla, que ocasionaba que los menús fueran sobredibujados por los restantes componentes de la pantalla del juego. Tampoco es posible utilizar ventanas de ningún otro tipo (InternalFrames, JOptionPanes, etc.) puesto que padecen de los mismos problemas de sobredibujado por el dibujo activo de la pantalla, o no son mostradas en primer plano debido a los monopolios del foco y la pantalla inherentes al modo pantalla completa.

Probablemente estos problemas hubieran podido solucionarse utilizando otras características de las ventanas Swing, como el *glass pane*, pero el problema con este tipo de soluciones es que el desarrollador se ve en la necesidad de programar mucha lógica de dibujo de interfaces gráficas que *ya está elaborada para Swing, pero que deja de funcionar en el modo pantalla completa.* Esto es algo que no se menciona explícitamente en ninguno de los documentos consultados por el autor, pero que es bien sabido por los desarrolladores de juegos en Java, al punto de que la solución al problema fue el uso de menús especiales desarrollados en [Brackeen2003].

4.3.2. El control para 2 jugadores

El applet a partir del cual se desarrolló el juego únicamente soportaba un jugador. Durante el seminario con el maestro Aguilar se implementó el modo 2 jugadores, con un jugador controlado por el ratón y otro por el teclado. Puesto que el juego debía usarse en terapia de rehabilitacíon, los mecanismos de control del juego debían tener una naturaleza más mecánica, lo que descartaba el uso del teclado. En virtud de ello y a petición del Dr. Leder y el Mtro. Aguilar el autor intentó utilizar 2 distintos ratones, uno para cada jugador, lo cual resultó inútil ya que Java no diferencia las señales provenientes de los 2 ratones, esto es, ambos funcionan, pero controlan el mismo apuntador y, por consiguiente, ambas raquetas del juego.

El único API para Java gratuito [2] que ofrecía esta funcionalidad que el autor localizó fue jinput[3]. La funcionalidad para diferenciar entre 2 distintos ratones solo se ofrecía en la versión para Windows. El autor no pudo compilar la biblioteca en Windows, a pesar de seguir todas las instrucciones que el sitio brinda al respecto y de obtener todas las bibliotecas de las que la compilación requiere. Utilizando la versión binaria de JInput disponible en linea el autor realizó una prueba con un programa Java para tratar de distinguir entre los 2

[2] Además de gratuito, JInput es un proyecto open source.

[3] `https://jinput.dev.java.net/`

ratones, pero sin conseguir buenos resultados: todas la información generada por los eventos de los 2 ratones proporcionada por el API de jinput era la misma. Finalmente el autor descartó la idea de poder utilizar el modo 2 jugadores en la misma PC, lo que llevo al modelo basado en RMI que el juego usa en su versión final.

Otra posible solución era utilizar un ataque al problema como el empleado por Reinkensmeyer, que emplea controladores ActiveX propietarios (FeelTheWeb de Immersion Corporation) para controlar el joystick con retroalimentación de fuerzas, y Netscape Live Connect para comunicar el estado del joystick al applet. Dado que el autor nunca contó con los dispositivos de juego a utilizar ni sus controladores, no pudo intentar este tipo de soluciones.

4.4. Conclusión

El juego incorpora exitosamente una serie de características configurables, las cuales se detallaron en la sección 4.1. El juego utiliza el dibujo activo de pantalla y el modo de pantalla completa, e incorpora el modo de juego para 2 jugadores a través de una LAN. Las características que no se concluyeron o no se implementaron son:

- Controlar la(s) raqueta(s) mediante dispositivos de terapia.
- Controlar al segundo jugador mediante un segundo ratón en la misma PC.
- Integración de widgets Swing en la interfaz gráfica.

Figura 4.3: *Pantalla inicial.*

Figura 4.4: *Configuración del teclado.*

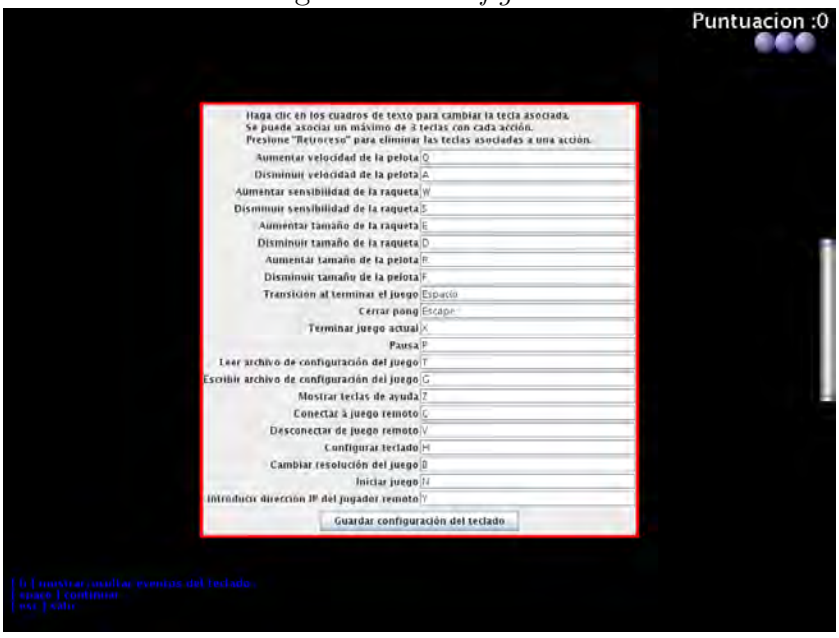


Figura 4.5: *Selección de host para 2 jugadores.*

Figura 4.6: *Juego un jugador.*

Figura 4.7: *Juego en pausa.*

Figura 4.8: *Fin del juego.*

Capítulo 5

Segundo juego: "¿¿¿Tú decides si fumas???"

Durante el periodo Noviembre de 2007 - Junio de 2008 el autor debió desarrollar un videojuego para para el *Programa de Prevención del Tabaquismo en Mujeres Adolescentes* de la Facultad de Psicología de la UNAM, a cargo de la Doctora Nazira Calleja Bello, como parte de sus labores en el DPID-SERUNAM-DGSCA[1]. El videojuego a desarrollar consistiría en un laberinto para un jugador en 2D, el cual sería utilizado para realizar labores educativas en contra del tabaquismo entre niñas.

5.1. Descripción del juego

En el laberinto se hallan desperdigados aleatoriamente 10 archivos. Al alcanzarlos el jugador y presionar una tecla, estos muestran su contenido: información respecto al tabaquismo(archivos "malos") y las estrategias mercadotécnicas utilizadas por las compañías tabacaleras(archivos "buenos"), ambas orientadas al público femenino. Luego de ver la información que contienen, el jugador puede decidir entre llevar el archivo o dejarlo. Los archivos buenos dan puntos al jugador al ser recolectados, los malos le quitan puntos. El objetivo del juego es recolectar todos los archivos buenos, maximizando la puntuación. El laberinto es recorrido constantemente por un policía. Si el jugador es alcanzado por el policía, pierde todos los archivos que ha recolec-

[1]Departamento de Productos Interactivos para la Docencia de la Subdirección de Servicios Educativos en Red de la Dirección General de Computo Académico de la UNAM

tado y debe comenzar de nuevo, a menos que haya recolectado una "placa de detective" que se halla en el laberinto, con la cual el policía es engañado y permite a la jugadora continuar su camino. El programador del videojuego fue el autor, el diseño gráfico y las ideas respecto a las reglas del juego fueron realizados por personal del DPID y la DGSCA, y la Dra. Calleja.

