UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

esi Escuela Superior de Informática

tsi Departamento de Tecnología y Sistemas de Información

tegnix

Desarrollo de Videojuegos Multiplataforma

Un enfoque práctico con NME

David Vallejo · Carlos González · David Frutos

Título: Desarrollo de Videojuegos Multiplataforma. Un enfoque práctico con NME.
Autores: David Vallejo Fernández, Carlos González Morcillo y David Frutos Talavera
ISBN papel: 978-84-686-4383-0
ISBN digital: 978-84-686-4384-7
Edita: Bubok Publishing S.L
Diseño: Carlos González Morcillo
Impreso en España

Este libro fue compuesto con LaTeX a partir de una plantilla de Carlos González Morcillo y Sergio García Mondaray. La portada y las entradillas fueron diseñadas con GIMP, Blender, InkScape y OpenOffice.

esi Escuela Superior de Informática

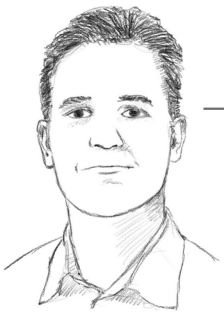
tsi Departamento de Tecnología y Sistemas de Información

tegnix

Creative Commons License: Usted es libre de copiar, distribuir y comunicar públicamente la obra, bajo las condiciones siguientes: 1. Reconocimiento. Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador. 2. No comercial. No puede utilizar esta obra para fines comerciales. 3. Sin obras derivadas. No se puede alterar, transformar o generar una obra derivada a partir de esta obra. Más información en: http://creativecommons.org/licenses/by-nc-nd/3.0/

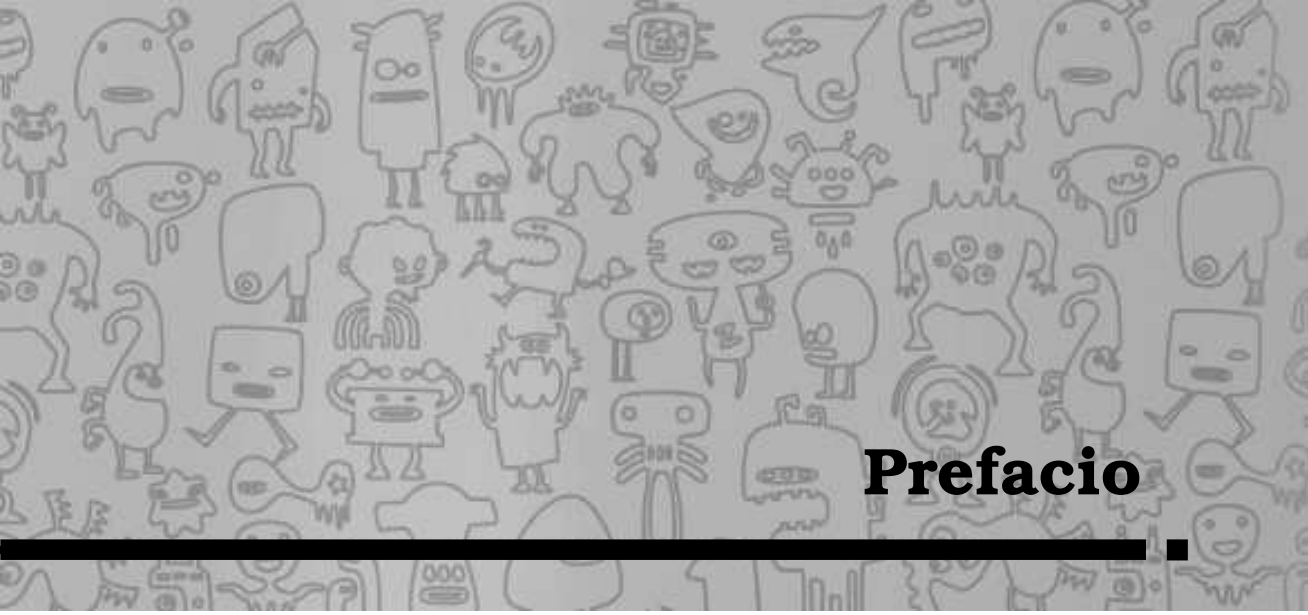
Autores

David Vallejo (2009, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en asignaturas relacionadas con Informática Gráfica, Programación y Sistemas Operativos desde 2007. Actualmente, su actividad investigadora gira en torno a la Vigilancia Inteligente, los Sistemas Multi-Agente y el Rendering Distribuido.



Carlos González (2007, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Titular de Universidad e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en asignaturas relacionadas con Informática Gráfica, Síntesis de Imagen Realista y Sistemas Operativos desde 2002. Actualmente, su actividad investigadora gira en torno a los Sistemas Multi-Agente, el Rendering Distribuido y la Realidad Aumentada.

David Frutos (Ingeniero Técnico en Informática de Sistemas, Universidad de Castilla-La Mancha). Experto en desarrollo de videojuegos para plataformas móviles con Haxe-NME. Apasionado del mundo de los videojuegos, obtuvo la máxima calificación académica como alumno de la primera edición del Curso de Experto en Desarrollo de Videojuegos de la Universidad de Castilla-La Mancha.



Prefacio

El desarrollo de videojuegos ha sufrido una evolución vertiginosa en los últimos años. Prueba de ello es la gran cantidad de entornos y dispositivos para los cuales se crean videojuegos. Desde redes sociales hasta plataformas móviles, pasando por las tradicionales consolas de sobremesa y PCs, los videojuegos están presentes en el día a día de cada vez más personas.

Este libro surge como respuesta al desarrollo multi-plataforma de videojuegos, especialmente en el ámbito de los dispositivos móviles. Concebido desde una perspectiva esencialmente práctica, este libro te ayudará a programar tu propio videojuego utilizando NME. Gracias a él, serás capaz de implementar tu propio bucle de juego, integrar recursos gráficos y física, añadir efectos de sonido e incluso desarrollar módulos de Inteligencia Artificial y Networking.

NME es un framework *open-source* multi-plataforma que tiene soporte para Windows, Mac, Linux, iOS, Android, BlackBerry, Flash y HTML5. En los últimos años, NME ha sufrido una evolución drástica y el potencial que ofrece es enorme debido al gran compromiso de la comunidad que lo soporta. NME se apoya en el moderno compilador del lenguaje de programación Haxe, el cual permite generar código para diversas plataformas sin sacrificar el rendimiento de la aplicación final.

La versión electrónica de este libro, junto con todos los ejemplos de código fuente, puede descargarse desde http://www.openflbook.com. El libro «físico» puede adquirirse desde la página web de la editorial online *Bubok* en http://www.bubok.es.

Desde aquí también te invitamos a que visites la web oficial del *Curso de Experto en Desarrollo de Videojuegos*, http://www.cedv.es, impartido en la Universidad de Castilla-La Mancha y cuyo material docente también está disponible en dicha web.

Programas y código fuente

Debido a que NME y Haxe son herramientas relativamente modernas, su evolución es constante. En este contexto, todos los ejemplos de código fuente discutidos en este libro, a fecha de Julio de 2013, se pueden compilar con la versión 3.5.5 de NME y la versión 2.10 de Haxe.

El código de los ejemplos del libro puede descargarse en la siguiente página web: http://www.openflbook.com. Salvo que se especifique explícitamente otra licencia, todos los ejemplos del libro se distribuyen bajo GPLv3.

Requisitos previos

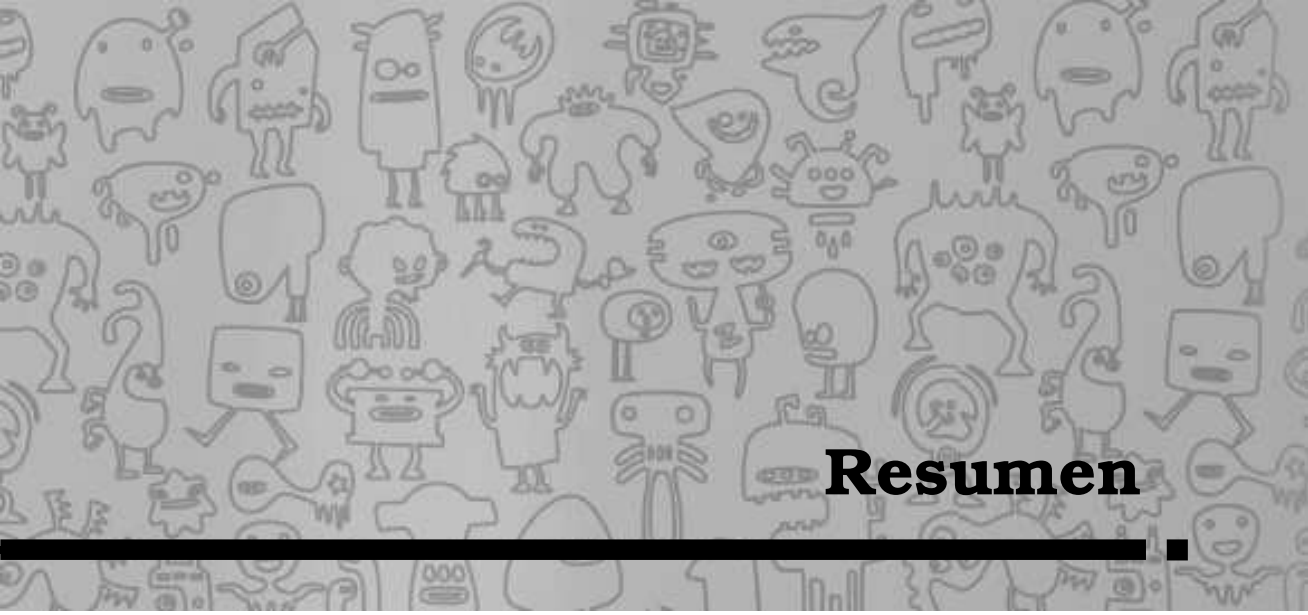
Este libro tiene un público objetivo con un perfil principalmente técnico. En otras palabras, este libro no está orientado para un público de perfil artístico (modeladores, animadores, músicos, etc) en el ámbito de los videojuegos.

Se asume que el lector tiene unos conocimientos de programación medios. En el libro se realiza una breve introducción al lenguaje de programación Haxe, pero no se discuten sus características ni sus diferencias con respecto a otros lenguajes de programación. De igual modo, se asume que el lector tiene conocimientos de estructuras de datos y algoritmia.

Agradecimientos

Los autores del libro quieren agradecer a Aula Tegnix y a la Xunta de Galicia la financiación del curso titulado *Programación Multimedia e Xogos*, cuyo material docente fue la base del presente libro. Dicho curso fue impartido por los profesores Carlos González Morcillo y David Vallejo Fernández, ambos coautores del mismo y profesores del Departamento de Tecnologías y Sistemas de Información de la Escuela Superior de Informática de la Universidad de Castilla-La Mancha.

Este agradecimiento también se hace extensivo a la Escuela de Informatica de Ciudad Real y al Departamento de Tecnologías y Sistema de Información de la Universidad de Castilla-La Mancha.



Resumen

Año tras año, el desarrollo de videojuegos se ha afianzado hasta convertirse en la industria del entrenimiento más importante, superando a las industrias cinematográfica y musical. Tanto es así, que la facturación generada en torno al mundo de los videojuegos supera los 30.000 millones de euros anuales. Una de las fracciones más relevantes de esta cifra está representada con el software, es decir, con el procesos de diseño y desarrollo de videojuegos.

La variedad de dispositivos hardware existentes (consolas, ordenadores, *smartphones*, *tablets*, ect) tiene como consecuencia directa que los desarrolladores de videojuegos hagan uso de herramientas o frameworks que faciliten el desarrollo multi-plataforma. El motivo es claro: obtener un mayor retorno de la inversión realizada y no depender de una única tecnología.

En este contexto, el principal objetivo de este libro consiste en estudiar, desde una perspectiva práctica, el diseño y desarrollo de un videojuego completo utilizando un *framework* multi-plataforma que permita la generación de ejecutables para distintas plataformas. En concreto, el *framework* utilizado es NME, el cual está basado en el popular lenguaje de programación multi-plataforma Haxe.

Así, el libro plantea una introducción al desarrollo de videojuegos, mostrando la arquitectura típica de un motor de juegos, y discute cómo diseñar y desarrollar un videojuego completo con NME mediante un tutorial incremental.

Índice general

1. Introducción **1**

1.1. El desarrollo de videojuegos 1

1.1.1. La industria del videojuego. Presente y futuro 1

1.1.2. Estructura típica de un equipo de desarrollo 4

1.1.3. El concepto de juego 7

1.1.4. Motor de juego 10

1.1.5. Géneros de juegos 12

1.2. Arquitectura del motor. Visión general 20

1.2.1. Hardware, *drivers* y sistema operativo 20

1.2.2. SDKs y *middlewares* 22

1.2.3. Capa independiente de la plataforma 23

1.2.4. Subsistemas principales 24

1.2.5. Gestor de recursos 25

1.2.6. Motor de *rendering* 27

1.2.7. Herramientas de depuración 30

1.2.8. Motor de física 31

1.2.9. Interfaces de usuario 32

1.2.10 *Networking* y multijugador 32

1.2.11 Subsistema de juego 33

1.2.12 Audio 36
1.2.13 Subsistemas específicos de juego 36

2. Entorno de Trabajo **37**
2.1. NME. Toma de contacto 37
2.1.1. ¿Qué es NME? 37
2.1.2. El lenguaje de programación Haxe 43
2.1.3. Instalación y configuración de NME 48
2.2. *Hello World!* con NME 53

3. Tutorial de Desarrollo con NME **61**
3.1. El bucle de juego 62
3.1.1. El bucle de renderizado 62
3.1.2. Visión general del bucle de juego 63
3.1.3. Arquitecturas típicas del bucle de juego 64
3.1.4. Gestión de estados de juego con NME 69
3.1.5. *BubbleDemo*: definición de estados concretos 75
3.2. Recursos Gráficos y Representación 83
3.2.1. Introducción 83
3.2.2. Sprites 88
3.2.3. Capas y Tiles 90
3.2.4. Animación de Sprites: TileClip 94
3.2.5. Texto con True Type 97
3.2.6. Scroll Parallax 98
3.2.7. Bee Adventures: Mini Juego 103
3.2.8. Sistemas de Partículas 114
3.3. Gestión de sonido 117
3.3.1. NME y su soporte de sonido 118
3.3.2. La clase SoundManager 118
3.3.3. Integrando sonido en Bee Adventures 125
3.4. Simulación Física 131
3.4.1. Algunos Motores de Simulación 133

3.4.2. Aspectos destacables 135
3.4.3. Conceptos Básicos 136
3.4.4. Formas de Colisión 137
3.4.5. Optimizaciones 140
3.4.6. Hola Mundo con Physaxe 141
3.4.7. Más allá del Hola Mundo 147
3.5. Inteligencia Artificial 156
3.5.1. Introducción 156
3.5.2. Aplicando el Test de Turing 157
3.5.3. Ilusión de inteligencia 159
3.5.4. ¿NPCs o Agentes? 160
3.5.5. Diseño de agentes basado en estados 162
3.5.6. Búsqueda entre adversarios 164
3.5.7. Caso de estudio. Un Tetris *inteligente* 174
3.5.8. Caso de estudio. Tres en raya (tic-tac-toe) con NME 177
3.6. Networking 184
3.6.1. Introducción 184
3.6.2. Consideraciones iniciales de diseño 185
3.6.3. Sockets TCP/IP 187
3.6.4. Gestión on-line de récords en Tic-Tac-Toe 191

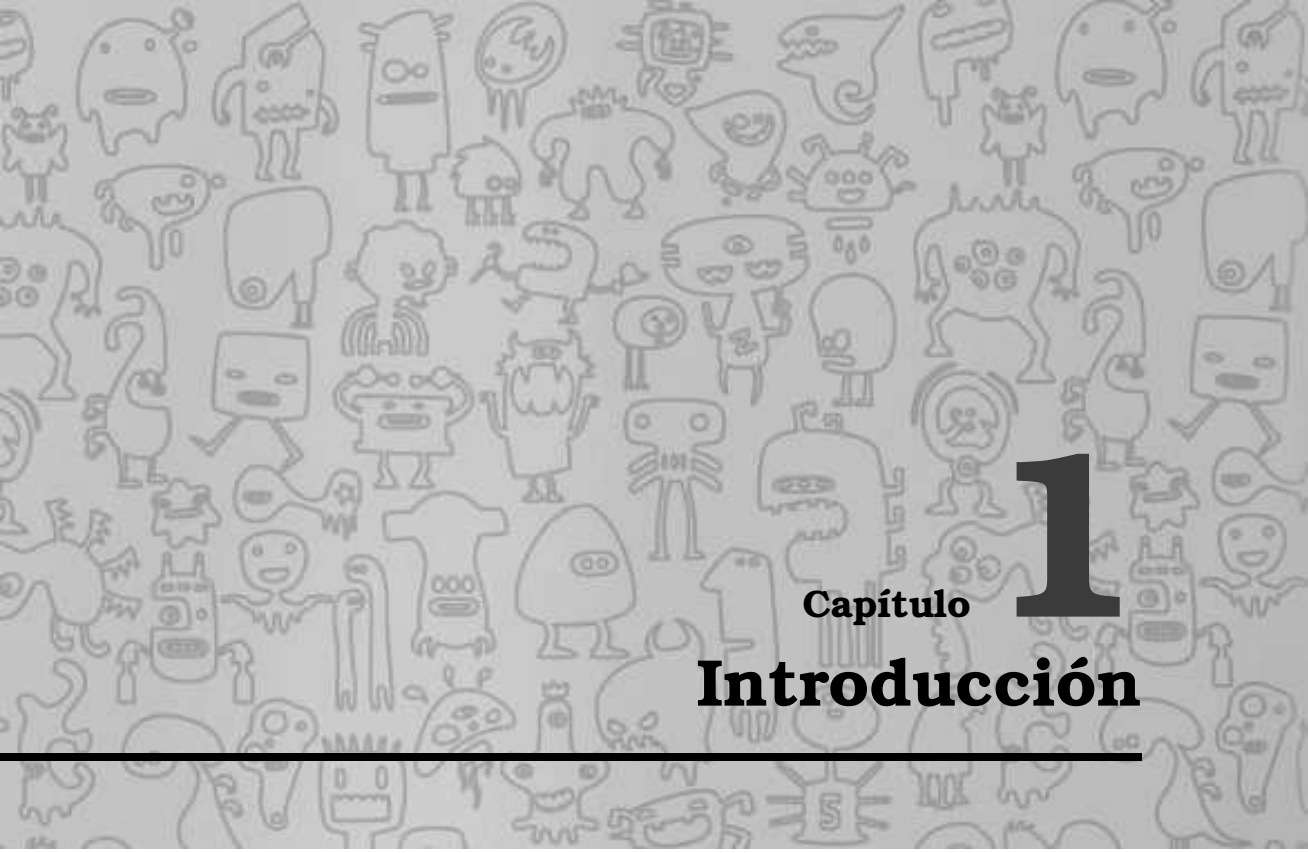
A. Despliegue y Distribución 199

A.1. Despliegue en Flash 200
A.1.1. Usando bibliotecas externas en Flash 200
A.1.2. Caso de estudio. Operaciones con una biblioteca externa. 203
A.1.3. Mochi Media, publicidad y *highscores* 206
A.2. Caso de estudio. Implementando MochiAPI en OfficeBasket 215
A.2.1. Añadiendo un final al juego y un tablón de récords . 216
A.2.2. Añadiendo publicidad al juego 219
A.3. Kongregate. Gestión de logros 221

A.3.1. Instalación y uso con NME 221
A.3.2. Carga del juego en Kongregate 223
A.4. Despliegue en Android. Google play 225
A.4.1. Firmando la APK 225
A.4.2. Subiendo la APK a Google Play. Distribución 226

Listado de acrónimos

API	Application Program Interface
AVD	Android Virtual Device
BSP	Binary Space Partitioning
E/S	Entrada/Salida
FPS	First Person Shooter
GPL	General Public License
GUI	Graphical User Interface
IA	Inteligencia Artificial
IBM	International Business Machines
IGC	In-Game Cinematics
KISS	Keep it simple, Stupid!
LTS	Long Term Support
MMOG	Massively Multiplayer Online Game
NPC	Non-Player Character
ODE	Open Dynamics Engine
OGRE	Object-Oriented Graphics Rendering Engine
OO	Orientación a Objetos
POO	Programación Orientada a Objetos
RPG	Role-Playing Games
RTS	Real-Time Strategy
SDK	Software Development Kit
SGBD	Sistema de Gestión de Base de Datos
STL	Standard Template Library



Capítulo 1

Introducción

Actualmente, la industria del videojuego goza de una muy buena salud a nivel mundial, rivalizando en presupuesto con las industrias cinematográfica y musical. En este capítulo se discute, desde una perspectiva general, el **desarrollo de videojuegos**, haciendo especial hincapié en su evolución y en los distintos elementos involucrados en este complejo proceso de desarrollo.

En la segunda parte del capítulo se introduce el concepto de **arquitectura del motor**, como eje fundamental para el diseño y desarrollo de videojuegos comerciales.

1.1. El desarrollo de videojuegos

1.1.1. La industria del videojuego. Presente y futuro

Lejos han quedado los días desde el desarrollo de los primeros videojuegos, caracterizados principalmente por su simplicidad y por el hecho de estar desarrollados completamente sobre hardware. Debido a los distintos avances en el campo de la informática, no sólo a nivel de desarrollo software y capacidad hardware sino también en la aplicación

de métodos, técnicas y algoritmos, la industria del videojuego ha evolucionado hasta llegar a cotas inimaginables, tanto a nivel de jugabilidad como de calidad gráfica, tan sólo hace unos años.

La **evolución** de la industria de los videojuegos ha estado ligada a una serie de hitos, determinados particularmente por juegos que han marcado un antes y un después, o por fenómenos sociales que han afectado de manera directa a dicha industria. Juegos como *Doom*, *Quake*, *Final Fantasy*, *Zelda*, *Tekken*, *Gran Turismo*, *Metal Gear*, *The Sims* o *World of Warcraft*, entre otros, han marcado tendencia y han contribuido de manera significativa al desarrollo de videojuegos en distintos géneros.

El videojuego *Pong* se considera como unos de los primeros videojuegos de la historia. Desarrollado por Atari en 1975, el juego iba incluido en la consola *Atari Pong*. Se calcula que se vendieron unas 50.000 unidades.

Por otra parte, y de manera complementaria a la aparición de estas obras de arte, la propia evolución de la informática ha posibilitado la vertiginosa evolución del desarrollo de videojuegos. Algunos **hitos clave** son por ejemplo el uso de la tecnología poligonal en 3D [1] en las consolas de sobremesa, el *boom* de los ordenadores personales como plataforma multi-propósito, la expansión de Internet, los avances en el desarrollo de microprocesadores, el uso de *shaders* programables [11], el desarrollo de motores de juegos o, más recientemente, la eclosión de las redes sociales y el uso masivo de dispositivos móviles.

Por todo ello, los videojuegos se pueden encontrar en ordenadores personales, consolas de juego de sobremesa, consolas portátiles, dispositivos móviles como por ejemplo los *smartphones*, o incluso en las redes sociales como medio de soporte para el entretenimiento de cualquier tipo de usuario. Esta diversidad también está especialmente ligada a distintos tipos o géneros de videojuegos, como se introducirá más adelante en esta misma sección.

La **expansión del videojuego** es tan relevante que actualmente se trata de una industria multimillonaria capaz de rivalizar con las industrias cinematográfica y musical. Un ejemplo representativo es el valor total del mercado del videojuego en Europa, tanto a nivel hardware como software, el cual alcanzó la nada desdeñable cifra de casi 11.000 millones de euros, con países como Reino Unido, Francia o Alemania a la cabeza. En este contexto, España representa el cuarto consumidor a nivel europeo y también ocupa una posición destacada dentro del *ranking* mundial.

A pesar de la vertiginosa evolución de la industria del videojuego, hoy en día existe un gran número de **retos** que el desarrollador de videojuegos ha de afrontar a la hora de producir un videojuego. En realidad, existen retos que perdurarán eternamente y que no están ligados a la propia evolución del hardware que permite la ejecución de los videojuegos. El más evidente de ellos es la necesidad imperiosa de ofrecer una experiencia de entretenimiento al usuario basada en la diversión, ya sea a través de nuevas formas de interacción, como por ejemplo la realidad aumentada o la tecnología de visualización 3D, a través de una mejora evidente en la calidad de los títulos, o mediante innovación en aspectos vinculados a la jugabilidad.

No obstante, actualmente la evolución de los videojuegos está estrechamente ligada a la **evolución del hardware** que permite la ejecución de los mismos. Esta evolución atiende, principalmente, a dos factores: i) la potencia de dicho hardware y ii) las capacidades interactivas del mismo. En el primer caso, una mayor potencia hardware implica que el desarrollador disfrute de mayores posibilidades a la hora de, por ejemplo, mejorar la calidad gráfica de un título o de incrementar la IA (Inteligencia Artificial) de los enemigos. Este factor está vinculado al **multiprocesamiento**. En el segundo caso, una mayor riqueza en términos de interactividad puede contribuir a que el usuario de videojuegos viva una experiencia más inmersiva (por ejemplo, mediante realidad aumentada) o, simplemente, más natural (por ejemplo, mediante la pantalla táctil de un *smartphone*).

Finalmente, resulta especialmente importante destacar la existencia de **motores de juego** (*game engines*), como por ejemplo *Quake*[1] o *Unreal*[2], *middlewares* para el tratamiento de aspectos específicos de un juego, como por ejemplo la biblioteca *Havok*[3] para el tratamiento de la física, o motores de renderizado, como por ejemplo *Ogre 3D* [8]. Este tipo de herramientas, junto con técnicas específicas de desarrollo y optimización, metodologías de desarrollo, o patrones de diseño, entre otros, conforman un aspecto esencial a la hora de desarrollar un videojuego. Al igual que ocurre en otros aspectos relacionados con la Ingeniería del Software, desde un punto de vista general resulta aconsejable el uso de todos estos elementos para agilizar el proceso de desarrollo y reducir errores potenciales. En otras palabras, no es necesario, ni productivo, reinventar la rueda cada vez que se afronta un nuevo proyecto.

[1] http://www.idsoftware.com/games/quake/quake/
[2] http://www.unrealengine.com/
[3] http://www.havok.com

1.1.2. Estructura típica de un equipo de desarrollo

El desarrollo de videojuegos comerciales es un proceso complejo debido a los distintos requisitos que ha de satisfacer y a la integración de distintas disciplinas que intervienen en dicho proceso. Desde un punto de vista general, un videojuego es una **aplicación gráfica en tiempo real** en la que existe una interacción explícita mediante el usuario y el propio videojuego. En este contexto, el concepto de tiempo real se refiere a la necesidad de generar una determinada tasa de *frames* o imágenes por segundo, típicamente 30 ó 60, para que el usuario tenga una sensación continua de realidad. Por otra parte, la interacción se refiere a la forma de comunicación existente entre el usuario y el videojuego. Normalmente, esta interacción se realiza mediante *joysticks* o mandos, pero también es posible llevarla a cabo con otros dispositivos como por ejemplo teclados, ratones, cascos o incluso mediante el propio cuerpo a través de técnicas de visión por computador o de interacción táctil.

> Tiempo real
>
> En el ámbito del desarrollo de videojuegos, el concepto de tiempo real es muy importante para dotar de realismo a los juegos, pero no es tan estricto como el concepto de tiempo real manejado en los sistemas críticos.

A continuación se describe la estructura típica de un equipo de desarrollo atendiendo a los distintos roles que juegan los componentes de dicho equipo [6]. En muchos casos, y en función del número de componentes del equipo, hay personas especializadas en diversas disciplinas de manera simultánea.

Los **ingenieros** son los responsables de diseñar e implementar el software que permite la ejecución del juego, así como las herramientas que dan soporte a dicha ejecución. Normalmente, los ingenieros se suelen clasificar en dos grandes grupos:

- Los **programadores del núcleo** del juego, es decir, las personas responsables de desarrollar tanto el motor de juego como el juego propiamente dicho.
- Los **programadores de herramientas**, es decir, las personas responsables de desarrollar las herramientas que permiten que el resto del equipo de desarrollo pueda trabajar de manera eficiente.

De manera independiente a los dos grupos mencionados, los ingenieros se pueden especializar en una o en varias disciplinas. Por ejemplo, resulta bastante común encontrar perfiles de ingenieros especializados

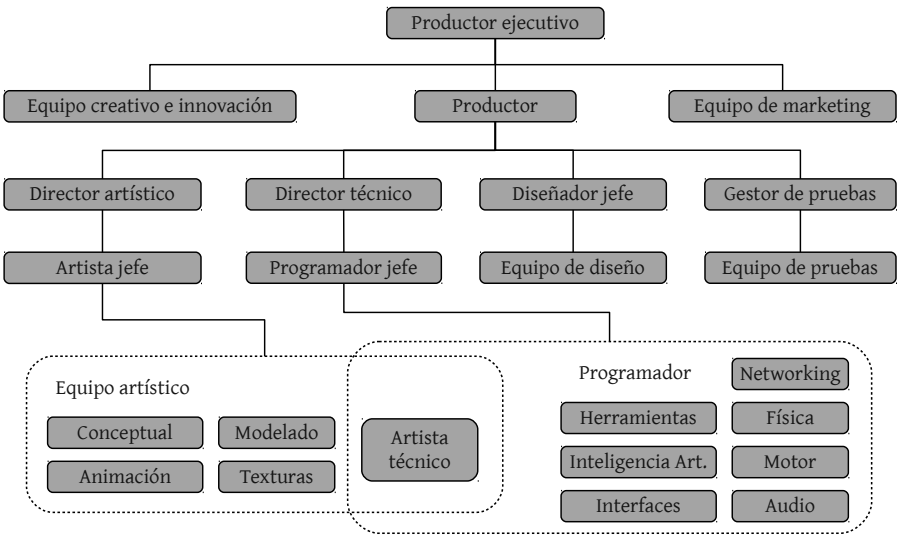


Figura 1.1: Visión conceptual de un equipo de desarrollo de videojuegos, considerando especialmente la parte de programación.

en programación gráfica o en *scripting* e IA. Sin embargo, tal y como se sugirió anteriormente, el concepto de *ingeniero transversal* es bastante común, particularmente en equipos de desarrollo que tienen un número reducido de componentes o con un presupuesto que no les permite la contratación de personas especializadas en una única disciplina.

En el mundo del desarrollo de videojuegos, es bastante probable encontrar ingenieros *senior* responsables de supervisar el desarrollo desde un punto de vista técnico, de manera independiente al diseño y generación de código. No obstante, este tipo de roles suelen estar asociados a la supervisión técnica, la gestión del proyecto e incluso a la toma de decisiones vinculadas a la dirección del proyecto. Así mismo, algunas compañías también pueden tener directores técnicos, responsables de la supervisión de uno o varios proyectos, e incluso un director ejecutivo, encargado de ser el director técnico del estudio completo y de mantener, normalmente, un rol ejecutivo en la compañía o empresa.

Los **artistas** son los responsables de la creación de todo el contenido audio-visual del videojuego, como por ejemplo los escenarios, los personajes, las animaciones de dichos personajes, etc. Al igual que ocurre en el caso de los ingenieros, los artistas también se pueden especializar en diversas cuestiones, destacando las siguientes:

General VS Específico

En función del tamaño de una empresa de desarrollo de videojuegos, el nivel de especialización de sus empleados es mayor o menor. Sin embargo, las ofertas de trabajo suelen incluir diversas disciplinas de trabajo para facilitar su integración.

- Artistas de concepto, responsables de crear bocetos que permitan al resto del equipo hacerse una idea inicial del aspecto final del videojuego. Su trabajo resulta especialmente importante en las primeras fases de un proyecto.
- Modeladores, responsables de generar el contenido 3D del videojuego, como por ejemplo los escenarios o los propios personajes que forman parte del mismo.
- Artistas de texturizado, responsables de crear las texturas o imágenes bidimensionales que formarán parte del contenido visual del juego. Las texturas se aplican sobre la geometría de los modelos con el objetivo de dotarlos de mayor realismo.
- Artistas de iluminación, responsables de gestionar las fuentes de luz del videojuego, así como sus principales propiedades, tanto estáticas como dinámicas.
- Animadores, responsables de dotar de movimientos a los personajes y objetos dinámicos del videojuego. Un ejemplo típico de animación podría ser el movimiento de brazos de un determinado carácter.
- Actores de captura de movimiento, responsables de obtener datos de movimiento reales para que los animadores puedan integrarlos a la hora de animar los personajes.
- Diseñadores de sonido, responsables de integrar los efectos de sonido del videojuego.
- Otros actores, responsables de diversas tareas como por ejemplo los encargados de dotar de voz a los personajes.

Al igual que suele ocurrir con los ingenieros, existe el rol de artista *senior* cuyas responsabilidades también incluyen la supervisión de los numerosos aspectos vinculados al componente artístico.

Los **diseñadores de juego** son los responsables de diseñar el contenido del juego, destacando la evolución del mismo desde el principio

hasta el final, la secuencia de capítulos, las reglas del juego, los objetivos principales y secundarios, etc. Evidentemente, todos los aspectos de diseño están estrechamente ligados al propio género del mismo. Por ejemplo, en un juego de conducción es tarea de los diseñadores definir el comportamiento de los coches adversarios ante, por ejemplo, el adelantamiento de un rival.

Los diseñadores suelen trabajar directamente con los ingenieros para afrontar diversos retos, como por ejemplo el comportamiento de los enemigos en una aventura. De hecho, es bastante común que los propios diseñadores programen, junto con los ingenieros, dichos aspectos haciendo uso de lenguajes de *scripting* de alto nivel, como por ejemplo *LUA*[4] o *Python*[5].

> **Scripting e IA**
>
> El uso de lenguajes de alto nivel es bastante común en el desarrollo de videojuegos y permite diferenciar claramente la lógica de la aplicación y la propia implementación. Una parte significativa de las desarrolladoras utiliza su propio lenguaje de *scripting*, aunque existen lenguajes ampliamente utilizados, como son *LUA* o *Python*.

Como ocurre con las otras disciplinas previamente comentadas, en algunos estudios los diseñadores de juego también juegan roles de gestión y supervisión técnica.

Finalmente, en el desarrollo de videojuegos también están presentes roles vinculados a la producción, especialmente en estudios de mayor capacidad, asociados a la planificación del proyecto y a la gestión de recursos humanos. En algunas ocasiones, los productores también asumen roles relacionados con el diseño del juego. Así mismo, los responsables de *marketing*, de administración y de soporte juegan un papel relevante. También resulta importante resaltar la figura de publicador como entidad responsable del *marketing* y distribución del videojuego desarrollado por un determinado estudio. Mientras algunos estudios tienen contratos permanentes con un determinado publicador, otros prefieren mantener una relación temporal y asociarse con el publicador que le ofrezca mejores condiciones para gestionar el lanzamiento de un título.

1.1.3. El concepto de juego

Dentro del mundo del entretenimiento electrónico, un **juego** normalmente se suele asociar a la evolución, entendida desde un punto de

[4] http://www.lua.org
[5] http://www.python.org

vista general, de uno o varios personajes principales o entidades que pretenden alcanzar una serie de objetivos en un mundo acotado, los cuales están controlados por el propio usuario. Así, entre estos elementos podemos encontrar desde superhéroes hasta coches de competición pasando por equipos completos de fútbol. El mundo en el que conviven dichos personajes suele estar compuesto, normalmente, por una serie de escenarios virtuales recreados en tres dimensiones y tiene asociado una serie de reglas que determinan la interacción con el mismo.

De este modo, existe una **interacción** explícita entre el jugador o usuario de videojuegos y el propio videojuego, el cual plantea una serie de retos al usuario con el objetivo final de garantizar la diversión y el entretenimiento. Además de ofrecer este componente emocional, los videojuegos también suelen tener un componente cognitivo asociado, obligando a los jugadores a aprender técnicas y a dominar el comportamiento del personaje que manejan para resolver los retos o puzzles que los videojuegos plantean.

Desde una perspectiva más formal, la mayoría de videojuegos suponen un ejemplo representativo de lo que se define como aplicaciones gráficas o **renderizado en tiempo real** [1], las cuales se definen a su vez como la rama más interactiva de la Informática Gráfica. Desde un punto de vista abstracto, una aplicación gráfica en tiempo real se basa en un bucle donde en cada iteración se realizan los siguientes pasos:

- El usuario visualiza una imagen renderizada por la aplicación en la pantalla o dispositivo de visualización.
- El usuario actúa en función de lo que haya visualizado, interactuando directamente con la aplicación, por ejemplo mediante un teclado.
- En función de la acción realizada por el usuario, la aplicación gráfica genera una salida u otra, es decir, existe una retroalimentación que afecta a la propia aplicación.

Caída de frames

Si el núcleo de ejecución de un juego no es capaz de mantener los *fps* a un nivel constante, el juego sufrirá una caída de frames en un momento determinado. Este hecho se denomina comúnmente como *ralentización*.

En el caso de los videojuegos, este ciclo de visualización, actuación y renderizado ha de ejecutarse con una frecuencia lo suficientemente elevada como para que el usuario se sienta inmerso en el videojuego, y no lo perciba simplemente como una sucesión de imágenes estáticas. En

este contexto, el **frame rate** se define como el número de imágenes por segundo, comúnmente *fps*, que la aplicación gráfica es capaz de generar. A mayor *frame rate*, mayor sensación de realismo en el videojuego. Actualmente, una tasa de 30 *fps* se considera más que aceptable para la mayoría de juegos. No obstante, algunos juegos ofrecen tasas que doblan dicha medida.

> Generalmente, el desarrollador de videojuegos ha de buscar un compromiso entre los *fps* y el grado de realismo del videojuego. Por ejemplo, el uso de modelos con una alta complejidad computacional, es decir, con un mayor número de polígonos, o la integración de comportamientos inteligentes por parte de los enemigos en un juego, o NPC (Non-Player Character), disminuirá los *fps*.

En otras palabras, los juegos son aplicaciones interactivas que están marcadas por el tiempo, es decir, cada uno de los ciclos de ejecución tiene un *deadline* que ha de cumplirse para no perder realismo.

Aunque el **componente gráfico** representa gran parte de la complejidad computacional de los videojuegos, no es el único. En cada ciclo de ejecución, el videojuego ha de tener en cuenta la evolución del mundo en el que se desarrolla el mismo. Dicha evolución dependerá del estado de dicho mundo en un momento determinado y de cómo las distintas entidades dinámicas interactúan con él. Obviamente, recrear el mundo real con un nivel de exactitud elevado no resulta manejable ni práctico, por lo que normalmente dicho mundo se aproxima y se simplifica, utilizando modelos matemáticos para tratar con su complejidad. En este contexto, destaca por ejemplo la simulación física de los propios elementos que forman parte del mundo.

Por otra parte, un juego también está ligado al comportamiento del personaje principal y del resto de entidades que existen dentro del mundo virtual. En el ámbito académico, estas entidades se suelen definir como **agentes** (*agents*) y se encuadran dentro de la denominada *simulación basada en agentes* [10]. Básicamente, este tipo de aproximaciones tiene como objetivo dotar a los NPC con cierta inteligencia para incrementar el grado de realismo de un juego estableciendo, incluso, mecanismos de cooperación y coordinación entre los mismos. Respecto al personaje principal, un videojuego ha de contemplar las distintas acciones realizadas por el mismo, considerando la posibilidad de decisiones impredecibles a priori y las consecuencias que podrían desencadenar.

En resumen, y desde un punto de vista general, el desarrollo de un juego implica considerar un gran número de factores que, inevitable-

Figura 1.2: El motor de juego representa el núcleo de un videojuego y determina el comportamiento de los distintos módulos que lo componen.

mente, incrementan la complejidad del mismo y, al mismo tiempo, garantizar una tasa de *fps* adecuada para que la inmersión del usuario no se vea afectada.

1.1.4. Motor de juego

Al igual que ocurre en otras disciplinas en el campo de la informática, el desarrollo de videojuegos se ha beneficiado de la aparición de herramientas que facilitan dicho desarrollo, automatizando determinadas tareas y ocultando la complejidad inherente a muchos procesos de bajo nivel. Si, por ejemplo, los SGBD han facilitado enormemente la gestión de persistencia de innumerables aplicaciones informáticas, los motores de juegos hacen la vida más sencilla a los desarrolladores de videojuegos.

Según [6], el término *motor de juego* surgió a mediados de los años 90 con la aparición del famosísimo juego de acción en primera persona *Doom*, desarrollado por la compañía *id Software* bajo la dirección de *John Carmack*[6]. Esta afirmación se sustenta sobre el hecho de que *Doom* fue diseñado con una **arquitectura orientada a la reutilización** mediante una separación adecuada en distintos módulos de los componentes fundamentales, como por ejemplo el sistema de renderizado gráfico, el sistema de detección de colisiones o el sistema de audio, y los elementos más *artísticos*, como por ejemplo los escenarios virtuales o las reglas que gobernaban al propio juego.

Este planteamiento facilitaba enormemente la reutilización de software y el concepto de motor de juego se hizo más popular a medida que otros desarrolladores comenzaron a utilizar diversos módulos o juegos previamente licenciados para generar los suyos propios. En otras palabras, era posible diseñar un juego del mismo tipo sin apenas modificar

[6] http://en.wikipedia.org/wiki/John_D._Carmack

Figura 1.3: John Carmack, uno de los desarrolladores de juegos más importantes, en el *Game Developer Conference* del año 2010.

el núcleo o *motor* del juego, sino que el esfuerzo se podía dirigir directamente a la parte artística y a las reglas del mismo.