5.2. Diseño del juego

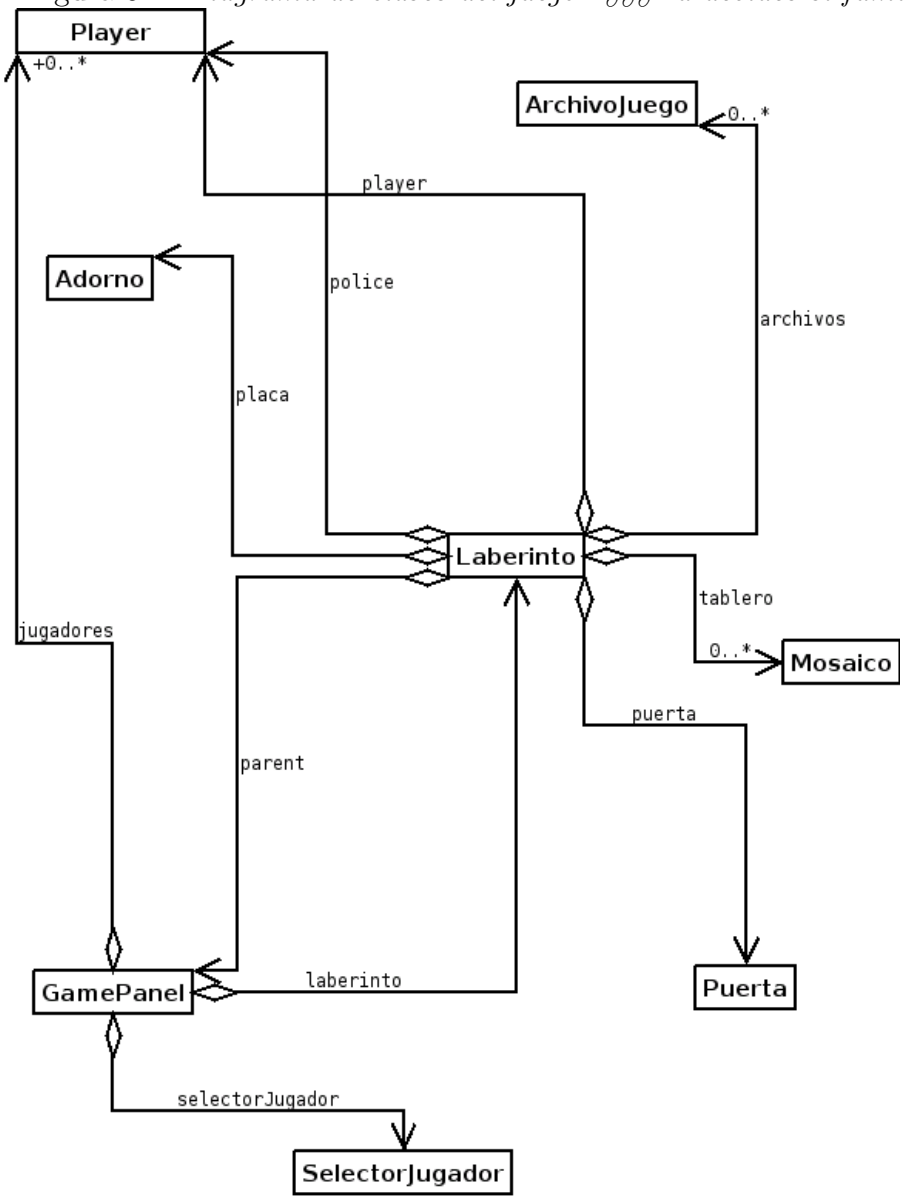
5.2.1. Herencia de *JComponent*

A diferencia de *Pong* en este juego no existe una separación entre modelo, vista y controlador, puesto que no se requería un controlador altamente configurable, ni se requería presentar la información del juego a través de distintas vistas, ni se trataba de un proyecto con la finalidad de ilustrar un patrón de diseño específico. Existen clases *Laberinto*, *Mosaico*, *Jugador*, *Archivo*, *Adorno*, *Puerta*, todas ellas herederas de JComponent, y cada una de ellas es responsable de su dibujado en pantalla. La experiencia ganada con *Pong* fue de gran utilidad aquí, al reusar mucho código de JComponent concerniente al dibujo de cajas en la pantalla(posición x, posición y, anchura y altura) en lugar de "reinventar la rueda".

5.2.2. Dibujo pasivo

El juego no utiliza el modo de pantalla completa, así que está más próximo a una aplicación Swing común que *pong*. El dibujo de la pantalla es realizado a través del método *repaint* de JComponent o alguna de sus herederas. Como se mencionó en el capítulo anterior, esto conlleva problemas de "parpadeo"de la imagen debido al pobre desempeño de los mecanismos de dibujo de Swing al ser utilizados en forma constante por el hilo de ejecución principal del juego. El juego no utiliza double buffering ya que los problemas de flickering experimentados no son muy acusados, por lo que no fueron un motivo de queja por parte de los usuarios finales del juego.

Figura 5.1: *Diagrama de clases del juego "¿¿¿Tú decides si fumas???".*



5.2.3. Aplicación de búsquedas: generación del laberinto y rutas del policía

Cada vez que el juego inicia el laberinto es generado utilizando un algoritmo de búsqueda a profundidad [2]. El policía se mueve en el laberinto escogiendo aleatoriamente una casilla a la cual se dirige y siguiendo posteriormente la ruta a ella. Ambos procesos son aplicaciones prácticas de búsquedas por profundidad en el laberinto.

5.2.4. Movimiento del policía en un hilo de ejecución independiente

El policía calcula la trayectoria que seguirá y se mueve a través de ella en un hilo de ejecución independiente al hilo en el que se ejecuta el ciclo principal del juego. El código del hilo de ejecución del policía se ve de esta manera muy simplificado, pues sus únicas labores son:

1. Mientras aun falten tramos por recorrer en la ruta del policía, continuar el movimiento del policía a través de la misma.

2. Si la ruta del policía ya ha sido completada, calcular una nueva ruta y volver a la tarea anterior.

5.3. Problemas del desarrollo

5.3.1. Detección de colisiones: Paredes

La lógica para que el jugador no "pasara sobre las paredes" fue uno de los aspectos del juego que requirieron más tiempo para su programación, pues cada casilla del laberinto encapsula las 4 paredes y las 4 esquinas que forman su perímetro, y su posición y existencia es parte de la lógica interna de la casilla(clase *Mosaico*), la cual no es directamente accesible desde la clase *Laberinto*. Este diseño facilitó la creación del laberinto, pero entorpeció la detección de colisiones del jugador y el policía con las paredes.

Dado que cada casilla encapsula a sus paredes, la creación del laberinto se ve facilitada en gran medida: basta hacer un recorrido a profundidad de las

[2] Consúltese [MazeGenerationAlgorithm]

casillas donde se decide en cada paso, de forma aleatoria, cual casilla será la siguiente en el recorrido, y se eliminan la paredes entre esta y la casilla actual.

La detección de las colisiones, por otra parte, se ve entorpecida: para averiguar si el jugador o un policía está haciendo colisión con alguna pared, es necesario interrogar a la casilla actual respecto a las paredes que existen en la misma y a si el jugador puede abandonar la casilla en un sentido determinado. Esto puede verse más claramente en el código que se encarga de esto:

```
    public boolean moveInMaze(int x, int y, Player object) {
        if (x < object.getX()) {
            object.setDirection(object.LEFT);
            if (x < object.getX() &&
                    (object.getX() > paredIzquierda(object) ||
                    (izquierdaLibre(object) &&
                     enEspacioVertical(object)))) {
                object.setX(object.getX() - 1);
                object.increaseStep();
                repaint();
                testForCollisions(object);
                return true;
            }
...
```

este segmento recibe las coordenadas a las que se intenta mover al jugador. El método *paredIzquierda* devuelve el límite derecho de la pared izquierda de la casilla en la que se encuentra el jugador. El método *izquierdaLibre* indica si existe o no la pared izquierda de la casilla en la que se encuentra el jugador. El método *enEspacioVertical* devuelve *true* si el jugador se encuentra por completo entre las paredes superior e inferior de la casilla en la que se encuentra.Estos chequeos se deben hace, *mutatis mutandi*, en cada una de las cuatro direcciones en que se mueve al jugador.

5.3.2. Detección de colisiones: policía-jugador

La presencia del hilo de ejecución encargado del movimiento del policía obligo a utilizar bloques *synchronized* para que el policía y el jugador interactuaran correctamente. El código sincroniza adecuadamente los tres hilos de ejecución presentes en el juego:

1. El despachador de eventos del AWT.
2. El ciclo principal del juego.
3. El hilo encargado del movimiento del policía.

Las secciones *synchronized* del código son suficientemente pequeñas, de suerte que su presencia no causa latencias en el juego, a pesar de que algunas de ellas se encuentran en el hilo de ejecución del ciclo principal del juego.

El autor carecía de experiencia real creando código *thread safe*, por lo que lograr el correcto funcionamiento de las secciones de código involucradas requirió de muchas horas realizando depuración lógica, la cual hubiera resultado mucho mas engorrosa de lo que fue sin la ayuda de las funciones para debugging de Netbeans[3].

5.4. Conclusión

El juego implementa exitosamente algoritmos para la generación aleatoria del laberinto y para el movimiento del policía. Las colisiones entre el jugador y todos los elementos del juego se detectan adecuadamente utilizando código *thread-safe*. El juego utiliza dibujo pasivo de la pantalla, lo cual resulta poco eficiente, pero suficiente para las necesidades del juego.

El juego fue probado en un grupo de 160 niñas, las medidas de susceptibilidad tabáquica disminuyeron significativamente del pretest al postest[4].

[3] `http://www.netbeans.org`

[4] Fuente: Comunicación de la Dra. Nazira Calleja Bello con el autor, 22 de Octubre de 2009.