Este enfoque ha ido evolucionando y se ha expandido, desde la generación de ***mods*** por desarrolladores independientes o *amateurs* hasta la creación de una gran variedad de herramientas, bibliotecas e incluso lenguajes que facilitan el desarrollo de videojuegos. A día de hoy, una gran parte de compañías de desarrollo de videojuego utilizan motores o herramientas pertenecientes a terceras partes, debido a que les resulta más rentable económicamente y obtienen, generalmente, resultados espectaculares. Por otra parte, esta evolución también ha permitido que los desarrolladores de un juego se planteen licenciar parte de su propio motor de juego, decisión que también forma parte de su política de trabajo.

Obviamente, la separación entre motor de juego y juego nunca es total y, por una circunstancia u otra, siempre existen dependencias directas que no permiten la reusabilidad completa del motor para crear otro juego. La dependencia más evidente es el genero al que está vinculado el motor de juego. Por ejemplo, un motor de juegos diseñado para construir juegos de acción en primera persona, conocidos tradicionalmente como *shooters* o *shoot'em all*, será difícilmente reutilizable para desarrollar un juego de conducción.

Una forma posible para diferenciar un motor de juego y el software que representa a un juego está asociada al concepto de **arquitectura dirigida por datos** (*data-driven architecture*). Básicamente, cuando un juego contiene parte de su lógica o funcionamiento en el propio código (*hard-coded logic*), entonces no resulta práctico reutilizarla para otro juego, ya que implicaría modificar el código fuente sustancialmente. Sin embargo, si dicha lógica o comportamiento no está definido a nivel de código, sino por ejemplo mediante una serie de reglas definidas a través de un lenguaje de *script*, entonces la reutilización sí es posible y, por lo tanto, beneficiosa, ya que optimiza el tiempo de desarrollo.

Los motores de juegos se suelen adaptar para cubrir las necesidades específicas de un título y para obtener un mejor rendimiento.

Como conclusión final, resulta relevante destacar la evolución relativa a la generalidad de los motores de juego, ya que poco a poco están haciendo posible su utilización para diversos tipos de juegos. Sin embargo, el compromiso entre generalidad y optimalidad aún está presente. En otras palabras, a la hora de desarrollar un juego utilizando un determinado motor es bastante común personalizar dicho motor para adaptarlo a las necesidades concretas del juego a desarrollar.

1.1.5. Géneros de juegos

Los motores de juegos suelen estar, generalmente, ligados a un tipo o género particular de juegos. Por ejemplo, un motor de juegos diseñado con la idea de desarrollar juegos de conducción diferirá en gran parte con respecto a un motor orientado a juegos de acción en tercera persona. No obstante, y tal y como se discutirá en la sección 1.2, existen ciertos módulos, sobre todo relativos al procesamiento de más bajo nivel, que son transversales a cualquier tipo de juego, es decir, que se pueden reutilizar en gran medida de manera independiente al género al que pertenezca el motor. Un ejemplo representativo podría ser el módulo de tratamiento de eventos de usuario, es decir, el módulo responsable de recoger y gestionar la interacción del usuario a través de dispositivos como el teclado, el ratón, el joystick o la pantalla táctil. Otros ejemplo podría ser el módulo de tratamiento del audio o el módulo de renderizado de texto.

A continuación, se realizará una descripción de los distintos géneros de juegos más populares atendiendo a las características que diferencian unos de otros en base al motor que les da soporte. Esta descripción resulta útil para que el desarrollador identifique los aspectos críticos de cada juego y utilice las técnicas de desarrollo adecuadas para obtener un buen resultado.

Mercado de *shooters*

Los FPS (First Person Shooter) gozan actualmente de un buen momento y, como consecuencia de ello, el número de títulos disponibles es muy elevado, ofreciendo una gran variedad al usuario final.

Probablemente, el género de juegos más popular es el de los los denominados FPS, abreviado como *shooters*, representado por juegos como *Quake*, *Half-Life*, *Call of Duty* o *Gears of War*, entre muchos otros. En este género, el usuario normalmente controla a un personaje con una vista en primera persona a lo largo de escenarios que tradicionalmente han sido interiores, como los típicos pasillos, pero que han ido evolucionando a escenarios exteriores de gran complejidad.

Figura 1.4: Captura de pantalla del juego *Tremulous*®, licenciado bajo GPL y desarrollado sobre el motor de *Quake III*.

Los FPS representan juegos con un desarrollo complejo, ya que uno de los retos principales que han de afrontar es la inmersión del usuario en un mundo hiperrealista que ofrezca un alto nivel de detalle, al mismo tiempo que se garantice una alta reacción de respuesta a las acciones del usuario. Este género de juegos se centra en la aplicación de las siguientes tecnologías [6]:

- Renderizado eficiente de grandes escenarios virtuales 3D.
- Mecanismo de respuesta eficiente para controlar y apuntar con el personaje.

- Detalle de animación elevado en relación a las armas y los brazos del personaje virtual.
- Uso de una gran variedad de arsenal.
- Sensación de que el personaje *flota* sobre el escenario, debido al movimiento del mismo y al modelo de colisiones.
- NPC con un nivel de IA considerable y dotados de buenas animaciones.
- Inclusión de opciones multijugador a baja escala, típicamente entre 32 y 64 jugadores.

Normalmente, la tecnología de renderizado de los FPS está especialmente optimizada atendiendo, entre otros factores, al tipo de escenario en el que se desarrolla el juego. Por ejemplo, es muy común utilizar estructuras de datos auxiliares para disponer de más información del entorno y, consecuentemente, optimizar el cálculo de diversas tareas. Un ejemplo muy representativo en los escenarios interiores son los árboles BSP (Binary Space Partitioning) (árboles de partición binaria del espacio) [1], que se utilizan para realizar una división del espacio físico en dos partes, de manera recursiva, para optimizar, por ejemplo, aspectos como el cálculo de la posición de un jugador. Otro ejemplo representativo en el caso de los escenarios exteriores es el denominado *occlusion culling* [1], que se utiliza para optimizar el proceso de renderizado descartando aquellos objetos 3D que no se ven desde el punto de vista de la cámara, reduciendo así la carga computacional de dicho proceso.

En el **ámbito comercial**, la familia de motores *Quake*, creados por *Id Software*, se ha utilizado para desarrollar un gran número de juegos, como la saga *Medal of Honor*, e incluso motores de juegos. Hoy es posible descargar el código fuente de *Quake*, *Quake II* y *Quake III* [7] y estudiar su arquitectura para hacerse una idea bastante aproximada de cómo se construyen los motores de juegos actuales.

Otra familia de motores ampliamente conocida es la de *Unreal*, juego desarrollado en 1998 por *Epic Games*. Actualmente, la tecnología *Unreal Engine* se utiliza en multitud de juegos, algunos de ellos tan famosos como *Gears of War*.

Más recientemente, la compañía *Crytek* ha permitido la descarga del CryENGINE 3 SDK (Software Development Kit)[8] para propósitos no comerciales, sino principalmente académicos y con el objetivo de crear una comunidad de desarrollo. Este kit de desarrollo para aplicaciones gráficas en tiempo real es exactamente el mismo que el utilizado por la

[7] http://www.idsoftware.com/business/techdownloads
[8] http://mycryengine.com/

propia compañía para desarrollar juegos comerciales, como por ejemplo *Crysis 2*.

Otro de los géneros más relevantes son los denominados ***juegos en tercera persona***, donde el usuario tiene el control de un personaje cuyas acciones se pueden apreciar por completo desde el punto de vista de la cámara virtual. Aunque existe un gran parecido entre este género y el de los FPS, los juegos en tercera persona hacen especial hincapié en la animación del personaje, destacando sus movimientos y habilidades, además de prestar mucha atención al detalle gráfico de la totalidad de su cuerpo. Ejemplos representativos de este género son *Resident Evil*, *Metal Gear*, *Gears of War* o *Uncharted*, entre otros.

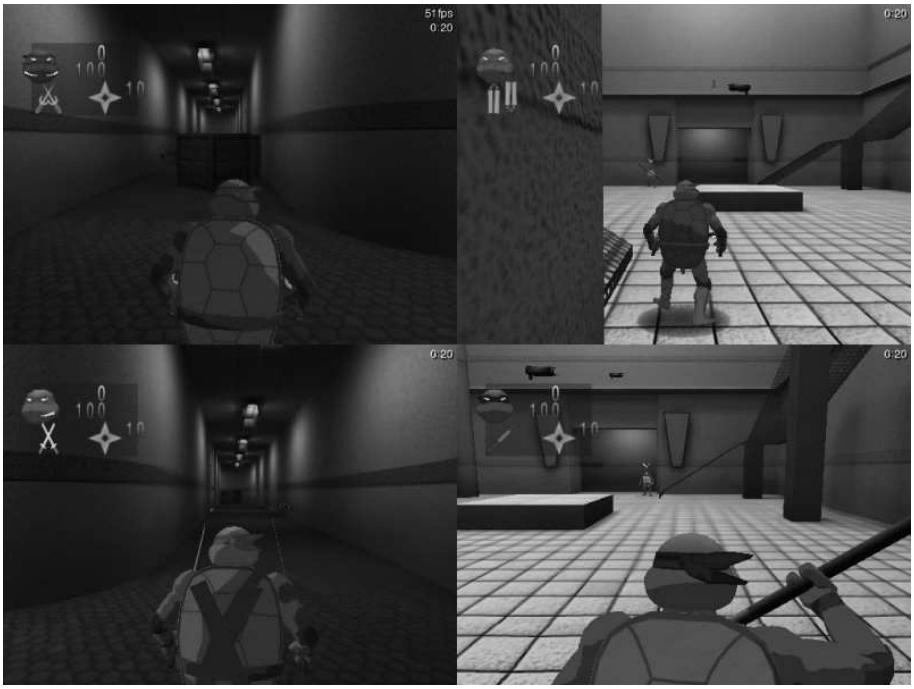


Figura 1.5: Captura de pantalla del juego *Turtlearena*®, licenciado bajo GPL y desarrollado sobre el motor de *Quake III*.

Dentro de este género resulta importante destacar los juegos de plataformas, en los que el personaje principal ha de ir avanzado de un lugar a otro del escenario hasta alcanzar un objetivo. Ejemplos representativos son las sagas de *Super Mario*, *Sonic* o *Donkey Kong*. En el caso particular de los juegos de plataformas, el avatar del personaje tiene normalmente un efecto de *dibujo animado*, es decir, no suele ne-

cesitar un renderizado altamente realista y, por lo tanto, complejo. En cualquier caso, la parte dedicada a la animación del personaje ha de estar especialmente cuidada para incrementar la sensación de realismo a la hora de controlarlo.

Super Mario Bros

El popular juego de Mario, diseñado en 1985 por Shigeru Miyamoto, ha vendido aproximadamente 40 millones de juegos a nivel mundial. Según el libro de los *Record Guinness*, es una de los juegos más vendidos junto a Tetris y a la saga de Pokemon.

En los juegos en tercera persona, los desarrolladores han de prestar especial atención a la aplicación de las siguientes tecnologías [6]:

- Uso de plataformas móviles, equipos de escalado, cuerdas y otros modos de movimiento avanzados.
- Inclusión de puzzles en el desarrollo del juego.
- Uso de cámaras de seguimiento en tercera persona centradas en el personaje y que posibiliten que el propio usuario las maneje a su antojo para facilitar el control del personaje virtual.
- Uso de un complejo sistema de colisiones asociado a la cámara para garantizar que la visión no se vea dificultada por la geometría del entorno o los distintos objetos dinámicos que se mueven por el mismo.

Gráficos 3D

Virtua Fighter, lanzado en 1993 por Sega y desarrollado por Yu Suzuki, se considera como el primer juego de lucha arcade en soportar gráficos tridimensionales.

Otro género importante está representado por los **juegos de lucha**, en los que, normalmente, dos jugadores compiten para ganar un determinado número de combatos minando la vida o *stamina* del jugador contrario. Ejemplos representativos de juegos de lucha son *Virtua Fighter*, *Street Fighter*, *Tekken*, o *Soul Calibur*, entre otros. Actualmente, los juegos de lucha se desarrollan normalmente en escenarios tridimensionales donde los luchadores tienen una gran libertad de movimiento. Sin embargo, últimamente se han desarrollado diversos juegos en los que tanto el escenario como los personajes son en 3D, pero donde el

movimiento de los mismos está limitado a dos dimensiones, enfoque comúnmente conocido como juegos de lucha de *scroll lateral*.

Debido a que en los juegos de lucha la acción se centra generalmente en dos personajes, éstos han de tener una gran calidad gráfica y han de contar con una gran variedad de movimientos y animaciones para dotar al juego del mayor realismo posible. Así mismo, el escenario de lucha suele estar bastante acotado y, por lo tanto, es posible simplificar su tratamiento y, en general, no es necesario utilizar técnicas de optimización como las comentadas en el género de los FPS. Por otra parte, el tratamiento de sonido no resulta tan complejo como lo puede ser en otros géneros de acción.

Los juegos del género de la lucha han de prestar atención a la detección y gestión de colisiones entre los propios luchadores, o entre las armas que utilicen, para dar una sensación de mayor realismo. Además, el módulo responsable del tratamiento de la entrada al usuario ha de ser lo suficientemente sofisticado para gestionar de manera adecuada las distintas combinaciones de botones necesarias para realizar complejos movimientos. Por ejemplo, juegos como *Street Fighter IV* incorporan un sistema de *timing* entre los distintos movimientos de un *combo*. El objetivo perseguido consiste en que dominar completamente a un personaje no sea una tarea sencilla y requiera que el usuario de videojuegos dedique tiempo al entrenaiento del mismo.

Los juegos de lucha, en general, han estado ligados a la evolución de técnicas complejas de síntesis de imagen aplicadas sobre los propios personajes con el objetivo de mejorar al máximo su calidad y, de este modo, incrementar su realismo. Un ejemplo representativo es el uso de *shaders* [11] sobre la armadura o la propia piel de los personajes que permitan implementar técnicas como el *bump mapping* [1], planteada para dotar a estos elementos de un aspecto más rugoso.

Otro género representativo en el mundo de los videojuegos es la **conducción**, en el que el usuario controla a un vehículo que normalmente rivaliza con más adversarios virtuales o reales para llegar a la meta en primera posición. En este género se suele distinguir entre *simuladores*, como por ejemplo *Gran Turismo*, y *arcade*, como por ejemplo *Ridge Racer* o *Wipe Out*.

Simuladores F1

Los simuladores de juegos de conducción no sólo se utilizan para el entretenimiento doméstico sino también para que, por ejemplo, los pilotos de Fórmula-1 conozcan todos los entresijos de los circuitos y puedan conocerlos al detalle antes de embarcarse en los entrenamientos reales.

Figura 1.6: Captura de pantalla del juego de conducción *Tux Racing*, licenciado bajo GPL por Jasmin Patry.

Mientras los simuladores tienen como objetivo principal representar con fidelidad el comportamiento del vehículo y su interacción con el escenario, los juegos arcade se centran más en la jugabilidad para que cualquier tipo de usuario no tenga problemas de conducción.

Los juegos de conducción se caracterizan por la necesidad de dedicar un esfuerzo considerable en alcanzar una calidad gráfica elevada en aquellos elementos cercanos a la cámara, especialmente el propio vehículo. Además, este tipo de juegos, aunque suelen ser muy lineales, mantienen una velocidad de desplazamiento muy elevada, directamente ligada a la del propio vehículo.

Al igual que ocurre en el resto de géneros previamente comentados, existen diversas técnicas que pueden contribuir a mejorar la eficiencia de este tipo de juegos. Por ejemplo, suele ser bastante común utilizar estructuras de datos auxiliares para dividir el escenario en distintos tramos, con el objetivo de optimizar el proceso de renderizado o incluso facilitar el cálculo de rutas óptimas utilizando técnicas de IA [12]. También se suelen usar imágenes para renderizar elementos lejanos, como por ejemplo árboles, vallas publicitarias u otro tipo de elementos.

Del mismo modo, y al igual que ocurre con los juegos en tercera persona, la cámara tiene un papel relevante en el seguimiento del juego. En este contexto, el usuario normalmente tiene la posibilidad de elegir el tipo de cámara más adecuado, como por ejemplo una cámara en primera persona, una en la que se visualicen los controles del propio vehículo o una en tercera persona.

Otro género tradicional son los juegos de **estrategia**, normalmente clasificados en tiempo real o RTS (Real-Time Strategy)) y por turnos (*turn-based strategy*). Ejemplos representativos de este género son *Warcraft*, *Command & Conquer*, *Comandos*, *Age of Empires* o *Starcraft*, entre otros.

Este tipo de juegos se caracterizan por mantener una cámara con una perspectiva isométrica, normalmente fija, de manera que el jugador tiene una visión más o menos completa del escenario, ya sea 2D o 3D. Así mismo, es bastante común encontrar un gran número de unidades

Figura 1.7: Captura de pantalla del juego de estrategia en tiempo real *0 A.D.*, licenciado bajo GPL por *Wildfiregames*.

virtuales desplegadas en el mapa, siendo responsabilidad del jugador su control, desplazamiento y acción.

Teniendo en cuenta las características generales de este género, es posible plantear diversas optimizaciones. Por ejemplo, una de las aproximaciones más comunes en este tipo de juegos consiste en dividir el escenario en una rejilla o *grid*, con el objetivo de facilitar no sólo el emplazamiento de unidades o edificios, sino también la planificación de movimiento de un lugar del mapa a otro. Por otra parte, las unidades se suelen renderizar con una resolución baja, es decir, con un bajo número de polígonos, con el objetivo de posibilitar el despliegue de un gran número de unidades de manera simultánea.

Finalmente, en los últimos años ha aparecido un género de juegos cuya principal característica es la posibilidad de jugar con un gran número de jugadores reales al mismo tiempo, del orden de cientos o incluso miles de jugadores. Los juegos que se encuadran bajo este género se denomina comúnmente MMOG (Massively Multiplayer Online Game). El ejemplo más representativo de este género es el juego *World of Warcarft*. Debido a la necesidad de soportar un gran número de jugadores en línea, los desarrolladores de este tipo de juegos han de realizar un gran esfuerzo en la parte relativa al *networking*, ya que han de proporcionar un servicio de calidad sobre el que construir su modelo de negocio, el cual suele estar basado en suscripciones mensuales o anuales por parte de los usuarios.

Al igual que ocurre en los juegos de estrategia, los MMOG suelen utilizar personajes virtuales en baja resolución para permitir la aparición de un gran número de ellos en pantalla de manera simultánea.

Además de los distintos géneros mencionados en esta sección, existen algunos más como por ejemplo los juegos deportivos, los juegos de rol o RPG (Role-Playing Games) o los juegos de puzzles.

Antes de pasar a la siguiente sección en la que se discutirá la arquitectura general de un motor de juego, resulta interesante destacar la existencia de algunas **herramientas libres** que se pueden utilizar para la construcción de un motor de juegos. Una de las más populares es

OGRE 3D[9]. Básicamente, OGRE es un motor de renderizado 3D bien estructurado y con una curva de aprendizaje adecuada. Aunque OGRE no se puede definir como un motor de juegos completo, sí que proporciona un gran número de módulos que permiten integrar funcionalidades no triviales, como iluminación avanzada o sistemas de animación de caracteres.

1.2. Arquitectura del motor. Visión general

En esta sección se plantea una visión general de la arquitectura de un motor de juegos [6], de manera independiente al género de los mismos, prestando especial importancia a los módulos más relevantes desde el punto de vista del desarrollo de videojuegos.

Como ocurre con la gran mayoría de sistemas software que tienen una complejidad elevada, los motores de juegos se basan en una **arquitectura estructurada en capas**. De este modo, las capas de nivel superior dependen de las capas de nivel inferior, pero no de manera inversa. Este planteamiento permite ir añadiendo capas de manera progresiva y, lo que es más importante, permite modificar determinados aspectos de una capa en concreto sin que el resto de capas inferiores se vean afectadas por dicho cambio.

A continuación, se describen los principales módulos que forman parte de la arquitectura que se expone en la figura 1.8.

1.2.1. Hardware, *drivers* y sistema operativo

La capa relativa al **hardware** está vinculada a la plataforma en la que se ejecutará el motor de juego. Por ejemplo, un tipo de plataforma específica podría ser una consola de juegos de sobremesa. Muchos de los principios de diseño y desarrollo son comunes a cualquier videojuego, de manera independiente a la plataforma de despliegue final. Sin embargo, en la práctica los desarrolladores de videojuegos siempre llevan a cabo optimizaciones en el motor de juegos para mejorar la eficiencia del mismo, considerando aquellas cuestiones que son específicas de una determinada plataforma.

[9] http://www.ogre3d.org/

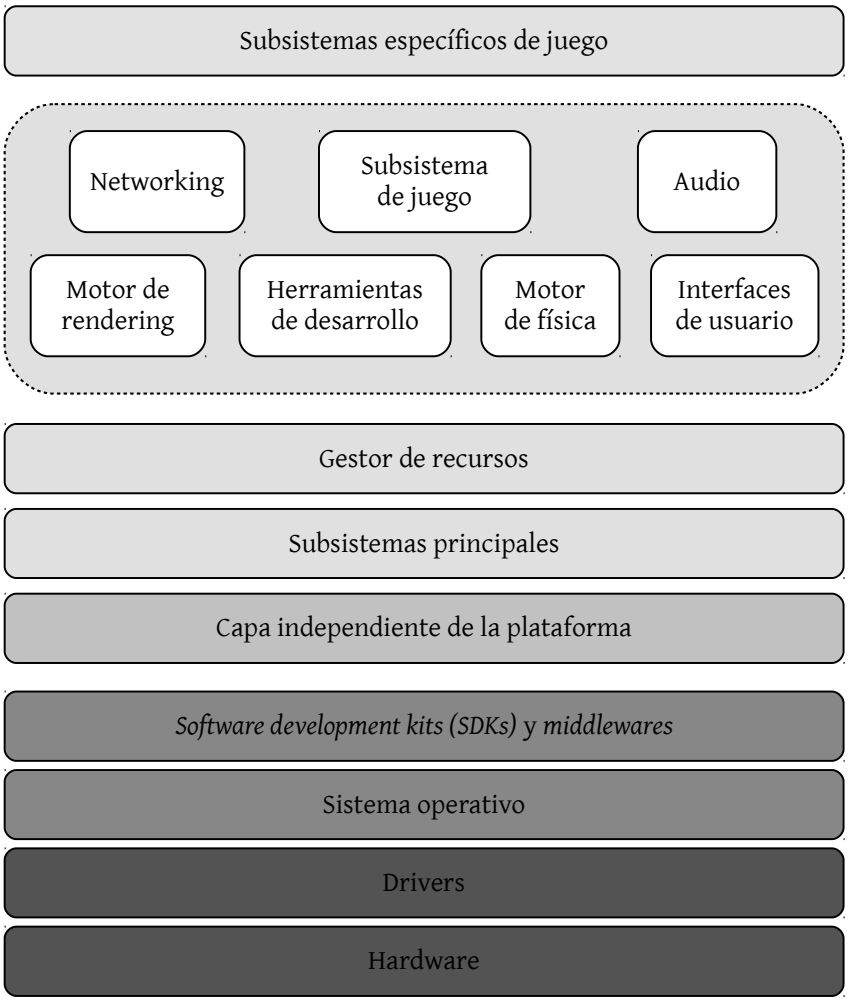


Figura 1.8: Visión conceptual de la arquitectura general de un motor de juegos. Esquema adaptado de la arquitectura propuesta en [6].

La arquitectura *Cell*

En arquitecturas más novedosas, como por ejemplo la arquitectura *Cell* usada en *Playstation 3* y desarrollada por *Sony*, *Toshiba* e IBM, las optimizaciones aplicadas suelen ser más dependientes de la plataforma final.

La capa de ***drivers*** soporta aquellos componentes software de bajo nivel que permiten la correcta gestión de determinados dispositivos, como por ejemplo las tarjetas de aceleración gráfica o las tarjetas de sonido.

La capa del **sistema operativo** representa la capa de comunicación entre los procesos que se ejecutan en el mismo y los recursos hardware asociados a la plataforma en cuestión. Tradicionalmente, en el mundo de los videojuegos los sistemas operativos se compilan con el propio juego para producir un ejecutable. Sin embargo, las consolas de última generación, como por ejemplo *Sony Playstation 3*® o *Microsoft XBox 360*®, incluyen un sistema operativo capaz de controlar ciertos recursos e incluso interrumpir a un juego en ejecución, reduciendo la separación entre consolas de sobremesa y ordenadores personales.

1.2.2. SDKs y *middlewares*

Al igual que ocurre en otros proyectos software, el desarrollo de un motor de juegos se suele apoyar en bibliotecas existentes y SDK para proporcionar una determinada funcionalidad. No obstante, y aunque generalmente este software está bastante optimizado, algunos desarrolladores prefieren personalizarlo para adaptarlo a sus necesidades particulares, especialmente en consolas de sobremesa y portátiles.

API (Application Program Interface)s gráficas

OpenGL y Direct3D son los dos ejemplos más representativos de APIs gráficas que se utilizan en el ámbito comercial. La principal diferencia entre ambas es la estandarización, factor que tiene sus ventajas y desventajas.

Un ejemplo representativo de biblioteca para el manejo de **estructuras de datos** es STL (Standard Template Library) [10]. STL es una biblioteca de plantillas estándar para C++, el cual representa a su vez el lenguaje más extendido actualmente para el desarrollo de videojuegos, debido principalmente a su portabilidad y eficiencia.

[10] http://www.sgi.com/tech/stl/

En el ámbito de los **gráficos 3D**, existe un gran número de bibliotecas de desarrollo que solventan determinados aspectos que son comunes a la mayoría de los juegos, como el renderizado de modelos tridimensionales. Los ejemplos más representativos en este contexto son las APIs gráficas *OpenGL*[11] y *Direct3D*, mantenidas por el grupo *Khronos* y *Microsoft*, respectivamente. Este tipo de bibliotecas tienen como principal objetivo ocultar los diferentes aspectos de las tarjetas gráficas, presentando una interfaz común. Mientras *OpenGL* es multiplataforma, *Direct3D* está totalmente ligado a sistemas *Windows*.

Otro ejemplo representativo de SDKs vinculados al desarrollo de videojuegos son aquellos que dan soporte a la detección y tratamiento de **colisiones** y a la gestión de la **física** de los distintas entidades que forman parte de un videojuego. Por ejemplo, en el ámbito comercial la compañía *Havok*[12] proporciona diversas herramientas, entre las que destaca *Havok Physics*. Dicha herramienta representa la alternativa comercial más utilizada en el ámbito de la detección de colisiones en tiempo real y en las simulaciones físicas. Según sus autores, *Havok Physics* se ha utilizado en el desarrollo de más de 200 títulos comerciales.

Por otra parte, en el campo del *Open Source*, ODE (Open Dynamics Engine) 3D[13] representa una de las alternativas más populares para simular dinámicas de cuerpo rígido [1].

Recientemente, la rama de la **Inteligencia Artificial** en los videojuegos también se ha visto beneficiada con herramientas que posibilitan la integración directa de bloques de bajo nivel para tratar con problemas clásicos como la búsqueda óptima de caminos entre dos puntos o la acción de evitar obstáculos.

1.2.3. Capa independiente de la plataforma

Gran parte de los juegos se desarrollan teniendo en cuenta su potencial lanzamiento en diversas plataformas. Por ejemplo, un título se puede desarrollar para diversas consolas de sobremesa y para PC al mismo tiempo. En este contexto, es bastante común encontrar una capa software que aisle al resto de capas superiores de cualquier aspecto que sea dependiente de la plataforma. Dicha capa se suele denominar *capa independiente de la plataforma*.

Aunque sería bastante lógico suponer que la capa inmediatamente inferior, es decir, la capa de SDK*s y middleware*, ya posibilita la independencia respecto a las plataformas subyacentes debido al uso de módulos estandarizados, como por ejemplo bibliotecas asociadas a C/C++,

[11] `http://http://www.opengl.org/`
[12] `http://www.havok.com`
[13] `http://www.ode.org`

la realidad es que existen diferencias incluso en bibliotecas estandarizadas para distintas plataformas.

> Aunque en teoría las herramientas multiplataforma deberían abstraer de los aspectos subyacentes a las mismas, como por ejemplo el sistema operativo, en la práctica suele ser necesario realizar algunos ajustos en función de la plataforma existente en capas de nivel inferior.

Algunos ejemplos representativos de módulos incluidos en esta capa son las bibliotecas de manejo de hijos o los *wrappers* o envolturas sobre alguno de los módulos de la capa superior, como el módulo de detección de colisiones o el responsable de la parte gráfica.

1.2.4. Subsistemas principales

La capa de subsistemas principales está vinculada a todas aquellas utilidades o bibliotecas de utilidades que dan soporte al motor de juegos. Algunas de ellas son específicas del ámbito de los videojuegos pero otras son comunes a cualquier tipo de proyecto software que tenga una complejidad significativa.

A continuación se enumeran algunos de los subsistemas más relevantes:

- **Biblioteca matemática**, responsable de proporcionar al desarrollador diversas utilidades que faciliten el tratamiento de operaciones relativas a vectores, matrices, cuaterniones u operaciones vinculadas a líneas, rayos, esferas y otras figuras geométricas. Las bibliotecas matemáticas son esenciales en el desarrollo de un motor de juegos, ya que éstos tienen una naturaleza inherentemente matemática.
- **Estructuras de datos y algoritmos**, responsable de proporcionar una implementación más personalizada y optimizada de diversas estructuras de datos, como por ejemplo listas enlazadas o árboles binarios, y algoritmos, como por ejemplo búsqueda u ordenación, que la encontrada en bibliotecas como STL. Este subsistema resulta especialmente importante cuando la memoria de la plataforma o plataformas sobre las que se ejecutará el motor está limitada (como suele ocurrir en consolas de sobremesa).
- **Gestión de memoria**, responsable de garantizar la asignación y liberación de memoria de una manera eficiente.

- **Depuración y *logging***, responsable de proporcionar herramientas para facilitar la depuración y el volcado de *logs* para su posterior análisis.

1.2.5. Gestor de recursos

Esta capa es la responsable de proporcionar una interfaz unificada para acceder a las distintas entidades software que conforman el motor de juegos, como por ejemplo la escena o los propios objetos 3D. En este contexto, existen dos aproximaciones principales respecto a dicho acceso: i) plantear el gestor de recursos mediante un enfoque centralizado y consistente o ii) dejar en manos del programador dicha interacción mediante el uso de archivos en disco.

> Ogre 3D
>
> El motor de *rendering* Ogre 3D está escrito en C++ y permite que el desarrollador se abstraiga de un gran número de aspectos relativos al desarrollo de aplicaciones gráficas. Sin embargo, es necesario estudiar su funcionamiento y cómo utilizarlo de manera adecuada.

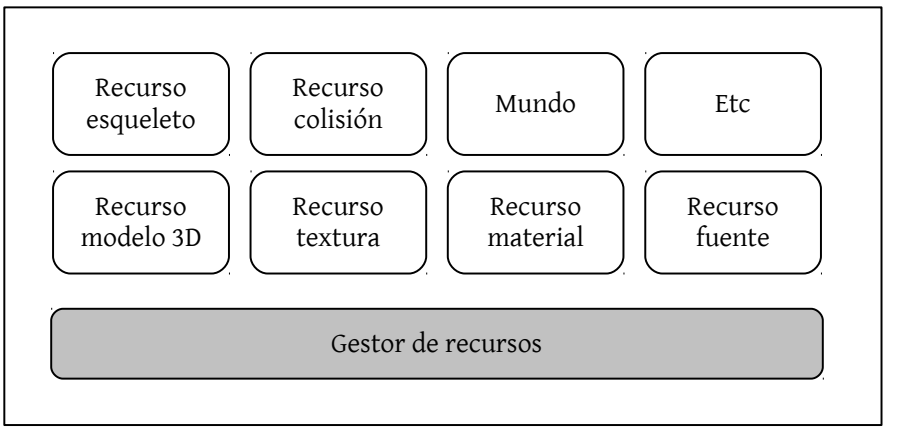


Figura 1.9: Visión conceptual del gestor de recursos y sus entidades asociadas. Esquema adaptado de la arquitectura propuesta en [6].

La figura 1.9 muestra una visión general de un gestor de recursos, representando una interfaz común para la gestión de diversas entidades como por ejemplo el mundo en el que se desarrolla el juego, los objetos 3D, las texturas o los materiales.

En el caso particular de *Ogre 3D* [8], el gestor de recursos está representado por la clase *Ogre::ResourceManager*, tal y como se puede apreciar en la figura 1.10. Dicha clase mantiene diversas especializaciones, las cuales están ligadas a las distintas entidades que a su vez gestionan distintos aspectos en un juego, como por ejemplo las texturas (clase *Ogre::TextureManager*), los modelos 3D (clase *Ogre::MeshManager*) o las fuentes de texto (clase *Ogre::FontManager*). En el caso particular de Ogre 3D, la clase *Ogre::ResourceManager* hereda de dos clases, *ResourceAlloc* y *Ogre::ScriptLoader*, con el objetivo de unificar completamente las diversas gestiones. Por ejemplo, la clase *Ogre::ScriptLoader* posibilita la carga de algunos recursos, como los materiales, mediante *scripts* y, por ello, *Ogre::ResourceManager* hereda de dicha clase.

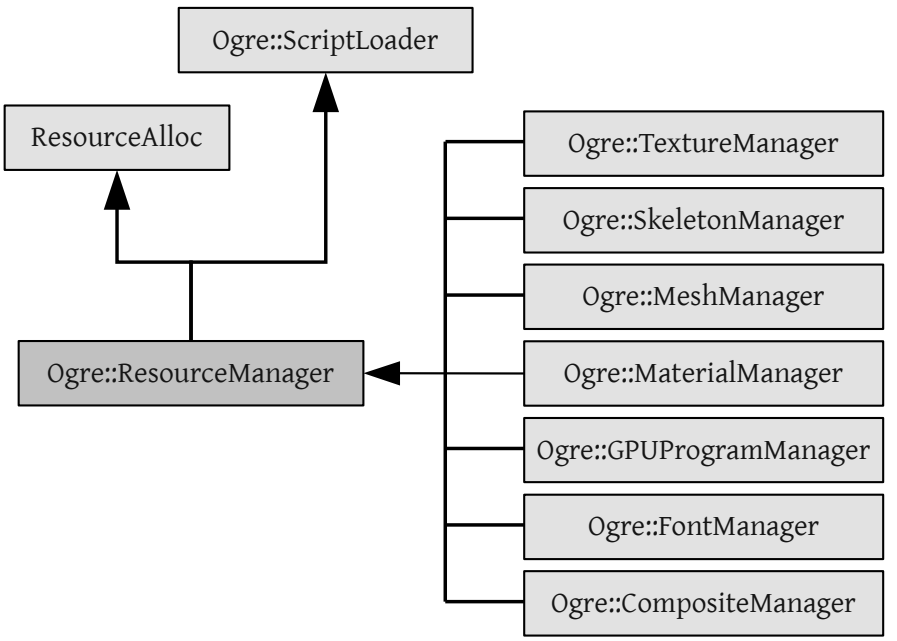


Figura 1.10: Diagrama de clases asociado al gestor de recursos de *Ogre 3D*, representado por la clase *Ogre::ResourceManager*.

1.2.6. Motor de *rendering*

Debido a que el componente gráfico es una parte fundamental de cualquier juego, junto con la necesidad de mejorarlo continuamente, el motor de renderizado es una de las partes más complejas de cualquier motor de juego.

> *Shaders*
>
> Un *shader* se puede definir como un conjunto de instrucciones software que permiten aplicar efectos de renderizado a primitivas geométricas. Al ejecutarse en las unidades de procesamiento gráfico (*Graphic Processing Units - GPUs*), el rendimiento de la aplicación gráfica mejora considerablemente.

Al igual que ocurre con la propia arquitectura de un motor de juegos, el enfoque más utilizado para diseñar el motor de renderizado consiste en utilizar una arquitectura multi-capa, como se puede apreciar en la figura 1.11.

A continuación se describen los principales módulos que forman parte de cada una de las capas de este componente.

La capa de **renderizado de bajo nivel** aglutina las distintas utilidades de renderizado del motor, es decir, la funcionalidad asociada a la representación gráfica de las distintas entidades que participan en un determinado entorno, como por ejemplo cámaras, primitivas de *rendering*, materiales, texturas, etc. El objetivo principal de esta capa reside precisamente en renderizar las distintas primitivas geométricas tan rápido como sea posible, sin tener en cuenta posibles optimizaciones ni considerar, por ejemplo, qué partes de las escenas son visibles desde el punto de vista de la cámara.

> Optimización
>
> Las optimizaciones son esenciales en el desarrollo de aplicaciones gráficas, en general, y de videojuegos, en particular, para mejorar el rendimiento. Los desarrolladores suelen hacer uso de estructuras de datos auxiliares para aprovecharse del mayor conocimiento disponible sobre la propia aplicación.

Esta capa también es responsable de gestionar la interacción con las APIs de programación gráficas, como *OpenGL* o *Direct3D*, simplemente para poder acceder a los distintos dispositivos gráficos que estén disponibles. Típicamente, este módulo se denomina *interfaz de dispositivo gráfico* (*graphics device interface*).

Así mismo, en la capa de renderizado de bajo nivel existen otros componentes encargados de procesar el dibujado de distintas primiti-

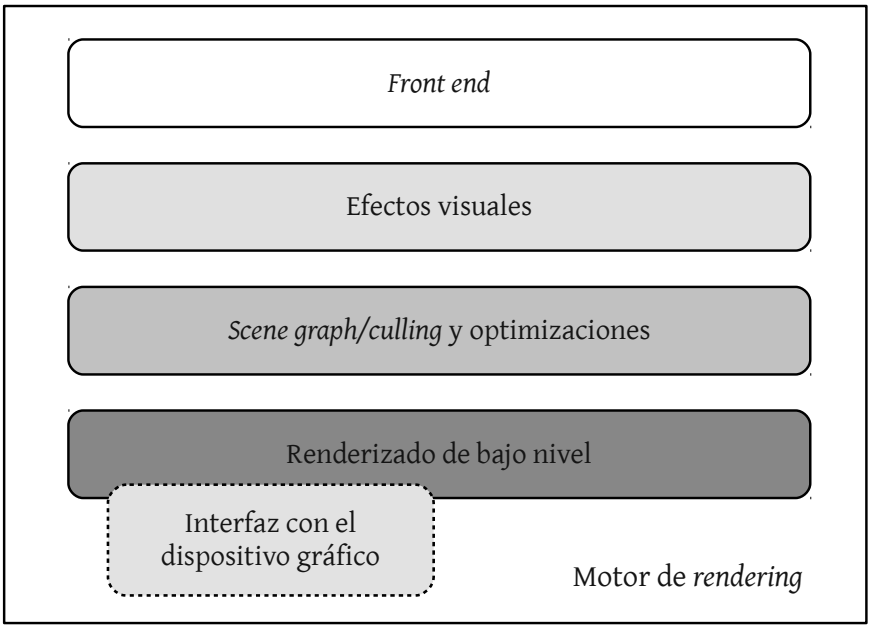


Figura 1.11: Visión conceptual de la arquitectura general de un motor de rendering. Esquema simplificado de la arquitectura discutida en [6].

vas geométricas, así como de la gestión de la cámara y los diferentes modos de proyección. En otras palabras, esta capa proporciona una serie de abstracciones para manejar tanto las primitivas geométricas como las cámaras virtuales y las propiedades vinculadas a las mismas.

Por otra parte, dicha capa también gestiona el estado del hardware gráfico y los *shaders* asociados. Básicamente, cada primitiva recibida por esta capa tiene asociado un material y se ve afectada por diversas fuentes de luz. Así mismo, el material describe la textura o texturas utilizadas por la primitiva y otras cuestiones como por ejemplo qué *pixel* y *vertex shaders* se utilizarán para renderizarla.

La capa superior a la de renderizado de bajo nivel se denomina ***scene graph/culling* y optimizaciones** y, desde un punto de vista general, es la responsable de seleccionar qué parte o partes de la escena se enviarán a la capa de *rendering*. Esta selección, u optimización, permite incrementar el rendimiento del motor de *rendering*, debido a que se limita el número de primitivas geométricas enviadas a la capa de nivel inferior.

Aunque en la capa de *rendering* sólo se dibujan las primitivas que están dentro del campo de visión de la cámara, es decir, dentro del *viewport*, es posible aplicar más optimizaciones que simplifiquen la complejidad de la escena a renderizar, obviando aquellas partes de la misma que no son visibles desde la cámara. Este tipo de optimizaciones son críticas en juegos que tenga una complejidad significativa con el objetivo de obtener tasas de *frames* por segundo aceptables.

Una de las optimizaciones típicas consiste en hacer uso de estructuras de datos de *subdivisión espacial* para hacer más eficiente el renderizado, gracias a que es posible determinar de una manera rápida el conjunto de objetos potencialmente visibles. Dichas estructuras de datos suelen ser árboles, aunque también es posible utilizar otras alternativas. Tradicionalmente, las subdivisiones espaciales se conocen como *scene graph* (grafo de escena), aunque en realidad representan un caso particular de estructura de datos.

Por otra parte, en esta capa también es común integrar métodos de *culling*, como por ejemplo aquellos basados en utilizar información relevante de las oclusiones para determinar qué objetos están siendo solapados por otros, evitando que los primeros se tengan que enviar a la capa de *rendering* y optimizando así este proceso.