Figura 5.2: *Pantalla Inicial*

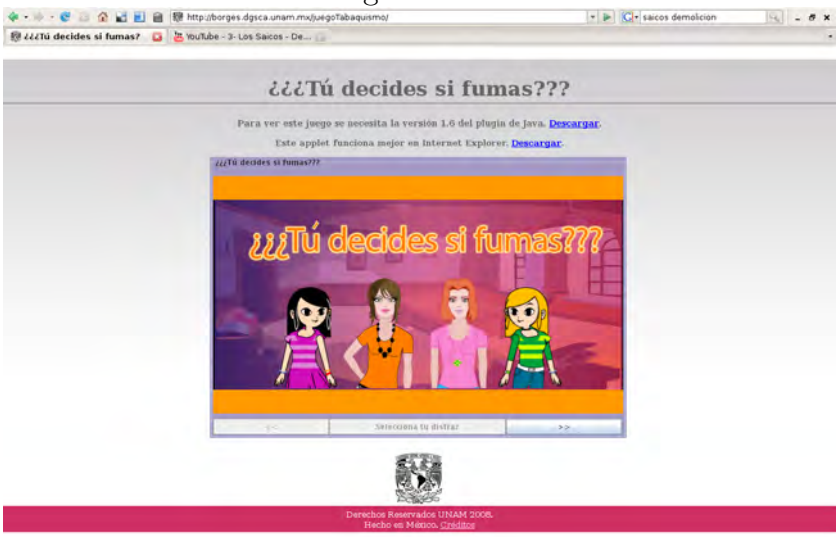


Figura 5.3: *Selección de jugadora*

Figura 5.4: *Juego*

Figura 5.5: *Pantalla al abrir un expediente*

Figura 5.6: *Pantalla al ser capturada la jugadora*

Figura 5.7: *Indicaciones de laberinto terminado*

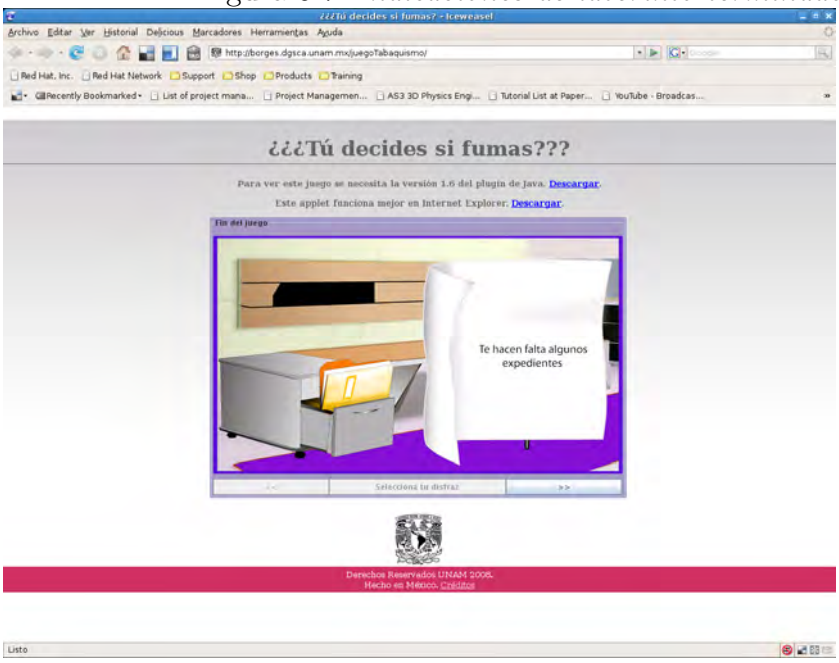
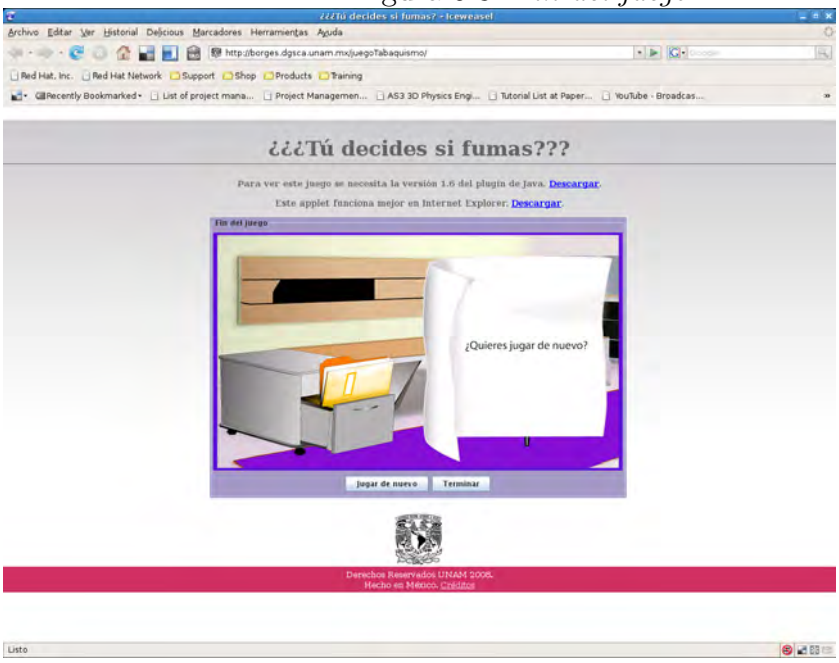


Figura 5.8: *Fin del juego*



Capítulo 6

Tercer juego: *¿¿¿Tú decides si fumas???* utilizando un motor de juego

El tercer videojuego desarrollado fue una nueva versión de *¿¿¿Tú decides si fumas???* utilizando un motor de juego. El motor de juego utilizado fue Java Instructional Gaming(JIG). El autor conoció el proyecto JIG al leer el artículo [Wallace2006]. El desarrollo del videojuego utilizando JIG tenía por objetivo explorar las capacidades y posibilidades de uso y adaptación de JIG a proyectos de desarrollo de videojuegos del DPI-SERUNAM-DGSCA. El desarrollo de la versión JIG del juego *¿¿¿Tú decides si fumas???* tuvo lugar en el periodo que va de Diciembre de 2008 a Febrero de 2009, durante el cual no constituyó la principal obligación laboral del autor.

Las razones para utilizar JIG en lugar de otro motor de juego Java fueron:

- Es el único motor de juego en Java que se menciona en los artículos consultados para la elaboración de este documento.
- La lectura de los tutores en linea del proyecto mostró que este motor en específico posee la mayoría de las características necesarias para reelaborar *¿¿¿Tú decides si fumas???*.
- La falta de tiempo para buscar otros motores de juego en Java y conocer sus características, y para el desarrollo en general de esta segunda versión de *¿¿¿Tú decides si fumas???*.

6.1. Diseño del juego

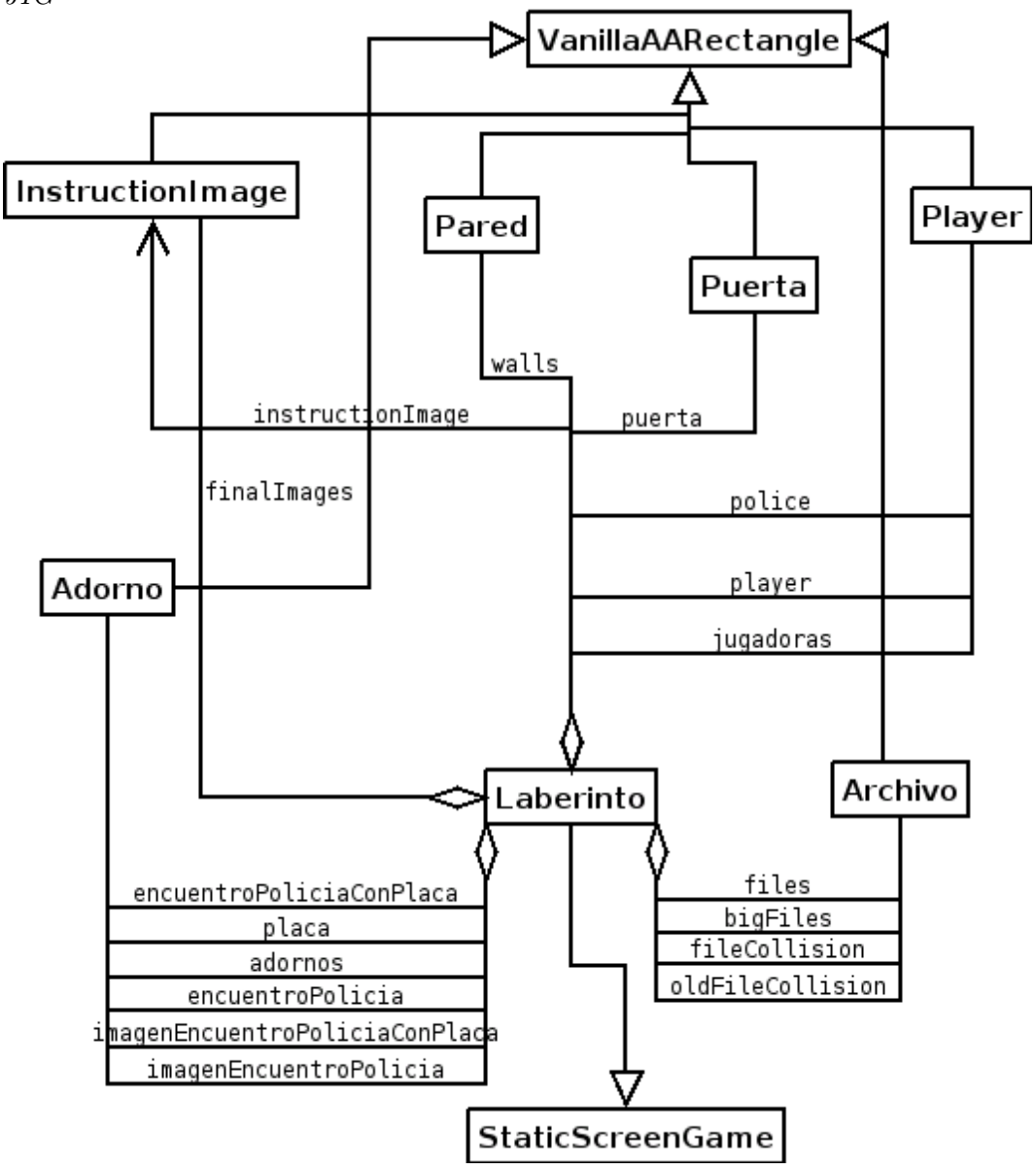
El motor de juegos JIG ofrece un marco de trabajo para el desarrollo de videojuegos 2D muy completo. Esta versión del juego hace uso de varias características de JIG:

- Los objetos de juego son Sprites
- El motor de juego se encarga del dibujo en pantalla de los objetos del juego y de actualizar esta representación gráfica de los mismos, en una versión que ya es eficiente y que evita al programador la carga de programar el double buffering o alguna otra técnica para el dibujo en pantalla sin flickering y otros defectos visuales desagradables.
- JIG cuenta con un componente de simulación física que permite, por ejemplo, simular la acción de la gravedad en juegos que lo requieran. Este componente fue utilizado en el juego, pero únicamente para el manejo de las colisiones.
- Organización de los objetos de juego en *Layer*s para el manejo de las colisiones y el dibujado en pantalla, lo que ayuda a organizar los componentes gráficos del juego en una pila cuyas capas son elementos con las mismas características.
- JIG cuenta con una API para obtener y desplegar imágenes en pantalla que ahorra al programador complicaciones de bajo nivel.

6.1.1. Sprites

Todos los objetos de juego heredan de *jig.engine.Sprite*(En realidad de *jig.engine.physics.vpe.VanillaAARectangle* , la cual es descendiente de *Sprite*), la cual encapsula toda la lógica necesaria para mover el objeto en la pantalla y animarlo, de suerte que para desarrollar nuevos elementos de juego basta simplemente con heredar de *Sprite* o de alguna de sus subclases y escribir únicamente la lógica a la medida necesaria. Esto tiene la ventaja de no entorpecer las relaciones de herencia al crear clases que hereden de *JComponent* o sus descendientes, a la vez que utiliza mucho código ya preexistente para la animación y movimiento y que es tedioso realizar por cuenta propia. Esto aunado a las clases que permiten obtener imágenes desde

Figura 6.1: *Diagrama de clases del juego ¿¿¿*Tú decides si fumas???*, versión JIG*



el sistema de archivos y a la lógica de animación ya existente en *Sprite* y sus herederas simplifica en alto grado la labor del programador, permitiendo que se concentre en la lógica del juego.

6.1.2. Detección de colisiones

A diferencia de la otra versión del laberinto, las paredes de este juego son sprites, objetos de juego en toda forma, de manera que el detectar las colisiones entre los jugadores y las paredes fue mucho más sencillo puesto que basta obtener el rectángulo que contiene al jugador y a las paredes en cuestión y probar si ambos se intersectan. Todas estas funcionalidades están ya presentes en las clases de JIG, por lo que para utilizarlas basto hacer que los *Player*s y las *Pared*es heredaran de *jig.engine.physics.vpe.VanillaAARectangle*. Lo mismo aplica a la detección de las colisiones entre el policía y la jugadora. Aquí la utilización del motor de juego y la experiencia adquirida al programar la versión previa del laberinto rindieron frutos: puesto que, por diseño de JIG, la actualización de la representación gráfica del juego se hace únicamente en el método *update* de *jig.engine.hli.AbstractSimpleGame*, no fue necesario utilizar otro hilo de ejecución para mover al policía en el laberinto, evitando así los problemas de concurrencia que surgieron en la otra versión del juego.

6.2. Problemas del desarrollo

6.2.1. La utilización de elementos Swing

La primera versión del juego *¿¿¿Tú decides si fumas???* utiliza distintos elementos Swing, como botones para navegar las instrucciones, y etiquetas con iconos en la interfaz de juego para mostrar el marcador y el tiempo. Su uso se debe a la necesidad de hacer que el juego tenga elementos de interfaz gráfica semejantes a los de la Internet, a los cuales los usuarios están acostumbrados. El uso de estos mismos elementos de interfaz gráfica en el juego en la versión utilizando JIG fue imposible, dado que en los tutores disponibles en el sitio de Internet no existen ejemplos de integración de elementos de interfaz gráfica de Swing en un juego elaborado con JIG. Otro de los motivos para no utilizar estos elementos de interfaz gráfica Swing fue la poca disponibilidad de tiempo para experimentar con el API de JIG, y para el desarrollo del juego en general. Las funciones de los mencionados elementos de interfaz

Figura 6.2: *Pantalla Inicial*

gráfica Swing se suplieron mediante el dibujo de instrucciones en texto en la pantalla del juego, las cuales tienen un atractivo visual menor, pero que dan una solución rápida al problema gracias a que este es uno de los aspectos que sí cubren los tutores en el sitio en Internet de JIG. Además de la falta de atractivo visual, esta solución resulta poco deseable puesto que numerosos juegos utilizan estos elementos de interfaz gráfica. La única solución posible parece ser la programación de estos elementos por cuenta del desarrollador, como hace [Brackeen2003] al programar los botones y menús utilizados en *Pong*.

6.2.2. El controlador

JIG utiliza *polling* para interactuar con el teclado y el ratón. A pesar de ello el juego experimenta retrasos notables entre las acciones del jugador a través del teclado y el momento en que el motor del juego realiza estás acciones en pantalla, los cuales no pudieron ser solucionados por el autor. El análisis del código con el profiler de Netbeans muestra que estos retrasos no son atribuibles al código realizado por el autor, sino que ocurren en el código de JIG.