Idealmente, esta capa debería ser independiente de la capa de renderizado, permitiendo así aplicar distintas optimizaciones y abstrayéndose de la funcionalidad relativa al dibujado de primitivas. Un ejemplo representativo de esta independencia está representado por OGRE (Object-Oriented Graphics Rendering Engine) y el uso de la filosofía *plug & play*, de manera que el desarrollador puede elegir distintos diseños de grafos de escenas ya implementados y utilizarlos en su desarrollo.

Filosofía *Plug & Play*

Esta filosofía se basa en hacer uso de un componente funcional, hardware o software, sin necesidad de configurar ni de modificar el funcionamiento de otros componentes asociados al primero.

Sobre la capa relativa a las optimizaciones se sitúa la capa de **efectos visuales**, la cual proporciona soporte a distintos efectos que, posteriormente, se puedan integrar en los juegos desarrollados haciendo uso del motor. Ejemplos representativos de módulos que se incluyen en esta capa son aquéllos responsables de gestionar los sistemas de partículos (humo, agua, etc), los mapeados de entorno o las sombras dinámicas.

Finalmente, la capa de ***front-end*** suele estar vinculada a funcionalidad relativa a la superposición de contenido 2D sobre el escenario 3D. Por ejemplo, es bastante común utilizar algún tipo de módulo que per-

mita visualizar el menú de un juego o la interfaz gráfica que permite conocer el estado del personaje principal del videojuego (inventario, armas, herramientas, etc). En esta capa también se incluyen componentes para reproducir vídeos previamente grabados y para integrar secuencias cinemáticas, a veces interactivas, en el propio videojuego. Este último componente se conoce como IGC (In-Game Cinematics) *system*.

1.2.7. Herramientas de depuración

Debido a la naturaleza intrínseca de un videojuego, vinculada a las aplicaciones gráficas en tiempo real, resulta esencial contar con buenas herramientas que permitan depurar y optimizar el propio motor de juegos para obtener el mejor rendimiento posible. En este contexto, existe un gran número de herramientas de este tipo. Algunas de ellas son herramientas de propósito general que se pueden utilizar de manera externa al motor de juegos. Sin embargo, la práctica más habitual consiste en construir herramientas de *profiling*, vinculadas al análisis del rendimiento, o depuración que estén asociadas al propio motor. Algunas de las más relevantes se enumeran a continuación [6]:

> Versiones *beta*
>
> Además del uso extensivo de herramientas de depuración, las desarrolladoras de videojuegos suelen liberar versiones betas de los mismos para que los propios usuarios contribuyan en la detección de *bugs*.

- Mecanismos para determinar el tiempo empleado en ejecutar un fragmento específico de código.
- Utilidades para mostrar de manera gráfica el rendimiento del motor mientras se ejecuta el juego.
- Utilidades para volcar *logs* en ficheros de texto o similares.
- Herramientas para determinar la cantidad de memoria utilizada por el motor en general y cada subsistema en particular. Este tipo de herramientas suelen tener distintas vistas gráficas para visualizar la información obtenida.
- Herramientas de depuración que gestión el nivel de información generada.
- Utilidades para grabar eventos particulares del juego, permitiendo reproducirlos posteriormente para depurar *bugs*.

1.2.8. Motor de física

La detección de colisiones en un videojuego y su posterior tratamiento resultan esenciales para dotar de realismo al mismo. Sin un mecanismo de detección de colisiones, los objetos se *traspasarían* unos a otros y no sería posible interactuar con ellos. Un ejemplo típico de colisión está representado en los juegos de conducción por el choque entre dos o más vehículos. Desde un punto de vista general, el sistema de detección de colisiones es responsable de llevar a cabo las siguientes tareas [1]:

1. La **detección de colisiones**, cuya salida es un valor lógico indicando si hay o no colisión.
2. La **determinación de la colisión**, cuya tarea consiste en calcular el punto de intersección de la colisión.
3. La **respuesta a la colisión**, que tiene como objetivo determinar las acciones que se generarán como consecuencia de la misma.

ED auxiliares

Al igual que ocurre en procesos como la obtención de la posición de un enemigo en el mapa, el uso extensivo de estructuras de datos auxiliares permite obtener soluciones a problemas computacionalmente complejos. La gestión de colisiones es otro proceso que se beneficia de este tipo de técnicas.

Debido a las restricciones impuestas por la naturaleza de tiempo real de un videojuego, los mecanismos de gestión de colisiones se suelen aproximar para simplificar la complejidad de los mismos y no reducir el rendimiento del motor. Por ejemplo, en algunas ocasiones los objetos 3D se aproximan con una serie de líneas, utilizando técnicas de intersección de líneas para determinar la existancia o no de una colisión. También es bastante común hacer uso de árboles BSP para representar el entorno y optimizar la detección de colisiones con respecto a los propios objetos.

Por otra parte, algunos juegos incluyen sistemas realistas o semirealistas de simulación dinámica. En el ámbito de la industria del videojuego, estos sistemas se suelen denominar ***sistema de física*** y están directamente ligados al sistema de gestión de colisiones.

Actualmente, la mayoría de compañías utilizan motores de colisión/-física desarrollados por terceras partes, integrando estos *kits* de desarrollo en el propio motor. Los más conocidos en el ámbito comercial son *Havok*, el cual representa el estándar de facto en la industria debido a su potencia y rendimiento, y *PhysX*, desarrollado por *NVIDIA* e integrado en motores como por ejemplo el del *Unreal Engine 3*.

En el ámbito del *open source*, uno de los más utilizados es ODE. Otra opción muy interesante es *Bullet*[14], el cual se utiliza actualmente en proyectos tan ambiciosos como la suite 3D *Blender*.

1.2.9. Interfaces de usuario

En cualquier tipo de juego es necesario desarrollar un módulo que ofrezca una abstracción respecto a la interacción del usuario, es decir, un módulo que principalmente sea responsable de procesar los **eventos de entrada** del usuario. Típicamente, dichos eventos estarán asociados a la pulsación de una tecla, al movimiento del ratón o al uso de un *joystick*, entre otros.

Desde un punto de vista más general, el módulo de interfaces de usuario también es responsable del **tratamiento de los eventos** de salida, es decir, aquellos eventos que proporcionan una retroalimentación al usuario. Dicha interacción puede estar representada, por ejemplo, por el sistema de vibración del mando de una consola o por la fuerza ejercida por un volante que está siendo utilizado en un juego de conducción. Debido a que este módulo gestiona los eventos de entrada y de salida, se suele denominar comúnmente *componente de entrada/salida del jugador* (*player I/O component*).

El módulo de interfaces de usuario actúa como un puente entre los detalles de bajo nivel del hardware utilizado para interactuar con el juego y el resto de controles de más alto nivel. Este módulo también es responsable de otras tareas importantes, como la asocación de acciones o funciones lógicas al sistema de control del juego, es decir, permite asociar eventos de entrada a acciones lógicas de alto nivel.

En la gestión de eventos se suelen utilizar patrones de diseño como el patrón *delegate* [4], de manera que cuando se detecta un evento, éste se traslada a la entidad adecuada para llevar a cabo su tratamiento.

1.2.10. *Networking* y multijugador

La mayoría de juegos comerciales desarrollados en la actualidad incluyen modos de juegos multijugador, con el objetivo de incrementar la jugabilidad y duración de los títulos lanzados al mercado. De hecho, algunas compañías basan el modelo de negocio de algunos de sus juegos en el **modo *online***, como por ejemplo *World of Warcraft* de *Blizzard Entertainment*, mientras algunos títulos son ampliamente conocidos por su exitoso modo multijugador *online*, como por ejemplo la saga *Call of Duty* de *Activision*.

[14] http://www.bulletphysics.com

El retraso que se produce desde que se envía un paquete de datos por una entidad hasta que otra lo recibe se conoce como *lag*. En el ámbito de los videojuegos, el *lag* se suele medir en milésimas de segundo.

Aunque el modo multijugador de un juego puede resultar muy parecido a su versión *single-player*, en la práctica incluir el soporte de varios jugadores, ya sea *online* o no, tiene un profundo impacto en diseño de ciertos componentes del motor de juego, como por ejemplo el modelo de objetos del juego, el motor de renderizado, el módulo de entrada/salida o el sistema de animación de personajes, entre otros. De hecho, una de las filosofías más utilizadas en el diseño y desarrollo de motores de juegos actuales consiste en tratar el modo de un único jugador como un caso particular del modo multijugador.

Por otra parte, el **módulo de *networking*** es el responsable de *informar* de la evolución del juego a los distintos actores o usuarios involucrados en el mismo mediante el envío de paquetes de información. Típicamente, dicha información se transmite utilizando *sockets*. Con el objetivo de reducir la latencia del modo multijugador, especialmente a través de Internet, sólo se envía/recibe información relevante para el correcto funcionamiento de un juego. Por ejemplo, en el caso de los FPS, dicha información incluye típicamente la posición de los jugadores en cada momento, entre otros elementos.

1.2.11. Subsistema de juego

El subsistema de juego, conocido por su término en inglés *gameplay*, integra todos aquellos módulos relativos al funcionamiento interno del juego, es decir, aglutina tanto las propiedades del mundo virtual como la de los distintos personajes. Por una parte, este subsistema permite la definición de las reglas que gobiernan el mundo virtual en el que se desarrolla el juego, como por ejemplo la necesidad de derrotar a un enemigo antes de enfrentarse a otro de mayor nivel. Por otra parte, este subsistema también permite la definición de la mecánica del personaje, así como sus objetivos durante el juego.

Este subsistema sirve también como capa de *aislamiento* entre las capas de más bajo nivel, como por ejemplo la de *rendering*, y el propio funcionamiento del juego. Es decir, uno de los principales objetivos de diseño que se persiguen consiste en independizar la lógica del juego de la implementación subyacente. Por ello, en esta capa es bastante común encontrar algún tipo de sistema de ***scripting*** o lenguaje de alto

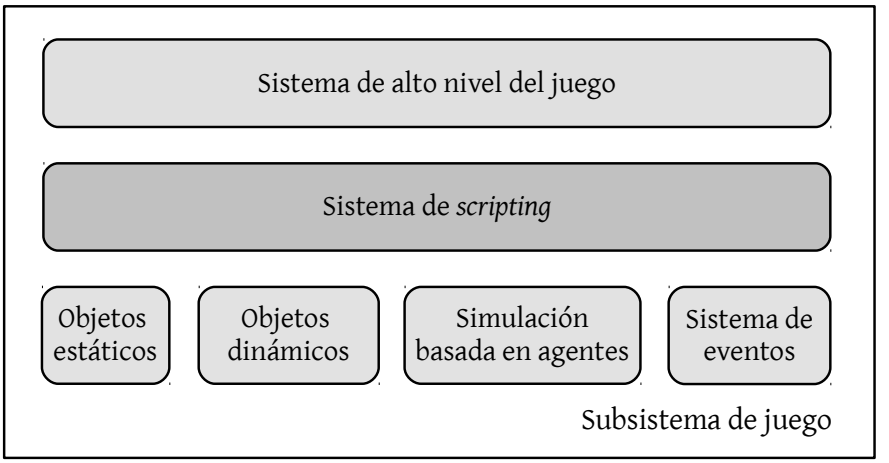


Figura 1.12: Visión conceptual de la arquitectura general del subsistema de juego. Esquema simplificado de la arquitectura discutida en [6].

nivel para definir, por ejemplo, el comportamiento de los personajes que participan en el juego.

> Los diseñadores de los niveles de un juego, e incluso del comportamiento de los personajes y los NPCs, suelen dominar perfectamente los lenguajes de script, ya que son su principal herramienta para llevar a cabo su tarea.

La capa relativa al subsistema de juego maneja conceptos como el *mundo del juego*, el cual se refiere a los distintos elementos que forman parte del mismo, ya sean estáticos o dinámicos. Los tipos de objetos que forman parte de ese mundo se suelen denominar *modelo de objetos del juego* [6]. Este modelo proporciona una simulación en tiempo real de esta colección heterogénea, incluyendo

- Elementos geométricos relativos a fondos estáticos, como por ejemplo edificios o carreteras.
- Cuerpos rígidos dinámicos, como por ejemplo rocas o sillas.
- El propio personaje principal.

- Los personajes no controlados por el usuario (NPCs).
- Cámaras y luces virtuales.
- Armas, proyectiles, vehículos, etc.

El modelo de objetos del juego está intimamente ligado al *modelo de objetos software* y se puede entender como el conjunto de propiedades del lenguaje, políticas y convenciones utilizadas para implementar código utilizando una filosofía de orientación a objetos. Así mismo, este modelo está vinculado a cuestiones como el lenguaje de programación empleado o a la adopción de una política basada en el uso de patrones de diseño, entre otras.

En la capa de subsistema de juego se integra el **sistema de eventos**, cuya principal responsabilidad es la de dar soporte a la comunicación entre objetos, independientemente de su naturaleza y tipo. Un enfoque típico en el mundo de los videojuegos consiste en utilizar una *arquitectura dirigida por eventos*, en la que la principal entidad es el evento. Dicho evento consiste en una estructura de datos que contiene información relevante de manera que la comunicación está precisamente guiada por el contenido del evento, y no por el emisor o el receptor del mismo. Los objetos suelen implementar manejadores de eventos (*event handlers*) para tratarlos y actuar en consecuencia.

Por otra parte, el **sistema de *scripting*** permite modelar fácilmente la lógica del juego, como por ejemplo el comportamiento de los enemigos o NPCs, sin necesidad de volver a compilar para comprobar si dicho comportamiento es correcto o no. En algunos casos, los motores de juego pueden seguir en funcionamiento al mismo tiempo que se carga un nuevo *script*.

Finalmente, en la capa del subsistema de juego es posible encontrar algún módulo que proporcione funcionalidad añadida respecto al tratamiento de la IA, normalmente de los NPCs). Este tipo de módulos, cuya funcionalidad se suele incluir en la propia capa de software específica del juego en lugar de integrarla en el propio motor, son cada vez más populares y permiten asignar comportamientos preestablecidos sin necesidad de programarlos. En este contexto, la *simulación basada en agentes* [18] cobra especial relevancia.

Este tipo de módulos pueden incluir aspectos relativos a problemas clásicos de la IA, como por ejemplo la búsqueda de caminos óptimos entre dos puntos, conocida como *pathfinding*, y típicamente vinculada al uso de algoritmos *A** [12]. Así mismo, también es posible hacer uso de información *privilegiada* para optimizar ciertas tareas, como por ejemplo la localización de entidades de interés para agilizar el cálculo de aspectos como la detección de colisiones.

1.2.12. Audio

Tradicionalmente, el mundo del desarrollo de videojuegos siempre ha prestado más atención al componente gráfico. Sin embargo, el apartado sonoro también tiene una gran importancia para conseguir una inmersión total del usuario en el juego. Por ello, el **motor de audio** ha ido cobrando más y más relevancia.

Así mismo, la aparición de nuevos formatos de audio de alta definición y la popularidad de los sistemas de cine en casa han contribuido a esta evolución en el cada vez más relevante apartado sonoro.

Actualmente, al igual que ocurre con otros componentes de la arquitectura del motor de juego, es bastante común encontrar desarrollos listos para utilizarse e integrarse en el motor de juego, los cuales han sido realizados por compañías externas a la del propio motor. No obstante, el apartado sonoro también requiere modificaciones que son específicas para el juego en cuestión, con el objetivo de obtener un alto de grado de fidelidad y garantizar una buena experiencia desde el punto de visto auditivo.

1.2.13. Subsistemas específicos de juego

Por encima de la capa de subsistema de juego y otros componentes de más bajo nivel se sitúa la capa de subsistemas específicos de juego, en la que se integran aquellos módulos responsables de ofrecer las características propias del juego. En función del tipo de juego a desarrollar, en esta capa se situarán un mayor o menor número de módulos, como por ejemplo los relativos al sistema de cámaras virtuales, mecanismos de IA específicos de los personajes no controlados por el usuario (NPCs), aspectos de renderizados específicos del juego, sistemas de armas, puzzles, etc.

Idealmente, la línea que separa el motor de juego y el propio juego en cuestión estaría entre la capa de subsistema de juego y la capa de subsistemas específicos de juego.

Capítulo 2

Entorno de Trabajo

2.1. NME. Toma de contacto

2.1.1. ¿Qué es NME?

Descripción general

En palabras de sus creadores, *NME es la mejor forma de crear juegos multi-plataforma*. El optimismo excesivo puede jugar, en algunas ocasiones, malas pasadas, pero en el caso particular de NME es cierto que representa uno de los *frameworks open-source* más potentes y fáciles de utilizar existentes en la actualidad.

El hecho de que NME sea un ***framework* multi-plataforma** permite la generación de distintos ejecutables, a partir del mismo código fuente, que podrán *correr* sobre distintos plataformas, desde ordenadores personales con Windows o GNU/Linux hasta navegadores web con soporte de HTML5 o Flash pasando por teléfonos y *tablets* con Android o iOS.

Figura 2.1: La primera versión liberada en Internet de NME fue el 1 de Marzo de 2007.

Compilación cruzada

Un compilador cruzado es capaz de generar código ejecutable para otra plataforma distinta a aquélla en la que se ejecuta.

Este proceso de abstracción que permite, reutilizando prácticamente el mismo código fuente, la generación de ejecutables asociados a diversos lenguajes de programación se consigue gracias a que NME hace uso del innovador **lenguaje de programación Haxe**. De hecho, el compilador de Haxe es capaz de generar el mismo código fuente a código C++, JavaScript y *bytecode* SWF sin realizar sacrificios importantes en la fiabilidad de dicho software. Como resultado, NME proporciona un buen rendimiento para Windows, Mac, GNU/Linux, iOS, Android y BlackBerry, ofreciendo también soporte para Flash Player y HTML5.

NME proporciona un buen balance entre un desarrollo ágil con un flujo de trabajo fácil de usar y la generación de juegos o aplicaciones de calidad.

Características

NME hace gala de una serie de características que lo convierten en una elección muy interesante para el desarrollo de videojuegos. A continuación se enumeran las principales cualidades que lo caracterizan.

Desde el punto de vista **gráfico**, NME

- ofrece soporte para *bitmaps*,

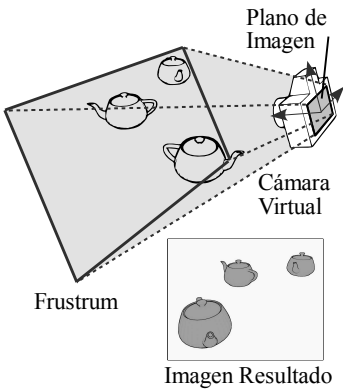


Figura 2.2: Visión general del **proceso de *rendering***, cuyo objetivo principal es la generación de imágenes 2D a partir de la descripción abstracta de una escena 3D.

- permite tratar con gráficos vectoriales,
- soporta manipulación a nivel de píxel,
- soporta *batch triangle rendering*,
- soporta *batch tile rendering*,
- permite un acceso directo a la API OpenGL para tener más control sobre el proceso de *rendering*.
- soporte para *rendering* a pantalla completa.

Desde el punto de vista **multimedia**, NME

- soporta la reproducción de audio,
- posibilita la generación dinámica de audio.

Desde el punto de vista de las redes de computadores, un *socket* representa un canal de comunicación entre procesos a través de una red. La programación de *sockets* se realiza, comúnmente, haciendo uso de los protocolos TCP o UDP.

Desde el punto de vista del **networking**, NME

- soporta peticiones HTTP,
- permite el uso de *sockets* de bajo nivel,

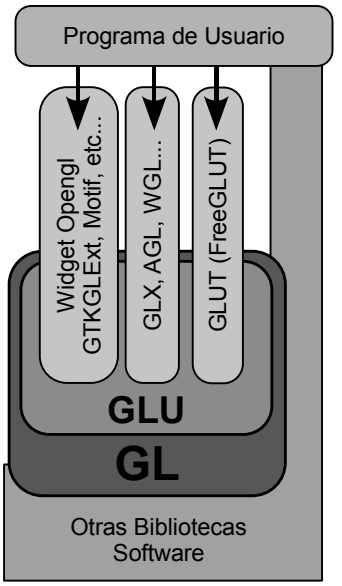


Figura 2.3: OpenGL es una API multiplataforma para el renderizado interactivo de gráficos en 2D/3D que fue inicialmente desarrollada por Silicon Graphics en 1992. El gráfico muestra la relación de la biblioteca asociada a OpenGL (GL) con otras bibliotecas relevantes en este contexto.

- proporciona soporte para cachés de datos.

Desde el punto de vista de la manipulación de **texto**, NME

- soporta el *rendering* de texto a nivel de dispositivo,
- soporta el *rendering* de fuentes empotradas,
- posibilita la personalización del *rendering* de fuentes,
- ofrece *rendering* HTML básico.

Desde el punto de vista de la **gestión de eventos**, NME

- facilita el tratamiento de eventos de teclado y ratón.
- soporta entrada *multitouch*.
- permite el uso de *joysticks*.

Por otra parte, resulta importante destacar que NME permite hacer uso de **extensiones nativas** a determinados lenguaje de programación que tengan dicho soporte. Por ejemplo, NME posibilita la programación multi-hilo para plataformas que proporcionen, de manera nativa, dicho

Figura 2.4: Un *sprite* es una imagen o animación 2D que se integra dentro de una escena mayor.

modelo de programación. De este modo, es posible mejorar la eficiencia, si así es necesario, atendiendo al dispositivo final sobre el cual se desplegará el videojuego desarrollado.

Además, NME se puede utilizar fácilmente con **otras bibliotecas** para integrar funcionalidad no trivial en muchas casos. En el contexto del desarrollo de videojuegos, existe cierta funcionalidad que es recurrente y se utiliza con mucha frecuencia, como por ejemplo las animaciones de *sprites*, la física de cuerpos rígidos o los efectos de partículas. Afortunadamente, existen bibliotecas que proporcionan esta funcionalidad (y mucha más) y que se integran fácilmente con NME.

Por último, NME es un *framework* que posibilita la integración con otros *frameworks* existentes para el desarrollo de videojuegos. En este contexto, algunos ejemplos representativos son *Stencyl*[1], *Flixel*[2], *HaxePunk*[3] o *awe6*[4].

Plataformas soportadas

Como se ha introducido anteriormente, NME soporta una **gran variedad** de plataformas software:

- Windows.
- Mac.
- GNU/Linux.
- iOS.

[1] http://www.stencyl.com
[2] http://flixel.org
[3] http://www.haxepunk.com
[4] http://awe6.org

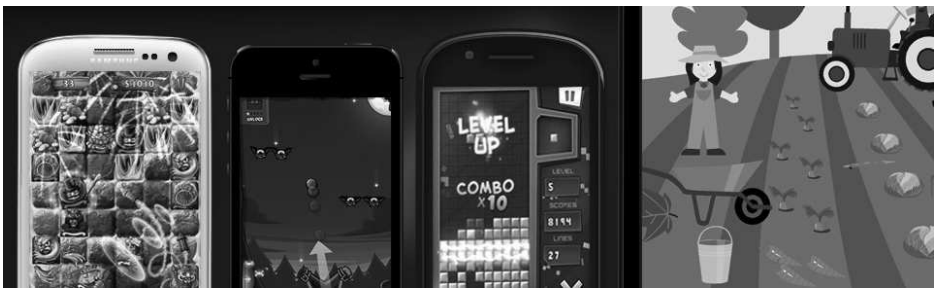


Figura 2.5: *Target Everything* es una de las señas de identidad del *framework* NME y, sin duda, uno de sus puntos fuertes.

- Android.
- BlackBerry.
- webOS.
- Flash.
- HTML5.

La consecuencia inmediata de esta característica es que el **mercado potencial** para desplegar un juego desarrollado con NME se multiplica por un factor más que relevante. Por ejemplo, un desarrollo de un videojuego con NME se podría ejecutar, con una mínima inversión de tiempo extra para resolver cuestiones específicas que puedan surgir, en un teléfono móvil con Android 3.0, un iPhone con iOS 5.0 y, por ejemplo, un navegador web como Chromium con soporte para HTML5.

No obstante, a veces todo no es tan directo y sencillo. Comúnmente, en los desarrollos multi-plataforma siempre hay que considerar **aspectos específicos** de las plataformas finales sobre las que se ejecutará el videojuego (o aplicación) desarrollado. Esto se debe fundamentalmente a dos factores: i) controlar que los aspectos implementación del juego están soportados por las plataformas finales y ii) incrementar la eficiencia en una determinada plataforma, lo cual implicará implementar código específico en la misma.

Cross-platform SW

El SW multi-plataforma se puede dividir en dos tipos: i) SW que ha de compilarse para la plataforma destino y ii) SW que se puede ejecutar directamente sobre la plataforma destino sin necesidad de una preparación especial.

Llegados a este punto, el lector podría ser reticente a pensar que NME permita la generación de código para múltiples plataformas y, que de manera simultánea, sea fácil de usar y realmente práctico para que el proceso de desarrollo multi-plataforma sea ágil. Sin embargo, y como se discutirá más adelante en la sección 2.1.3, realmente lo es. De hecho, y tras realizar con éxito la instalación de NME y su configuración inicial, generar el código en C++ de un videojuego que se ejecutará en sistemas GNU/Linux es tan sencillo como ejecutar la siguiente instrucción:

```
$ nme test "Ejemplo.nmml" linux
```

Con esta instrucción, NME es capaz de generar el código C++ a partir del implementado en Haxe y de crear un ejecutable binario que pueda ejecutarse.

A continuación se describe, desde un punto de vista general, uno de los pilares de NME que permite que todo esto sea posible: el lenguaje de programación multi-plataforma Haxe.

2.1.2. El lenguaje de programación Haxe

Descripción general

Haxe[5] es un lenguaje de programación de código abierto. Sin embargo, mientras otros lenguajes están estrechamente ligados a una plataforma, como por ejemplo Java a su máquina virtual (*Java Virtual Machine*), Haxe es un lenguaje multi-plataforma. Esto significa que Haxe puede utilizarse con el objetivo de generar código para plataformas como Javascript, Flash, NekoVM[6] (*Neko Virtual Machine*), PHP, C++, C# y Java.

Haxe representa el corazón de NME.

La filosofía de Haxe se basa en permitir que el programador decida la mejor plataforma destino para un programa. Este enfoque incrementa de manera evidente la **flexibilidad** a la hora de desarrollar, ya que cada plataforma está típicamente asociada a un lenguaje de programación

[5] http://www.haxe.org

[6] Neko es un lenguaje de programación dinámicamente tipado y de alto nivel. Normalmente, se utiliza como lenguaje de *scripting*.

Figura 2.6: La versión actual de Haxe es la 2.10. Desde que en 2005 apareciese su primera versión, la comunidad Haxe ha logrado desarrollar un lenguaje de programación multi-plataforma de alto nivel que gana adeptos cada día.

concreto. Los tres pilares sobre los que esta filosofía se fundamenta se enumeran a continuación:

1. Lenguaje de programación estandarizado.
2. Biblioteca estándar de funciones que funciona de manera idéntica en todas las plataformas.
3. Bibliotecas específicas de plataforma, posibilitando el acceso a las mismas desde código Haxe.

Antes de describir desde un punto de vista general las características más relevantes de Haxe como lenguaje de programación, resulta interesante distinguir entre dos grupos a la hora de estudiar el soporte multi-plataforma proporcionado por Haxe: i) *client-side* (*lado* del cliente) y ii) *server-side* (*lado* del servidor).

Por una parte, desde el punto de vista del **cliente de una aplicación**, Haxe proporciona la siguiente funcionalidad:

- Es posible compilar a JavaScript generando un único archivo .js. Además, se permite el *debug* interactivo directamente en código Haxe y es posible ajustar el tamaño final del script resultante.
- Es posible compilar a código C++, el cual se puede compilar a su vez a binarios nativos de plataformas móviles, como iOS o Android. De hecho, NME es un caso representativo de este enfoque.
- Es posible general a Flash generando un archivo .swf. Además, se permite el *debug* interactivo y es posible integrar bibliotecas externas SWF.

Por otra parte, desde el punto de vista del **servidor de aplicaciones**, Haxe proporciona la siguiente funcionalidad:

Figura 2.7: El soporte multi-plataforma de Haxe es muy amplio y permite mejorar drásticamente el retorno de una inversión.

- Es posible compilar a NodeJS y a otras tecnologías relacionadas con JavaScript desde el punto de vista del servidor.
- Es posible compilar a código PHP 5.
- Es posible compilar para la máquina virtual de Neko generando un único archivo binario .n.

Características del lenguaje

En este apartado se enumeran las principales características del lenguaje de programación Haxe con el objetivo de que el lector pueda tener una primera impresión de lo que ofrece y lo que no ofrece Haxe.

- Modelo de **POO (Programación Orientada a Objetos)**, el cual facilita el diseño, desarrollo y mantenimiento de proyectos software con una complejidad significativa. Haxe da soporte, mediante distintos elementos, a las características típicas de la OO (Orientación a Objetos) (herencia, encapsulación y polimorfismo).

> Tipado fuerte
>
> Un lenguaje de programación se considera fuertemente tipado cuando no se puede utilizar una variable de un tipo determinado, al menos de manera directa, como si fuera de otro tipo distinto.

- Tipado estricto pero con un **soporte dinámico de tipos**. Esta característica garantiza la interoperabilidad con bibliotecas dependiente de una plataforma en concreto.
- Soporte a **paquetes y módulos** que permitan organizar el código de la manera más adecuada posible.

- Integración de **tipos genéricos** para permitir una programación genérica. Por ejemplo, la clase *Array* de Haxe se puede instanciar para albergar distintos tipos de datos en función de las necesidades de un programa.

> *Type inference*
>
> El compilador de Haxe es capaz de traducir la instrucción $varf = 1{,}0$ a $varf : Float = 1{,}0$ sin necesidad de que el desarrollador especifique el tipo de una variable.

- **Inferencia de tipos** que permite que el compilador de Haxe *deduzca* el tipo de una variable a partir del valor que le fue asignado previamente. Esta característica libera al desarrollador de las restricciones de un lenguaje fuertemente tipado.

- Soporte a **parámetros opcionales y por defecto** en funciones, permitiendo establecer esquemas por defecto a la hora de invocar a funciones.

- Integración del modificador **inline** tanto a nivel de variable como a nivel de función. En el caso de las variables, el uso de *inline* provoca que el compilador sustituya el nombre de la misma por su valor. En el caso de las llamadas a funciones, el uso de *inline* provoca que el compilador sustituya la llamada a una función por el código que conforma su cuerpo. Este modificador es especialmente relevante para incrementar el rendimiento de una aplicación, sobre todo en el caso de variables o funciones que se utilizan con una gran frecuencia.

- Manejo de **excepciones** mediante los típicos bloques *try-catch*.

- Integración de **metadatos** para, por ejemplo, definir un comportamiento específico. Los metadatos se pueden recuperar en tiempo de ejecución.

> *getX()* y *setX()*
>
> Las operaciones *setters* (escritura) y *getters* (lectura) definen el tipo de acceso a una variable miembro de una clase.

- Integración de **propiedades** para, por ejemplo, implementar distintos tipos de características como las variables miembro accedidas mediante *setters* o *getters*.

- Soporte a **compilación condicional** a través de macros de compilación condicional. Este esquema facilita la integración de código específico para una determinada plataforma y es esencial en lenguajes multi-plataforma como Haxe.
- Soporte nativo de **iteradores** para el recorrido directo de contenedores, posibilitando la implementación de iteradores personalizados.

Esta lista no es una lista exhaustiva de las características de Haxe pero sirve para que el lector se haga una idea de la potencia de este lenguaje de alto nivel y de las facilidades que proporciona para el desarrollo de software[7].

Como todo buen lenguaje, Haxe tiene asociado una **biblioteca estándar** que comprende los tipos básicos del lenguaje, los contenedores, el soporte matemático y de expresiones regulares, la serialización o el manejo de flujos de E/S, entre otros muchos elementos[8].

Hola Mundo! con Haxe

Tradicionalmente, el primer programa que se discute cuando alguien se adentra en el estudio de un lenguaje de programación es el típico *Hola Mundo!*, cuya funcionalidad es la de simplemente imprimir un mensaje por la salida estándar (la pantalla).

El siguiente listado de código muestra este programa implementado con Haxe. Como se puede apreciar, la función *main* define el punto de entrada del programa en la línea 2, mientras que la función *trace* de la línea 3 se puede utilizar para mostrar la traza de un programa.

Listado 2.1: *Hola Mundo!* implementado con Haxe.

```
1 class Test {
2   static function main() {
3     trace("Hola Mundo!");
4   }
5 }
```

En este caso, *trace* se ha utilizado para mostrar el mensaje *Hola Mundo!* por la salida, la cual dependerá del *target* o lenguaje elegido a la hora de compilar. Note cómo la función *main* está enmarcada en la clase *Test*, definida en la línea 1.

[7]En http://haxe.org/doc/features se muestra un listado más exhaustivo de las características de Haxe, de su biblioteca estándar y de las herramientas y bibliotecas más relevantes asociadas a dicho lenguaje.

[8]Vea http://haxe.org/api para una descripción más en profundidad de la API de Haxe.

El siguiente paso para ejecutar nuestro primer programa consiste en definir cuál será el lenguaje cuyo código generará el compilador de Haxe. Si se elige, por ejemplo, C++, entonces Haxe generará el código necesario para, posteriormente, compilarlo y ejecutarlo con el compilador de C++ instalado de manera nativa en la máquina. Para ello, es necesario crear un archivo *compile.hxml* cuyo contenido sea el que se muestra en el siguiente listado.

Listado 2.2: Archivo con opciones de compilación de Haxe.

```
1 -cpp cpp
2 -debug
3 -main Test
```

Básicamente, Haxe hace uso del anterior archivo para conocer cuál será el lenguaje de programación de destino (línea 1), si se activará el modo de depuración (línea 2) y cuál será la clase en la que se encuentra el punto de entrada o función principal (línea 3).

Para compilar y generar un ejecutable para C++, tan sólo es necesario ejecutar el siguiente comando desde la línea de órdenes:

```
$ haxe compile.hxml
```

La figura 2.8 muestrar la jerarquía de archivos y directorios generados después de la ejecución del comando anterior.

Si todo ha ido bien, este ejemmplo se puede probar mediante el siguiente comando:

```
$ ./cpp/Test-debug
```

La salida de este programa debería ser

```
Test.hx:3: Hola Mundo!
```

Note cómo la función *trace* tiene como objetivo principal servir como un mecanismo de traza o *log* ya que, además de mostrar el mensaje asociado, imprime el número de línea y la clase en la que se ejecutó.

2.1.3. Instalación y configuración de NME

Esta sección discute la instalación y configuración de NME considerando la generación de código para plataformas GNU/Linux (C++) y Android (Java). Para realizar estos procesos se ha elegido el sistema operativo Ubuntu 12.04 LTS (Long Term Support) con una arquitectura de 32 bits.

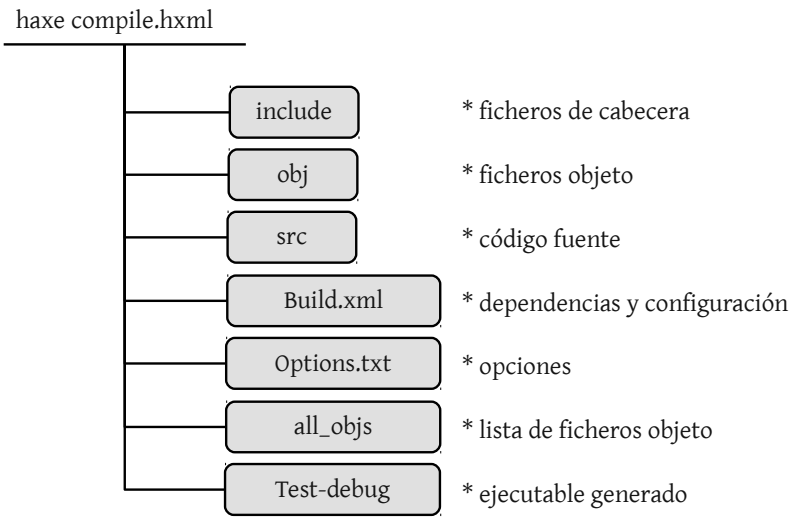


Figura 2.8: Estructura de archivos y directorios generados tras compilar el básico *Hola Mundo!*.

El primer paso consiste en descargar la versión 2.10 de Haxe, disponible en *http://haxe.org/download*. Una vez descargada, es necesario descomprimirla mediante la siguiente instrucción:

```
$ tar xzf hxinst-linux.tar.gz
```

Al descomprimir el anterior archivo, ya es posible ejecutar el script de instalación con permisos de superusuario para instalar Haxe. Para ello, simplemente hay que ejecutar la siguiente instrucción:

```
$ sudo ./hxinst-linux
```

Ahora ya es posible **instalar NME**, junto con algunas bibliotecas, a través de la utilidad *haxelib*, instalada junto con Haxe. Para ello, tan solo hay que ejecutar las siguientes instrucciones:

```
$ sudo haxelib install nme
$ sudo haxelib install hxcpp
$ sudo haxelib install actuate
```

La utilidad haxelib es una herramienta de gestión de biblitoecas realmente potente que permite instalar, desinstalar y buscar el software que necesitemos para implementar una aplicación.

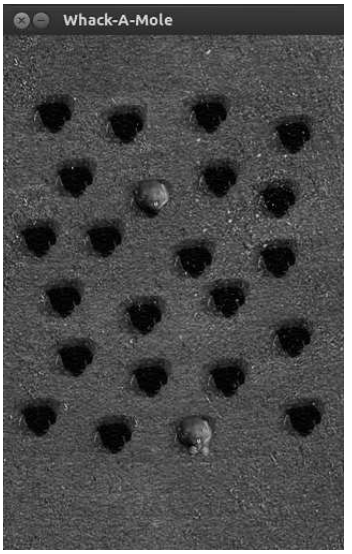


Figura 2.9: Captura de pantalla de un juego tipo *Whack-A-Mole* desarrollado con NME por Joshua Granick. El objetivo del juego es acumular puntos al realizar *click* con el ratón sobre los castores cuando éstos salen de los agujeros.

A continuación, ya es posible realizar la **configuración de NME** mediante el comando *nme setup* y en función del *target* para el cual se generará código fuente. Por ejemplo, es posible llevar a cabo la configuración para **sistemas GNU/Linux** a través del siguiente comando:

```
$ sudo nme setup linux
```

En el caso particular de Ubuntu, dicha instrucción delegará en el gestor de paquetes *apt* la instalación de todas las herramientas y bibliotecas necesarias para generar código fuente en C++ para sistemas GNU/Linux.

Para comprobar si la instalación y configuración de NME para sistemas GNU/Linux ha sido adecuada, el lector puede descargar un sencillo juego, titulado *Whack A Mole!*[9], y generar el ejecutable a partir de la generación previa de código en C++. Para ello, simplemente hay que descomprimir el código fuente, implementado en Haxe, y ejecutar la siguiente instrucción:

```
$ nme test "Whack A Mole.nmml" linux
```

Si todo ha funcionado correctamente, el resultado de la ejecución del juego debería ser similar al de la figura 2.9.

En el presente curso, además de utilizar GNU/Linux como *target* para generar los ejecutables asociados a los distintos prototipos de juegos,

[9] http://www.joshuagranick.com/blog/2011/07/01/game-tutorial-whack-a-mole/

también se generarán paquetes que puedan ejecutarse sobre el sistema operativo **Android**. En este contexto, la generación de código para otra plataforma hace necesaria configuración de NME para dicha plataforma.