Figura 6.3: *Instrucciones*

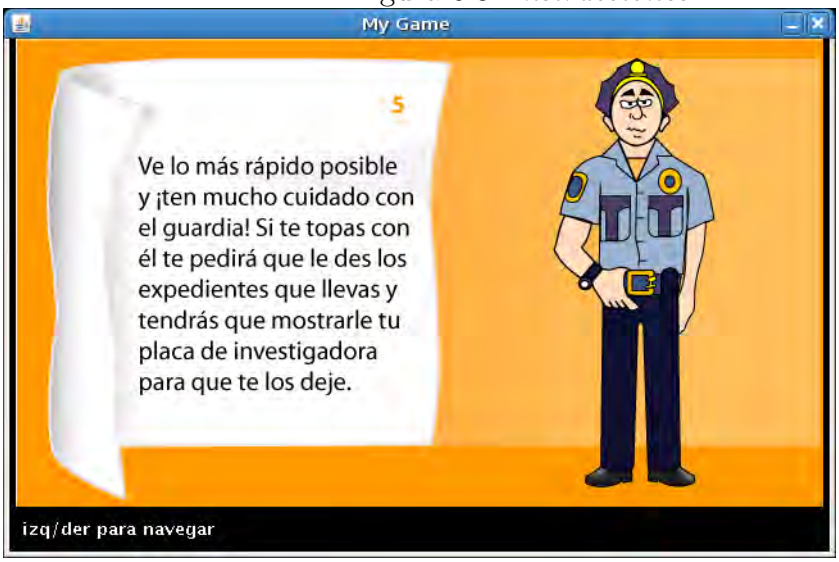


Figura 6.4: *Selección de jugadora*

Figura 6.5: *Pantalla de juego*

Figura 6.6: *Leyendo un archivo*

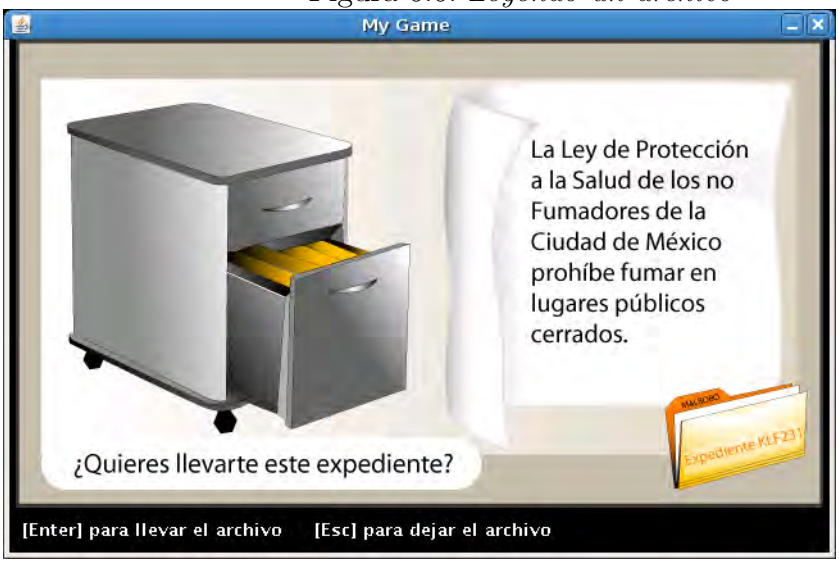
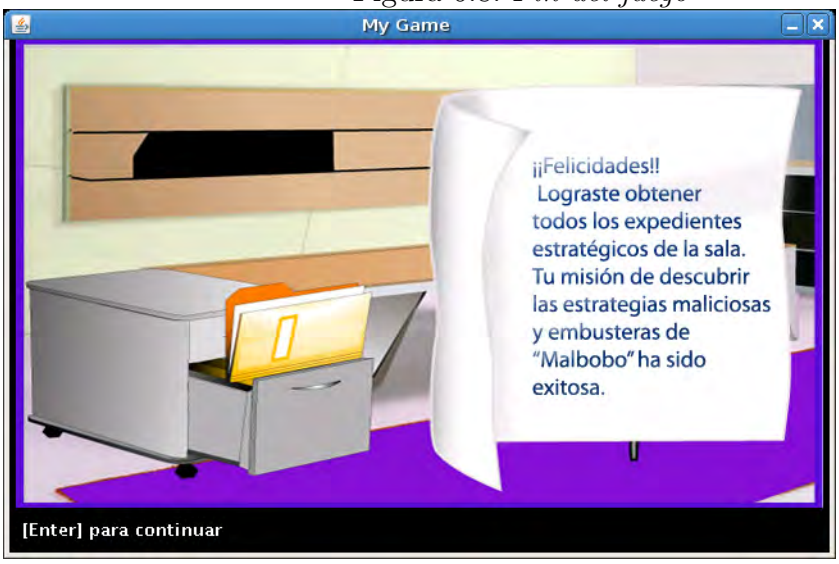


Figura 6.7: *Capturada por el policía*

Figura 6.8: *Fin del juego*



6.3. Conclusión

El juego implementa las mismas características que la versión que no utiliza JIG. El uso del motor de juego redujo dramáticamente los tiempos de desarrollo, y la complejidad de tareas como la detección de colisiones, obtención de imágenes desde el sistema de archivos, y entrada desde el teclado. El juego no incorpora elementos de interfaz gráfica de Swing, lo cual podría constituir un defecto serio en algunos proyectos, los cuales fueron suplidos mediante mensajes de texto en este juego en particular, lo que resulta suficiente para los requerimientos del mismo.

Capítulo 7

Conclusiones

Los 3 juegos constituyen aplicaciones prácticas de los fundamentos de las ciencias computacionales, como son el uso adecuado de estructuras de datos y de algoritmos de búsqueda, además de ejemplos de la conjugación de muy distintas disciplinas de las ciencias computacionales (graficación, concurrencia, inteligencia artificial, por citar algunas) en proyectos entretenidos para los usuarios y el desarrollador.

Consideraciones de eficiencia por uso de red El juego a través de Internet plantea al programador problemas de eficiencia serios. En el caso de *Pong* el uso de llamadas RMI en la vista se vio dificultado por los retrasos que su uso a través de Internet impone. Para solucionar estos problemas se debió construir un objeto *Serializable* que resumiera los datos del modelo necesarios a la vista y cuya transmisión por red se realizara únicamente al producirse un cambio en el modelo(Clase *EngineState*). Si este tipo de problemas comienzan a experimentarse en un juego tan sencillo como *Pong*, es de esperarse que impongan limitaciones más serias a juegos de mayor complejidad o enfocados más específicamente al juego en linea. Probablemente la solución a estos problemas radica en el uso de APIs especializadas, pero la comprobación de ello rebasa los alcances de este documento. Si bien el autor desconoce la magnitud de estos posibles problemas debido a las latencias por tráfico en la red, es necesario decir que el API de RMI de Java hace que la programación de juegos distribuidos sea transparente al desarrollador (la única diferencia entre un objeto local y uno remoto es que el objeto remoto implementa *Remote* y lanza *RemoteException*s),

de suerte que su uso para realizar estas tareas facilita en gran medida el trabajo del programador.

MVC en juegos y los controladores Utilizar MVC en *pong* puede parecer artificial , pues los objetos del modelo se corresponden normalmente a objetos de la vista, y en realidad lo es, tratándose de un juego tan simple como *pong* y que no requiere de muchas instrucciones a través del teclado. Sin embargo ya en un juego tan sencillo los beneficios son obvios: MVC vuelve al teclado – y a cualquier otro dispositivo de entrada si fuera necesario – altamente configurable, concentrando estos cambios únicamente en el código del controlador, o volviendo el controlador enteramente configurable, como es el caso con el teclado de *pong*. Este grado elevado de configurabilidad en el controlador (normalmente teclado y ratón) es una de las características a las que los usuario de juegos actuales están acostumbrados, y, como ha mostrado la experiencia con *pong*, su implementación y funcionamiento adecuado es posible utilizando Java.

En el caso del motor de juegos JIG, las latencias en la respuesta del controlador constituyen un obstáculo serio para su uso en proyectos profesionales. Si bien el motor de juegos es sencillo de utilizar, las latencias en la respuesta del controlador hacen que el juego resulte poco atractivo para el jugador, quien experimentará frustración al jugar un juego que responde de manera pobre a sus instrucciones.

Motores de juego Si bien mi experiencia respecto a este tipo de software se limita al uso de JIG en la segunda versión de *¿¿¿Tú decides si fumas???*, mi recomendación para quienes intenten por primera vez desarrollar un juego es que utilicen, siempre que sea posible, un motor de juego. El motor de juegos automatiza una serie de tareas sencillas pero que requieren muchas lineas de código: obtención de imágenes desde el sistema de archivos, programación de toda la lógica necesaria para el funcionamiento de los sprites, dibujo eficiente en pantalla, etc. Al tener todas estas tareas de bajo nivel ya implementadas, los desarrolladores pueden concentrarse en implementar la lógica del juego, que es la parte del desarrollo que requiere de mayor creatividad y cuyo diseño e implementación cuidadosa impactan en mayor grado en la experiencia del usuario final. El decidir respecto a usar o no un motor de juegos debe, en mi opinión, basarse en el tiempo disponible para el desarrollo

y en si los motores de juego disponibles cuentan con todas las características requeridas para el desarrollo o pueden integrar fácilmente a otras herramientas que cubran estas necesidades. Si bien el motor de juego JIG experimento problemas en cuanto a velocidad de respuesta al controlador, la existencia de numerosos motores de juego para Java hace que esto no constituya una limitante para el empleo del lenguaje para el desarrollo de videojuegos.

Multihilos y sincronización El uso de múltiples hilos en la primera versión de *¿¿¿Tú decides si fumas???* tuvo por objetivo simplificar la programación de tareas que, en un análisis rápido y poco acertado, parecieron independientes al autor (ciclo de juego, respuesta al usuario, movimiento del policía en el laberinto). Si bien esta separación de las tareas en distintos hilos simplificó la programación independiente de cada una de ellas, el resultado final fue que se invirtió mucho tiempo en la sincronización adecuada de estas tareas. La versión basada en JIG no requirió de estos hilos múltiples y los problemas de sincronización implícitos, demostrando que una solución más cuidadosa del problema pudo evitarlos. Esto es aplicable al desarrollo de cualquier videojuego, es decir, que a primera vista el utilizar múltiples hilos para las distintas tareas del juego simplificará el diseño. El precio a pagar por ello será tener que resolver los problemas de sincronización de dichos hilos. Esto puede resolverse únicamente mediante un diseño más cuidadoso del juego, como fue el caso en la versión de *¿¿¿Tú decides si fumas???* basada en JIG. Las herramientas que Java provee para ello (bloques *synchonized*, debuggers) ayudarán a realizar la tarea, pero no eliminan las fallas causadas por un diseño apresurado o poco cuidadoso.

Uso de widgets de Swing Como se mencionó en las secciones correspondientes, el uso de widgets Swing en los juegos resulto problemático. El el caso de *Pong* el problema se solucionó al usar los menús y botones desarrollados por Brackeen. En el caso de la versión basada en JIG de *¿¿¿Tú decides si fumas???* el problema muy probablemente fue únicamente la falta de tiempo para experimentar la posibilidad de incorporar los widgets en la interfaz de juego. Estos problemas(funcionalidad de widgets Swing en modo pantalla completa usando dibujo activo de pantalla, incorporación de widgets Swing a JIG) no se mencionan en la documentación consultada, y constituyen serias limitantes en el desa-

rrollo de juegos Java. La mejor solución parcial encontrada – los botones y menús desarrollados por Brackeen – obligan a los desarrolladores de videojuegos a programar de nuevo funcionalidades de los widgets Swing ya existentes.

Glosario

Application Programming Interface (API) La interfaz para programación de aplicaciones es la interfaz que define la forma en que un programa de aplicación puede solicitar servicios a bibliotecas o al sistema operativo. El API determina el vocabulario y las convenciones de llamada que el programador debe seguir para utilizar los servicios. Puede incluir especificaciones de rutinas, estructuras de datos, clases y protocolos utilizados para la comunicación entre el software solicitante y la biblioteca.

Double buffering Un *buffer* es una área de memoria utilizada para dibujar que no es desplegada en pantalla. La técnica de *double buffering* consiste en dibujar en el buffer en lugar de directamente a la pantalla. Cuando todo el dibujo se ha realizado, el contenido del buffer es copiado a la pantalla en una operación atómica, de manera que la pantalla solo se actualiza una vez y se evita el *flickering*.

Flickering Literalmente "parpadeo". Es un efecto visual desagradable al ojo que ocurre frecuentemente en software, sin que el problema tenga un origen en el hardware. Es causado por fallas en un programa al mantener su estado gráfico, normalmente porque el programa está dibujando directamente a la pantalla, de manera que hay momentos muy cortos en los que algunos elementos de la pantalla no han sido aun dibujados, lo cual es percibido por el ojo.

Graphical User Interface (GUI) Una interfaz gráfica de usuario es una interfaz de usuario que muestra las acciones e información disponibles al usuario como íconos gráficos e indicadores visuales, en contraposición a las interfaces basadas en texto. Normalmente el usuario realiza acciones al manipular dichos elementos gráficos.

Look-and-feel Es un termino utilizado para describir productos en campos como la mercadotecnia y la publicidad, designa la experiencia del usuario utilizando el producto, así como las características principales de su aspecto e interfaces.

RMI Siglas de Remote Method Invocation. Es un mecanismo ofrecido por Java para invocar un método de manera remota. Forma parte del entorno estándar de ejecución de Java y provee de un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas basadas exclusivamente en Java.

Role Playing Game Juego de interpretación de papeles. Género de videojuegos que incluye una amplia variedad de sistemas y estilos de juego. Actualmente, predomina la propuesta de videojuego donde se controla y representa cabalmente a un personaje (o varios), que debe cumplir con una serie de objetivos o misiones bien establecidos por los programadores; usualmente, se crea un mundo perteneciente a un tema de fantasía épica. Para ello, se viene utilizando cada vez más una vistosa interfaz gráfica para utilizar un sofisticado inventario de poderes humanos y sobrenaturales (que el jugador desarrolla poco a poco con práctica y muchas horas de juego), recursos monetarios y objetos diversos en propiedad (comprados o encontrados de manera fortuita), para el logro de las metas.Los principales atractivos que persiguen los fans de ésta clase de videojuego, son, entre otros, la jugabilidad del combate tipo medieval cada vez más complejo y realista (en ciertos aspectos, como movimientos y rasgos humanos más naturales en general), los bienes virtuales que se poseen (en especial armas y efectos de guerra encantados o mágicos, que facilitan el juego notablemente, o que simplemente, se presumen), el detalle en las estadísticas que arroja la aventura y los reconocimientos al tiempo invertido (nivel de habilidades alcanzado, mismo que define la respetabilidad del jugador ante los otros aficionados).

Scripting languaje La traducción literal es "lenguaje de guión". También conocido como *script language* ("lenguaje guión") o *extension language*("lenguaje de extensión"), se trata de un lenguaje de programación que controla a uno o mas programas de aplicación. Los scripts son diferentes al código principal de la aplicación, la cual es frecuentemente escrita en otro lenguaje. Los scripts son, frecuentemente, creados

o modificados por el usuario final. Con frecuencia son interpretados, mientras que las aplicaciones que controlan son compiladas. Su nombre se deriva del guión de las artes escénicas, usado para transmitir el diálogo que los actores en escena deben decir. Los primeros lenguajes de scripting se denominaron *batch lenguages* o *job control languages*, y se utilizaron para automatizar el proceso de edición-compilación-ligado-ejecución. Un ejemplo de una aplicación que utiliza un lenguaje de scripting es el navegador de Internet Firefox, escrito en C/C++ y controlable a través del lenguaje de scripting Javascript.