Antes de hacer uso de la opción *setup* del comando *nme* es necesario instalar el SDK de Java. En este curso se hará uso de Java 7 y de las facilidades que proporciona Ubuntu para llevar a cabo su instalación:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java7-installer
```

A continuación, ya es posible proceder a la **configuración de NME para Android** haciendo uso del comando *nme*:

```
$ sudo nme setup android
```

El proceso de configuración de NME para Android es algo más tedioso ya que involucra la instalación del SDK, el NDK y Apache Ant. La instalación del SDK lanza la aplicación gráfica *Android SDK Manager* (ver figura 2.10) para que el usuario pueda elegir qué herramientas y versiones del API de Android instalar. En el caso particular del proceso de configuración descrito en esta sección se optó por la versión 2.2 de Android.

El NDK de Android es un conjunto de herramientas que posibilita la implementación de partes de un programa haciendo uso de C y C++. Esta práctica puede resultar útil en términos de reutilización de código y/o mejora de la eficiencia.

La instalación de Apache Ant se puede realizar mediante el siguiente comando y de manera independiente a la configuración de NME para Android:

```
$ sudo apt-get install ant
```

Además de llevar a cabo la instalación de todas estas herramientas, el comando *nme setup android* también permite establecer, como paso final, la ruta a los directorios que contienen todas las herramientas y bibliotecas instaladas, junto con la ruta a las utilidades de Java 7 y Ant[10].

[10]En el caso de Ubuntu 12.04 LTS, los directorios del SDK y NDK de Android se instalaron en /opt. La ruta a las utilidades de Java 7 y Ant se estableció a /usr.

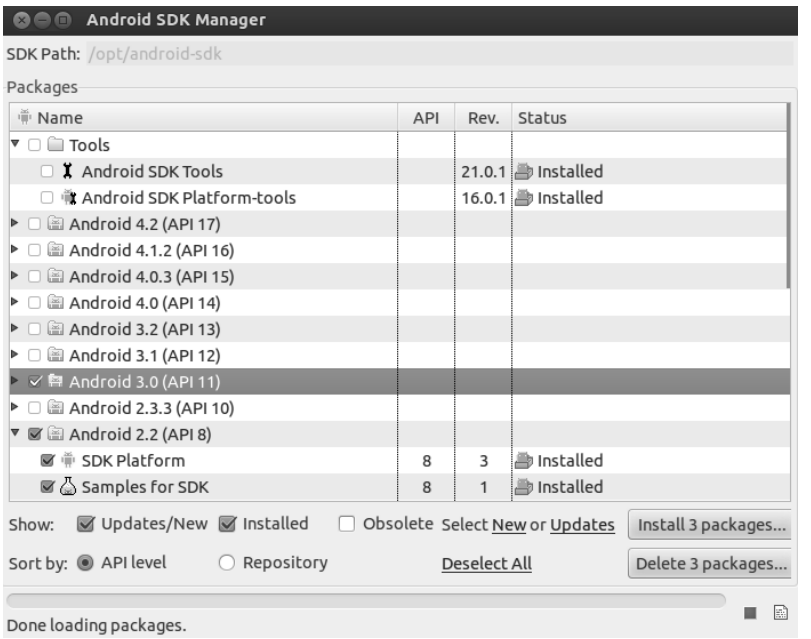


Figura 2.10: Captura de pantalla del *Android SDK Manager*. Esta herramienta permite administrar cómodamente qué herramientas y versiones del API de Android instalar en el sistema.

Finalmente, e igual que se discutió anteriormente para el caso de sistemas GNU/Linux, es posible comprobar si la instalación y configuración de NME para Android se completó de manera satisfactoria generando el paquete que contiene el ejecutable del juego *Whack A Mole!*. Para ello, simplemente es necesario ejecutar el siguiente comando:

```
$ nme test "Whack A Mole.nmml" android
```

Si todo ha funcionado con éxito, entonces se habrá generado un archivo denominado *WhackAMole-debug.apk* en el directorio *bin/android/bin/bin*, el cual se puede transferir directamente a un *smartphone* que haga uso de Android para poder ejecutarlo en el propio dispositivo[11].

Por otra parte, también es posible hacer uso del **emulador de Android** que incorpora el propio SDK previamente instalado y que se encuentra en el directorio *tools*. Sin embargo, antes es necesario crear un

[11] Note que previamente es necesario activar la opción *Orígenes Desconocidos* en el menú de *Aplicaciones* del teléfono

Figura 2.11: Captura de pantalla del emulador de Android incluido en el SDK.

perfil o AVD (Android Virtual Device) que represente la configuración del dispositivo que se pretende emular (ver figura 2.11).

Este perfil o AVD se puede crear de manera sencilla desde el menú *Tools->Manage AVDs...* del propio *Android SDK Manager*.

2.2. *Hello World!* con NME

Esta sección discute un **primer ejemplo** desarrollado con NME que sirva como punto de partida del lector para afrontar el tutorial de desarrollo del capítulo 3.

Básicamente, este primer *Hello World!* con NME integra dos aspectos esenciales en el desarrollo de videojuegos: i) la **animación básica** y ii) la **integración de sonido**. En el primer caso de este sencillo programa que se discutirá a continuación, la animación está asociada al movimiento de una imagen a lo largo del tiempo sobre la pantalla (ver figura 2.12). En el segundo caso, la integración de sonido está representada por la reproducción de un archivo de audio.

En ambos casos se ha hecho uso extensivo de las facilidades que proporciona NME para el desarrollo de videojuegos o, desde un punto de vista general, el desarrollo de aplicaciones gráficas interactivas.

Antes de discutir el código fuente de este ejemplo tan clásico, resulta esencial discutir la estructura de los archivos que contienen la información de un proyecto realizado con NME. El siguiente listado muestra el **documento XML** que contiene la información básica relativa a dicho ejemplo. La extensión de este tipo de archivos es NMML (por ejemplo, HolaMundo.nmml).

Como puede apreciar, este documento es bastante legible y contiene información que podríamos clasificar como i) general y ii) específica sobre un proyecto desarrollado con NME.

Listado 2.3: Archivo del proyecto NME para el ejemplo *Hola Mundo!*.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <project>
3
4    <app title="Hola Mundo! con NME" file="HolaMundoNME"
5         main="HolaMundoNME" package="org.nme.HolaMundoNME"
6         version="1.0" />
7
8    <set name="fullscreen" if="target_webos" />
9    <set name="fullscreen" if="target_ios" />
10   <set name="fullscreen" if="target_android" />
11
12   <window width="640" height="480" fps="30"
13       orientation="portrait" resizable="true" />
14   <window width="0" height="0" if="fullscreen" />
15
16   <set name="BUILD_DIR" value="Export" />
17   <classpath name="Source" />
18
19   <haxelib name="nme" />
20   <haxelib name="actuate" />
21
22   <assets path="Assets" rename="assets"
23       include="*" exclude="nme.svg" />
24   <assets path="Assets/music" rename="assets/music"
25       type="music" include="*" />
26
27   <ndll name="std" />
28   <ndll name="regexp" />
29   <ndll name="zlib" />
30   <ndll name="nme" haxelib="nme" />
31
32   <icon name="Assets/nme.svg" />
33
34 </project>
```

Por ejemplo, la información relativa a las líneas 4–6 está asociada al *título* de la aplicación, es decir, la etiqueta textual que se mostrará en la ventana, el fichero de código fuente (*file*) y el punto de entrada vinculado a la clase principal (*main*), junto con el nombre del paquete y la versión del proyecto.

La etiqueta *window* de las líneas 12–14 también contiene información general. En este caso concreto, en el documento XML se especifica la resolución del juego, 640x480, el *frame rate*, 30 fps, la orientación y si es posible o no redimensionar la ventana del juego.

Por otra parte, también es posible especificar información dependiente del dispositivo final sobre el que se ejecutará el juego a través de sentencias *if*, como se muestra en las líneas 8–10. En este caso particular, el documento XML permite que NME sepa cuándo, es decir, sobre qué entornos (WebOS, iOS y Android), tiene que ejecutar el juego a pantalla completa.

La inclusión de bibliotecas y de *assets* también se gestiona a través del documento XML. Por ejemplo, las líneas *19-20* denotan el uso de las bibliotecas *nme* y *actuate*, cuya funcionalidad será utilizada en el ejemplo del *Hola Mundo!*. La gestión de estas bibliotecas se realiza a través de *haxelib*, una herramienta asociada al lenguaje Haxe que permite compartir información de una manera práctica sobre el código y las bibliotecas software vinculadas a Haxe[12].

Así mismo, la integración de *assets*, como por ejemplo recursos gráficos o sonoros se realiza a través de la etiqueta *assets*, como se puede apreciar en las líneas 22–25. Por ejemplo, en la línea 24 se crea un alias *assets/music* asociado al directorio *Assets/music*, que a su vez contiene los archivos de sonido reales que se utilizarán en el juego.

Finalmente, la etiqueta *ndll* se utiliza para enlazar bibliotecas nativas al proyecto NME, mientras que *icon* se usa para asociarle un icono a dicho proyecto.

Antes de pasar a discutir el código del programa *¡Hola Mundo!*, resulta muy importante destacar la relevancia de la **compilación condicional**.

Haxe, y por consiguiente NME, dispone de la directiva **define** (definición) que pueden modificar la compilación de forma condicional. Si vamos a realizar una serie de operaciones en nuestro juego que sólo pueden ser compiladas para una plataforma, podemos usar *define*. Esto hará que el compilador sólo compile dicho código si ese flag está definido.

El desarrollador puede usar las sentencias **if, else, elseif** y **end** para controlar que líneas serán procesadas por el compilador según los valores o flags ya definidos. Estas sentencias están precedidas por el símbolo **#**.

Por ejemplo, para nuestros casos de uso, si queremos ejecutar un código sólo cuando estemos compilando para flash, por cuestiones de dependencias, usaremos la siguiente estructura:

```
#if flash
//...código flash
#else
//....código no flash
#end
```

Se pueden combinar múltiples valores o flags concatenándolos gracias a operadores lógicos, **||** y **&&**, además de excluir valores con **!**.

[12]En http://haxe.org/doc/haxelib/using_haxelib se discute cómo hacer uso de haxelib.

Por defecto, NME tiene definidos los valores del nombre de la plataforma, el tipo de la plataforma y el lenguaje usado, tal y como se enumera a continuación.

1. **Plataformas:**
   a) windows
   b) mac
   c) linux
   d) ios
   e) android
   f) blackberry
   g) webos
   h) flash
   i) html5
2. **Tipos de plataforma:**
   a) web
   b) mobile
   c) desktop
3. **Lenguaje de programación:**
   a) js
   b) cpp
   c) neko
   d) flash

En ciertos casos hará falta añadir nuestros propios valores, como por ejemplo en el caso de usar bibliotecas excluyentes para la misma plataforma. Por este motivo, existen varias formas de crear nuestros propios valores.

Podemos añadir el valor al archivo .nmml:

```
<haxedef name="valor" />
```

En la línea de órdenes, podemos añadir el argumento *-D* seguido del valor definido.

```
nme test flash -Dvalor
```

Este caso puede crearse también en el archivo .nmml añadiendo el flag por código.

```
<compilerflag name="-Dvalor" if="flash" />
```

A continuación, y después de esta reflexión, el siguiente listado de código muestra la cabecera de la clase *HolaMundoNME*, la cual se instanciará posteriormente, sus variables de clase y el constructor.

En primer lugar, note cómo se define una **clase *HolaMundoNME*** que hereda de la clase *Sprite*, la cual forma parte del conjunto de clases que conforman la implementación de NME. Esta relación de herencia se ha establecido para facilitar el *rendering* de distintos elementos gráficos, como el logo asociado.

Figura 2.12: Resultado visual obtenido al ejecutar el *Hola Mundo!* con NME discutido en esta sección.

La clase *Sprite* de NME representa un bloque básico de construcción de listas de representación, es decir, representa un nodo que puede mostrar recursos gráficos y que puede albergar hijos.

A continuación se definen las **variables de clase** *logo* (línea 2) y *sound* (línea 3), las cuales son de tipo *Bitmap* y *Sound*. La primera de ellas se utilizará para cargar el logo, mientras que la segunda se usará para cargar la información asociada a un *track* de audio.

Por último, el **constructor** (función *new()* de la línea 6) se encarga de llamar al constructor de *Sprite* mediante *super()* (línea 8) y define el manejador de entrada mediante la función *this_onAddedToStage* (línea 10). Este manejador está asociado a un evento de NME de la categoría *Event.ADDED_TO_STAGE*, es decir, el código de la función denominada *this_onAddedToStage* se ejecutará cuando se produzca un evento de tipo *Event.ADDED_TO_STAGE*.

Listado 2.4: *Hola Mundo!* con NME. Constructor.

```
1  class HolaMundoNME extends Sprite {
2
3    private var logo:Bitmap;
4    private var sound:Sound;
5
6    public function new () {
7      /* Delega en el constructor de Sprite (clase padre) */
8      super ();
9      /* Define el manejador de entrada */
10     addEventListener (Event.ADDED_TO_STAGE, this_onAddedToStage);
11   }
```

Note cómo *addEventListener*, en la línea *10*, realiza la asociación entre dicho evento y el manejador, el cual representa el *event listener* asociado al mismo.

El concepto de *event listener* es muy común en el desarrollo de vidoejuegos y representa la relación entre la ocurrencia de un evento, por ejemplo el redimensionado de una ventana, y el código o manejador que se ejecutará cuando el primero tenga lugar. Este planteamiento es muy escalable y posibilita la delegación de eventos.

Precisamente, el siguiente listado de código muestra la implementación del manejador *this_onAddedToStage()* que, a su vez, delega la lógica en la función *construct()*. En el desarrollo de videojuegos, es común independizar la parte de **inicialización** y finalización de un videojuego en funciones explícitamente definidas para ello. Por este motivo se plantea la función *construct()*.

Listado 2.5: *Hola Mundo!* con NME. Función *onAddedToStage()*.

```
1 private function this_onAddedToStage (event:Event):Void {
2   construct ();
3 }
```

La función *construct()*, como muestra el siguiente listado, es la responsable de realizar la **carga efectiva del logo** en la línea 6 y de centrarlo en la pantalla con un nivel de transparencia inicial de 0 (líneas 9–11). Para ello, solamente es necesario acceder a las variables x e y de logo, el cual es de tipo *Bitmap*.

A continuación, en la línea 12 se añade el logo a la lista de representación que maneja internamente la clase *Sprite*, de la cual hereda *HolaMundoNME*.

Finalmente, es necesario asociar el **archivo de audio** que se reproducirá al ejecutar la aplicación y realizar la animación del logo. El primer problema se resuelve fácilmente gracias a la función *getSound()* de la clase *Assets* y a la función *play()* de la clase *Sound*. El segundo problema, es decir, la animación del logo, se delega en la función *tween* de la clase *Actuate*.

La **animación** juega con el nivel de transparencia del logo para que su aparición y desaparición sea suave, en lugar de que aparezca y desaparezca súbitamente. Para ello, los argumentos especificados en la línea 19 permiten que, por ejemplo, el logo pase de un estado de transparencia total a totalmente visible en 2 segundos.

La biblioteca *Actuate* maneja de manera cómoda qué debe ocurrirle a un objeto entre el intervalo de tiempo que transcurre desde el estado actual hasta un estado futuro.

Listado 2.6: *Hola Mundo!* con NME. Función *construct()*.

```
1  private function construct ():Void {
2    stage.align = StageAlign.TOP_LEFT;
3    stage.scaleMode = StageScaleMode.NO_SCALE;
4
5    /* Carga del logo */
6    logo = new Bitmap (Assets.getBitmapData ("assets/tegnix.png"));
7
8    /* Centra el logo en el centro de la pantalla con transparencia a 0
         */
9    logo.x = (stage.stageWidth - logo.width) / 2;
10   logo.y = (stage.stageHeight - logo.height) / 2;
11   logo.alpha = 0;
12   addChild (logo); /* Incorpora el logo a la display list */
13
14   /* Obtiene el sonido y lo reproduce*/
15   sound = Assets.getSound ("assets/music/elvuelodelmoscardon.ogg");
16   sound.play ();
17
18   /* Delega la animacion en la biblioteca Actuate */
19   Actuate.tween (logo, 2, { alpha: 1 } ).onComplete (moveLogo);
20 }
```

Para mover el logo por la pantalla se utiliza la función *moveLogo*. Note cómo está función se ejecuta cada vez que se completa la función *tween*. Esta relación se establece mediante la función *onComplete*.

Listado 2.7: *Hola Mundo!* con NME. Función *moveLogo()*.

```
1  private function moveLogo ():Void {
2      var randomX = Math.random () * (stage.stageWidth - logo.width);
3      var randomY = Math.random () * (stage.stageHeight - logo.height);
4      Actuate.tween (logo, 2, { x: randomX, y: randomY } )
5      .ease (Elastic.easeOut)
6      .onComplete (moveLogo);
7  }
```

Básicamente, el movimiento del logo se realiza de manera aleatoria alterando las coordenadas x e y (líneas 2–3) de la posición de éste. De nuevo, se delega en *Actuate* la parte de animación mediante una ecuación cuadrática que aproxima la *parada* del logo (línea 5).

El **punto de entrada** de este sencillo *Hola Mundo!* es la función *main* que, simplemente, crea una instancia de la clase definida.

Listado 2.8: *Hola Mundo!* con NME. Función *main()*.

```
1  public static function main () {
2    Lib.current.addChild (new HolaMundoNME ());
3  }
```

Capítulo 3

Tutorial de Desarrollo de Videojuegos con NME

Este capítulo discute aspectos esenciales a la hora de abordar el diseño y desarrollo de videojuegos. Para ello, este capítulo está planteado como un **tutorial incremental** de desarrollo, en el cual se estudiarán distintos ejemplos autocontenidos de código fuente sobre aspectos clave que se abordarán en las siguientes secciones:

- **Bucle de juego**, en el que se discutirán distintos esquemas y se estudiará un caso concreto de gestión de estados con NME.
- **Recursos gráficos y representación 2D/3D**, donde se planteará cómo integrar recursos gráficos en el juego y cómo representarlos para que el usuario pueda visualizarlos.
- **Simulación física**, donde se discutirá un esquema sencillo de simulación física para dotar de realismo a los juegos desarrollados.
- **Gestión de sonido**, que servirá para integrar sonido y efectos utilizando la funcionalidad ofrecida por NME para llevar a cabo dicho objetivo.

- **Inteligencia Artificial**, donde se discutirán técnicas que permitan acercar el modelo de razonamiento humano a un juego.
- **Networking**, que estudiará cómo implementar funcionalidad online en un caso de estudio concreto.

Este enfoque facilita la adquisición de conocimientos de manera incremental y permite que el lector se concentre, de manera individual, en los módulos más relevantes que conforman la arquitectura general de un juego.

3.1. El bucle de juego

3.1.1. El bucle de renderizado

Hace años, cuando aún el **desarrollo de videojuegos 2D** era el estándar en la industria, uno de los principales objetivos de diseño de los juegos era minimizar el número de píxeles a dibujar por el *pipeline* de renderizado con el objetivo de maximizar la tasa de *fps* del juego. Evidentemente, si en cada una de las iteraciones del bucle de renderizado el número de píxeles que cambia es mínimo, el juego *correrá* a una mayor velocidad.

Esta técnica es en realidad muy parecida a la que se plantea en el desarrollo de interfaces gráficas de usuario (GUI (Graphical User Interface)), donde gran parte de las mismas es estática y sólo se producen cambios, generalmente, en algunas partes bien definidas. Este planteamiento, similar al utilizado en el desarrollo de videojuegos 2D antiguos, está basado en *redibujar* únicamente aquellas partes de la pantalla cuyo contenido cambia.

En el **desarrollo de videojuegos 3D**, aunque manteniendo la idea de dibujar el mínimo número de primitivas necesarias en cada iteración del bucle de renderizado, la filosofía es radicalmente distinta. En general, al mismo tiempo que la cámara se mueve en el espacio tridimensional, el contenido audiovisual cambia continuamente, por lo que no es viable aplicar técnicas tan simples como la mencionada anteriormente.

La consecuencia directa de este esquema es la necesidad de un bucle de renderizado que muestre las distintas imágenes o *frames* percibidas por la cámara virtual con una velocidad lo suficientemente elevada para transmitir una sensación de realidad.

El siguiente listado de código muestra la **estructura general** de un bucle de renderizado.

Listado 3.1: Esquema general de un bucle de renderizado.

```
1  while (true) {
2    // Actualizar la cámara,
3    // normalmente de acuerdo a un camino prefijado.
4    update_camera ();
5
6    // Actualizar la posición, orientación y
7    // resto de estado de las entidades del juego.
8    update_scene_entities ();
9
10   // Renderizar un frame en el buffer trasero.
11   render_scene ();
12
13   // Intercambiar el contenido del buffer trasero
14   // con el que se utilizará para actualizar el
15   // dispositivo de visualización.
16   swap_buffers ();
17  }
```

3.1.2. Visión general del bucle de juego

Como ya se introdujo en la sección 1.2, en un **motor de juegos** existe una gran variedad de subsistemas o componentes con distintas necesidades. Algunos de los más importantes son el motor de renderizado, el sistema de detección y gestión de colisiones, el subsistema de juego o el subsistema de soporte a la Inteligencia Artificial.

La mayoría de estos componentes han de actualizarse periódicamente mientras el juego se encuentra en ejecución. Por ejemplo, el sistema de animación, de manera sincronizada con respecto al motor de renderizado, ha de actualizarse con una frecuencia de 30 ó 60 Hz con el objetivo de obtener una tasa de *frames* por segundo lo suficientemente elevada para garantizar una sensación de realismo adecuada. Sin embargo, no es necesario mantener este nivel de exigencia para otros componentes, como por ejemplo el de Inteligencia Artificial.

De cualquier modo, es necesario un planteamiento que permita actualizar el estado de cada uno de los subsistemas y que considere las restricciones temporales de los mismos. Típicamente, este planteamiento se suele abordar mediante el **bucle de juego**, cuya principal responsabilidad consiste en actualizar el estado de los distintos componentes del motor tanto desde el punto de vista interno (ej. coordinación entre subsistemas) como desde el punto de vista externo (ej. tratamiento de eventos de teclado o ratón).

Antes de discutir algunas de las arquitecturas más utilizadas para modelar el bucle de juego, resulta interesante estudiar el siguiente listado de código, el cual muestra una manera muy simple de gestionar el bucle de juego a través de una sencilla estructura de control iterati-

va. Evidentemente, la complejidad actual de los videojuegos comerciales requiere un esquema que sea más general y escalable. Sin embargo, es muy importante conservar la simplicidad del mismo para garantizar su mantenibilidad.

> La filosofía KISS (Keep it simple, Stupid!) se adapta perfectamente al planteamiento del bucle de juego, en el que idealmente se implementa un enfoque sencillo, flexible y escalable para gestionar los distintos estados de un juego.

Listado 3.2: Esquema general del bucle de juego.

```
1  // Pseudocódigo de un juego tipo "Pong".
2  int main (int argc, char* argv[]) {
3    init_game();              // Inicialización del juego.
4
5    // Bucle del juego.
6    while (1) {
7      capture_events();       // Capturar eventos externos.
8
9      if (exitKeyPressed())   // Salida.
10       break;
11
12     move_paddles();         // Actualizar palas.
13     move_ball();            // Actualizar bola.
14     collision_detection(); // Tratamiento de colisiones.
15
16     // ¿Anotó algún jugador?
17     if (ballReachedBorder(LEFT_PLAYER)) {
18       score(RIGHT_PLAYER);
19       reset_ball();
20     }
21     if (ballReachedBorder(RIGHT_PLAYER)) {
22       score(LEFT_PLAYER);
23       reset_ball();
24     }
25
26     render();               // Renderizado.
27   }
28 }
```

3.1.3. Arquitecturas típicas del bucle de juego

La arquitectura del bucle de juego se puede implementar de diferentes formas mediante distintos planteamientos. Sin embargo, la mayoría de ellos tienen en común el uso de uno o varios **bucles de control** que gobiernan la actualización e interacción con los distintos componentes

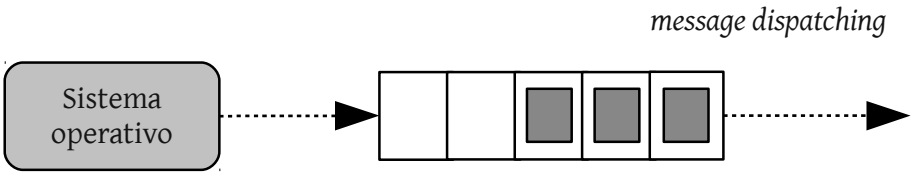


Figura 3.1: Esquema gráfico de una arquitectura basada en *message pumps*.)

del motor de juegos. En esta sección se realiza un breve recorrido por las alternativas más populares, resaltando especialmente un planteamiento basado en la gestión de los distintos estados por los que puede atravesar un juego. Esta última alternativa se discutirá con un caso de estudio detallado cuya implementación hace uso de NME.

Tratamiento de mensajes en Windows

En plataformas Windows™, los juegos han de atender los mensajes recibidos por el propio sistema operativo y dar soporte a los distintos componentes del propio motor de juego. Típicamente, en estas plataformas se implementan los denominados ***message pumps*** [6], como responsables del tratamiento de este tipo de mensajes (ver figura 3.1).

Desde un punto de vista general, el planteamiento de este esquema consiste en atender los mensajes del propio sistema operativo cuando llegan, interactuando con el motor de juegos cuando no existan mensajes del sistema operativo por procesar. En ese caso se ejecuta una iteración del bucle de juego y se repite el mismo proceso.

La principal consecuencia de este enfoque es que los mensajes del sistema operativo tienen prioridad con respecto a aspectos críticos como el bucle de renderizado. Por ejemplo, si la propia ventana en la que se está ejecutando el juego se arrastra o su tamaño cambia, entonces el juego se *congelará* a la espera de finalizar el tratamiento de eventos recibidos por el propio sistema operativo.

Esquemas basados en retrollamadas

El concepto de retrollamada o ***callback*** consiste en asociar una porción de código para atender un determinado evento o situación. Este concepto se puede asociar a una función en particular o a un objeto. En este último caso, dicho objeto se denominará *callback object*, término muy usado en el desarrollo de interfaces gráficas de usuario.

A continuación se muestra un ejemplo de uso de funciones de retrollamada utilizando OpenGL para tratar de manera simple eventos básicos.

Listado 3.3: Ejemplo de uso de retrollamadas con OpenGL.

```
1  #include <GL/glut.h>
2  #include <GL/glu.h>
3  #include <GL/gl.h>
4
5  // Se omite parte del código fuente...
6
7  void update (unsigned char key, int x, int y) {
8    Rearthyear += 0.2;
9    Rearthday += 5.8;
10   glutPostRedisplay();
11 }
12
13 int main (int argc, char** argv) {
14     glutInit(&argc, argv);
15
16     glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
17     glutInitWindowSize(640, 480);
18     glutCreateWindow("Session #04 - Solar System");
19
20     // Definición de las funciones de retrollamada.
21     glutDisplayFunc(display);
22     glutReshapeFunc(resize);
23     // Eg. update se ejecutará cuando el sistema
24     // capture un evento de teclado.
25     // Signatura de glutKeyboardFunc:
26     // void glutKeyboardFunc(void (*func)
27     // (unsigned char key, int x, int y));
28     glutKeyboardFunc(update);
29
30     glutMainLoop();
31
32     return 0;
33 }
```

Desde un punto de vista abstracto, las funciones de retrollamada se suelen utilizar como mecanismo para *rellenar* el código fuente necesario para tratar un determinado tipo de evento. Este esquema está directamente ligado al concepto de ***framework***, entendido como una aplicación construida parcialmente y que el desarrollador ha de *completar* para proporcionar una funcionalidad específica.

Tratamiento de eventos

En el ámbito de los juegos, un **evento** representa un cambio en el estado del propio juego o en el entorno. Un ejemplo muy común está representado por el jugador cuando pulsa un botón del *joystick*, pero

también se pueden identificar eventos a nivel interno, como por ejemplo la reaparición o *respawn* de un NPC en el juego.

Gran parte de los motores de juegos incluyen un subsistema específico para el tratamiento de eventos, permitiendo al resto de componentes del motor, o incluso a entidades específicas, registrarse como partes interesadas en un determinado tipo de evento.

> Con el objetivo de independizar los publicadores y los suscriptores de eventos, se suele utilizar el concepto de *canal de eventos* como mecanismo de abstracción.

El tratamiento de eventos es un aspecto transversal a otras arquitecturas diseñadas para tratar el bucle de juego, por lo que es bastante común integrarlo dentro de otros esquemas más generales, como por ejemplo el que se discute a continuación y que está basado en la gestión de distintos estados dentro del juego.

Esquema basado en estados

Desde un punto de vista general, los juegos se pueden dividir en una serie de etapas o estados que se caracterizan no sólo por su funcionamiento, sino también por la interacción con el usuario o jugador. Típicamente, en la mayor parte de los juegos es posible diferenciar los siguientes estados:

- **Introducción** o presentación, en el que se muestra al usuario aspectos generales del juego, como por ejemplo la temática del mismo o incluso cómo jugar.
- **Menú principal**, en la que el usuario ya puede elegir entre los distintos modos de juegos y que, normalmente, consiste en una serie de entradas textuales identificando las opciones posibles.
- **Juego**, donde ya es posible interactuar con la propia aplicación e ir completando los objetivos marcados.
- **Finalización** o *game over*, donde se puede mostrar información sobre la partida previamente jugada.

Evidentemente, esta clasificación es muy general ya que está planteada desde un punto de vista abstracto. Por ejemplo, si consideramos aspectos más específicos como el uso de dispositivos como *PlayStation*

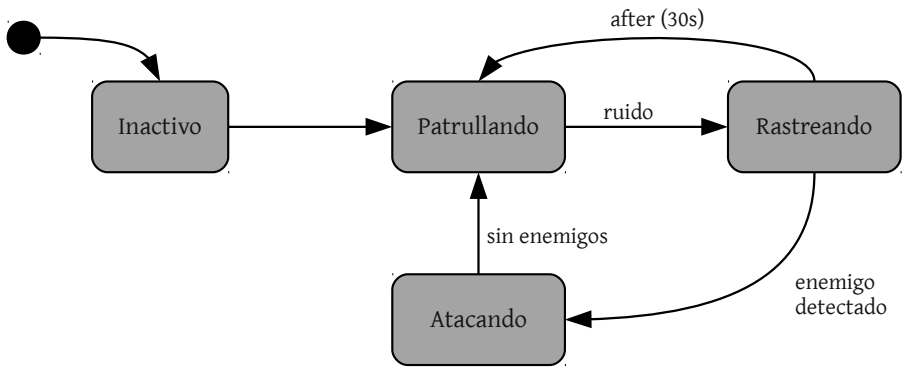


Figura 3.2: Visión general de una máquina de estados finita que representa los estados más comunes en cualquier juego.

Move™, *Wiimote*™o *Kinect*™, sería necesario incluir un estado de calibración antes de utilizar estos dispositivos de manera satisfactoria.

Por otra parte, existe una relación entre cada uno de estos estados que se manifiesta en forma de **transiciones** entre los mismos. Por ejemplo, desde el estado de *introducción* sólo será posible acceder al estado de *menú principal*, pero no será posible acceder al resto de estados. En otras palabras, existirá una transición que va desde *introducción* a *menú principal*. Otro ejemplo podría ser la transición existente entre *finalización* y *menú principal* (ver figura 3.2).

Este planteamiento basado en estados también debería poder manejar varios estados de manera simultánea para, por ejemplo, contemplar situaciones en las que sea necesario ofrecer algún tipo de menú sobre el propio juego en cuestión.

> Las máquinas de estados o autómatas representan modelos matemáticos utilizados para diseñar programas y lógica digital. En el caso del desarrollo de videojuegos se pueden usar para modelar diagramas de estados para, por ejemplo, definir los distintos comportamientos de un personaje.

En la siguiente sección se discute en profundidad un caso de estudio en el que se utiliza NME para implementar un sencillo mecanismo basado en la gestión de estados. En dicha discusión se incluye un gestor responsable de capturar los eventos externos, como por ejemplo las pulsaciones de teclado o la interacción mediante el ratón.

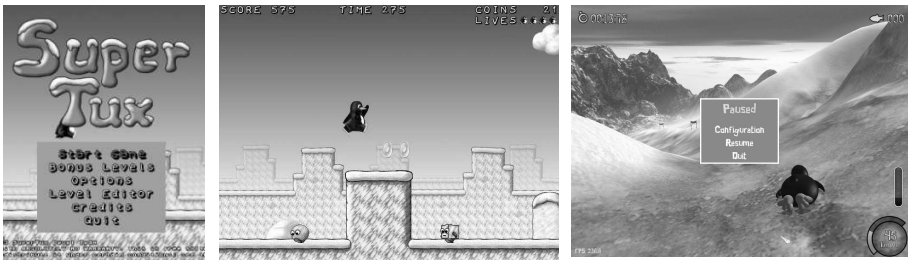


Figura 3.3: Capturas de pantallas de los diferentes estados del juego SuperTux.)

3.1.4. Gestión de estados de juego con NME

Como ya se ha comentado, los juegos normalmente atraviesan una serie de estados durante su funcionamiento habitual. En función del tipo de juego y de sus características, el número de estados variará significativamente. Sin embargo, es posible plantear un esquema común, compartido por todos los estados de un juego, que sirva para definir un **modelo de gestión general**, tanto para interactuar con los estados como para efectuar transiciones entre ellos.

La solución discutida en esta sección[1] se basa en definir una clase abstracta, *GameState*, que contiene una serie de funciones a implementar en los estados específicos de un juego. Estos estados se gestionan por parte de una clase denominada *GameManager*, la cual es la responsable, entre otros aspectos, de gestionar las transiciones entre estados y de delegar los eventos detectados en el estado actual.

El diagrama de clases que muestra el diseño de la gestión de estados discutida en esta sección se muestra gráficamente en la figura 3.4. A continuación se discuten las distintas clases que conforman dicho diseño.

La primera de ellas, la cual representa el concepto abstracto de *estado* está representada por la clase **GameState**. En el siguiente listado de código se muestra el esqueleto de la misma. Como se puede apreciar, el constructor de esta clase es privado. Por lo tanto, no es posible crear instancias de la misma desde fuera de la misma. Sin embargo, es posible extenderla, como se discutirá más adelante, para definir estados de juego concretos. Esta clase comprende una serie de funciones miembro que, idealmente, deberán implementarse en cualquier especialización de la misma. Dichas funciones se pueden dividir en tres grandes bloques:

[1] La solución discutida aquí se basa en el artículo *Managing Game States in C++*.

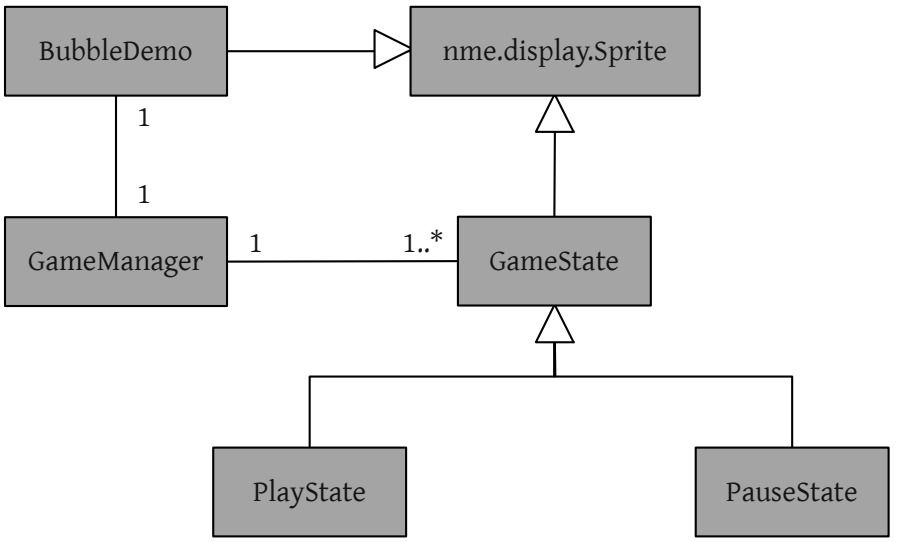


Figura 3.4: Diagrama de clases del esquema de gestión de estados de juego con NME.

1. **Gestión básica del estado** (líneas 10–13), para definir qué hacer cuando se entra, sale, pausa o reanuda el estado.
2. **Gestión básica de tratamiento de eventos** (líneas 17–19), para definir qué hacer cuando se recibe un evento básico de teclado o de ratón.
3. **Gestión básica de eventos antes y después del renderizado** (líneas 24–25), para asociar código antes y después de cada iteración del bucle de renderizado.

Adicionalmente, existe otro bloque de funciones relativas a la **gestión básica de transiciones** (líneas 30–40), con operaciones para cambiar de estado, añadir un estado a la pila de estados y volver a un estado anterior, respectivamente. Las transiciones implican una interacción con la entidad *GameManager*, que se discutirá a continuación.

Note cómo la figura 3.4 muestra la relación de la clase *GameState* con el resto de clases, así como dos posibles especializaciones de la misma. Como se puede observar, esta clase está relacionada con *GameManager*, responsable de la gestión de los distintos estados y de sus transiciones. Antes de pasar a discutir cómo especializar la clase *GameState* para modelar estados concretos de juego, se estudiará la estructura general de la clase *GameManager*.

Listado 3.4: Clase GameState.

```
1  class GameState extends Sprite {
2
3    // Constructor privado (clase abstracta).
4    private function new () {
5      super();
6    }
7
8    // Gestión básica del estado.
9
10   public function enter  () : Void { }
11   public function exit   () : Void { }
12   public function pause  () : Void { }
13   public function resume () : Void { }
14
15   // Gestión básica de eventos de teclado y ratón.
16
17   public function keyPressed    (event:KeyboardEvent) : Void { }
18   public function keyReleased   (event:KeyboardEvent) : Void { }
19   public function mouseClicked  (event:MouseEvent)    : Void { }
20
21   // Gestión básica para el manejo
22   // de eventos antes y después de renderizar un frame.
23
24   public function frameStarted (event:Event) : Void { }
25   public function frameEnded   (event:Event) : Void { }
26
27   // Gestión básica de transiciones entre estados.
28   // Se delega en el GameManager.
29
30   public function changeState (state:GameState) : Void {
31     GameManager.getInstance().changeState(state);
32   }
33
34   public function pushState (state:GameState) : Void {
35     GameManager.getInstance().pushState(state);
36   }
37
38   public function popState () : Void {
39     GameManager.getInstance().popState();
40   }
41
42 }
```

Precisamente, el siguiente listado de código muestra dicha clase, la cual representa la entidad principal de gestión del esquema basado en estados de juego. Como se puede apreciar, esta clase maneja como estado interno dos elementos:

1. Una **instancia del tipo *GameManager*** (línea 4), es decir, del tipo de la propia clase, que será la única instancia disponible de la misma. En otras palabras, la clase *GameManager* implementa el patrón de diseño *Singleton* [4] para garantizar que sólo exista una instancia disponible para dicha clase.

2. Una **pila de estados**, representada por una estructura del tipo *Array* del lenguaje de programación Haxe (línea 6).

Por otra parte, el método constructor simplemente se encarga de instanciar dicha pila de estados, como se muestra en las líneas 8–10. Finalmente, el método *getInstance()* (líneas 13–17) sirve como interfaz pública para que el resto de entidades puedan obtener la única instancia disponible de la clase *GameManager*.

Los gestores de recursos suelen implementar el patrón *Singleton* para garantizar que sólo exista una instancia de los mismos.

Listado 3.5: Clase GameManager. Singleton y constructor.

```
1  class GameManager {
2
3    // Variable estática para implementar Singleton.
4    public static var _instance:GameManager;
5    // Pila de estados.
6    private var _states:Array<GameState>;
7
8    private function new () {
9      _states = new Array<GameState>();
10   }
11
12   // Patrón Singleton.
13   public static function getInstance() : GameManager {
14     if (GameManager._instance == null)
15       GameManager._instance = new GameManager();
16     return GameManager._instance;
17   }
18
19   // Más código aquí...
20
21 }
```

Una de las funciones más relevantes de la clase *GameManager* es *start*, la cual se muestra en el siguiente listado de código. El principal cometido de esta función consiste en inicializar aspectos transversales del juego, como por ejemplo el modo de alineado y escalado (líneas 2–3), registrar los denominados *event listeners* (líneas 7–14) y realizar una transición al estado inicial (línea 17).

La **inicialización** de aspectos transversales es esencial y permite llevar a cabo la carga de recursos o aspectos de configuración que serán utilizados por el motor de juegos (o NME en este caso) posteriormente.

Listado 3.6: Clase GameManager. Función start().

```
1  public function start (state:GameState) : Void {
2    Lib.current.stage.align = StageAlign.TOP_LEFT;
3    Lib.current.stage.scaleMode = StageScaleMode.NO_SCALE;
4
5    // Registro de event listeners.
6    // Permiten asociar evento y código de tratamiento.
7    Lib.current.stage.addEventListener(Event.ENTER_FRAME,
8                        frameStarted);
9    Lib.current.stage.addEventListener(MouseEvent.CLICK,
10                       mouseClicked);
11   Lib.current.stage.addEventListener(KeyboardEvent.KEY_DOWN,
12                       keyPressed);
13   Lib.current.stage.addEventListener(KeyboardEvent.KEY_UP,
14                       keyReleased);
15
16   // Transición al estado inicial.
17   changeState(state);
18 }
```

El **registro de *event listeners*** permite asociar un evento determinado, como un *click* de ratón (por ejemplo *MouseEvent.CLICK* en NME) a una función de retrollamada definida por el programador (por ejemplo *mouseClicked*). Esta función de retrollamada contendrá el código necesario para atender el evento en cuestión. Por ejemplo, el evento de pulsación de la tecla **Esc** se podría asociar a una función de retrollamada *salir* que liberara los recursos asociados al juego en ejecución y finalizara de manera correcta el mismo.

En el anterior fragmento de código se han contemplado cuatro de los eventos más representativos en un juego, los cuales permiten asociar código i) de manera previa al *rendering* o generación de un *frame* (líneas 7–8), ii) cuando se haga *click* con el ratón (líneas 9–10), iii) cuando se pulse una tecla (líneas 11–12) o iv) cuando se libere una tecla previamente pulsada (líneas 13–14).

La **transición al estado inicial** posibilita que el *GameManager* arranque de manera explícita el estado inicial del juego. Éste será típicamente un estado en el que se muestre una pantalla de presentación y un menú. Note cómo en la función *start*, este estado se pasa como parámetro (línea 1) y cómo se utiliza para hacer efectiva la transición mediante la función *changeState* (línea 17), la cual se discutirá más adelante.

Anteriormente se introdujo la variable miembro *_states*, la cual representa la pila de estados gestionada por el *GameManager*. En esencia, esta pila refleja las transiciones entre los distintos estados de un juego. Por ejemplo, la figura 3.5 muestra cómo cambiará dicha estructura de datos si existe un cambio desde el estado de pausa al estado de juego.

Desde un punto de vista más general, para cambiar de un estado A a otro B, suponiendo que A sea la cima de la pila, habrá que realizar las

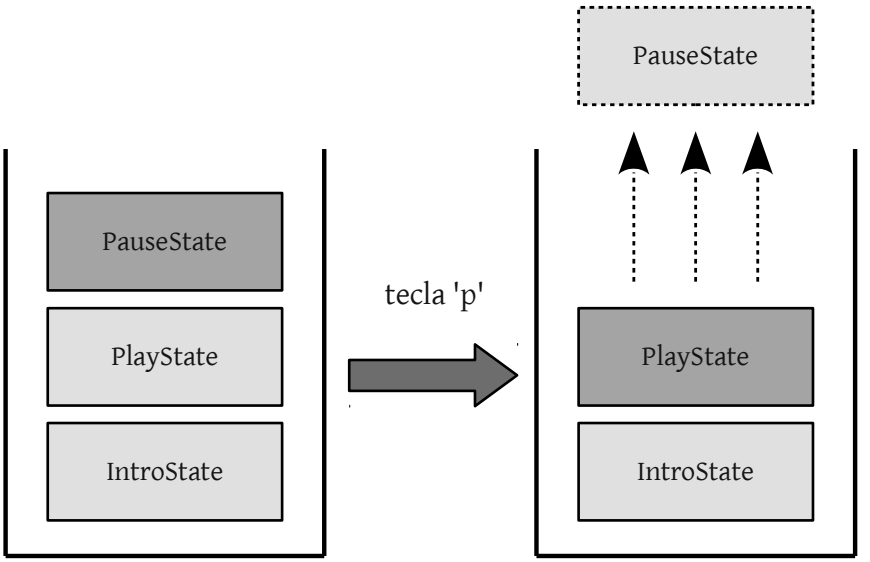


Figura 3.5: Ejemplo de actualización de la pila de estados para reanudar el juego desde el estado de *pause* (evento de tecla 'pulsación de tecla p'.)

operaciones siguientes:

1. Ejecutar *exit()* sobre A.
2. Desapilar A.
3. Apilar B (pasaría a ser el estado activo).
4. Ejecutar *enter()* sobre B.

El siguiente listado de código muestra una posible implementación de la **función *changeState()*** de la clase *GameManager*. Note cómo la estructura de pila de estados permite un acceso directo al estado actual (cima) para llevar a cabo las operaciones de gestión necesarias. Las transiciones se realizan con las típicas operaciones de *push* y *pop*.

Como se puede apreciar en el código fuente, en la línea 4 se accede a la cima de la pila mediante la función *pop()*. Esta función desapila el elemento superior de la pila o cima y lo devuelve. En esa misma instrucción, se invoca la función *exit()* sobre el elemento devuelto, es decir, sobre el estado a desapilar. El objetivo es que antes de pasar a un nuevo estado se garantice la ejecución del código de *exit()* o salida del estado a desapilar.

Listado 3.7: Clase GameManager. Función changeState().

```
1  public function changeState (state:GameState) : Void {
2    // Limpieza del estado actual.
3    if (_states.length > 0) {
4      _states.pop().exit();
5    }
6
7    // Transición al nuevo estado.
8    _states.push(state);
9    state.enter();
10 }
```

Por otra parte, la transición efectiva al nuevo estado se realiza con la función *push()* (línea 8), que permite añadir un nuevo estado a la pila. Justo después se invoca a la función *enter()* del nuevo estado (línea 9), la cual contendrá el código que se ha de ejecutar al entrar o transicionar al mismo.

En este punto concreto, resulta interesante recalcar que se ha planteado un **diseño general** que sirve para cualquier estado concreto, ya que la clase *GameManager* maneja objetos del tipo *GameState* que, en un juego en concreto, serán realmente instancias específicas de estados particulares (por ejemplo *PlayState* o *PauseState*). Este aspecto se discutirá a continuación a través de la definición de estados concretos en un ejemplo sencillo de juego.

Recuerde que el polimorfimo es uno de los pilares de la POO y posibilita mantener una misma interfaz para distintos tipos de datos.

3.1.5. *BubbleDemo*: definición de estados concretos

Este esquema de gestión de estados general, el cual contiene una clase genérica *GameState*, permite la definición de estados específicos vinculados a un juego en particular. En la figura 3.4 se muestra gráficamente cómo la clase *GameState* se extiende para definir dos estados:

- **PlayState**, que define el estado principal del juego y en el que se desarrollará la lógica del mismo.
- **PauseState**, que define un estado de pausa típico en cualquier tipo de juego.

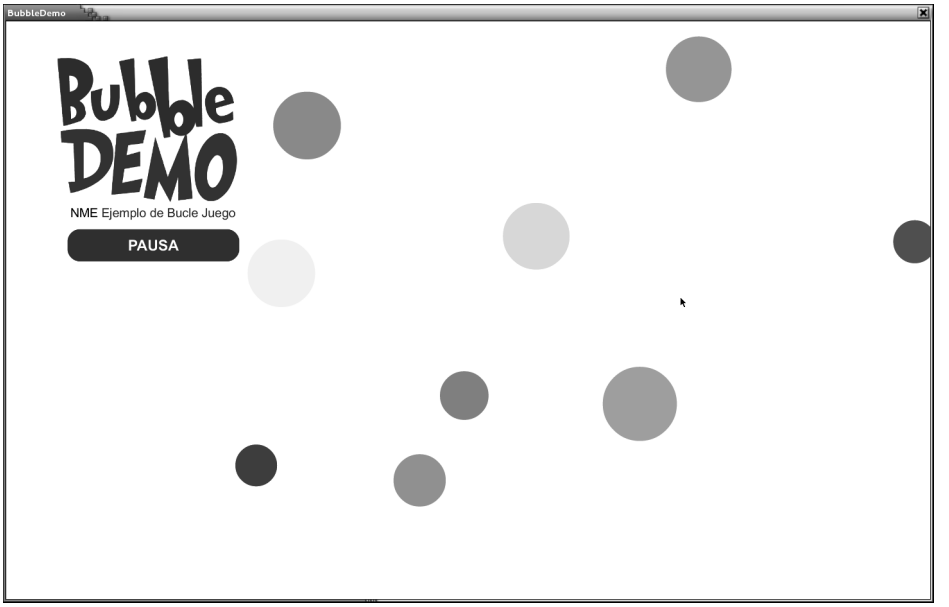


Figura 3.6: Captura de pantalla del estado de pausa de *BubbleDemo*.

La implementación de estos estados permite hacer explícito el comportamiento del juego en cada uno de ellos, al mismo tiempo que posibilita las transiciones entre los mismos. Por ejemplo, una transición típica será pasar del estado *PlayState* al estado *PauseState*.

En ambos casos, a la hora de llevar a cabo dicha implementación, se ha optado por utilizar el patrón *Singleton*, mediante la definición de un constructor privado y una función *getInstance()* que permitirá recuperar la única instancia de clase existente. Note que si ésta no existe previamente, dicha función se encarga de instanciarla.

Para ejemplificar la creación de estados concretos se ha implementado *BubbleDemo*, un **sencillo juego con NME** que consiste en generar, de manera aleatoria, círculos o burbujas que tienen su propia animación. La figura 3.6 muestra una captura de pantalla del aspecto visual de *BubbleDemo*. En realidad, este juego es simplemente un ejemplo para mostrar cómo se ha hecho uso del esquema de gestión de estados previamente discutido.

La **funcionalidad** de *BubbleDemo* es muy sencilla y se puede estructurar en los dos estados mencionados anteriormente:

- En el **estado *PlayState***, el jugador puede hacer *click* sobre el fondo o sobre un círculo. En el primer caso se creará un nuevo círculo

con un color, tamaño y animación aleatorios. En el segundo caso, el círculo sobre el que se hizo *click* oscurecerá su color.

- En el **estado *PauseState***, el jugador puede hacer *click* fácilmente sobre los círculos para cambiar su color, sin la dificultad añadida de la animación de los mismos en el estado de juego.

Para pausar el juego, el jugador sólo ha de pulsar la tecla **Espacio**. Del mismo modo, para reanudar el juego puede hacer uso de la misma tecla.

La clase *PlayState*

El siguiente listado de código muestra una posible definición de la clase *PlayState* y su **constructor**. En esta clase se implementa la lógica necesaria para poder jugar a *BubbleDemo.*

Listado 3.8: Clase PlayState. Constructor.

```
1  class PlayState extends GameState {
2
3    public static var _instance:PlayState;
4    private var _circles:Array<Circle>;
5    private var _logo:Bitmap;
6
7    private function new () {
8      super();
9      _circles = new Array<Circle>();
10     _logo = new Bitmap (Assets.getBitmapData
11             ("assets/background.png"));
12   }
13
14   // Más código aquí...
15
16 }
```

Como se puede apreciar, el estado de la clase está formado por, además de la única referencia a la misma (línea 3), un *array* de objetos de la clase *Circle* (línea 4) y un elemento *_logo* de la clase *Bitmap* definida por NME (línea 5). Este último permite cargar la información del fondo junto con el texto que se muestra en la parte superior izquierda de la figura 3.6. Por otra parte, el *array* de círculos sirve como estructura de datos para almacenar los distintos círculos que se irán creando en el juego.

Como se comentó anteriormente, es necesario definir la lógica de **gestión básica del estado** que controle lo que ocurre cuando se se entra, se sale, se pausa o se reanuda un estado (vea la implementación de la

clase *GameState*). Para ello, la clase *PlayState* sobreescribe las cuatro funciones que se exponen en el siguiente listado de código.

Listado 3.9: Clase PlayState. Funciones de gestión del estado.

```
1  override function enter() : Void {
2    Lib.current.stage.align = StageAlign.TOP_LEFT;
3    Lib.current.stage.scaleMode = StageScaleMode.NO_SCALE;
4    // Al entrar añade el logo para que lo renderice.
5    addChild(_logo);
6  }
7
8  override function exit() : Void {
9    removeChild(_logo);
10 }
11
12 // Pause la animación de los círculos creados.
13 override function pause() : Void {
14   removeChild(_logo);
15   Actuate.pause(this);
16   for (circle in _circles) {
17     circle.pause();
18   }
19 }
20
21 // Reanuda la animación de los círculos creados.
22 override function resume() : Void {
23   addChild(_logo);
24   Actuate.resume(this);
25   for (circle in _circles) {
26     circle.resume();
27   }
28 }
```

La función *enter()* comprende las líneas 1-6 y básicamente se encarga de añadir como hijo del propio estado el *bitmap* que contiene la información del fondo de pantalla a renderizar. Por el contrario, la función *exit()* lo elimina de la lista de hijos de la clase *PlayState*. Este planteamiento mejora la eficiencia de la aplicación de manera que no sea necesario renderizar o dibujar elementos de un estado que no sea el actual. En este punto, es interesante recordar que la clase *GameState*, clase padre de *PlayState*, hereda de *Sprite*. A su vez, la clase *Sprite*[2] de NME hereda de *DisplayObjectContainer*[3], la cual mantiene como estado interno una lista de hijos que representan los elementos que se renderizarán en la pantalla.

Por otra parte, las funciones *pause()* y *resume()* se encargan de pausar y reanudar, respectivamente, las animaciones asociadas a los círculos que estén desplegados en el juego. En este caso, su implementación se

[2]http://www.haxenme.org/api/types/nme/display/Sprite.html
[3]http://www.haxenme.org/api/types/nme/display/DisplayObjectContainer.html

basa en utilizar la biblioteca *Actuate*[4], la cual permite animar entidades gráficas de una manera sencilla y flexible.

Note cómo todas las funciones utilizan el modificador *override* de Haxe para informar de manera explícita al compilador de que están sobreescribiendo funciones ya definidas en la clase padre *GameState*.

La **interacción** con el juego se realiza a nivel de teclado, para cambiar de un estado a otro, y a nivel de ratón, para crear u oscurecer los círculos animados. En el siguiente listado se muestran las funciones *keyPressed()* (líneas 1–6) y *keyReleased()* (línea 8), que definen lo que ha de ocurrir cuando se pulsa y se libera una tecla, respectivamente.

Listado 3.10: Clase PlayState. Eventos de teclado.

```
1 override function keyPressed (event:KeyboardEvent) : Void {
2   // Transición al estado de pausa.
3   if (event.keyCode == Keyboard.SPACE) {
4     pushState(PauseState.getInstance());
5   }
6 }
7
8 override function keyReleased (event:KeyboardEvent) : Void { }
```

En el caso particular de *keyPressed()* se controla si la tecla pulsada fue la tecla **Espacio** para realizar una transición, a través de la función *pushState()* (línea 4), al estado *PauseState*. Recuerde que la transición efectiva se delega en la clase *GameManager* discutida anteriormente.

El siguiente listado de código es relevante, ya que muestra la implementación relativa a la **creación o modificación de los círculos** del juego en base a la interacción con el ratón. Básicamente, la función *mouseClicked()* se encarga de detectar si el jugador hizo *click* sobre alguno de los círculos en movimiento para, en ese caso, cambiar su color (líneas 7–13). Note cómo es necesario recoger la posición exacta del ratón (líneas 3–4) para, posteriormente, comprobar si hubo impacto o no sobre alguno de los círculos.

La funcionalidad relativa a detectar un impacto sobre un círculo o la de modificar su color se ha encapsulado en una clase *Circle*, que se mostrará más adelante. Este enfoque es deseable desde el punto de vista del diseño, ya que permite estructurar de manera adecuada el código fuente y delegar el procesamiento de eventos en la propia clase *Circle*.

[4]http://lib.haxe.org/p/actuate

Listado 3.11: Clase PlayState. Eventos de ratón.

```
1  // Crea un nuevo círculo u oscurece uno ya existente.
2  override function mouseClicked (event:MouseEvent) : Void {
3    var x:Float = event.stageX;
4    var y:Float = event.stageY;
5
6    // ¿Colisión sobre algún círculo?
7    for (circle in PlayState.getInstance().getCircles()) {
8      if (circle.impact(x, y)) {
9        // Si hay colisión, oscurece el color del círculo.
10       circle.fillInBlack();
11       return;
12     }
13   }
14
15   // Crea un círculo cuando se hace click con el ratón
16   // y no hay colisión.
17   createCircle ();
18 }
```

Si, por el contrario, hizo *click* sobre el fondo, entonces se creará un nuevo círculo que pasará a formar parte del juego. Esta funcionalidad se delega en la función *createCircle()*, la cual se muestra a continuación.

Listado 3.12: Clase PlayState. Función createCircle().

```
1 // Crea y un círculo y lo anima.
2 private function createCircle ():Void {
3   var circle:Circle = new Circle ();
4   _circles.push(circle);
5   addChild(circle);
6   circle.animate();
7 }
```

Finalmente, sólo queda por discutir la implementación de la **clase *Circle***, la cual se utiliza para instanciar las distintas burbujas o círculos animados de los que se compone el juego. El siguiente listado de código muestra la implementación del constructor y detalla las variables de clase.

Un círculo mantiene como estado una posición en el espacio, un elemento que permite dibujarlo en la pantalla (ambos heredados de *Sprite*), un tamaño y unos niveles de transparencia y *motion blur* (líneas 3–6 y 16–19). Note cómo todas ellas se inicializan de manera aleatoria. Para ello, se recurre a la función *random()* de la biblioteca matemática *Math*[5] proporcionada por Haxe. Esta asignación aleatoria de valores añade dinamismo a *BubbleDemo*.

[5]http://haxe.org/api/math

Listado 3.13: Clase Circle. Constructor y variables de clase.

```
1  class Circle extends Sprite {
2
3    private var _size  : Float;
4    private var _blur  : Float;
5    private var _alpha : Float;
6
7    public function new () {
8      super ();
9
10     // Generación aleatoria de info básica del círculo.
11     _size = 5 + Math.random () * 35 + 20;
12     _blur = 3 + Math.random () * 12;
13     _alpha = 0.2 + Math.random () * 0.6;
14
15     // Heredadas de Sprite.
16     this.graphics.beginFill (Std.int (Math.random () * 0xFFFFFF));
17     this.graphics.drawCircle (0, 0, _size);
18     this.x = Math.random () * Lib.current.stage.stageWidth;
19     this.y = Math.random () * Lib.current.stage.stageHeight;
20   }
21
22   // Más código aquí...
23
24 }
```

Por otra parte, es necesario llevar a cabo la **animación real** de los círculos para simular su movimiento continuo sobre el fondo. La función *animate* se encarga de esta tarea y se muestra en el siguiente listado. En esencia, esta función delega la animación en la biblioteca *Actuate* en las líneas 7–8.

Listado 3.14: Clase Circle. Función animate().

```
1 public function animate () : Void {
2   var duration:Float = 1.5 + Math.random () * 4.5;
3   var targetX:Float = Math.random () * Lib.current.stage.stageWidth;
4   var targetY:Float = Math.random () * Lib.current.stage.stageHeight;
5
6   // La animación se delega en la biblioteca Actuate.
7   Actuate.tween (this, duration, { x: targetX, y: targetY }, false)
8     .ease (Quad.easeOut).onComplete (animate);
9 }
```

Como aspecto representativo, también se discute la implementación de la función que permite comprobar si se realizó *click* sobre un círculo o no. Dicha implementación se basa en comprobar si las coordenadas del ratón se encuentran dentro o no del cuadrado imaginario que enmarca al círculo sobre el cual se calcula el impacto potencial.

Listado 3.15: Clase Circle. Función impact().

```
1 // ¿Impacto sobre el círculo en el espacio 2D?
2 public function impact (x:Float, y:Float) : Bool {
3   if ((x < this.x + this._size) && (x > this.x - this._size)) {
4     if ((y < this.y + this._size) && (y > this.y - this._size)) {
5       return true;
6     }
7   }
8   return false;
9 }
```

La clase *PauseState*

La clase *PauseState* es más sencilla que la clase *PlayState*, ya que sólo ha de controlar la transición de vuelta al estado de juego, mediante la tecla **Espacio**, y los eventos de ratón para, en su caso, cambiar el color de un círculo que se encuentre en un estado de pausa. Esta última funcionalidad ya se ha discutido en el caso de la clase *PlayState*.

A continuación se muestra el código de la función *keyPressed()*, cuya implementación se basa en volver al estado anterior (*PlayState*) mediante la función *popState()* (línea 4). En este caso, dicha operación tendrá como consecuencia que el elemento de la cima de la pila se elimine y, de este modo, sea posible transitar al estado que anteriormente ocupaba la cima de la pila. Recuerde que desde el estado *PlayState* se pasó al estado *PauseState* mediante la función *pushState()* (ver figura 3.5).

Listado 3.16: Clase PauseState. Función keyPressed().

```
1 override function keyPressed (event:KeyboardEvent) : Void {
2   // Al estado anterior... (PlayState)
3   if (event.keyCode == Keyboard.SPACE) {
4     popState();
5   }
6 }
```

3.2. Recursos Gráficos y Representación

Los recursos gráficos son esenciales en cualquier aplicación multimedia. El caso de los videojuegos resulta especialmente relevantes por los requisitos de interactividad y las altas expectativas de los jugadores actuales. En esta sección estudiaremos los conceptos fundamentales para el despliegue gráfico en videojuegos basados en *Sprites*, prestando especial atención al uso de animaciones, el desarrollo de algunos efectos ampliamente utilizados (como el *Scroll Parallax* o los sistemas de partículas). Para terminar construiremos un pequeño videojuego que ponga en práctica los conceptos estudiados en la sección.

3.2.1. Introducción

Gracias al lenguaje HaXe y a la combinación con la Máquina Virtual de Neko, es posible desarrollar potentes aplicaciones multimedia y videojuegos. La biblioteca SDL (*Simple Direct Media Library* [6]), distribuida como un módulo *ndll*, forma el corazón del motor NME.

El entorno de ejecución de Neko ha sido desarrollado en C para proporcionar un entorno eficiente que pueda ser utilizado en el desarrollo de videojuegos. En la actualidad existen multitud de frameworks y tecnologías para el desarrollo de videojuegos multiplataforma. En concreto, ejecutando pruebas de rendimiento de sprites animados [7] en diferentes plataformas, los resultados obtenidos por NME son impactantes.

La siguiente tabla resume una prueba de rendimiento realizada con un móvil HTC Nexus One, con Android 2.3 y AIR 2.7. Para la compilación de NME se utilizó Haxe (2.07) y NME. Se utilizaron dos versiones para las pruebas; por un lado un sprite de 64x64 píxeles (con 30x50 elementos en pantalla simultáneos), y por otro lado un sprite animado de 32x32 píxeles, en el que debía reproducirse en modo de matriz de 60x100 elementos. Se generó una aplicación nativa para Android con diversos frameworks, y se obtuvieron los resultados de la Tabla 3.1

Flash utiliza métodos de despliegue basados en software. En la mayor parte de los casos, se utiliza la técnica denominada *Blitting* (originalmente de *Bit Blit*), que es una primitiva mediante la que se combinan dos mapas de bits en uno. En NME se utiliza por defecto rendering con soporte Hardware, con soporte de múltiples instancias de *Tiles*. La técnica de *Blitting* es, en general, significativamente más lenta que la manipulación de sprites (aunque es mucho más flexible, y no está limitada al tamaño específico de los Sprites a desplegar).

[6] Ver web oficial: http://www.libsdl.org/
[7] Ver pruebas en el blog de *Krzysztof Rozalski*: http://blog.krozalski.com/?p=1

Tecnología	FPS	Memoria
Corona: Matriz 30x50	3	0.01 Mb
Flash/AIR: Matriz 30x50	22	4.0 Mb
NME: Matriz 30x50	58	0.59 Mb
Corona: Matriz 60x100	-	-
Flash/AIR: Matriz 60x100	8	4.55 Mb
NME: Matriz 60x100	25	2.2 Mb

Cuadro 3.1: Comparativa de Frames por Segundo (FPS) y Memoria Utilizada por el caso de prueba empleando diferentes tecnologías.

En el capítulo 2 utilizamos los primeros recursos gráficos en NME. Los recursos a utilizar y la configuración general de la aplicación se realiza en base a un archivo XML, que permite especificar algunas opciones de compilación y configuración. El siguiente listado muestra un ejemplo de configuración de archivo NMML[8].

Listado 3.17: Ejemplo de archivo NMML.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <project>
3    <meta title="Bee Adventures" package="BeeGame" version="1.0" />
4    <app main="BeeGame" file="com.juego.BeeGame" path="bin" />
5
6    <window background="#000000" width="950" height="550" />
7    <window resizable="false" if="desktop" />
8    <window width="950" height="550" unless="mobile" />
9    <window orientation="landscape" vsync="true" if="mobile" />
10   <window fps="50" />
11   <window fps="60" if="desktop" />
12
13   <source path="src" />
14
15   <haxelib name="nme" />
16   <haxelib name="tilelayer" />
17
18   <assets path="assets" include="*" exclude="nme.svg" />
19   <icon path="assets/nme.svg" />
20
21   <ndll name="std" />
22   <ndll name="regexp" />
23   <ndll name="zlib" />
24   <ndll name="nme" haxelib="nme" />
25 </project>
```

[8]El valor de *package* ha de estar asociado a la jerarquía de directorios utilizada para generar correctamente el APK.

El nodo **<app>** permite definir la configuración general de la aplicación, como el punto de entrada (función *main*), el número de versión de la aplicación, o el título de la misma.

El nodo **<window>** controla las características de despliegue de la aplicación en cada plataforma de publicación concreta. Si una propiedad no está soportada en una plataforma, simplemente será ignorada (como por ejemplo la aceleración *Hardware* cuando se publica en Flash). Algunas de las propiedades más relevantes del nodo *window* son:

- **background**: Color de fondo de la aplicación en hexadecimal (por defecto *0xFFFFFF*).
- **fps**: Define el número de frames por segundo deseados para la aplicación. Si es posible, NME garantizará ese número de FPS estables (siempre que el sistema pueda proporcionar ese número de FPS o superior).
- **hardware**: Indica si se utilizará aceleración hardware (despliegue con soporte en GPU). Por defecto "*true*".
- **width**, **height**: Ancho y alto de la aplicación de píxeles. Si se quiere forzar el despliegue en pantalla completa, se debe indicar 0 (cero) en ambos valores.
- **orientation**: Orientación de la aplicación. Puede tomar valores de "*portrait*" o "*landscape*".
- **resizable**: Indica si la ventana de la aplicación será reescalable o no. Por defecto "*true*".

Los nodos anteriormente definidos soportan atributos adicionales de tipo "**if**" y "**unless**", que pueden emplearse para la configuración y compilación condicional. En el ejemplo del listado anterior, en las líneas 12–13 se indica que el número de FPS deseados para la aplicación en el caso de ser compilada para escritorio es superior al de otras plataformas (móviles). De forma similar, se puede indicar que unos parámetros de configuración se aplicarán a todas las plataformas menos a una en concreto (mediante *unless*, como se muestra en la línea 8).

El nodo **<haxelib>** permite indicar qué bibliotecas adicionales vamos a utilizar. En el caso de los ejemplos de esta sección, como veremos a continuación, utilizaremos la biblioteca adicional "*tilelayer*" (ver línea 16 del listado anterior).

El nodo **<assets>** (ver línea 18) permite indicar los recursos que serán incluidos en el paquete de nuestra aplicación. Mediante los modificadores "if" y "unless" descritos anteriormente es posible indicar diferentes grupos de recursos dependiendo de la plataforma final de publi-

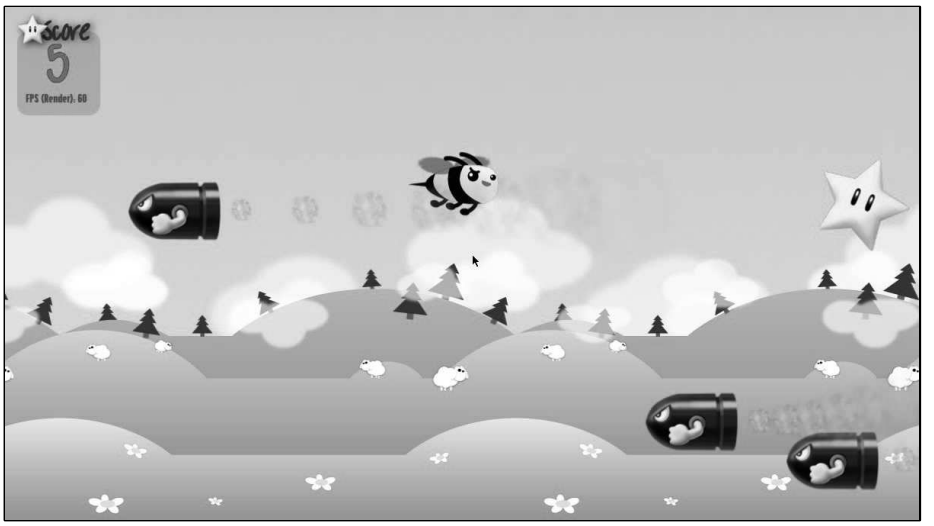


Figura 3.7: La biblioteca TileLayer soporta el uso de hojas de sprites, que facilita la carga eficiente de animaciones y elementos estáticos que intervienen en el juego. En el caso del Mini-Juego "*Bee Adventures*" se emplean 14 sprites, de los cuales 3 son animados.

cación. Estudiaremos su uso concreto en el videojuego que desarrollaremos al final del curso. El tipo de cada fichero se determina automáticamente en base a la extensión del mismo, aunque se pueden especificar de forma manual mediante la propiedad *type* (en este caso, se admiten los valores "*sound*", "*music*", "*font*" e "*image*". Las propiedades *path*, *include* y *exclude* permiten indicar la ruta de los recursos y los archivos que serán incluidos y excluidos en el paquete final.

Por último, el nodo **<ndll>** permite indicar las bibliotecas nativas precompiladas que se utilizarán en el proyecto.

En los ejemplos de esta sección utilizaremos la biblioteca TileLayer, desarrollada por *Philippe Elsass*. Es necesario su instalación como se detalla a continuación.

Para facilitar el desarrollo de aplicaciones, HaXe dispone de multitud de bibliotecas adicionales desarrolladas por empresas y colaboradores de la comunidad. Mediante la utilidad `haxelib` es posible gestionar las bibliotecas instaladas en el sistema.

Un ejemplo de biblioteca para el trabajo con sprites gráficos en *TileLayer*. Esta biblioteca proporciona un interfaz optimizado para el trabajo con Sprites, con algunas opciones muy interesantes como el soporte de animaciones, efecto espejo, y *batching* (ver Sección 3.2.2) de modo que pueden emplearse múltiples sprites en una única hoja (mejorando en rendimiento enormemente).

Para instalar la biblioteca, basta con ejecutar desde el terminal:

```
sudo haxelib install tilelayer
```

E incorporar `haxelib name="tilelayer"` en el archivo NMML (como hemos visto en el listado anteriormente). En las siguientes secciones estudiaremos cómo utilizar las clases proporcionadas por esta biblioteca.

Haxelib proporciona otros comandos de utilidad para buscar otras bibliotecas, actualizar las existentes o listar las actualmente instaladas. Llamando a `haxelib` sin argumentos obtendremos un listado de las opciones disponibles. Podemos buscar otras bibliotecas interesantes para el desarrollo de videojuegos tecleando en el terminal:

```
haxelib search game
```

Obtendremos un listado de bibliotecas relacionadas con el desarrollo de videojuegos. Para obtener más información sobre una en concreto, teclearemos `haxelib info "nombre"`. Por ejemplo, podemos obtener más información sobre *gm2d* [9] mediante:

```
#haxelib info gm2d

Name: gm2d
Tags: cross, flash, game, nme, svg
Desc: GM2D helper classes for rapid game making in 2D.
Website: http://code.google.com/p/gm2d/
License: BSD
Owner: gamehaxe
Version: 1.1
Releases:
   2011-07-27 17:23:22 1.0 : Initial release, including tilemaps, bitmap
        fonts, svg and swf renderers.
   2012-03-04 17:27:12 1.1 : Add tile add blend mode.  Improved SVG
       parser - now looks at root node. Add FileOpen code.  Start on
       docking framework.  Flash decoding frame number fixes.  Fix {}
       compile error.  Add some waxe integration.  Added a binary gfx
       format.
```

[9] Aunque no la utilizaremos en el curso, *gm2d* es una potente biblioteca compatible con nme que facilita el desarrollo de videojuegos en 2D. Queda propuesto como *ejercicio* para el lector que ejecute los ejemplos que vienen integrados con el paquete oficial.

3.2.2. Sprites

El *Sprite*[10] es una de las primitivas gráficas más simples. Un sprite es una imagen que se mueve por la pantalla. El puntero del ratón puede considerarse un sprite. Este tipo de mapa de bits a menudo son pequeños y parcialmente transparentes, por lo que no es necesario que definan regiones totalmente rectangulares. En la década de los 80 comenzaron a utilizarse ampliamente debido a que algunos ordenadores, como el MSX, el Commodore 64 y Amiga, incorporaban hardware específico para su tratamiento (sin necesidad de realizar cálculos adicionales en la CPU).

Los sprites se siguen utilizando ampliamente en la actualidad, incluso en gráficos 3D. Las técnicas de IBR (*Image-Based Renderig*) utilizan imágenes para simular ciertos efectos de rendering. Entre otras, los sistemas de partículas y billboards (polígonos que se mantienen paralelos al plano de imagen de la cámara) se implementan con sprites desplegados sobre polígonos en el espacio objeto. La principal ventaja de esta aproximación es que el tiempo de render es únicamente proporcional al número de píxeles, por lo que pueden ser altamente eficientes.

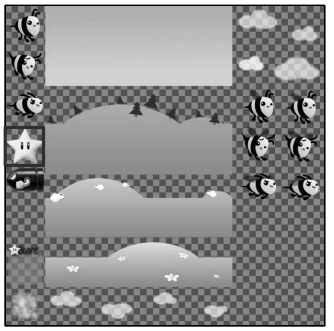
Para generar una animación es suficiente con desplegar una sucesión de diferentes sprites. Para optimizar el despliegue de Sprites se emplean técnicas como el *Sprite Batching*. Gracias al uso de esta técnica se *agrupan* los *sprites* en una única llamada a la GPU, de modo que la llama efectiva al despliegue de los mismos se realiza sin necesidad de sobrecarga extra a la CPU.

La carga efectiva de la textura se realiza en base a lo que se denomina *Texture Atlas*. Un *Texture Atlas* es una imagen que contiene más de una subimagen. Por cuestiones de eficiencia en el tratamiento posterior por la GPU es recomendable utilizar imágenes cuadradas de tamaño 2^n píxeles. Formatos habituales incluyen 512x512, 1024x1024 y 2048x2048 píxeles.

Existen multitud de formatos de especificación de *Texture Atlas*. Uno de los más utilizados en la comunidad libre es el empleado por el motor libre *Sparrow*[11]. El formato de especificación de Sparrow se basa en la construcción de un XML, que estudiaremos a continuación.

[10] En inglés, fuera del ámbito técnico de la informática gráfica, significa *duendecillo*, *trasgo*.

[11] Sparrow es un motor de desarrollo de videojuegos libre para iOS. Más información en: http://gamua.com/sparrow/



```
<TextureAtlas imagePath='bee.png'>
  <SubTexture name='beeSt' x='1' y='1' width='126' height='127'/>
  <SubTexture name='beeBk' x='1' y='129' width='126' height='127'/>
  <SubTexture name='beeFw' x='1' y='257' width='126' height='127'/>
  <SubTexture name='star' x='1' y='385' width='126' height='126'/>
  <SubTexture name='enemy' x='1' y='516' width='127' height='79'/>
  <SubTexture name='sky' x='129' y='0' width='598' height='256'/>
  <SubTexture name='trees' x='129' y='257' width='598' height='282'/>
  <SubTexture name='sheep' x='129' y='541' width='597' height='198'/>
  <SubTexture name='flowers' x='129' y='741' width='598' height='158'/>
  <SubTexture name='clouds' x='129' y='901' width='598' height='118'/>
  <SubTexture name='2cloud1' x='729' y='0' width='288' height='126'/>
  <SubTexture name='2cloud2' x='729' y='128' width='288' height='126'/>
  <SubTexture name='smoke' x='1' y='900' width='126' height='126'/>
</TextureAtlas>
```

Figura 3.8: Ejemplo de *Texture Atlas* definido en el formato XML de Sparrow. Cada sprite tiene asociado un nombre y unas dimensiones.

> Cuidado con la resolución del *Texture Atlas*. Dependiendo de la plataforma final de publicación, existen limitaciones hardware específicas de cada procesador. Por ejemplo, el *iPhone 4* cuenta con una memoria de 512MB, una resolución de pantalla de 960x640 píxeles y un tamaño de textura máximo de 2048x2048 píxeles. Sin embargo, el *iPad 3* permite cargar texturas de 4096x4096 píxeles para ser desplegadas en su pantalla de 2048x1536 píxeles.

La Figura 3.8 muestra el resultado de definir un fichero de tipo *Texture Atlas* en el formato XML de Sparrow. El formato XML requiere indicar para cada sprite el nombre (que será utilizado cuando recuperemos el gráfico mediante código), las coordenadas X, Y del vértice superior izquierdo del sprite y el ancho y alto del mismo. Así, en el ejemplo de la Figura anterior, el sprite de la estrella (*star*) tiene un tamaño de 126x126 píxeles, y comienza en el píxel (1,385) de la imagen.

Como veremos en la sección 3.2.4, el propio nombre de los elementos permite especificar animaciones.

La creación de estos *Texture Atlas* puede realizarse de una forma automática empleando herramientas específicas para su creación. Una de las más versátiles, *Texture Packer* [12], y disponible para multitud de plataformas (Windows, Mac y GNU/Linux) desafortunadamente no es libre. Una alternativa libre a *Texture Packer* es *Sprite Mapper* [13].

[12] Web oficial de Texture Packer: http://www.codeandweb.com/texturepacker
[13] Sprite Mapper: Descargable en http://opensource.cego.dk/spritemapper/

3.2.3. Capas y Tiles

Una escena puede ser entendida como un conjunto de capas. Esta aproximación es la habitual en animación de películas 2D, donde cada elemento animado es un objeto formado por diferentes partes enlazadas en un esquema jerárquico. Veremos en la sección 3.2.6 que esta aproximación de capas permite implementar esquemas ampliamente utilizados en el desarrollo de videojuegos, como el *Scroll Parallax*.

La biblioteca *TileLayer* cuenta con varias clases de utilidad para el tratamiento de capas de Sprites y Tiles. En concreto, define las siguientes clases:

La clase **TileLayer** define una capa de despliegue. Al menos nuestra escena deberá tener un elemento de este tipo, que debe ser añadido a la escena principal. Es importante minimizar el número de capas de despliegue en la escena, porque optimizará el volcado gráfico. Cada *TileLayer* tendrá asociado un conjunto de sprites definidos en un *TileSheet* (como veremos en el siguiente ejemplo, es posible que varias capas compartan el mismo conjunto de sprites). En realidad *TileLayer* es una clase hija de **TileGroup** (está formada por un conjunto de tiles, que pueden ser accedidos empleando los métodos de la clase padre).

La clase **TileGroup** gestiona grupos de *Sprites*. Como se ha comentado anteriormente, *TileLayer* es una subclase que especializa la implementación de *TileGroup*, por lo que todos estos métodos y atributos son accesibles por ambas clases. Algunas de las variables públicas y métodos de esta clase son:

- **children: array<TileBase>**. Esta variable *pública* contiene la lista de todos los Sprites añadidos al grupo. Puede ser accedido empleando los métodos estándar de tratamiento de arrays o algunos de los métodos públicos de esta clase. El elemento de posición 0 en el array será el primero que se dibuje (será ocultado por los elementos de posiciones superiores).

- **x**, **y**. Posición del *TileGroup*.

- **width**, **height**. Ancho y alto del grupo (sólo lectura).

- **visible**. Indica si el grupo será desplegado o no. Es de tipo Bool.

- **addChild(tile)**, **addChildAt(tile, index)**. Permite añadir un Sprite al grupo. El segundo método permite especificar el orden dentro del grupo (en el array *children*).

- **removeChild(tile)**, **removeChildAt(index)**. Estos métodos eliminan un Sprite del grupo.

- **removeAllChildren()**. Elimina todos los Sprites hijos de este grupo.
- **getChildIndex(tile)**. Obtiene la posición del Sprite en el array *children* del grupo.

Las clases **TileSprite** y **TileClip** se utilizan para la recuperación de Sprites individuales. La principal diferencia es que *TileSprite* no permite animaciones. A continuación describiremos los principales atributos de ambas clases:

- **tile**. Atributo de tipo *String* que contiene el nombre simbólico del Sprite.
- **x**, **y**. Posición del Sprite en pantalla.
- **rotation**. Rotación del sprite (tipo Float).
- **scale**, **scaleX**, **scaleY**. Tamaño del sprite. Por defecto, 1.0. Puede establecerse uniformemente (mediante *scale*) o de forma independiente (*scaleX* y *scaleY*).
- **width**, **height**. Ancho y alto. Propiedades de solo lectura.
- **mirror**. Efecto espejo. Por defecto, 0 (sin mirror). Si vale 1, se indica inversión en horizontal. Un valor 2 implica espejo en vertical.
- **visible**. Indica si el Sprite será visible.

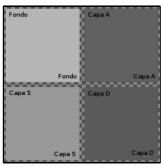
Para la gestión de animaciones, la clase **TileClip** cuenta con los siguientes atributos públicos:

- **animated**. De tipo *Bool*. Indica si está siendo animado (si *loop* está a *false*, este atributo se pondrá igualmente a *false* cuando se haya reproducido un ciclo.
- **loop**. De tipo *Bool*, indica si la animación se reproducirá en bucle.
- **currentFrame**. Indica el frame que se está dibujando actualmente.
- **totalFrames**. Número total de frames de la animación.
- **play()**, **stop()**. Métodos para reproducir o parar la animación.

A continuación estudiaremos un ejemplo básico de gestión de capas de sprites con la biblioteca *TileLayer*. El primer paso a la hora de utilizar sprites es cargar los recursos gráficos. En el siguiente listado, las líneas 11–13 cargan un objeto de tipo *SparrowTilesheet* que integra la información del XML y los gráficos descritos en la imagen 3.9.

capas.xml

```
<TextureAtlas imagePath='capas.png'>
  <SubTexture name='Fondo' x='0' y='0' width='128' height='128'/>
  <SubTexture name='CapaA' x='128' y='0' width='128' height='128'/>
  <SubTexture name='CapaS' x='0' y='128' width='128' height='128'/>
  <SubTexture name='CapaD' x='128' y='128' width='128' height='128'/>
</TextureAtlas>
```



capas.png
256x256

Salida en Terminal

```
LayerExample.hx:87: Posición S: 1
LayerExample.hx:87: Posición D: 1
LayerExample.hx:87: Posición D: 2
LayerExample.hx:87: Posición A: 0
```

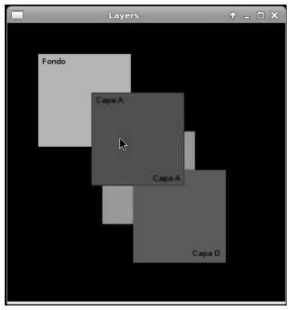


Figura 3.9: Ejemplo de gestión básica de capas: definición del xml y salida por terminal.

Hecho esto, pueden utilizarse los sprites referenciándolos por el nombre que aparece en el XML.

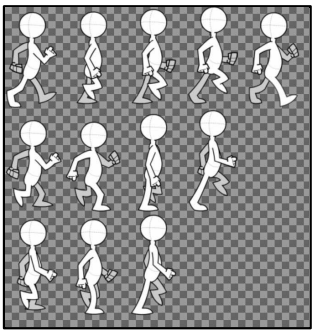
En el ejemplo se han empleado dos capas (miembros de la clase), declarados en las líneas 3-4. Estas capas utilizan el mismo archivo de Sprites (*tilesheet*), definido en la línea 14-15. La capa *layerFg* incluye tres sprites que serán reordenados según se pulsen las teclas A, S y D (ver método onKeyPressed en las líneas 48-63).

La tabla hash *hashLayer* almacena referencias a los objetos de tipo *TileSprite* que son añadidos a la capa *layerFg*. Esto nos permite recuperarlos fácilmente por el nombre y reordenarlos. Así, se ha definido un método propio auxiliar para añadir los sprites, tanto a la capa como a la tabla Hash (ver implementación en las líneas 33-38). El Sprite se crea a partir del nombre simbólico almacenado en el XML (ver línea 35), se indican las coordenadas X, Y iniciales (relativas al centro del Sprite), y se añade tanto a la capa como a la tabla Hash.

Listado 3.18: Ejemplo de gestión de capas básico.