Sprite La traducción literal es "duendecillo". En graficación se designa con este termino a una imagen o animación integrada a una escena. Es un gráfico que se mueve independientemente por la pantalla, pudiendo estar animado, por lo que se mueve y cambia de imagen al mismo tiempo, encapsulando la lógica que ambas funciones requieren.

Widget En el sentido usado en este documento, un *widget* es un elemento de interfaz gráfica, como botones, menús, campos de texto, *radio-button*s, *check-boxes*, etc.

Bibliografía

[Barbagallo2004] Barbagallo, Ralph.
Wireless game development in Java with MIDP 2.0
Wordware publishing
2004

[Brackeen2003] Brackeen, David; Barker, Bret; Vanhelsuwé, Laurence
Developing Games in Java
New Riders
2003

[Bendas2002] Bendas, Dan; Myllyaho, Mauri.
Wireless Games - Review and experiment
`http://virtual.vtt.fi/virtual/proj1/projects/moose/docs/wireless%20games%20-%20review%20and%20experiment1.pdf`
PROFES 2002, LNCS 2559, pp. 587-600, 2002.
Springer-Verlag, 2002

[Davison2003] Davison, Andrew
Games programming with Java and Java3D
Borrador para el libro "Killer Game Programming in Java"
`http://fivedots.coe.psu.ac.th/~ad/jg/intro/`
Enero 14, 2003

[Davison2005] Davison, Andrew
Killer game programming in Java
O'Reilly, 2005

[Dodson2003] Dodson, Sean
Rune to move
`http://www.webcitation.org/5gM1kotDm`
Diciembre 11, 2003

[GameEngine] *Game engine*
`http://en.wikipedia.org/wiki/Game_engine`
Artículo de la wikipedia

[Gamma1995] *Design Patterns: Elements of Reusable Object-Oriented Software*
Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides
Addison-Wesley
1995

[Heiss2004] Heiss, Janice J.
The Utility Model for Online Games: Part Two of a Conversation with Chief Gaming Officer, Chris Melissinos
`http://www.j2ee.me/developer/technicalArticles/Interviews/games2/`
Juli, 2004

[Huebner2000] Robert Huebner
Postmortem of Nihilistic Software's. Vampire: The Masquerade – Redemption
`http://www.gamasutra.com/features/20000802/huebner_01.htm`
Agosto 2, 2000

[Java3D] *Java 3D*
`http://en.wikipedia.org/wiki/Java_3D`
Artículo de la wikipedia

[JavaRemoteMethodInvocation] *Java Remote Method Invocation*
`http://es.wikipedia.org/wiki/Java_Remote_Method_Invocation`
Artículo de la wikipedia

[JIG] *Java Instructional Gaming Main Page*
`http://ai.vancouver.wsu.edu/jig/wiki/index.php/Main_Page`
Sitio en Internet del proyecto Java Instructional Gaming

[ListOfGameEngines] *List of game engines*
`http://en.wikipedia.org/wiki/List_of_game_engines`
Artículo de la wikipedia

[Marner2002] Marner, J.
Evaluating Java for game development
`http://java.coe.psu.ac.th/FreeOnline/Evaluating%20Java%20for%20Game%20Development.pdf`
University of Copenhagen, Denmark.
March 4, 2002

[Martak2008] Martak, Michael.
Lesson: Full-Screen Exclusive Mode API
`http://java.sun.com/docs/books/tutorial/extra/fullscreen/index.html`
Capítulo del "Java Tutorial"

[MazeGenerationAlgorithm] *Maze generation algorithm*
`http://en.wikipedia.org/wiki/Maze_generation_algorithm`
Artículo de la wikipedia

[Model-View-Controller] *Model-view-controller*
`http://java.sun.com/blueprints/patterns/MVC-detailed.html`
Java Blue Prints

[MVC] *Model-view-controller*
`http://en.wikipedia.org/wiki/Model-view-controller`
Artículo de la wikipedia

[Potel2003] Potel, Michael J.
Samurai Romanesque, J2ME, and the battle for mobile cyberspace
IEEE Computer Graphics January/February 2003 (vol. 23 no. 1) pp. 16-23 `http://www.cs.vu.nl/~eliens/online/media/course/local/mobile.pdf`

[Reinkensmeyer2002] Reinkensmeyer, D.J.; Pang, C.T.; Nessler, J.A.; Painter, C.C.
Web-based telerehabilitation for the upper extremity after stroke.
Neural Systems and Rehabilitation Engineering, IEEE Transactions

on
Volume 10, Issue2
pp. 102-108
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1031978

[Runescape] *Runescape*
http://en.wikipedia.org/wiki/RuneScape
Artículo de la wikipedia

[RunescapeDevelopmentDiaries] *Runescape Development Diaries*
http://www.runescape.com/kbase/view.ws?guid=dev_diary

[ScriptingLanguage] *Scripting Language*
http://en.wikipedia.org/wiki/Scripting_language
Artículo de la wikipedia

[Sweedyk2005] Sweedyk, E., deLaet, M., Slattery, M. C., and Kuffner, J.
Computer games and CS education: why and how
Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education(St. Louis, Missouri, USA, February 23 - 27, 2005). SIGCSE '05. ACM, New York, NY, 256-257.
http://doi.acm.org/10.1145/1047344.1047433

[tesis] *tesis*
http://publab03.coseac.unam.mx/~ozamudio/tesis
El código de los 3 juegos

[TrailRMI] *Trail: RMI*
http://java.sun.com/docs/books/tutorial/rmi/index.html
Capítulo del "Java Tutorial"

[Twilleager2004] Twilleager, Doug; Kesselman, Jeff; Goldberg, Athomas; Petersen, Daniel; Soto, Juan Carlos; Melissinos, Chris.
Java Technologies For Games
http://portal.acm.org/citation.cfm?id=1008213.1008242
ACM Computers in Entertainment, Volume 2, Number 2, April 2004, Article 8.

[VideojuegoDeRol] *Videojuego de Rol*
`http://es.wikipedia.org/wiki/Videojuego_de_rol`
Artículo de la wikipedia

[Wallace2006] Scott A. Wallace, Andrew Nierman
Addressing the need for a java based game curriculum
Journal of Computing Sciences in Colleges 22, 2(Dec. 2006), 20-26.
`http://portal.acm.org/citation.cfm?id=1181901.1181905`

[Wang2007] Wang, Y., Wu, I., and Jiang, J
A portable AWT-Swing architecture for Java game development.
Software – Practice & Experience
Volume 37 , Issue 7(June 2007)
pp. 727 - 745
`http://www3.interscience.wiley.com/journal/113451017/abstract?CRETRY=1&SRETRY=0`