```
1  class LayerExample extends Sprite{
2
3    var _layerBg:TileLayer;           // Capa del Background (Bg)
4    var _layerFg:TileLayer;           // Capa del Foreground (Fg)
5    var _hashLayer:Hash<TileSprite>;  // Hash de los sprites capa Fg
6
7    // Constructor ==================================================
8    private function new() {
9      super();
10
11     var sheetData:String = Assets.getText("assets/capas.xml");
12     var tilesheet:SparrowTilesheet = new SparrowTilesheet
13       (Assets.getBitmapData("assets/capas.png"), sheetData);
14     _layerBg = new TileLayer(tilesheet);
```

```
15      _layerFg = new TileLayer(tilesheet);
16      _hashLayer = new Hash<TileSprite>();
17
18      addSprite(_layerBg, "Fondo", 100, 100);
19      addChild(_layerBg.view);
20
21      addSprite (_layerFg, "CapaA", 150, 150, _hashLayer);
22      addSprite (_layerFg, "CapaS", 200, 200, _hashLayer);
23      addSprite (_layerFg, "CapaD", 250, 250, _hashLayer);
24      addChild(_layerFg.view);
25
26      Lib.current.stage.addEventListener
27        (KeyboardEvent.KEY_DOWN, onKeyPressed);
28      Lib.current.stage.addEventListener
29        (Event.ENTER_FRAME, onEnterFrame);
30    }
31
32    // addSprite ======================================================
33    function addSprite(layer:TileLayer, sprname:String, x:Int, y:Int,
34              hash:Hash<TileSprite>=null):Void {
35      var spr = new TileSprite(layer, sprname);
36      spr.x = x; spr.y = y; layer.addChild(spr);
37      if (hash != null) hash.set(sprname, spr);  // Aniadimos a Hash
38    }
39
40    // Update =========================================================
41    function onEnterFrame(event:Event):Void {
42      _layerBg.render();
43      for (e in _hashLayer) e.x += 1-Std.random(3);
44      _layerFg.render();
45    }
46
47    // Events =========================================================
48    function onKeyPressed(event:KeyboardEvent):Void {
49      var spr : TileSprite = null;
50      var keyString : String = null;
51      switch (event.keyCode) {
52      case Keyboard.A: keyString = "A";
53      case Keyboard.S: keyString = "S";
54      case Keyboard.D: keyString = "D";
55      }
56      if (keyString != null) {
57        spr = _hashLayer.get("Capa" + keyString);
58        trace ("Posicion " + keyString + " : " + _layerFg.indexOf(spr))
            ;
59        _layerFg.removeChild(spr);
60        _layerFg.addChildAt(spr, _layerFg.numChildren);
61      }
62    }
63
64    // Main ===========================================================
65    public function main() {
66      Lib.current.addChild(new LayerExample());
67    }
68  }
```



```
<TextureAtlas imagePath='atlas.png'>
  <SubTexture name='walk_00' x='0' y='0' height='150' width='87'/>
  <SubTexture name='walk_01' x='0' y='151' height='150' width='87'/>
  <SubTexture name='walk_02' x='0' y='302' height='150' width='87'/>
  <SubTexture name='walk_03' x='88' y='0' height='150' width='87'/>
  <SubTexture name='walk_04' x='176' y='0' height='150' width='87'/>
  <SubTexture name='walk_05' x='264' y='0' height='150' width='87'/>
  <SubTexture name='walk_06' x='352' y='0' height='150' width='87'/>
  <SubTexture name='walk_07' x='88' y='151' height='150' width='87'/>
  <SubTexture name='walk_08' x='88' y='302' height='150' width='87'/>
  ...
</TextureAtlas>
```

Figura 3.10: Ejemplo de definición de un Clip animado mediante un Texture Atlas.

Cuando las capas han sido definidas (se han incluido los Sprites que van a formar parte del grupo), deben añadirse a la escena que será gestionada por NME. Para ello, habrá que llamar a “addChild”, indicando el atributo *view* (ver líneas 19 y 24). Este atributo público, de tipo *Sprite*, es el que necesita NME para su correcto despliegue.

En el método que se llama automáticamente en el bucle principal *onEnterFrame* (ver Líneas 40–45) basta con llamar al método *render* de cada capa. Antes de dibujar los Sprites de la capa *layerFg* se aplica una animación aleatoria respecto de X.

Empleando la información almacenada en la tabla Hash, recuperamos el Sprite que vamos a reordenar. En la línea 57 obtenemos la referencia a la capa que tiene un determinado nombre. En la línea 58 se muestra información sobre la tecla pulsada (y la posición del Sprite dentro del grupo de Sprites en la capa *layerFg*). A menor índice en el grupo de sprites el sprite se dibujará más *profundo*. Para reordenar las capas, se elimina la referencia en la capa (línea 59), y se inserta de forma ordenada en la *cima* del array (*layerFg.numChildren* nos devuelve el número de Sprites que han sido añadidos).

3.2.4. Animación de Sprites: TileClip

Como se ha comentado en la sección anterior, la clase *TileClip* permite trabajar con Sprites animados. La carga de recursos animados es muy sencilla; basta con indicar el mismo nombre base del recurso, seguido de “_” y el número de frame dentro de la secuencia (ver Figura 3.10). En el constructor del objeto de tipo *TileClip* se puede especificar el número de frames por segundo al que queremos reproducir la animación (puede cambiarse también modificando la variable miembro *fps*).

Listado 3.19: Uso de TileClip (Fragmento).

```
1  class Ciclo extends Sprite {
2    var _layer:TileLayer;       // Capa principal de dibujado
3    var _character:TileClip;    // Clip del personaje animado
4    var _fpstext:FpsLabel;      // Clase propia para mostrar los FPs
5
6    function createScene():Void {  // Crear Escena ==================
7      var sheetData:String = Assets.getText("assets/atlas.xml");
8      var tilesheet:SparrowTilesheet = new SparrowTilesheet
9        (Assets.getBitmapData("assets/atlas.png"), sheetData);
10     _layer = new TileLayer(tilesheet);
11
12     // Carga del Clip animado (nombre "walk2 a 16 fps)
13     _character = new TileClip(_layer, "walk", 16);
14     _character.x = 0;  _character.y = stage.stageHeight / 2.0;
15     _layer.addChild(_character);
16     // Creación del objeto FpsLabel para mostrar los FPS
17     _fpstext = new FpsLabel(20, 20, 0x606060, "assets/cafeta.ttf");
18     // Añadir capa y objeto FPS a la escena...
19     addChild(_layer.view); addChild(_fpstext);
20   }
21
22   // Update ==========================================================
23   function goInside(c:TileClip, w:Int, h:Int, d: Float):Bool {
24     // Devuelve true si está dentro o camina al interior
25     if ((c.x > w + c.width / 2) && (d > 0)) return false;
26     if ((c.x < - (c.width / 2)) && (d < 0)) return false;
27     return true;
28   }
29
30   function updateCharacter():Void {
31     // Ajuste del incremento en X segun la animacion
32     var delta:Float = _character.fps * 0.12;
33     if (_character.mirror == 1) delta *= -1;
34     if (goInside(_character, stage.stageWidth,
35          stage.stageHeight, delta)) _character.x += delta;
36   }
37
38   function onEnterFrame(event:Event):Void {
39     _fpstext.update();  // Actualizamos FPS
40     updateCharacter();  // Actualizamos el personaje
41     _layer.render();    // Despliegue de la capa principal
42   }
43
44   // Events ==========================================================
45   function onKeyPressed(event:KeyboardEvent):Void {
46     switch (event.keyCode) {
47     case (Keyboard.A): if (_character.fps < 50) _character.fps++;
48     case (Keyboard.Z): if (_character.fps > 2)  _character.fps--;
49     }
50     trace ("FPS del personaje: " + _character.fps);
51   }
52
53   function onMouseClicked(event:MouseEvent):Void {
54     _character.mirror = 1-_character.mirror;
55   }
```

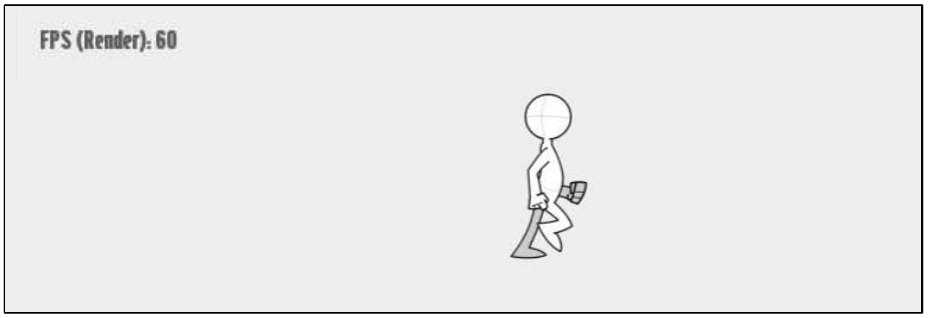


Figura 3.11: Salida del ejemplo que muestra la animación del Ciclo de Andar. El color de fondo de pantalla se ha modificado en el ejemplo cambiando el valor de la propiedad `background` en el archivo nmml.

En este fragmento de código, la carga de recursos gráficos (líneas 7–10) es similar a la estudiada en la sección anterior. Para cargar el clip animado, basta con indicar el nombre del Sprite (especificado en el archivo XML de "*atlas.xml*") sin la parte relativa al número de frame. Como se muestra en la Figura 3.10, en este caso únicamente se ha definido un Sprite animado de nombre "*walk*". El segundo parámetro del constructor de *TileClip* es opcional (línea 12), y sirve para indicar el número de FPS a los que se reproducirá la animación. Por defecto, la animación se reproduce en un bucle infinito, aunque como hemos visto en la sección anterior se puede cambiar este comportamiento cambiando el atributo de *loop* al *false*.

La velocidad de reproducción de la animación se cambia atendiendo a la pulsación de las teclas A y Z (ver líneas 45–49), en un intervalo entre 50 y 2 FPS, simplemente modificando el valor del atributo público "*fps*".

El efecto de cambio de sentido se consigue fácilmente, en el evento de pulsación del ratón, cambiando el valor del atributo "*mirror*" (ver línea 54).

Por último, la función *updateCharacter* se encarga de calcular la nueva posición del Sprite. En este caso, se ha utilizado un valor de 0.12 píxeles por cada frame (debido a que, reproduciendo los 12 frames de la animación original, el personaje avanzaba 26 píxeles). Como NME mantiene fijo el número de FPS que dibujaremos, podemos multiplicar directamente el número de fps por esa constante para evitar el efecto "*Moonwalker*" en el personaje.

La función de utilidad *goInside* (ver líneas 23–28) sirve para calcular si el personaje camina hacia el interior de la escena. Cuando el personaje sale por uno de los extremos de la pantalla (su posición en X es mayor que el ancho del Sprite en cualquiera de los dos extremos), no se actualizará la posición. De esta forma, cuando el usuario pinche de nuevo sobre la ventana, inmediatamente después aparecerá de nuevo entrando a la pantalla.

3.2.5. Texto con True Type

En la esquina superior izquierda del ejemplo de la sección anterior se hace uso de una clase propia de utilidad que muestra el número de frames por segundo a los que se está realizando el despliegue. Esta clase llamada *FpsLabel* hereda de la clase *TextField* de NME. El siguiente listado muestra su implementación.

Listado 3.20: Implementación de FpsLabel.

```
1  class FpsLabel extends TextField {
2    public var _fps:Int;   // Frames por segundo calculados
3    var _nFrames:Int;      // Numero de frames transcurridos
4    var _firstTime:Int;    // Tiempo desde la ultima vez
5    // Rango en el que se calculan los FPS de forma efectiva
6    static inline var RESETTIME : Int = 2000;
7    static inline var MINTIME : Int = 400;
8
9    // Constructor ==================================================
10   public function new(XPos:Int=20, YPos:Int=20, col:Int = 0xFFFFFF,
11               font:String = ""):Void {
12     super();
13     x = XPos;  y = YPos;  width = 300;  height = 80;
14     _nFrames = 0;  _firstTime = Lib.getTimer();
15     selectable = false;   // El texto no se puede seleccionar
16     embedFonts = true;    // La fuente se incrusta en Flash
17     if (font == "") font = "_sans";
18     else font = Assets.getFont (font).fontName;
19     var tf:TextFormat = new TextFormat(font, 16, col, true);
20     defaultTextFormat = tf;
21     htmlText = "FPS: ";
22   }
23   // update =======================================================
24   public function update():Void {
25     var now = Lib.getTimer();  var delta = now - _firstTime;
26     _nFrames++;
27     if (delta > MINTIME) _fps = Std.int(_nFrames * 1000 / delta);
28     htmlText = "FPS (Render): " + _fps;
29     if (delta > RESETTIME) {_firstTime = now; _nFrames = 1;}
30   }
31 }
```

Los objetos de tipo *TextField* tienen asociado un formato de texto (*TextFormat*), donde se indica el tipo de fuente, el tamaño, color y otras propiedades adicionales. En las líneas 18–20 del listado anterior se especifica que el tamaño es de 16 puntos y el nombre de la fuente es la que se indica por argumento en la función. En el constructor del ejemplo de la sección anterior, en la línea 17 se creaba un objeto FpsLabel especificando el nombre de la fuente True Type "*cafeta.ttf*" como argumento. La variable miembro *htmlText* permite indicar el texto que se renderizará con esa fuente.

> La clase *TextFormat* permite especificar con un alto nivel de detalle el formato de los objetos de tipo *TextField*. Es posible incluso aplicar hojas de estilo CSS, además de tags HTML. Los elementos de texto pueden incluir elementos multimedia (como películas, archivos SWF o imágenes). El texto se colocará alrededor de los elementos multimedia como lo haría un navegador. En la API de NME [14] puedes consultar más detalles sobre el soporte de texto HTML.

El comportamiento de la clase *FpsLabel* simplemente hace uso de temporizadores para calcular cuántos frames por segundo se están desplegando. La media se realiza en intervalos que se resetean entre 0.4 y 2 segundos (constantes definidas en las líneas 6–7). Cuando el tiempo es menor que 0.4 segundos, no se tiene en cuenta la muestra para evitar grandes fluctuaciones del valor mostrado. Cuando el tiempo del intervalo es mayor que 2 segundos, se inicia un nuevo intervalo de cálculo (ver línea 29).

3.2.6. Scroll Parallax

En esta sección estudiaremos uno de los efectos más ampliamente utilizados en videojuegos. El *Scroll Parallax* es una técnica que se basa en la definición de diversos planos de fondo en los que cada uno se desplaza a una velocidad diferente, creando la ilusión de profundidad en la escena. El primer ejemplo de Scroll Parallax en el mundo de los videojuegos se debe a *Moon Patrol*, publicado en 1982. Desde entonces, multitud de videojuegos de todos los estilos hacen uso de esta técnica, desde plataformas como *Super Mario* o *Sonic* en cualquiera de sus secuelas, pasando por videojuegos de lucha (como *Street Fighter II*), carreras de coches (*Out Run*) y un larguísimo etcétera.

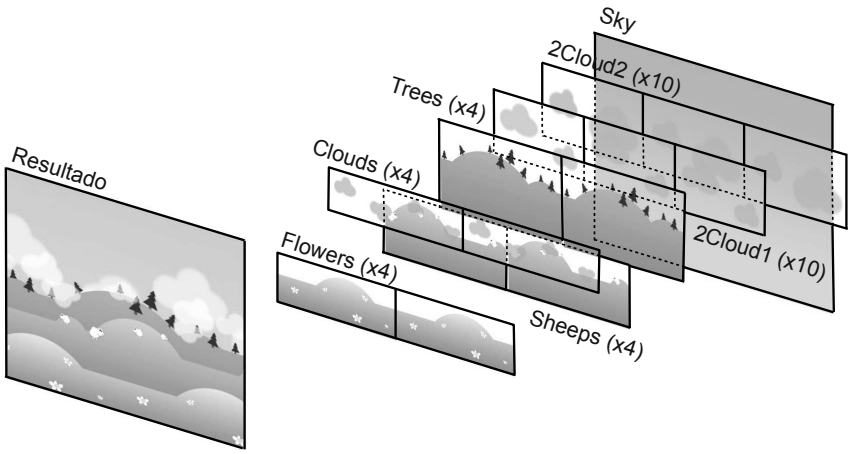


Figura 3.12: Esquema de Scroll Parallax. El resultado se obtiene de la composición de diversas capas que se desplazan a diferentes velocidades.

La forma más sencilla de implementar un Scroll Parallax es empleando grupos de sprites en diferentes capas que son controlados de forma independiente. Para conseguir el efecto Parallax simplemente tenemos que desplazar los elementos de cada capa a una velocidad diferente (ver Figura 3.12). A continuación se muestra un ejemplo de uso de una clase de utilidad que hemos creado llamada *ParallaxManager*, y que estudiaremos a continuación.

Listado 3.21: Ejemplo de uso Parallax.hx (Fragmento).

```
1  import graphics.ParallaxManager;
2
3  class Parallax extends Sprite {
4    var parallaxManager:ParallaxManager;   // Manager del Efecto
5
6    function createScene():Void {
7      loadAssets();  // Carga los recursos y crea la capa principal
8      parallaxManager = new ParallaxManager(layer);
9      parallaxManager.addSky("sky");
10     parallaxManager.addBackgroundStrip("2cloud2", 10, 1.7, 300, -5);
11     parallaxManager.addBackgroundStrip("2cloud1", 10, 2, 320, 10);
12     parallaxManager.addBackgroundStrip("trees", 4, 1, 220, 40);
13     parallaxManager.addBackgroundStrip("sheeps", 4, 1, 160, 140);
14     parallaxManager.addBackgroundStrip("clouds", 4, 1.3, 280, 80);
15     parallaxManager.addBackgroundStrip("flowers", 4, 1, 70, 300);
16   }
17 }
```

El uso de la clase *ParallaxManager* es muy sencilla. Basta con clear un objeto de ese tipo. El constructor (ver línea 8) recibe como parámetro la referencia a la capa de dibujado principal (de tipo *TileLayer* estudiada anteriormente). La implementación actual del objeto permite añadir dos tipos de elementos: el fondo (Sky) (ver línea 9) que será reescalado para que ocupe toda la ventana, o capas de Scroll llamadas *Background Strip*. Los sprites utilizados en este ejemplo son los definidos en la Figura 3.8.

> La implementación actual de la clase ParallaxManager realiza el scroll únicamente en horizontal, repitiendo los mismos sprites infinitamente. Queda como ejercicio propuesto para el lector que modifique la implementación de la clase para que permita cambiar los sprites aleatoriamente entre una lista de sprites especificados como argumento y la realización de scroll en vertical.

Listado 3.22: ParallaxManager.hx (Fragmento).

```
1  package graphics;
2  import graphics.sprites.Sky;
3  import graphics.sprites.BackgroundStrip;
4  import aze.display.TileLayer;
5
6  class ParallaxManager {
7    var _root:TileLayer;      // Capa principal de despliegue
8    var _sky:Sky;             // Objeto de tipo Sky (Fondo)
9    var _vBackgroundStrip:Array<BackgroundStrip>;  // Lista de Strips
10   // Constructor ==================================================
11   public function new(layer:TileLayer) {
12     _root = layer;
13     _vBackgroundStrip = new Array<BackgroundStrip>();
14   }
15   // Añadir Elementos ============================================
16   public function addSky(id:String) {
17     _sky = new Sky(_root, id);  _root.addChild(_sky);
18   }
19   public function addBackgroundStrip(id:String, nTiles:Int,
20           sc:Float, yPos:Int, speed:Float) {
21     var bgStrip = new BackgroundStrip(_root, id, nTiles, sc, yPos,
           speed);
22     _root.addChild(bgStrip);
23     bgStrip.update();    // Sin argumentos: Primer posicionamiento
24     _vBackgroundStrip.push(bgStrip);
25   }
26   // Update ======================================================
27   public function update(w:Int, h:Int, eTime:Int):Void {
28     _sky.update(w, h);
29     for (bgStrip in _vBackgroundStrip) bgStrip.update(w, h, eTime);
30   }
31 }
```

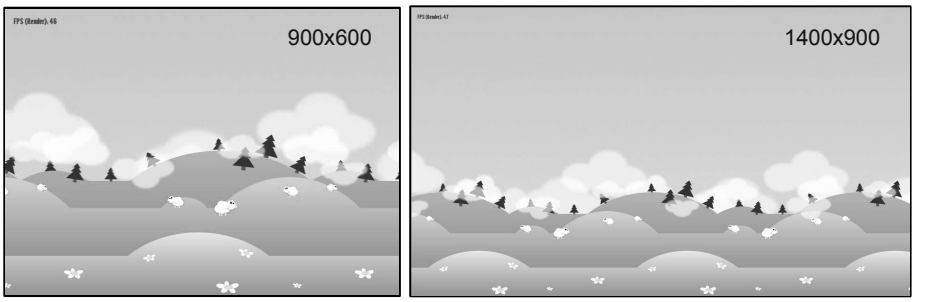


Figura 3.13: La clase ParallaxManager permite trabajar de forma independiente de la resolución. Las capas se posicionan de forma relativa con el borde inferior de la ventana.

Veamos a continuación los aspectos más relevantes de la implementación de la clase *ParallaxManager*.

La clase mantiene una referencia a un objeto de tipo *Sky* (ver línea 7), que está definido en el paquete *graphics.sprites.Sky* de nuestro ejemplo (línea 2). De forma análoga, se mantiene una lista de todos los *BackgroundStrip* que añade el usuario (ver línea 8). Estos elementos se añaden a la capa principal de despliegue (pasada como argumento del constructor en la línea 11).

> En la implementación actual de la clase ParallaxManager, el orden en el que se crean las capas (mediante la llamada a *addBackgroundStrip*) es el que se utilizará para su posterior despliegue. La primera capa creada estará situada debajo del resto.

Cada *BackgroundStrip* requiere cinco parámetros (ver líneas 19–20). En *id* se especifica el nombre del Sprite (especificado en el Texture Atlas). El parámetro *nTiles* indica el número de repeticiones del strip (debe cubrir toda la pantalla y permitir el scroll hasta que se alcance la mitad del primer sprite y pueda resetearse la posición). El factor de escala *sc* se aplicará a todos los sprites de esa capa. Como el scroll se realiza únicamente en horizontal, la posición *yPos* se especifica desde el borde inferior de la pantalla. De esta forma, si maximizamos la ventana, el suelo siempre quedará "*pegado*" al borde inferior de la misma. Finalmente el parámetro *speed* indica la velocidad a la que realizaremos el scroll en esa capa.

El posicionamiento efectivo de los elementos que forman cada strip tiene que realizarse después de que sean añadidos a la capa (ver líneas 22-23). Por esta razón, es necesario separar la creación de la capa con su posicionamiento efectivo.

El siguiente listado muestra la implementación completa de la clase Sky. Como se ha comentado anteriormente, utiliza un Sprite que es posicionado ocupando el área total de la pantalla. Como los atributos de *width* y *height* son de lectura, es necesario calcular el factor de escala *scaleX* y *scaleY* (ver línea 10).

Listado 3.23: Clase de Utilidad Sky.hx.

```
 1 package graphics.sprites;
 2 import aze.display.TileSprite;
 3 import aze.display.TileLayer;
 4
 5 class Sky extends TileSprite {
 6   public function new(l:TileLayer, id:String):Void {
 7     super(l, id); x = 0; y = 0;
 8   }
 9   public function update(w:Int, h:Int):Void {
10     scaleX = 2*w / width;   scaleY = 2*h / height;
11   }
12 }
```

El siguiente listado muestra los aspectos más importantes de la implementación de la clase *BackgroundStrip*. La clase es en realidad una clase hija de *TileGroup* por lo que hereda todos sus atributos y métodos.

Cada Strip está formado por un array de *TileSprite*, que se mantiene con el fin de actualizar su posición en el método *update*. Como se ha comentado anteriormente, es necesario separar el posicionamiento de los Sprites hasta que no se añadan a la capa general. Con el fin de eliminar acoplamiento, esta clase no recibe como parámetro la capa general sobre la que será desplegada, por lo que es necesario diferenciar entre la primera llamada a *update* y las siguientes. Para ello, se han utilizado los parámetros por defecto en la función (ver línea 24).

Si el método se llama sin parámetros, recibirá como *eTime=0*, por lo que entrará en la primera parte del if (líneas 28-32). Esta parte del código inicaliza la posición de los subTiles que forman el grupo.

Por su parte, la actualización del Strip de forma general se realiza en las líneas 35-36. El efecto de *scroll* infinito se consigue reseteando su posición cuando la posición en X sea menor que la mitad del ancho. En ese caso, el strip vuelve a la posición inicial de x=0, por lo que comienza de nuevo su desplazamiento hacia la izquierda.

Listado 3.24: Fragmento de BackgroundStrip.hx.

```
1  package graphics.sprites;
2  import aze.display.TileGroup;
3  import aze.display.TileSprite;
4  import aze.display.TileLayer;
5
6  class BackgroundStrip extends TileGroup {
7    var _vbgTiles:Array<TileSprite>;   // Array del grupo
8    public var _yPos:Int;              // Pos. Vertical
9    public var _speed:Float;           // Velocidad del Scroll
10
11   public function new(l:TileLayer, id:String, nTiles:Int,
12               fScale:Float, yPos:Int, speed:Float) {
13     super();
14     this._yPos = yPos;  this._speed = speed;
15     _vbgTiles = new Array<TileSprite>();
16     for (i in 0...nTiles) {  // Añadimos los subtrips
17       var bgTile = new TileSprite(l, id);
18       bgTile.scale = fScale;
19       addChild(bgTile);
20       _vbgTiles.push(bgTile);
21     }
22   }
23
24   public function update(w:Int=0, h:Int=0, eTime:Int=0):Void {
25     var pos:Int;
26
27     if (eTime == 0) {   // Inicializacion de los substrips
28       for (i in 0..._vbgTiles.length) {
29         if (i == 0) pos = Std.int(_vbgTiles[i].width / 2.0);
30         else pos = Std.int(_vbgTiles[i-1].x + _vbgTiles[i-1].width);
31         _vbgTiles[i].x = pos;
32       }
33     }
34     else { // Actualización del strip (general)
35       x -= (eTime / 1000.0) * _speed;
36       if (x < -width / _vbgTiles.length) x = 0;
37     }
38     y = h - _yPos;
39   }
40
41 }
```

3.2.7. Bee Adventures: Mini Juego

En esta subsección desarrollaremos como caso de estudio concreto un Mini-Juego que utilice las clases desarrolladas anteriormente. En concreto será un videojuego de *Scroll* horizontal con el objetivo de recoger la mayor cantidad de objetos de tipo estrella y evitar los proyectiles. El videojuego no tiene final; cada estrella sumará un punto y cada vez que el jugador choque con un proyectil se restarán dos puntos. La Figura 3.14 resume el diagrama de clases del juego.

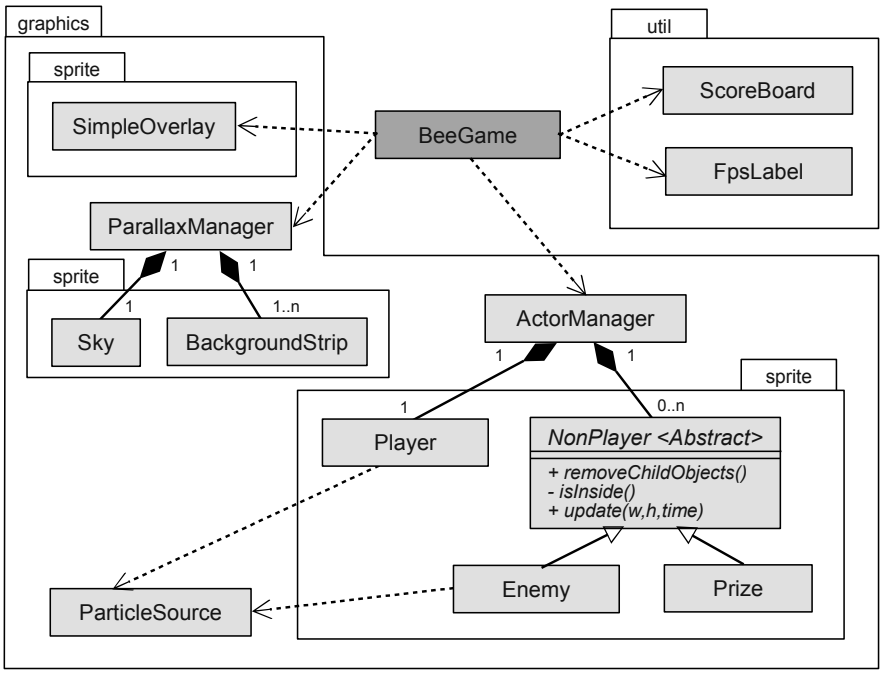


Figura 3.14: Diagrama de clases general de Bee Adventures.

El código está estructurado en dos paquetes principales. El paquete **util** contiene las clases relativas a la gestión del marcador *ScoreBoard* y la clase para mostrar el rendimiento gráfico del sistema *FpsLabel*, que ha sido explicada con anterioridad.

Por su parte, el paquete **graphics** contiene las clases relativas a la parte gráfica y de comportamiento del videojuego. Dentro del paquete *graphcis* hay tres clases de gestión general: el *ParallaxManager* estudiado en la sección anterior, el *ActorManager* que se encarga de la gestión de los actores del juego (tanto jugador como actores no jugadores), y el *ParticleSource* que se encarga de gestionar las fuentes de partículas.

El paquete **sprite** contiene las clases específicas a nivel de elemento individual (habitualmente móvil), como *SimpleOverlay* (empleado para desplegar elementos estáticos en la pantalla, como el fondo del marcador de la puntuación), el fondo *Sky*, las capas de Scroll *BackgroundStrip*, el jugador principal *Player*, y el resto de actores *NonPlayer* y sus clases hijas *Enemy* y *Prize*.

Estudiemos a continuación los aspectos más relevantes de la clase principal del ejemplo: BeeGame (ver siguiente listado).

Listado 3.25: Fragmento de BeeGame.hx.

```
1  class BeeGame extends Sprite {
2    var _scoreBoard:ScoreBoard;           // Actualiza los puntos
3    var _parManager:ParallaxManager;      // ParallaxScroll Manager
4    var _actorManager:ActorManager;       // Manager de Actores
5    var _layer:TileLayer;                 // Capa Pcpal. de dibujado
6
7    // Creación de la Escena ========================================
8    function createScene():Void {
9      loadAssets();   // Llamamos a función propia para cargar Assets
10     // Creamos el ScrollManager igual que estudiamos en el ejemplo
11     // de la sección anterior, añadiendo las capas y el Sky.
12       _actorManager = new ActorManager(_layer, stage.stageWidth,
13             stage.stageHeight, _scoreBoard);
14     _actorManager.addPlayer("bee");
15   }
16
17   // Bucle Principal ==============================================
18   function mainLoop(t:Int):Void {
19     _parManager.update(stage.stageWidth, stage.stageHeight, t);
20     _actorManager.update(stage.stageWidth, stage.stageHeight, t);
21     _layer.render();
22   }
23
24   // Listeners ====================================================
25   function addListeners():Void {
26     stage.addEventListener(Event.ENTER_FRAME, onEnterFrame);
27     stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove); }
28
29   function onMouseMove(event:MouseEvent):Void {
30     _actorManager.onMouseMove(event); }
31
32   function onEnterFrame(event:Event):Void {
33     var now = Lib.getTimer(); var elapsedTime = now - _previousTime;
34     _previousTime = now;
35     mainLoop(elapsedTime); }
36 }
```

La clase suscribe dos manejadores de eventos en *addListeners* (líneas 26–27). El bucle principal se llamará cada vez que se ejecute *mainLoop* (líneas 18–22) asociado al evento *ENTER_FRAME*. Por su parte, los eventos de ratón y teclado serán directamente propagados al *ActorManager*, que se encargará a su vez de propagarlos a la clase *Player*. Esto facilita que los eventos sean tratados por los objetos que implementen la lógica para darles tratamiento.

El *mainLoop* por su parte se encarga de llamar a los métodos de actualización de cada gestor (*ParallaxManager* y *ActorManager*, líneas 19–20).

A continuación estudiaremos el objeto de tipo *ActorManager*, que se encarga de mantener la lista de todos los actores del videojuego.

Listado 3.26: Fragmento de ActorManager.hx.

```
1  class ActorManager {
2    var _root:TileLayer;            // Capa de dibujado
3    var _w:Int;  var _h:Int;        // Ancho y alto de _root
4    var _player:Player;             // Objeto de jugador Pcpal.
5    var _vPrize:Array<NonPlayer>;   // Array de estrellas activas
6    var _vEnemy:Array<NonPlayer>;   // Array de enemigos activos
7    var _scoreBoard:ScoreBoard;     // Objeto para mostrar puntuacion
8
9    inline static var PROBPRIZE:Int = 10;   // Probabilidad de Estrella
10   inline static var PROBENEMY:Int = 10;   // Probabilidad de Enemigo
11   // Constructor ==================================================
12   public function new(layer:TileLayer, w:Int, h:Int, sb:ScoreBoard) {
13     _root = layer; _w = w; _h = h; _scoreBoard = sb;
14     _vPrize = new Array<NonPlayer>();
15     _vEnemy = new Array<NonPlayer>();
16   }
17   // Añadir Elementos ============================================
18   public function addPlayer(id:String) {
19     _player = new Player(_root, id,_scoreBoard,_vPrize,_vEnemy);
20   }
21   function addPrize(id:String) {
22     var p = new Prize(id, _root, _w,_h, 127,127,Std.random(10));
23     _root.addChild(p);  _vPrize.push(p);
24   }
25   function addEnemy(id:String) {
26     var p = new Enemy(id, _root,_w,_h, 127,79, 5+Std.random(10));
27     _root.addChild(p);  _vEnemy.push(p);
28   }
29   // Update ======================================================
30   function updateNonPlayer(v:Array<NonPlayer>, eTime:Int):Void {
31     for (p in v)
32       if (p.update(_w, _h, eTime) == false) {
33          _root.removeChild(p); v.remove(p);
34       }
35   }
36   public function update(w:Int, h:Int, eTime:Int):Void {
37     _player.update(w,h,eTime); _w = w; _h = h;
38     if (Std.random(1000) < PROBPRIZE) addPrize("star");
39     if (Std.random(1000) < PROBENEMY) addEnemy("enemy");
40     updateNonPlayer(_vPrize, eTime);
41     updateNonPlayer(_vEnemy, eTime);
42   }
43   public function onMouseMove(event:MouseEvent):Void {
44     _player.onMouseMove(event);  }
45 }
```

El *ActorManager* mantiene dos listas de actores no jugadores; por un lado los *Premios* (línea 5) que en este caso son las estrellas que debe recoger el jugador, y por otro lado los enemigos (línea 6) que debe evitar. La clase se encarga igualmente de la gestión del jugador principal *Player* (línea 4).

La creación de enemigos y premios se realiza aleatoriamente en la función *update* (ver líneas 38–39) en base a una probabilidad definida en las constantes de las líneas 9–10. Mediante la llamada al método *updateNonPlayer* se ejecuta el método *update* de cada objeto NonPlayer (línea 32). La llamada a *update* devuelve "*false*" si el objeto detecta que debe ser eliminado (en el caso de que salga fuera de las coordendas de la ventana), por lo que se elimina del array de *NonPlayer* y de la capa principal de dibujado (línea 33).

La clase *NonPlayer* ha sido diseñada como una clase abstracta. El método de *removeChildObjects* puede ser sobreescrito si el objeto hijo de *NonPlayer* cuenta a su vez con otros hijos (como sistemas de partículas, que estudiaremos en la sección 3.2.8). De igual modo, la clase proporciona una implementación general para comprobar si el objeto está dentro de los límites de la pantalla en *isInside* (ver línea 13).

Finalmente, la clase *update* (líneas 18–20) debe ser sobreescrito por todas las clases hijas, devolviendo *true* si tiene que seguir actualizándose o *false* si ha finalizado.

Listado 3.27: Fragmento de NonPlayer.hx.

```
1  class NonPlayer extends TileClip {
2    var _w:Int;           // Ancho de la pantalla
3    var _h:Int;           // Alto de la pantalla
4    var _root:TileLayer;  // Capa principal de dibujado
5
6    public function new(id:String, layer:TileLayer, w:Int, h:Int) {
7      super(layer, id); _w = w; _h = h; _root = layer;  }
8
9    // Metodo para eliminar objetos hijo (si los tuviera)
10   public function removeChildObjects():Void { }
11
12   // Funcion general para determinar si el objeto esta en pantalla
13   function isInside():Bool { return (x > -width / 2); }
14
15   // Este método será sobreescrito por las clases hijas.
16   // Devuelve true si tiene que seguir actualizándose y false
17   // en caso contrario (ha finalizado y debe ser eliminado).
18   public function update(w:Int, h:Int, eTime:Int):Bool {
19     return (true);   }
20 }
```

Como se estudió en el diagrama de la Figura 3.14, hay dos clases hijas de la clase abstracta *NonPlayer*. La clase *Enemy* sirve para representar los proyectiles, mientras que la clase *Prize* representa las estrellas que debe recoger el jugador. A continuación veremos los aspectos más importantes de su implementación.

Listado 3.28: Fragmento de Prize.hx.

```
1  class Prize extends NonPlayer {
2    var _speed:Int;      // Velocidad de movimiento
3    var _deltaY:Int;     // Desplazamiento en vertical
4    // Constructor ==================================================
5    public function new(id:String, layer:TileLayer, w:Int, h:Int,
6                spw:Int, sph:Int, speed:Int) {
7      super(id, layer, w, h);
8      _speed = speed + 1;
9      _deltaY = Std.random(5) + 1;
10     y = Std.int(sph/2.0 + Std.random(Std.int(h - sph)));
11     x = Std.int(w + spw);
12   }
13   // Actualización ================================================
14   public override function update(w:Int, h:Int, eTime:Int):Bool {
15     x-= _speed;   y += _deltaY;  rotation += _speed * 0.002;
16     // Rebote superior
17     if (y < height /2) { y = height/2; _deltaY *= -1; }
18     // Rebote inferior
19     if (y > h - height /2) { y = h - height/2; _deltaY *= -1;  }
20     return (isInside());   // Utiliza método de la clase NonPlayer
21   }
22 }
```

La clase *Prize* únicamente define dos variables miembro: *speed* que representa la velocidad de desplazamiento del objeto, y *deltaY* que describe el incremento de posición en Y en cada frame (ver líneas 2-3). El valor de velocidad se pasa como argumento al constructor de la clase, mientras que el valor de *deltaY* se decide aleatoriamente (línea 9).

El constructor define igualmente la posición inicial de este tipo de objetos en el exterior de la pantalla (margen derecho), indicando como coordenada Y un valor aleatorio dentro de la resolución de la ventana (ver líneas 10-11.

Por su parte, la clase *Enemy* descrita en el siguiente listado define, además de la velocidad de desplazamiento del objeto (*speed*), utiliza un sistema de partículas para dejar el rastro de humo (ver Figura 3.16). Este sistema de partículas se crea en la línea 10, indicando que será de tipo *Linear*. Describiremos los sistemas de partículas en la sección 3.2.8.

Si el enemigo alcanza el extremo izquierdo de la ventana, dejamos que siga dibujándose 800 píxeles más (línea 15) para que se actualice correctamente el sistema de partículas y evitar que se elimine antes de que se haya completado de dibujar la estela de humo.

El método de actualización (líneas 29-33) llama a un método específico para actualizar la posición del sistema de partículas. Si las partículas lanzadas en el sistema se han agotado (línea 24), se añaden 50 partículas adicionales con 100 frames de intervalo entre ellas.

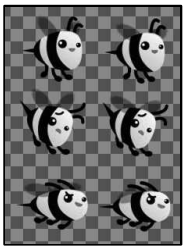
Listado 3.29: Fragmento de Enemy.hx.

```
1  class Enemy extends NonPlayer {
2    var _speed:Int;                    // Velocidad de desplazamiento
3    var _particles:ParticleSource;     // Fuente de partículas
4
5    // Constructor ==================================================
6    public function new(id:String, layer:TileLayer,  w:Int, h:Int,
7                spw:Int, sph:Int, speed:Int) {
8      super(id, layer, w, h);  _speed = speed ;
9      y = Std.int(sph/2.0 + Std.random(Std.int(h - sph)));
10     x = Std.int(w + spw);
11     _particles = new ParticleSource("smoke", _root, x, y, Linear);
12   }
13   // Métodos sobreescritos en Enemy ================================
14   override function isInside():Bool {
15     return (x > -800);  }  // 800 Para actualizar partículas
16
17   public override function removeChildObjects():Void {
18     _particles.clearParticles();  }   // Elimina las partículas
19
20   // Actualización de partículas ===================================
21   function updateParticles(w:Int, h:Int, eTime:Int) {
22     _particles.setPosition(x+width/2.0,y);  // Posición de fuente
23     _particles.update(w, h, eTime);         // Actualiza todo
24     if (_particles.getNParticles() == 0)    // Si no quedan...
25       _particles.addParticles(50, 100);     // Añade más!
26     if (isInside() == false) removeChildObjects();
27   }
28   // Actualización del enemigo =====================================
29   public override function update(w:Int, h:Int, eTime:Int):Bool {
30     x-= _speed;                   // Actualiza posicion del enemigo
31     updateParticles(w,h,eTime);   // Redibuja o añade particulas
32     return (isInside());          // Devuelve si está dentro
33   }
34 }
```

Por último, en el siguiente listado, estudiaremos la clase *Player* que modela el comportamiento del jugador principal. El jugador mantiene la lista de referencias a los objetos de tipo *NonPlayer* (líneas 15,16) para comprobar si hubo colisión con ellos e implementar la lógica que dé tratamiento a esta situación.

En la clase del jugador principal se han definido tres estados internos en los que puede encontrarse, modelados en un tipo de datos enumerado *PState* (*Player State*, en las líneas 2-6.

La carga de recursos gráficos asociados a esos estados internos se realiza empleando *Type*, la API de Reflexión de Haxe que permite obtener información sobre variables, clases y tipos Enum en tiempo de ejecución. Así, en las líneas 35-39 del constructor, se obtiene en tiempo de ejecución cada identificador del *Enum PState* (línea 35), cargando el *TileClip* que se llama igual que el nombre del jugador concatenado con el guión y con el nombre del estado.



```
<TextureAtlas imagePath='bee.png'>
  // Eliminadas algunas entradas...
  <SubTexture name='bee-Sforward_00' x='750' y='518' width='126' height='127'/>
  <SubTexture name='bee-Sforward_01' x='888' y='521' width='126' height='127'/>
  <SubTexture name='bee-Sstatic_00' x='750' y='260' width='126' height='127'/>
  <SubTexture name='bee-Sstatic_01' x='886' y='264' width='126' height='127'/>
  <SubTexture name='bee-Sback_00' x='750' y='380' width='126' height='127'/>
  <SubTexture name='bee-Sback_01' x='887' y='383' width='126' height='127'/>
</TextureAtlas>
```

Figura 3.15: Convenio de nombrado de los sprites para facilitar la carga en tiempo de ejecución empleando *Type* en Haxe.

Así, como se muestra en la Figura 3.15, hay una animación de dos frames definidas para cada estado. La animación utilizada en el avance del personaje principal se obtiene automáticamente con la cadena *bee-Sforward* (recordemos que la implementación de *TileClip* concatena automáticamente si es necesario el guión bajo "_" y el número de cada frame).

La clase del jugador principal utiliza igualmente un sistema de partículas (en este caso de tipo Radial) para desplegar un efecto cuando colisiona con un objeto de tipo *Enemy* (ver línea 41 del constructor). A continuación estudiaremos los métodos implementados para la detección de colisiones.

Listado 3.30: Fragmento de Player.hx (Constructor).

```
 1 // Estados del jugador ============================================
 2 enum PState {
 3   Sstatic;    // Estático (El ratón no se mueve NFRAMESCSTATE)
 4   Sforward;   // Avanzando (El Sprite está detrás del ratón)
 5   Sback;      // Retrocediendo (El Sprite está delante del ratón)
 6 }
 7
 8 class Player { // Class Player ====================================
 9   var _root:TileLayer;              // Capa pcpal. de despliegue
10   var _scoreBoard:ScoreBoard;       // Puntuación
11   var _x:Float;  var _y:Float;      // Posición del jugador
12   var _hashState:Hash<Int>;         // Id de estados del jugador
13   var _vStateClip:Array<TileClip>;  // Animaciones de cada estado
14   var _cState:PState;               // Estado actual (currentState)
15   var _vPrize:Array<NonPlayer>;     // Array de Premios (estrellas)
16   var _vEnemy:Array<NonPlayer>;     // Array de Enemigos
17   var _mouseX:Float;                // Posición del Ratón
18   var _mouseY:Float;
19   var _particles:ParticleSource;    // Partículas (choque Enemigo)
20
21   inline static var NFRAMESCSTATE:Int = 50;  // Frames para Sstatic
22   inline static var SPRITESIZE:Int = 64;     // Tamaño del jugador
23
```

```
24   // Constructor ==================================================
25   public function new(layer:TileLayer, id:String, sb:ScoreBoard,
26               vPrize:Array<NonPlayer>,
27               vEnemy:Array<NonPlayer>):Void {
28     _root = layer; _scoreBoard = sb;
29     _x = 200; _y = 200; _mouseX = _x; _mouseY = _y;
30     _vPrize = vPrize; _vEnemy = vEnemy; _cState = Sstatic;
31     _vStateClip = new Array<TileClip>();
32     _hashState = new Hash<Int>();
33
34     var i = 0; // Cargamos las animaciones asociadas a cada estado
35     for (value in Type.getEnumConstructs(PState)) {
36       var sClip = new TileClip(layer, id + "-" + value);
37       _root.addChild(sClip);
38       _vStateClip.push(sClip);
39       _hashState.set(value, i);  i++;
40     }
41     _particles = new ParticleSource("smoke", _root, _x, _y, Radial);
42     updateVisibleClip();
43   }
```

El método *checkCollision* (líneas 3-7) comprueba si las coordenadas del sprite del jugador solapan con las coordenadas del objeto *NonPlayer*, devolviendo true en este caso. Por su parte, el método *checkCollisionArray* recorre el array de *NonPlayer* pasado como argumento para comprobar si hay colisión individual con cada objeto. Este método se utiliza para comprobar la colisión con el array de *Prize* y con el array de *Enemy*.

Listado 3.31: Fragmento de Player.hx (Collision).

```
1  // checkCollision ==================================================
2  function checkCollision(e:NonPlayer):Bool {
3    if (Math.abs(_x - e.x) < (e.width / 2.0)) {
4      if (Math.abs(_y - e.y) < (e.height / 2.0)) return true;  }
5    return false;
6  }
7
8  // checkCollisionArray ==========================================
9  // Recorre array NonPlayer para ver si el jugador choca con ellos
10 function checkCollisionArray(v:Array<NonPlayer>):Int {
11   var i = 0;
12   for (e in v) {
13     if (checkCollision(e)) {    // Si hay choque
14       v.remove(e);              // Eliminamos el objeto
15       _root.removeChild(e);
16       // Si el objeto es un enemigo, creamos partículas de humo
17       var straux = Type.getClassName(Type.getClass(e));
18       if (straux.indexOf("Enemy")>0) _particles.addParticles(40);
19       e.removeChildObjects();   // Eliminamos sus objetos hijo
20       i++;
21     }
22   }
23   return i;
24 }
```

De nuevo se utiliza parte de la funcionalidad de introspección de Haxe para obtener el nombre de la clase que se está estudiando. Únicamente si el objeto es de tipo *Enemy* (ver línea 19), se añaden partículas a la escena para mostrar la colisión. En el caso de colisión con un objeto de tipo *Prize* no se añade ningún tipo de realimentación visual.

Por último estudiaremos la actualización del objeto *Player*. El siguiente fragmento de código muestra los principales métodos relacionados.

Listado 3.32: Fragmento de Player.hx (Update).

```
1   // Trabajo con Clips ==========================================
2   function getClipIndex(id:PState):Int {
3     if (_hashState.exists(Std.string(id)))
4       return (_hashState.get(Std.string(id)));
5     return -1;  }
6
7   function updateVisibleClip():Void {
8     for (elem in _vStateClip) elem.visible = false;
9     _vStateClip[getClipIndex(_cState)].visible = true;  }
10
11  // Update =====================================================
12  function updateFromMousePosition():Void {
13    var v = new Point(_mouseX - _x, _mouseY - _y);
14    // Actualizamos posición del objeto (más rápido si adelante)
15    if (Math.abs(v.length) > 0.3) {
16      if (v.x > 0) _x += v.x * 0.05;
17      else _x += v.x * 0.02;
18      _y += v.y * 0.05;
19      if (v.x > 0) _cState = Sforward; else _cState = Sback;
20    }
21    else _cState = Sstatic;
22  }
23
24  function updateParticles(ps: ParticleSource, w:Int,
25              h:Int, eTime:Int) {
26    ps.setPosition(_x,_y); ps.update(w, h, eTime); }
27
28  public function update(w:Int, h:Int, eTime:Int):Void {
29    updateFromMousePosition();
30    updateVisibleClip();
31    updateParticles(_particles, w, h, eTime);
32    _vStateClip[getClipIndex(_cState)].x = _x;
33    _vStateClip[getClipIndex(_cState)].y = _y;
34    var nprize = checkCollisionArray(_vPrize);
35    _scoreBoard.update(nprize);
36    var nenemy = checkCollisionArray(_vEnemy);
37    _scoreBoard.update(nenemy * -2);
38  }
39
40  // Events =====================================================
41  public function onMouseMove(event:MouseEvent):Void {
42    _mouseX = event.localX; _mouseY = event.localY;  }
```

Cada vez que se mueve el ratón en pantalla, se almacenan en dos variables miembro *_mouseX* y *_mouseY* la última posición del ratón (ver líneas 41–42). Estas variables son utilizadas en cada frame para calcular la posición del jugador en el método *updateFromMousePosition* (ver líneas 12–22).

Definimos un *vector* empleando la clase *Point* (línea 13) desde la posición actual del jugador apuntando hacia la posición del ratón. Si la longitud del vector es mayor que 0.3 (línea 15), actualizamos la posición del jugador. En otro caso, el jugador está *muy cerca* de la posición del ratón, y no necesitamos actualizar su posición, aunque cambiamos el estado interno del personaje a estático (línea 21).

Si la componente X del vector es mayor que cero, es porque la posición del ratón está *delante* del jugador. En caso contrario, el personaje debe *retroceder* hasta la posición del ratón. Actualizamos el estado interno del personaje en base a esta información (línea 19), y desplazamos la posición del personaje en la misma dirección del vector (líneas 16–18), multiplicando por un determinado factor. En el caso de que el personaje se mueva hacia delante, el ajuste en esa dirección se realiza más rápidamente (multiplicando por 0.05, en la línea 16). Si el personaje tiene que retroceder, simulamos que ese movimiento es más costoso, multiplicando por 0.02 (línea 17). El movimiento en el eje vertical siempre se realiza igual de rápido (línea 18).

Dependiendo del estado interno del personaje, mostraremos únicamente uno de los clips cargados. Para ello, se mantiene un array de clips que actualizaremos dependiendo del estado interno del personaje en el método *updateVisibleClip* (líneas 7–9). La idea es ocultar todos los clips y dejar visible únicamente el correspondiente al estado actual, actualizando la posición del mismo según las coordenadas *x,y* del jugador (ver líneas 32–33).

Finalmente, en el método *update* se actualiza la puntuación. La llamada a *checkCollisionArray* devuelve el número de colisiones encontradas en el array pasado como argumento. Dependiendo del número de choques detectado con cada tipo de objetos (líneas 34–37), se actualizará la puntuación sumando un punto por cada *Prize* y restando dos puntos por cada *Enemy*.

Para la creación de multitud de efectos especiales como fuego, explosiones, humo, lluvia y muchos más se emplean los denominados sistemas de partículas. Estas fuentes de Sprites se basan en la creación de un alto número de entidades que se actualizan de forma independiente una vez que son producidas. En la próxima sección estudiaremos la implementación utilizada en *Bee Adventures*.

3.2.8. Sistemas de Partículas

La clase desarrollada para el soporte de sistemas de partículas se ha denominado *Particle Source*. La implementación actual soporta dos tipos de partículas, pero puede ser fácilmente extendida para manejar nuevos tipos. Queda como ejercicio propuesto para el lector añadir nueva funcionalidad.

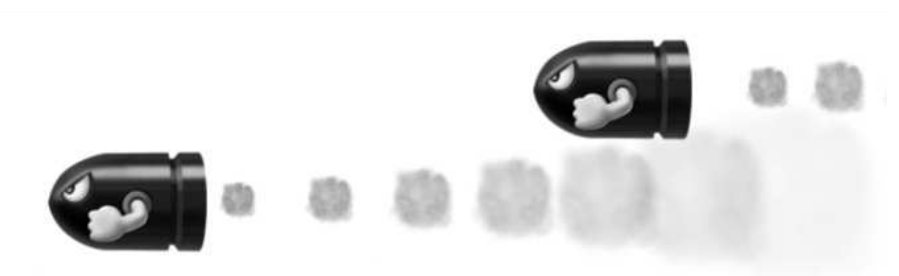


Figura 3.16: Ejemplo de utilización de la clase ParticleSource. Creación de una fuente Lineal.

> Aunque en esta sección hemos utilizado directamente nuestra propia implementación de sistemas de partículas, en NME existen bibliotecas específicas para su creación, como FLiNT (flint-particles.org).

Los sistemas de partículas son entidades que se enlazan a la Escena. En nuestra implementación actual se adjuntan a una capa de despliegue. Una vez que se han emitido las partículas, estas pasan a formar parte de la escena de forma independiente, por lo que una vez que han sido creadas, aunque se se mueva el punto de emisión del sistema, las partículas no se verán afectadas. Esto es interesante si, por ejemplo, se quiere dejar una estela de humo (como en el ejemplo de la Figura 3.16).

Los sistemas de partículas deben definir una cantidad límite de partículas (en muchos motores se denominan *quota*). Una vez alcanzada esta cantidad, el sistema dejará de emitir hasta que se eliminen algunas de las partículas antiguas.

Los sistemas de partículas pueden rápidamente convertirse en un cuello de botella que requiere mucho tiempo de cómputo. Es importante dedicar el tiempo suficiente a optimizarlos, por el buen rendimiento de la aplicación.

A continuación estudiaremos la implementación de la clase *ParticleSource*, distinguiendo entre el fragmento de código de la creación y de la actualización.

Los *ParticleSource* pueden ser de dos tipos, definidos en un tipo de datos enumerado *PType* (ver líneas 1-4). Cada partícula se mantiene en un array de forma independiente, en la lista de partículas *_pList* (línea 10). En el caso de definir un sistema de partículas lineal, es necesario controlar el tiempo transcurrido entre el lanzamiento de cada partícula (variable *_time* en la línea 11), y el número de partículas que han sido lanzadas *_spart* con respecto del total a lanzar *_npart*. A continuación veremos algunos de los métodos pricipales de esta clase.

Listado 3.33: Fragmento de ParticleSource.hx.

```
 1 enum PType {    // Tipos de Partículas soportados
 2   Linear;
 3   Radial;
 4 }
 5 class ParticleSource {
 6   public var _x:Float;      // Posición x,y de la fuente
 7   public var _y:Float;
 8   var _root:TileLayer;      // Capa principal de dibujado
 9   var _id:String;           // Grafico para la particula
10   var _pList:Array<TileSprite>;  // Graficos (1 por particula)
11   var _time:Float;          // Tiempo desde la ultima particula
12   var _npart:Int;           // Numero de particulas a lanzar
13   var _spart:Int;           // Numero de particulas lanzadas
14   var _interval:Float;      // Intervalo de tiempo entre particulas
15   var _type:PType;          // Tipo del sistema de particulas
16   // Constructor ==================================================
17   public function new(id:String, layer:TileLayer,
18               posX:Float, posY:Float, type:PType) {
19     _id = id; _x = posX; _y = posY; _root = layer; _type = type;
20     _pList = new Array<TileSprite>();
21
22     _time = 0; _npart = 0; _spart = 0;  _interval = 0;
23   }
24   // getNParticles ================================================
25   public function getNParticles():Int { return _pList.length; }
26   // Eliminar todas las partículas ================================
27   public function clearParticles() {
28     for (p in _pList) _root.removeChild(p);
29     _pList.splice(0,_pList.length);  }
30
```

```
31   // Añadir partículas ==========================================
32   function addIndividualParticle():Void {
33     _time = 0;
34     var p = new TileSprite(_root, _id);
35     p.alpha = 0.5; p.scale = 0.25;
36     p.x = _x; p.y = _y;
37     _pList.push(p);
38     _root.addChild(p);
39     _spart++;
40   }
41   public function addParticles(nparticles: Int, interval: Int = 0) {
42     clearParticles();
43     _time=0; _spart=0; _npart = nparticles; _interval = interval;
44     switch (_type) {
45     case Linear: addIndividualParticle();  // De una en una
46     case Radial:  // Si es radial se añaden todas a la vez
47       for (i in 0 ... _npart) addIndividualParticle();
48     }
49   }
50   public function setPosition(posX:Float, posY:Float):Void {
51     _x = posX; _y = posY; }
```

Cada partícula se maneja individualmente. En el caso del tipo de fuente *Linear* resulta especialmente relevante, ya que hay que dejar transcurrir un determinado tiempo entre la creación de cada partícula individual (línea 45). La creación de un tipo de sistema *Radial* (línea 46–47) implica que todas las partículas se añadan en el mismo instante.

El método privado *addIndividualParticles* (líneas 32–40) se encarga de crear cada una de las partículas de la fuente. La implementación actual no distingue entre la creación de una partícula de tipo Lineal o Radial, especificando en cualquier caso los mismos valores inciales de tamaño, posición y transparencia (líneas 35–36).

La función de actualización *update* (líneas 16–23) sí tiene en cuenta el tipo de la fuente de partículas. En el caso de una fuente *Linear*, si hay que añadir más partículas y se ha superado el tiempo de lanzamiento entre partículas, se llama al método estudiado anteriormente *addInvidualParticle*.

El método privado *updateParticleList* se encarga de actualizar la posición de cada partícula de la lista. En el caso de ser una fuente *Linear* (líneas 5–6) únicamente actualizamos el tamaño y el valor de transparencia (la partícula se mantiene en la misma posición). En el caso de ser una fuente *Radial* (líneas 7–9) se actualiza además la posición x, y con una nueva posición aleatoria.

El criterio empleado para eliminar una partícula de la lista es el valor de transparencia *Alpha*. Si el valor es positivo, seguimos actualizando la partícula. En el caso de llegar a un valor cero o negativo, la partícula se elimina de la lista (línea 12).

Listado 3.34: Fragmento de ParticleSource.hx (Update).

```
function updateParticleList():Void {
  for (p in _pList) {
    if (p.alpha > 0) {
      switch (_type) {
      case Linear:
        p.alpha -= .01;  p.scale *= 1.04;
      case Radial:
        p.x += 6 - Std.random(12); p.y += 6 - Std.random(12);
        p.alpha -= .02;  p.scale *= 1.04;
      }
    }
    else { _root.removeChild(p); _pList.remove(p);  }
  }
}

public function update(w:Int, h:Int, eTime:Int):Void {
  _time += eTime;
  if ((_type == Linear) && (_time > _interval)
      && (_spart < _npart)) {   // + partículas?
    addIndividualParticle();
  }
  updateParticleList();
}
```

3.3. Gestión de sonido

El sonido es un **aspecto fundamental** en el contexto del desarrollo de videojuegos. Piense en su juego favorito y, a continuación, siléncio lo y continúe jugando durante un momento. En términos generales, usted sentirá que el juego está incompleto y perderá gran parte de su esencia. Esto se debe a que la música y los efectos de sonido son compañeros indiscutibles de la mecánica del juego. Además, facilitan enormemente la sensación de inmersión en el mismo.

En función del género al cual pertenezca un juego, la música y los efectos de sonido cobrarán más o menos relevancia. Por ejemplo, en un juego de terror la música es esencial para transmitir la sensación de terror al jugador. Por otra parte, en un *shooter* los efectos de sonido, además de servir para proporcionar una verdadera inmersión, ayudan al jugador a determinar la dirección de la que provienen los disparos.

En esta sección se discute la gestión básica de sonido haciendo uso del framework NME. Básicamente, esta gestión consistirá en reproducir tanto **música en segundo plano** como **efectos de sonido** puntuales. En este segundo caso, resulta esencial que los efectos de sonido se reproduzcan de manera simultánea a la música principal con el objetivo de garantizar la inmersión del jugador en el videojuego.

3.3.1. NME y su soporte de sonido

El soporte de sonido proporcionado por NME, incluido en el paquete *nme.media*, tiene como núcleo 5 clases definidas en Haxe:

- **Sound**[15], clase que permite trabajar con sonido en una aplicación.
- **SoundChannel**[16], clase que controla un sonido dentro de una aplicación.
- **SoundTransform**[17], clase que permite la gestión de propiedades esenciales, como el volumen o el balance.
- **SoundLoaderContext**[18], clase que proporciona controles de seguridad para archivos que cargan sonido.
- **ID3Info**[19], clase que encapsula metadatos asociados a una canción, como su título o su autor.

En este punto, resulta importante considerar las restricciones asociadas al *target* final sobre el cual se desplegará el videojuego desarrollado con NME. Por ejemplo, si se compila el juego para *Flash*, entonces habrá que tener en cuenta la frecuencia de muestreo de los archivos de audio que se utilizarán en el juego[20]. Por otra parte, plataformas como Android pueden establecer restricciones en el número de canales de audio a utilizar por la aplicación. Típicamente, este tipo de sistema soporta al menos dos canales. Uno de ellos permite reproducir la música en segundo plano, mientras que el segundo permite reproducir un número elevado de efectos de sonido puntuales.

3.3.2. La clase SoundManager

Con el objetivo de gestionar los recursos de sonido en videojuegos desarrollados con NME, en esta sección se discute la **clase *SoundManager***[21], la cual ha sido implementada desde cero para llevar a cabo la gestión de sonido. Esta clase, que se puede utilizar directamente para simplificar la integración de sonido con NME, proporciona la siguiente **funcionalidad**:

[15] http://www.haxenme.org/api/types/nme/media/Sound.html
[16] http://www.haxenme.org/api/types/nme/media/SoundChannel.html
[17] http://www.haxenme.org/api/types/nme/media/SoundTransform.html
[18] http://www.haxenme.org/api/types/nme/media/SoundLoaderContext.html
[19] http://www.haxenme.org/api/types/nme/media/ID3Info.html
[20] Flash soporta archivos WAV de 5512, 11025, 22050 y 4410 Hz. Otras calidades pueden causar problemas
[21] La clase *SoundManager* se puede interpretar como una fachada de las clases de NME que dan soporte al sonido.

- Carga y reproducción de música en segundo plano.
- Reproducción de efectos de sonido.
- Control básico de volumen.
- Control básico de balance.

Antes de pasar a discutir la implementación que da soporte a esta funcionalidad, resulta interesante destacar que la clase *SoundManager* implementa el **patrón *Singleton***. De este modo, se garantiza que sólo existirá una instancia de dicha clase, centralizando así la gestión de recursos de sonido. A continuación, el siguiente listado muestra el estado de la clase *SoundManager*, es decir, las variables miembro que conforman su parte de datos.

Listado 3.35: Clase SoundManager. Variables miembro.

```
1  // Clase encargada de la gestión del sonido.
2  class SoundManager {
3
4    // Variable estática para implementar Singleton.
5    public static var _instance:SoundManager;
6
7    private var _backgroundMusic:Sound; // Música de fondo.
8    private var _channel:SoundChannel;  // Canal de la música de fondo.
9    private var _volume:Float;          // Nivel de volumen.
10   private var _balance:Float;         // Nivel de balance.
11
12   // Más código aquí...
13
14 }
```

Note cómo en la línea 5 se declara la variable estática *_instance* de tipo *SoundManager*, la cual representará la única instancia existente de dicha clase (accesible con la función *getInstance()*). Por otra parte, en las líneas 7-8 se declaran las variables miembro *_backgroundMusic* y *_channel*, las cuales representan, respectivamente, los recursos asociados a la música de fondo y al canal de dicha música. La primera de ellas es del tipo *nme.media.Sound*, mientras que la segunda es del tipo *nme.media.SoundChannel*. Aunque se discutirá más adelante, la variable *_channel* es la que permite controlar las propiedades del sonido que se reproduce. Por el contrario, *_backgroundMusic* gestiona funcionalidad de más alto nivel, como la carga y reproducción de un *track* de audio.

Las otras dos variables de clase son *_volume* y *_balance*. La primera representa el valor actual del volumen del juego mientras que la segunda hace referencia al balance, es decir, a la distribución de sonido estéreo.

El **constructor de *SoundManager*** inicializa todas estas variables miembro, tal y como se muestra en el siguiente listado de código.

Recuerde que el constructor de una clase ha de inicializar todo el estado de las instancias creadas a partir de dicha clase.

Listado 3.36: Clase SoundManager. Constructor.

```
1 private function new () {
2   _backgroundMusic = null;
3   _channel = null;
4   _volume = 1.0;
5   _balance = 0.0;
6 }
```

Al igual que se discutió en la sección 3.1.4 relativa a la clase *GameManager* en el bucle de juego, el constructor de *SoundManager* tiene una visibilidad privada para evitar la creación de instancias desde fuera de dicha clase.

Note cómo las variables *_volume* y *_balance* se inicializan, respectivamente, a los valores por defecto 1.0 (máximo volumen) y 0.0 (balance centrado) en las líneas 4–5. Sin embargo, *_backgroundMusic* y *_channel* se inicializan a *null*. El lector podría suponer que, idealmente, el constructor debería cargar directamente un *track* de audio como música en segundo plano e inicializar también el canal asociado al mismo para que la música comenzara a reproducirse. Sin embargo, se ha optado por delegar esta funcionalidad en las funciones *loadBackgroundMusic()* y *playBackgroundMusic()*, facilitando así la carga cuando realmente sea necesaria y posibilitando el cambio de la música en segundo plano cuando así lo requiera el juego.

Gestión de música en segundo plano

El siguiente listado muestra la implementación de las funciones *loadBackgroundMusic()* y *playBackgroundMusic()* (líneas 1–3 y 5–11) y de la función *stopBackgroundMusic()* (líneas 13–20). La función *loadBackgroundMusic()* es trivial, ya que delega en la función *getSound()* de la clase *Assets* de NME. Básicamente, esta función permite obtener una instancia de un sonido empotrado a partir de la ruta pasada como argumento (línea 2). La clase *Assets*[22] de NME es especialmente relevante, ya que proporciona una interfaz multi-plataforma para acceder a imágenes, fuentes, sonidos y otros tipos de recursos.

[22] http://www.haxenme.org/api/types/nme/Assets.html

Listado 3.37: Clase SoundManager. Funciones de gestión de sonido.

```
1  public function loadBackgroundMusic (url:String) : Void {
2    _backgroundMusic = nme.Assets.getSound(url);
3  }
4
5  public function playBackgroundMusic () : Void {
6    // Reproducción de la música de fondo.
7    _channel = _backgroundMusic.play();
8    // Retrollamada para efecto loop.
9    _channel.addEventListener(Event.SOUND_COMPLETE,
10                  channel_onSoundComplete);
11 }
12
13 public function stopBackgroundMusic () : Void {
14   if (_channel != null) {
15     _channel.removeEventListener(Event.SOUND_COMPLETE,
16                   channel_onSoundComplete);
17     _channel.stop();
18     _channel = null;
19   }
20 }
```

Por otra parte, la función *playBackgroundMusic()* de la clase *SoundManager* es la que realmente efectúa la reproducción de música en segundo plano. Para ello, simplemente utiliza la **función *play()*** (línea 7) de la clase *Sound* de NME, la cual devuelve un objeto de tipo *SoundChannel* que se almacena en la variable miembro *_channel*.

Esta última función *play()* tiene la siguiente declaración:

Listado 3.38: Función play() de nme.media.Sound.

```
1 function play(startTime : Float = 0,
2           loops : Int = 0,
3           ?sndTransform : SoundTransform) : SoundChannel;
```

El primer parámetro, *startTime*, representa la posición inicial en milisegundos en el que la música debe comenzar y tiene un valor por defecto de 0. El segundo parámetro, *loops*, define el número de veces que la música en segundo plano se repetirá y tiene un valor por defecto de 0. Finalmente, *sndTransform* representa un objeto de la clase *SoundTransform* utilizado para controlar el sonido. Este último parámetro es opcional, y así se denota en Haxe haciendo uso del símbolo de interrogación delante del parámetro. Al no pasar argumentos a la función *play()*, ésta usa los valores por defecto para gestionar la reproducción de música en segundo plano.

Por otra parte, y con el objetivo de garantizar que la canción principal vuelva a reproducirse desde el principio, a este canal se le asocia una función de retrollamada *channel_onSoundComplete()* que se ejecu-

Figura 3.17: Algunos juegos se basan en que el jugador asuma el rol de un guitarrista, de manera que el componente musical es el punto principal del juego. La imagen muestra una captura de pantalla de *Frets on Fire*, un clon Open Source del famosísimo Guitar Hero™.

tará cuando dicha canción termine. Este hecho se modela con el evento *Event.SOUND_COMPLETE*. La asociación entre evento y función de retrollamada se hace efectiva, como en otras ocasiones ya discutidas, mediante la función *addEventListener()*.

El siguiente listado muestra la implementación de la función *channel_onSoundComplete()* en la clase *SoundManager*. Note cómo simplemente se vuelve a llamar a la función *playBackgroundMusic()* para reproducir de nuevo la canción principal.

Listado 3.39: Función channel_onSoundComplete().

```
1 private function channel_onSoundComplete (event:Event) {
2   playBackgroundMusic();
3 }
```

Finalmente, la función *stopBackgroundMusic()* (líneas 13-20) elimina el *event listener* y para la reproducción de la música en segundo plano.

Gestión de efectos de sonido

Los efectos de sonido, al contrario que la música en segundo plano, se reproducen de manera puntual. En otras palabras, su duración está muy bien acotada y suelen estar ligados a **situaciones concretas en un juego**. Por ejemplo, la detección de una colisión en un juego de conducción generará un efecto de sonido que recreará dicha colisión.

La clase *SoundManager* proporciona esta funcionalidad a través de la función *playSoundEffect()*, la cual se muestra en el siguiente listado de código.

Listado 3.40: Función playSoundEffect().

```
1 public function playSoundEffect (url:String) : Void {
2   var soundFX = nme.Assets.getSound(url);
3   // Sound Transform para conservar volumen y balance.
4   soundFX.play(0, 0, new SoundTransform(_volume, _balance));
5 }
```

Básicamente, la filosofía es muy similar a la reproducción de sonido en segundo plano pero, al tratarse de un evento de sonido puntual, la carga del mismo y la reproducción se realizan en una única función. Esta función tiene como parámetro la ruta al archivo de audio que contiene el efecto de sonido (línea 1). En la línea 2 se obtiene la instancia de la clase *Sound* para, posteriormente, reproducirla con la función *play()* en la línea 4.

En el caso particular de la clase *SoundManager*, desarrollada explícitamente para la gestión de sonido, los parámetros de la función *play()*, en el caso de los efectos de sonido, reproducen siempre los efectos desde el principio, sin repetición y definiendo un objeto de la clase *SoundTransform* que recupere los valores actuales de volumen y balance. La línea 4 muestra cómo se hace uso de las variables miembro *_volume* y *_balance* para tal fin.

Control de sonido

El control de sonido planteado en *SoundManager* es sencillo y proporciona la siguiente funcionalidad básica:

- Establecer un volumen concreto (función *setVolume()*).
- Subir y bajar el volumen de manera incremental (funciones *turnVolumeUp()* y *turnVolumeDown()*).
- Quitar y reanudar el volumen (funciones *mute()* y *unmute()*).

El siguiente listado de código muestra la implementación de las funciones que gestionan el control de sonido en la clase *SoundManager*.

Listado 3.41: Clase SoundManager. Control de sonido.

```
1  public function setVolume (volume:Float) : Void {
2    _volume = volume > 1 ? 1 : volume;
3    if (_channel != null) {
4      _channel.soundTransform = new SoundTransform(_volume, _balance);
5    }
6  }
7
8  public function getVolume () : Float {
9    return _volume;
10 }
11
12 public function turnVolumeUp (delta:Float = 0.1) : Void {
13   setVolume(_volume + delta);
14 }
15
16 public function turnVolumeDown (delta:Float = 0.1) : Void {
```

```
17   setVolume(_volume - delta);
18 }
19
20 public function mute () : Void {
21   setVolume(0.0);
22   _channel.removeEventListener(Event.SOUND_COMPLETE,
23                     channel_onSoundComplete);
24 }
25
26 public function unmute () : Void {
27   setVolume(1.0);
28   _channel.addEventListener(Event.SOUND_COMPLETE,
29                  channel_onSoundComplete);
30 }
31
32 public function isMuted () : Bool {
33   return _volume == 0.0;
34 }
```

Las funciones son sencillas y **abstraen al programador** que las utilice de la interacción con NME. Por ejemplo, la función *setVolume()* es especialmente relevante (líneas 1-6), ya que se encarga de controlar que el volume pasado como parámetro no excede el máximo gestionado internamente por NME (1.0). Posteriormente, es necesario modificar la instancia de tipo *Sound Transform* para establecer de manera efectiva el nuevo volumen, manteniendo en balance actual (línea 4).

Control de balance

La implementación del control de balance, integrada también en la clase *SoundManager*, es prácticamente idéntica a la del control de sonido y se muestra en el siguiente listado de código. En este caso, estas funciones proporcionan la siguiente funcionalidad básica:

- Establecer un balance concreto (función *setBalance()*; líneas 1-6).
- Modificar el balance (funciones *increaseRightBalance()* e *increaseLeftBalance()*; líneas 12-15 y 17-20, respectivamente).

Listado 3.42: Clase SoundManager. Control de balance.

```
1 public function setBalance (balance:Float) : Void {
2   _balance = balance;
3   if (_channel != null) {
4     _channel.soundTransform = new SoundTransform(_volume, _balance);
5   }
6 }
7
8 public function getBalance () : Float {
```

```
 9   return _balance;
10 }
11
12 public function increaseRightBalance (delta:Float = 0.1) : Void {
13   if (_balance + delta <= 1.0)
14     setBalance(_balance + delta);
15 }
16
17 public function increaseLeftBalance (delta:Float = 0.1) : Void {
18   if (_balance - delta >= -1.0)
19     setBalance(_balance - delta);
20 }
```

3.3.3. Integrando sonido en Bee Adventures

Una vez discutida la implementación de la clase *SoundManager*, en esta sección se discute cómo utilizarla para integrar sonido en el juego *Bee Adventures*, discutido previamente en la sección 3.2.7.

Básicamente, esta integración consiste en dar soporte a los siguientes **requisitos de sonido**:

- Reproducción de música en segundo plano.
- Reproducción de efectos de sonido cuando se produzcan determinados eventos.
- Control básico de volumen y de balance.

A continuación se estudia cómo se ha modificado el código fuente original de *Bee Adventures* para dar soporte a la funcionalidad necesaria para cumplir con los requisitos anteriores. Este estudio permitirá al lector incluir fácilmente música y efectos de sonido en futuros desarrollos con NME.

Reproducción de música en segundo plano

Para reproducir música en segundo plano, tan sólo es necesario cargar el recurso de sonido asociado y ejecutar *play()* sobre el mismo después de una carga correcta. Desde el punto de vista de la implementación, en la clase *BeeGame* se ha incluido una **función *loadMusic()*** que se invoca desde la función *init()* en la línea 9. El siguiente listado muestra el código necesario para llevar a cabo la carga y reproducción de música.

Listado 3.43: Clase BeeGame. Carga de sonido.

```
1  class BeeGame extends Sprite {
2
3    // Variables miembro y constructor...
4
5    function init():Void {
6      createScene();
7      addListeners();
8      _previousTime = Lib.getTimer();
9      loadMusic();
10   }
11
12   // ...
13
14   private function loadMusic () {
15     SoundManager.getInstance().
16       loadBackgroundMusic("assets/stars.mp3");
17     SoundManager.getInstance().playBackgroundMusic();
18   }
19
20 }
```

Note que la reproducción de música en segundo plano es una cuestión general, por lo que su carga y reproducción se ha realizado desde una clase general, como es el caso de *BeeGame*. Así mismo, resulta interesante recordar que la reproducción del *track* principal volverá a reanudarse cuando finalice (y así sucesivamente) hasta que el jugador salga del juego.

Reproducción de efectos de sonido

Los efectos de sonido en *Bee Adventures*, al contrario de lo que ocurre con la música en segundo plano, están asociados a **eventos puntuales del juego**. Por lo tanto, su reproducción también es puntual. En concreto, se han integrado cuatro efectos de sonido en *Bee Adventures*:

- Cuando la abeja inicia un desplazamiento hacia adelante, el juego reproduce un sonido simulando el esfuerzo de dicho desplazamiento.
- Cuando la abeja choca con un enemigo, el juego reproduce un sonido de explosión.
- Cuando la abeja recoge una estrella, el juego reproduce un sonido asociado a la recolección de la misma.
- Cuando la puntuación alcanza un valor numérico que sea un múltiplo de 5, el juego reproduce un sonido que simula los aplausos de un grupo de personas.

En primer lugar se discute el **desplazamiento hacia adelante de la abeja**. Debido a que es un evento propio de la abeja, se ha optado por incluir la lógica necesaria para gestionarlo en la propia clase *Player*, tal y como muestra el siguiente listado de código. Simplemente es necesario asociar la reproducción del efecto, mediante la función *playSoundEffect()* en la línea 8, cuando se detecta una aceleración del personaje (línea 5) y el estado anterior era el de moverse hacia atrás (línea 7).

Listado 3.44: Clase Player. Efecto de desplazamiento.

```
1  function updateFromMousePosition() : Void {
2
3    // ...
4
5    if (v.x > 0) {
6      // Efecto de sonido al cambiar de estado.
7      if (_cState == Sback) {
8        SoundManager.getInstance().playSoundEffect("assets/blip.wav");
9      }
10     _cState = Sforward;
11   }
12
13   // ...
14
15 }
```

En segundo lugar se discute la **interacción de la abeja**, desde el punto de vista del sonido, con respecto a algún otro elemento móvil. En este caso, el elemento móvil puede ser una estrella o un enemigo. No obstante, la lógica requerida es idéntica y sólo es necesario cambiar el nombre del archivo que contiene el efecto de sonido (ver líneas 14–19). El siguiente listado de código muestra cómo se ha modificado la función *checkCollisionArray()* de la clase *Player* para dar soporte a dicha funcionalidad.

Finalmente, el último efecto de sonido añadido consiste en reproducir a un **grupo de personas aplaudiendo** cuando la puntuación alcanza un valor numérico que sea un múltiplo de 5. Este efecto, además de incrementar la inmersión en el juego, puede *animar* al jugador a seguir recolectando estrellas.

En este caso, el evento de sonido está asociado a un evento concreto de la puntuación. Debido a que la gestión de la puntuación está encapsulada en la clase *ScoreBoard*, la reproducción del efecto de sonido asociado se ha integrado en la función que actualiza el marcador.

Listado 3.45: Clase Player. Efectos de sonido.

```
1  function checkCollisionArray (v:Array<NonPlayer>) : Int {
2
3    var i = 0;
4    for (e in v) {
5      if (checkCollision(e)) {     // Si hay choque
6        v.remove(e);               // Eliminamos el objeto
7        _root.removeChild(e);
8        // Si el objeto es un enemigo, creamos partículas de humo
9        var straux = Type.getClassName(Type.getClass(e));
10
11       // ...
12
13       // Añadimos un efecto de sonido.
14       if (straux.indexOf("Enemy") > 0) {
15     SoundManager.getInstance().playSoundEffect("assets/bomb.wav");
16       }
17       if (straux.indexOf("Prize") > 0) {
18     SoundManager.getInstance().playSoundEffect("assets/coin.wav");
19       }
20     }
21   }
22
23   // ...
24
25 }
```

El código del siguiente listado muestra la nueva implementación de la función *update()* de la clase *ScoreBoard*. Note cómo dicho código controla que si la puntuación alcanza un valor igual al previamente definido (por ejemplo 5) o un múltiple de éste (por ejemplo 10, 15 ó 20), entonces se reproduce el efecto de sonido.

Listado 3.46: Clase Scoreboard. Efecto de sonido (aplausos).

```
1  public function update(delta:Int = 0):Void {
2    _score += delta;
3    htmlText = Std.string(_score);
4
5    // Llega al siguiente incremento?
6    if ((_score % _applausesScore == 0) && (_score > 0)) {
7      // Se aplaudió antes con el valor actual?
8      if (_applausesOn) {
9        SoundManager.getInstance().
10     playSoundEffect("assets/applause.wav");
11       _applausesOn = false;
12     }
13   }
14   else
15     _applausesOn = true;
16 }
```

Control de volumen y balance

Por último, la funcionalidad desde el punto de vista de la gestión del sonido de *Bee Adventures* ha sido extendida incluyendo un control básico de volumen y balance. Ahora es posible silenciar (*mute*) y reanudar el sonido (*unmute*) fácilmente de dos formas distintas: i) utilizando la tecla m o ii) interactuando con el ratón sobre el icono del altavoz que se encuentra en la parte superior izquierda de la pantalla.

El siguiente listado de código muestra la asociación de eventos de teclado y la interacción con *SoundManager* mediante la función de retrollamada *onKeyPressed()*.

Listado 3.47: Clase BeeGame. Función *onKeyPressed()*.

```
1  function onKeyPressed(event:KeyboardEvent):Void {
2
3    switch (event.keyCode) {
4
5    case Keyboard.DOWN:
6      SoundManager.getInstance().turnVolumeDown();
7
8    case Keyboard.UP:
9      SoundManager.getInstance().turnVolumeUp();
10
11   case Keyboard.RIGHT:
12     SoundManager.getInstance().increaseRightBalance();
13
14   case Keyboard.LEFT:
15     SoundManager.getInstance().increaseLeftBalance();
16
17   case 109: // Tecla 'm'
18     mute_unmute();
19
20   default:
21     _actorManager.onKeyPressed(event);
22
23   }
24
25 }
```

Note cómo la opción de silenciar o reanudar el sonido se delega en la **función *mute_unmute()***, la cual tiene una visibilidad privada en la clase *BeeGame*. Esta función también hace visible o invisible, en función del estado anterior, el *sprite* asociado al icono del altavoz activo o silenciado, respectivamente.

Listado 3.48: Clase BeeGame. Función *mute_unmute()*.

```
1  // Comprueba si el sonido estaba activado o no,
2  // y activa/desactiva overlays de iconos de sonido.
3
4  private function mute_unmute () : Void {
5
6    if (SoundManager.getInstance().isMuted()) {
7      _muted_speaker.visible = false;
8      _speaker.visible = true;
9      SoundManager.getInstance().unmute();
10   }
11   else {
12     _speaker.visible = false;
13     _muted_speaker.visible = true;
14     SoundManager.getInstance().mute();
15   }
16
17 }
```

Finalmente, también es interesante destacar que es posible silenciar el sonido con el ratón, haciendo *click* sobre el icono del altavoz situado en la parte superior izquierda del juego. En este caso, se ha implementado la función de retrollamada *onMouseClicked()* en la clase *Player*, tal y como se muestra en el siguiente listado.

Listado 3.49: Clase BeeGame. Función *onMouseClicked()*.

```
1  // Para activar/desactivar el sonido.
2
3  function onMouseClicked (event:MouseEvent):Void {
4
5    // Delega en Rectangle.
6    var rectangle:Rectangle =
7      new Rectangle(_speaker.x - _speaker.width / 2,
8            _speaker.y - _speaker.height / 2,
9            _speaker.width,
10           _speaker.height);
11
12   if (rectangle.contains(event.localX, event.localY)) {
13     mute_unmute();
14   }
15
16 }
```

Para detectar si el usuario hace *click* con el ratón sobre dicho icono, se ha delegado dicha detección en la clase *Rectangle* de NME, la cual implementa la función *contains()*. Esta función permite comprobar fácilmente si un punto está o no dentro del rectángulo que *envuelve* al *sprite*.

3.4. Simulación Física

En prácticamente cualquier videojuego (tanto 2D como 3D) es necesaria la detección de colisiones y, en muchos casos, la simulación realista de dinámica de cuerpo rígido. En esta sección estudiaremos la relación existente entre sistemas de detección de colisiones y sistemas de simulación física, y veremos algunos ejemplos de uso del motor de simulación física *Physaxe*.

La mayoría de los videojuegos requieren en mayor o menor medida el uso de técnicas de detección de colisiones y simulación física. Desde un videojuego clásico como *Arkanoid*, hasta modernos juegos automovilísticos como *Gran Turismo* requieren definir la interacción de los elementos en el mundo físico.

Definimos un **cuerpo rígido** como un objeto sólido ideal, infinitamente duro y no deformable.

El motor de *simulación física* puede abarcar una amplia gama de características y funcionalidades, aunque la mayor parte de las veces el término se refiere a un tipo concreto de simulación de la **dinámica de cuerpos rígidos**. Esta dinámica se encarga de determinar el movimiento de estos cuerpos rígidos y su interacción ante la influencia de fuerzas.

En el mundo real, los objetos no pueden pasar a través de otros objetos [23]. En nuestro videojuego, a menos que tengamos en cuenta las colisiones de los cuerpos, tendremos el mismo efecto. El *sistema de detección de colisiones*, que habitualmente es un módulo del motor de simulación física, se encarga de calcular estas relaciones, determinando la relación espacial existente entre cuerpos rígidos.

La mayor parte de los videojuegos actuales incorporan ciertos elementos de simulación física básicos. Algunos títulos se animan a incorporar ciertos elementos complejos como simulación de telas, cuerdas, pelo o fluidos. Algunos elementos de simulación física son precalculados y almacenados como animaciones estáticas (y pueden desplegarse en videojuegos 2D como sprites animados), mientras que otros necesitan ser calculados en tiempo real para conseguir una integración adecuada.

Como hemos indicado anteriormente, las tres tareas principales que deben estar soportadas por un motor de simulación física son la detección de colisiones, su resolución (junto con otras restricciones de los

[23] salvo casos específicos convenientemente documentados en la revista *Más Allá*

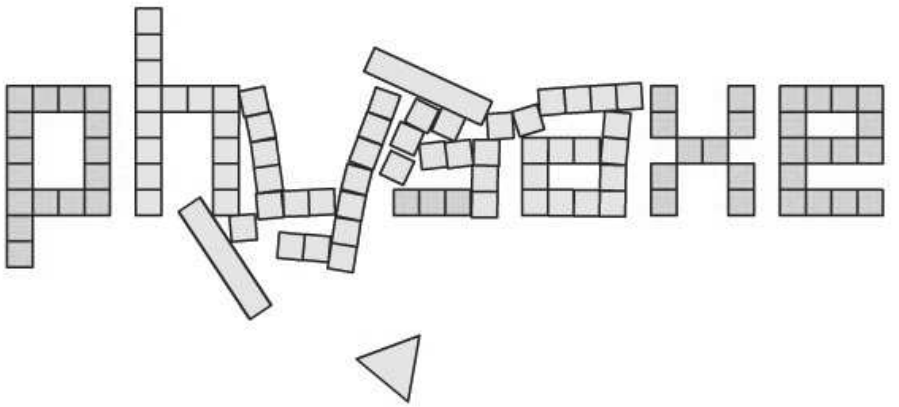


Figura 3.18: Ejemplo de mundo físico dinámico creado con Physaxe. El logotipo de la biblioteca se destruye mediante diversas formas convexas dinámicas.

objetos) y calcular la actualización del mundo tras esas interacciones. De forma general, las características que suelen estar presentes en motores de simulación física son:

- Detección de colisiones entre objetos dinámicos de la escena. Esta detección podrá ser utilizada posteriormente por el módulo de simulación dinámica.
- Cálculo de lineas de visión y tiro parabólico, para la simulación del lanzamiento de proyectiles en el juego.
- Definición de geometría estática de la escena (cuerpos de colisión) que formen el escenario del videojuego. Este tipo de geometría puede ser más compleja que la geometría de cuerpos dinámicos.
- Especificación de fuerzas (tales como viento, rozamiento, gravedad, etc...), que añadirán realismo al videjuego.
- Simulación de destrucción de objetos: paredes y objetos del escenario.
- Definición de diversos tipos de articulaciones, tanto en elementos del escenario (bisagras en puertas, raíles...) como en la descripción de las articulaciones de personajes.
- Especificación de diversos tipos de motores y elementos generadores de fuerzas, así como simulación de elementos de suspensión y muelles.

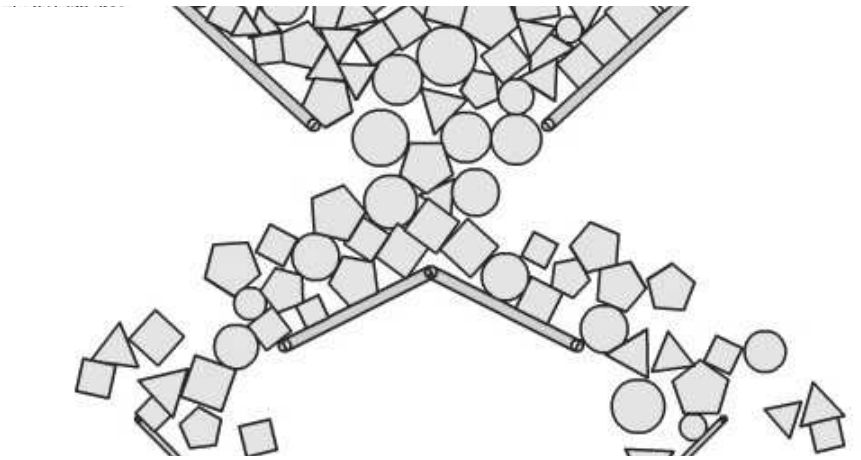


Figura 3.19: La utilización de formas estáticas permiten definir fácilmente obstáculos en el mundo sobre el que interaccionarán los elementos dinámicos. Esta imagen forma parte de los ejemplos oficiales de Physaxe.

3.4.1. Algunos Motores de Simulación

El desarrollo de un motor de simulación física desde cero es una tarea compleja y que requiere gran cantidad de tiempo. Afortunadamente existen gran variedad de motores de simulación física muy robustos, tanto basados en licencias libres como comerciales. A continuación se describirán brevemente algunas de las bibliotecas más utilizadas:

- **Bullet**. Bullet es una biblioteca de simulación física ampliamente utilizada tanto en la industria del videojuego como en la síntesis de imagen realista (Blender, Houdini, Cinema 4D y LightWave las utilizan internamente). Bullet es multiplataforma, y se distribuye bajo una licencia libre zlib compatible con GPL.

- **ODE**. ODE (*Open Dynamics Engine*) `www.ode.org` es un motor de simulación física desarrollado en C++ bajo doble licencias BSD y LGPL. El desarrollo de ODE comenzó en el 2001, y ha sido utilizado como motor de simulación física en multitud de éxitos mundiales, como el aclamado videojuego multiplataforma *World of Goo*, *BloodRayne 2* (PlayStation 2 y Xbox), y *TitanQuest* (Windows).

- **PhysX**. Este motor privativo, es actualmente mantenido por NVidia con aceleración basada en hardware (mediante unidades específicas de procesamiento físico PPUs *Physics Processing Units* o mediante núcleos CUDA. Las tarjetas gráficas con soporte de CUDA

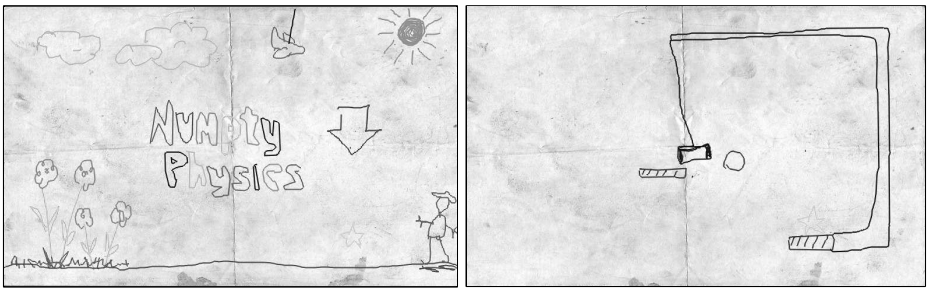


Figura 3.20: Capturas de pantalla de *Numpty Physics*, un clon libre del videojuego *Crayon Physics Deluxe*, igualmente basado en Box2D, cuyo objetivo es dirigir la bola al destino dibujando elementos de interacción física sobre la pantalla.

(siempre que tengan al menos 32 núcleos CUDA) pueden realizar la simulación física en GPU. Este motor puede ejecutarse en multitud de plataformas como PC (GNU/Linux, Windows y Mac), PlayStation 3, Xbox y Wii. El SDK es gratuito, tanto para proyectos comerciales como no comerciales. Existen multitud de videojuegos comerciales que utilizan este motor de simulación.

- **Havok**. El motor Havok se ha convertido en el estándar de facto en el mundo del software privativo, con una amplia gama de características soportadas y plataformas de publicación (PC, Videoconsolas y Smartphones). Desde que en 2007 Intel comprara la compañía que originalmente lo desarrolló, Havok ha sido el sistema elegido por más de 150 videojuegos comerciales de primera línea. Títulos como *Age of Empires*, *Killzone 2 & 3*, *Portal 2* o *Uncharted 3* avalan la calidad del motor.

- **Box2D**. Es un motor de simulación física 2D[24] libre que utilizaremos en este curso. Ha sido el corazón del fenómeno multiplataforma *Angry Birds*, que supuso más de 5 millones de ingresos a los propietarios sólo en concepto de publicidad. Entre otros títulos comerciales se encuentran *Crayon Physics Deluxe* (ver Figura 3.20), *Indredibots* y *Tiny Wings* (que se convirtió en la aplicación más vendidad del App Stores en Febrero de 2011).

Existen algunas bibliotecas específicas para el cálculo de colisiones (la mayoría distribuidas bajo licencias libres). Por ejemplo, *I-Collide*, desarrollada en la Universidad de Carolina del Norte permite calcular

[24]Web oficial en: http://box2d.org/

intersecciones entre volúmenes convexos. Existen versiones menos eficientes para el tratamiento de formas no convexas, llamadas *V-Collide* y *RAPID*.

Estas bibliotecas pueden utilizarse como base para la construcción de nuestro propio conjunto de funciones de colisión para videojuegos que no requieran funcionalidades físicas complejas.

La biblioteca **Physaxe** que utilizaremos en esta sección es, en realidad, una capa de recubrimiento sobre Box2D. La funcionalidad de Box2D está directamente accesible a través de los objetos de bajo nivel de la biblioteca.

3.4.2. Aspectos destacables

El uso de un motor de simulación física en el desarrollo de un videojuego conlleva una serie de aspectos que deben tenerse en cuenta relativos al diseño del juego, tanto a nivel de jugabilidad como de módulos arquitectónicos:

- **Predictibilidad**. El uso de un motor de simulación física afecta a la predictibilidad del comportamiento de sus elementos. Además, el ajuste de los parámetros relativos a la definición de las características físicas de los objetos (coeficientes, constantes, etc...) son difíciles de visualizar.
- **Realización de pruebas**. La propia naturaleza caótica de las simulaciones (en muchos casos no determinista) dificulta la realización de pruebas en el videojuego.
- **Integración**. La integración con otros módulos del juego puede ser compleja. Por ejemplo, ¿qué impacto tendrá en la búsqueda de caminos de Inteligencia Artificial el uso de simulaciones físicas? ¿cómo garantizar el determinismo en un videojuego multijugador?.
- **Realismo gráfico**. El uso de un motor de simulación puede dificultar el uso de ciertas técnicas de representación realista (como el uso de sprites con objetos que pueden ser destruidos). Además, el uso de cajas límite puede producir ciertos resultados poco realistas en el cálculo de colisiones.
- **Exportación**. La definición de objetos con propiedades físicas añade nuevas variables y constantes que deben ser tratadas por las herramientas de exportación de los datos del juego.

Figura 3.21: Gracias al uso de PhysX, las baldosas del suelo en *Batman Arkham Asylum* pueden ser destruidas (derecha). En la imagen de la izquierda, sin usar PhysX el suelo permanece inalterado, restando realismo y espectacularidad a la dinámica del juego.

- **Interfaz de Usuario**. Es necesario diseñar interfaces de usuario adaptados a las capacidades físicas del motor (¿cómo se especifica la fuerza y la dirección de lanzamiento de una granada?, ¿de qué forma se interactúa con objetos que pueden recogerse del suelo?).

3.4.3. Conceptos Básicos

A principios del siglo XVII, Isaac Netwon publicó las tres **leyes fundamentales del movimiento**. A partir de estos tres principios se explican la mayor parte de los problemas de dinámica relativos al movimiento de cuerpos y forman la base de la mecánica clásica. En el estudio de la dinámica resulta especialmente interesante la segunda ley de Newton, que puede ser escrita como $F = m \times a$, donde F es la fuerza resultante que actúa sobre un cuerpo de masa m, y con una aceleración lineal a aplicada sobre el centro de gravedad del cuerpo.

Desde el punto de vista de la mecánica en videojuegos, la **masa** puede verse como una medida de la resistencia de un cuerpo al movimiento (o al cambio en su movimiento). A mayor masa, mayor resistencia para el cambio en el movimiento. Según la segunda ley de Newton que hemos visto anteriormente, podemos expresar que $a = F/m$, lo que nos da una impresión de cómo la masa aplica resistencia al movimiento. Así, si aplicamos una **fuerza** constante e incrementamos la masa, la aceleración resultante será cada vez menor.

El **centro de masas** (o de **gravedad**) de un cuerpo es el punto espacial donde, si se aplica una fuerza, el cuerpo se desplazaría sin aplicar ninguna rotación.

Un **sistema dinámico** puede ser definido como cualquier colección de elementos que cambian sus propiedades a lo largo del tiempo. En el caso particular de las simulaciones de cuerpo rígido nos centraremos en el cambio de posición y rotación.

Así, nuestra **simulación** consistirá en la ejecución de un modelo matemático que describe un sistema dinámico en un ordenador. Al utilizar modelos, se simplifica el sistema real, por lo que la simulación no describe con total exactitud el sistema simulado.

Habitualmente se emplean los términos de interactividad y tiempo real de modo equivalente, aunque no lo son. Una **simulación interactiva** es aquella que consigue una tasa de actualización suficiente para el control por medio de una persona. Por su parte, una **simulación en tiepo real** garantiza la actualización del sistema a un número fijo de frames por segundo. Habitualmente los motores de simulación física proporcionan tasas de frames para la simulación interactiva, pero no son capaces de garantizar Tiempo Real.

3.4.4. Formas de Colisión

Como hemos comentado anteriormente, para calcular la colisión entre objetos, es necesario proporcionar una representación geométrica del cuerpo que se utilizará para calcular la colisión. Esta representación interna se calculará para determinar la posición y orientación del objeto en el mundo. Estos datos, con una descripción matemática mucho más simple y eficiente, son diferentes de los que se emplean en la representación visual del objeto (como hemos visto en la sección 3.2, basados en sprites) que cuentan con un mayor nivel de detalle).

Habitualmente se trata de simplificar al máximo la forma de colisión. Aunque el SDC (*Sistema de Detección de Colisiones*) soporte objetos complejos, será preferible emplear tipos de datos simples, siempre que el resultado sea aceptable. La Figura 3.22 muestra algunos ejemplos de aproximación de formas de colisión para ciertos objetos del juego.

Multitud de motores de simulación física separan la forma de colisión de la transformación interna que se aplica al objeto. De esta forma, como muchos de los objetos que intervienen en el juego son dinámicos, basta con aplicar la transformación a la forma de un modo computacionalmente muy poco costoso. Además, separando la transformación de la forma de colisión es posible que varias entidades del juego compartan la misma forma de colisión.

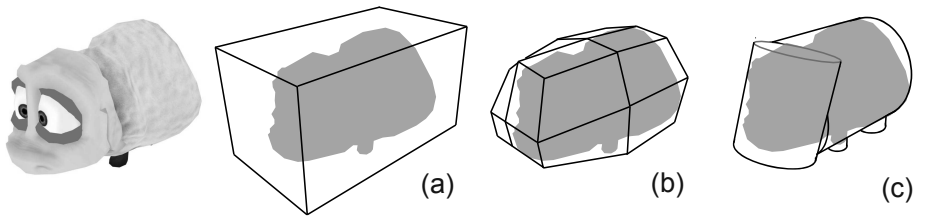


Figura 3.22: Diferentes formas de colisión para el objeto de la imagen. (a) Aproximación mediante una caja. (b) Aproximación mediante un volumen convexo. (c) Aproximación basada en la combinación de varias primitivas de tipo cilíndrico.

> Algunos motores de simulación física permiten compartir la misma descripción de la forma de colisión entre entidades. Esto resulta especialmente útil en juegos donde la forma de colisión es compleja, como en simuladores de carreras de coches.

Como se muestra en la Figura 3.22, las entidades del juego pueden tener diferentes formas de colisión, o incluso pueden compartir varias primitivas básicas (para representar por ejemplo cada parte de la articulación de un brazo robótico).

El **Mundo Físico** sobre el que se ejecuta el SDC mantiene una lista de todas las entidades que pueden colisionar empleando habitualmente una estructura global *Singleton*. Este *Mundo Físico* es una representación del mundo del juego que mantiene la información necesaria para la detección de las colisiones. Esta separación evita que el SDC tenga que acceder a estructuras de datos que no son necesarias para el cálculo de la colisión.

Los SDC mantienen estructuras de datos específicas para manejar las colisiones, proporcionando información sobre la *naturaleza* del contacto, que contiene la lista de las formas que están intersectando, su velocidad, etc...

Para gestionar de un modo más eficiente las colisiones, las formas que suelen utilizarse con convexas. Una **forma convexa** es aquella en la que un rayo que surja desde su interior atravesará la superfice una única vez. Las superficies convexas son mucho más simples y requieren menor capacidad computacional para calcular colisiones que las formas cóncavas. Algunas de las primitivas soportadas habitualmente en SDC son:

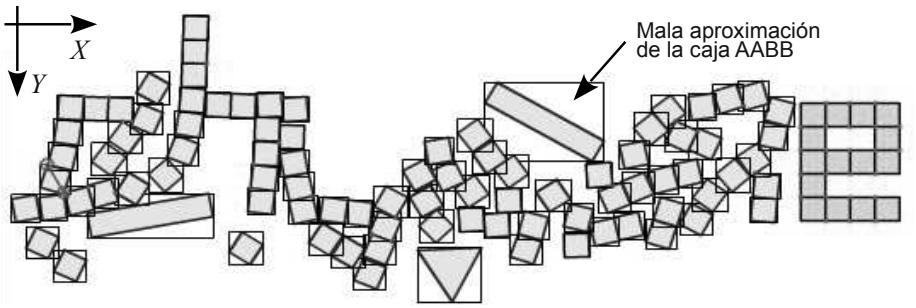


Figura 3.23: Gestión de la forma de un objeto empleando cajas límite alineadas con el sistema de referencia universal AABBs. Como se muestra en la figura, dependiendo de la rotación de la figura, puede ser que la aproximación mediante una caja AABB sea bastante pobre.

- **Esferas/Círculos**. Son las primitivas más simples y eficientes; basta con definir su centro y radio.
- **Cajas/Cuadrados**. Por cuestiones de eficiencia, se suelen emplear cajas límite alineadas con los ejes del sistema de coordenadas (AABB o ***Axis Aligned Bounding Box***). Las cajas AABB se definen mediante las coordenadas de dos extremos opuestos. El principal problema de las cajas AABB es que, para resultar eficientes, requieren estar alineadas con los ejes del sistema de coordenas global. Esto implica que si el objeto rota, como se muestra en la Figura 3.23, la aproximación de forma puede resultar de baja calidad. Por su eficiencia, este tipo de cajas suelen emplearse para realizar una primera aproximación a la intersección de objetos para, posteriormente, emplear formas más precisas en el cálculo de la colisión.
- **Cilindros/Cápsulas**. Los cilindros son ampliamente utilizados. Se definen mediante dos puntos y un radio. Las cápsulas pueden verse como el volumen resultante de desplazar una esfera entre dos puntos. El cálculo de la intersección con cápsulas es más eficiente que con esferas o cajas, por lo que se emplean para el modelo de formas de colisión en formas que son aproximadamente cilíndricas (como las extremidades del cuerpo humano).
- **Volúmenes/Áreas convexas**. La mayoría de los SDC permiten trabajar con volúmenes y áreas convexas (ver Figura 3.22). La forma del objeto suele representarse internamente mediante un conjunto de n planos (en el espacio 3D) o un conjunto de vértices (si se trabaja en 2D). Aunque este tipo de formas es menos eficiente que las primitivas estudiadas anteriormente, flexibilizan mucho el trabajo y son ampliamente utilizadas en desarrollo de videojuegos.

3.4.5. Optimizaciones

La detección de colisiones es, en general, una tarea que requiere el uso intensivo de la CPU. Por un lado, los cálculos necesarios para determianr si dos formas intersecan no son triviales. Por otro lado, muchos juegos requiren un alto número de objetos en la escena, de modo que el número de test de intersección a calcular rápidamente crece. En el caso de n objetos, si empleamos un algoritmo de fuerza bruta tendríamos una complejidad $O(n^2)$. Es posible utilizar ciertos tipos de optimizaciones que mejoran esta complejidad inicial:

- **Coherencia Temporal**. Este tipo de técnica de optimización (también llamada *coherencia entre frames*), evita recalcular cierto tipo de información en cada frame, ya que entre pequeños intervalos de tiempo los objetos mantienen las posiciones y orientaciones en valores muy similares.
- **Particionamiento Espacial**. El uso de estructuras de datos de particionamiento especial permite comprobar rápidamente si dos objetos podrían estar intersecando si comparten la misma celda de la estructura de datos. Algunos esquemas de particionamiento jerárquico, como árboles octales, BSPs o árboles-kd permiten optimizar la detección de colisiones en el espacio. Estos esquemas tienen en común que el esquema de particionamiento comienza realizando una subdivisión general en la raíz, llegando a divisiones más finas y específicas en las hojas. Los objetos que se encuentran en una determinada rama de la estructura no pueden estar colisionando con los objetos que se encuentran en otra rama distinta.
- **Barrido y Poda (SAP)**. En la mayoría de los motores de simulación física se emplea un algoritmo Barrido y Poda (*Sweep and Prune*). Esta técnica ordena las cajas AABBs de los objetos de la escena y comprueba si hay intersecciones entre ellos. El algoritmo *Sweep and Prune* hace uso de la *Coherencia temporal frame a frame* para reducir la etapa de ordenación de $O(n \times log(n))$ a $O(n)$.

En muchos motores, como en Box2D, se utilizan varias capas o pasadas para detectar las colisiones. Primero suelen emplearse cajas AABB para comprobar si los objetos pueden estar potencialmente en colisión (detección de la colisión amplia o en Inglés *Broadphase*). A continuación, en una segunda capa se hacen pruebas con volúmenes generales que engloban los objetos (por ejemplo, en un objeto compuesto por varios subobjetos, se calcula una esfera que agrupe a todos los subobjetos). Si esta segunda capa de colisión da un resultado positivo, en una tercera pasada se calculan las colisiones empleando las formas finales (colisión de granularidad fina).

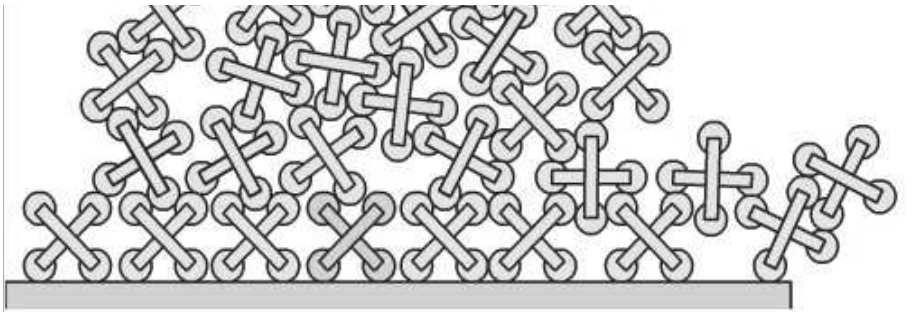


Figura 3.24: Physaxe soporta segmentos, polígonos con esquinas redondeadas, como forma de colisión básica.

3.4.6. Hola Mundo con Physaxe

Una vez estudiados los aspectos generales de los motores de simulación física, en esta sección presentaremos un ejemplo totalmente funcional con la biblioteca Physaxe, un wrapper escrito en Haxe del motor Box2D y de Glaze. Su autor destaca, entre otras, las siguientes características:

- Soporte de cuerpos rígidos basados en diversas formas de colisión: cajas (`phx.Shape.makeBox`), círculos (`phx.Circle`), Polígonos convexos de cualquier tipo (`phx.Polygon`) y segmentos (polígonos con esquinas redondeadas `phx.Segment`, ver Figura 3.24).
- Diversas fases de detección *Broadphase*.
- Activación y desactivación de objetos.
- Multitud de propiedades de simulación asociadas a los cuerpos rígidos (fricción, límites de movimiento, restricciones, etc...).

> Antes de comenzar con los ejemplos, debes instalar Physaxe. Como vimos en las secciones anteriores, basta con ejecutar `sudo haxelib install physaxe`.

Como se ha comentado al inicio de la sección, el motor de simulación física es independiente del motor de representación gráfica. El caso de Physaxe no es una excepción, y será necesario desarrollar clases de utilidad que conecten ambos motores. Veamos a continuación el primer ejemplo básico tipo *Hola Mundo* con Physaxe.

Listado 3.50: Clase principal de HelloPhysic (Fragmento).

```
1  class HelloPhysic extends Sprite{
2    var _layer:TileLayer;              // Capa principal de dibujado
3    var _world:World;                  // Mundo de simulación física
4    var _vPhSprite:Array<PhSprite>;    // Array de Physical Sprite
5    // Constructor ====================================================
6    function init():Void {
7      loadResources();       // Carga los recursos gráficos (omitido)
8      _vPhSprite = new Array<PhSprite>();
9      createPhysicWorld();   // Crea el mundo de simulación física
10     addListeners();        // Añade los Listeners (omitido)
11   }
12   // Physic World ===================================================
13   function createPhysicWorld():Void {
14     var size = new AABB(-1000, -1000, 1000, 1000);  // Límites
15     var bp = new SortedList();        // Método de Broadphase
16     _world = new World(size, bp);     // Creación del mundo
17     createLimits(_sw, _sh);           // Crea cajas estáticas
18     _world.gravity = new phx.Vector(0,0.9);   // Gravedad
19   }
20   function createLimits(w:Float, h:Float):Void {
21     // Creamos los límites del mundo: makeBox(Ancho, Alto, X, Y)
22     // La X e Y de la caja tiene origen en esquina superior izqda.
23     _world.addStaticShape(Shape.makeBox(w,40,0,h));
24     _world.addStaticShape(Shape.makeBox(w,40,0,-40));
25     _world.addStaticShape(Shape.makeBox(40,h,-40,0));
26     _world.addStaticShape(Shape.makeBox(40,h,w,0));
27   }
28   // Update =========================================================
29   function onEnterFrame(e:Event):Void {
30     _world.step(1,10);            // Avanzamos un paso de simulación
31     for (p in _vPhSprite) p.update(_sw, _sh);
32     _layer.render(); _fpsText.update();
33   }
34   // Events =========================================================
35   function onMouseClick(e:MouseEvent):Void {
36     var p = new PhSprite("box", _layer, _world, e.localX, e.localY);
37     _vPhSprite.push(p);
38   }
39 }
```

La clase cuenta con un objeto de tipo *World* (ver línea 3). Esta clase forma parte de la distribución de *Physaxe* y contendrá los elementos que intervendrán en la simulación física (cuerpos, restricciones, propiedades, etc...). Aunque el motor interno de *Physaxe* (*Box2D*) soporta la creación de múltiples mundos, habitualmente no es necesario (ni siquiera deseable) tener varias instancias de esta clase en ejecución.

En la línea 5 se define el *Array* de objetos de tipo *PhSprite* (*Physical Sprite*). Esta clase de utilidad ha sido desarrollada para los ejemplos a modo de conector entre el motor de dibujado (basado en Sprites gráficos) y el motor de simulación física. En este ejemplo, esta clase será utilizada para añadir cajas de tamaño aleatorio en la escena.

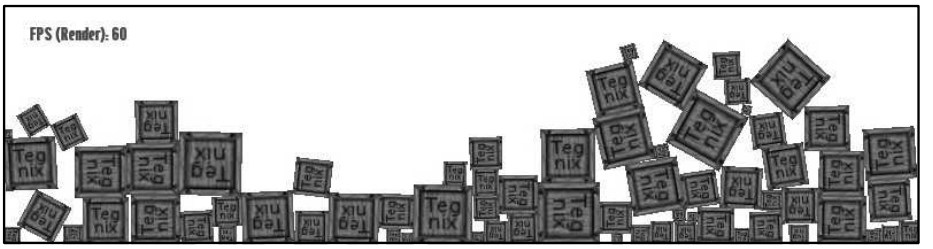


Figura 3.25: Resultado de la ejecución del *Hello Physics*. Cada vez que se pincha con el ratón sobre la ventana se añade una caja con dimensión y rotación aleatoria.

En el ejemplo de la siguiente sección, la clase *PhSprite* será rediseñada para soportar otras formas de colisión.

Las llamadas relativas a la creación del mundo de simulación física se han encapsulado en la llamada al método *createPhysicWorld* (ver líneas 13–19). La primera llamada en este método (línea 14) define los límites del mundo como una caja AABB. Estas dimensiones se especifican mediante dos vértices; el superior izquierdo y el inferior derecho. En este caso, únicamente se tendrán en cuenta los objetos físicos situados entre las coordenadas (-1000,-1000) y la (1000,1000).

Límites del Mundo. Internamente, *Physaxe* llama al método *boundCheck*, para determinar si los cuerpos se encuentran dentro de los límites del mundo. En otro caso, serán eliminados (ejecutando su método *onDestroy*).

A continuación se define el método de detección de colisión *BroadPhase* (línea 15). Esta primera etapa de detección de colisiones evita el tener que comparar todos los pares de formas de colisión, reduciendo el número de comparaciones a realizar. En *Physaxe* hay disponibles tres métodos de deteción *BroadPhase*:

- **BruteForce**. Este método almacena las formas de colisión en una lista y realiza la comprobación entre todos los pares. Este método es el más lento y no es aconsejable su uso salvo con fines de depuración.
- **SortedList**. Almacenando las formas de colisión en una lista ordenada, sólo se realizan las comprobaciones entre las formas que colisionan en el eje Y.

- **Quantize**. Es el método más avanzado de los tres. Divide el espacio del mundo en cuadrados de 2^n unidades. Cada forma se almacena en los cuadrados que intersecan con su caja límite (*bounding box*), comparando con el resto de formas que comparten el mismo cuadrado. Este método es recomendable si tenemos mundos muy grandes con una densidad pequeña de objetos dinámicos.

> Si te encuentras con fuerza, puedes implementar tu propio método de *Broadphase* teniendo en cuenta el interfaz general definido en `phx.col.Broadphase`. Como recomienda el autor, puedes partir del método *Bruteforce* para orientar tu implementación.

Para la creación del mundo de simulación física (ver línea 16) es necesario especificar únicamente el tamaño del mundo y el método de optimización de *Broadphase* definidos en las líneas anteriores.

En la línea 18 se especifica la fuerza de la gravedad del mundo. Este vector puede cambiarse en cualquier momento en tiempo de ejecución.

La llamada al método auxiliar de la línea 17 sirve para crear los límites de colisión del mundo mediante llamadas a *addStaticShape*. Los cuerpos estáticos no colisionan con otros objetos estáticos y no pueden desplazarse (se comportan como si tuvieran masa infinita; en realidad *Box2D* almacena un valor de cero para la masa).

Mediante la llamada al método *makeBox* se crea una caja de colisión. Es importante señalar que las coordenadas de los objetos estáticos deben ser universales (coordenadas globales). De este modo, sabiendo que el (0,0) está situado en la esquina superior izquierda y que las coordenadas X e Y de la caja se indican con respecto de la esquina superior izquierda, las líneas 23–26 definen cuatro cajas situadas en los bordes de la ventana. Por ejemplo, la línea 23 define la caja que sirve como suelo (la caja queda *fuera* de la ventana, el borde superior de la caja coincide con el borde inferior de la ventana).

Cada vez que *renderizamos* un frame en el motor de representación gráfico tenemos que avanzar igualmente un paso en la simulación física. La actualización de la simulación física se realiza mediante la llamada al método *step* (línea 30). El primer argumento de la función es la cantidad de tiempo transcurrida desde la última actualización (especificado en 1/60 Segundos). Como el despliegue gráfico se realiza a 60fps, llamaremos al método con un valor de 1. El segundo argumento indica el número de subpasos de simulación para el cálculo de colisiones. Este valor permite garantizar que las colisiones se calculan de forma correcta

(evitando el denominado efecto *túnel*[25]). Valores mayores del número de subpasos obtienen mejores resultados, pero con un coste computacional mayor. En el caso de este ejemplo se han utilizado 10 subpasos.

De igual forma, en cada paso de simulación llamaremos al método *update* (ver línea 31) de la clase auxiliar *PhSprite*. Cada vez que el usuario haga *click* con el ratón, crearemos una nueva instancia de objeto *PhSprite* (ver línea 36), indicando como primer parámetro el nombre del *Sprite* que queremos utilizar ("*box*" en este caso), la capa de dibujado, el objeto del mundo físico y las coordenadas donde se añadirá el objeto. Este nuevo objeto se incluirá en el array *vPhSprite* (línea 37).

A continuación estudiaremos la implementación de los aspectos más relevantes de la clase *PhSprite*. Como se ha comentado anteriormente, esta clase será adaptada en el siguiente ejemplo para permitir cualquier forma de colisión (la implementación actual únicamnete soporta cajas).

La clase mantiene un cuerpo de simulación física (de la clase *Body*, ver línea 4) asociado a cada Sprite. El método update *update* (ver líneas 18-22) se encarga de obtener las propiedades de posición y rotación del cuerpo de simulación física y copiarlas al *Sprite*.

Listado 3.51: Clase PhSprite (Verión 1).

```
1  class PhSprite extends TileSprite {
2    var _root:TileLayer;   // Capa de dibujado de este Sprite
3    var _world:World;      // Mundo de simulación física
4    var _body:Body;        // Cuerpo de simulación asociado al Sprite
5    // Constructor ==================================================
6    public function new(id:String, l:TileLayer, w:World, px:Float,
7                py:Float):Void {
8      super(l, id); _root = l; _world = w; x = px; y = py;
9      _root.addChild(this);   // Añadir Sprite a la capa de dibujado
10     _body = new Body(x,y);  // Crear cuerpo de simulación física
11     scale = 0.25 + Std.random(75) / 50;   // Tamaño aleatorio
12     _body.addShape(Shape.makeBox(width, height));
13     _body.a = (Math.PI / 180.0) * Std.random(360);  // Rotación!
14     _body.updatePhysics();  // Actualización del cuerpo
15     _world.addBody(_body);  // Añadimos el cuerpo al mundo
16   }
17   // Update =======================================================
18   public function update(w:Int, h:Int):Void {
19     _body.updatePhysics();   // Actualizamos el cuerpo en físicas
20     x =_body.x; y =_body.y;  // Copiamos coordenadas al Sprite
21     rotation = _body.a;      // Copiamos rotación en Sprite
22   }
23 }
```

[25]El efecto túnel ocurre cuando los objetos no colisionan con otros objetos. Imaginemos que tenemos un objeto de colisión muy estrecho. Si otro objeto se mueve a una alta velocidad en su dirección y las colisiones se calculan de forma discreta es posible que, en los pasos de simulación concretos, ambos objetos no lleguen nunca a colisionar. De este modo, es como si el objeto hubiera pasado por un *túnel*, evitando así la colisión.

Es posible mover manualmente un cuerpo durante la simulación. Debes tener en cuenta en este caso que es necesario llamar al método `world.sync(body)` para notificar el cambio realizado al motor de simulación física.

La implementación del constructor (líneas 8–15) es muy sencilla. La línea 10 crea un cuerpo genérico, en las coordenas X,Y pasadas al constructor de *PhSprite*. En la línea 11 se obtiene un valor aleatorio para la escala del objeto. Mediante la llamada a *addShape* (línea 12) se especifica la forma de colisión de ese objeto. En este caso utilizamos una caja con las mismas dimensiones del Sprite. El atributo a del cuerpo permite indicar la rotación en radianes (ver línea 13). En este caso asignamos un valor aleatorio entre 0 y 360. La línea 15 añade el objeto al mundo de simulación física.

Dulces Sueños... Cuando un cuerpo permanece estático y su energía cinética disminuye a un valor menor que la definida en `world.sleepEpsilon`, el cuerpo pasa al estado de *dormido*. Un cuerpo dormido no puede moverse, salvo que lo despertemos manualmente. En este caso, basta con llamar a `world.activate(body)` para desprenderlo de los brazos de Morfeo.

Como veremos en la siguiente sección "*Más Allá del Hola Mundo*", los cuerpos y formas de colisión en *Physaxe* cuentan con multitud de parámetros que pueden definirse. En el caso del siguiente Mini-Juego definiremos materiales con propiedades físicas específicas y definiremos manualmente formas de colisión convexas. Además, definiremos un vector de velocidad lineal según la posición del puntero en tiempo de ejecución.

Identificadores únicos. Cada objeto de tipo `Body` y `Shape` cuentan con identificadores (campo *id* únicos). Aunque en la implementación son atributos públicos, no es conveniente modificarlos (aunque sí es muy útil consultarlos para comprobar si ha ocurrido una colisión entre objetos).

Figura 3.26: *Office Basket* en acción. El objetivo del juego es encestar la pelota de papel. Cada vez el origen se situará en una nueva posición del espacio.

3.4.7. Más allá del Hola Mundo

En esta sección estudiaremos el ejemplo del Mini-Juego *Office Basket* (ver Figura 3.26). En el desarrollo de este ejemplo hemos utilizado, además de la clase *PhSprite* convenientemente rediseñada, una clase auxiliar para el dibujado de la flecha (llamada *VectorArrow*). En el siguiente listado se muestran los aspectos más relevantes de la clase principal del ejemplo.

Listado 3.52: Clase principal de Office Basket (Fragmento).

```
1  class OfficeBasket extends Sprite{
2    var _previousTime:Int;              // Tiempo desde lanzamiento
3    var _sw:Int; var _sh:Int;           // Ancho y alto de la pantalla
4    var _spx:Float; var _spy:Float;     // Coordenadas de la bola
5    var _ball:PhPaperBall = null;       // Objeto para la bola
6    var _bin:PhPaperBin = null;         // Objeto para la papelera
7    var _arrow:VectorArrow;             // Clase para dibujar flecha
8
9    // Métodos Auxiliares ==========================================
10   function generateRandomPaperPosition():Void {
11     _spx =_sw/2 - 50 + Std.random(Std.int(_sw/2));
12     _spy = 50 + Std.random(Std.int(_sh/3));  }
13   function createArrow(px:Float, py:Float):Void {
14     _arrow = new VectorArrow(_layerArrow, "ball", "dot", "arrow",
15                  px, py, 10, 100);  }
```

```
16    function createBasket(w:Float, h:Float):Void {
17      var xaux = 50 + Std.random(Std.int(w/2));
18      _bin = new PhPaperBin("basket",_layerBin,_world, xaux, h-52); }
19    // Physic World ================================================
20    function createPhysicWorld():Void {
21      var size = new AABB(-1000, -1000, 1000, 1000);
22      var bp = new SortedList();
23      _world = new World(size, bp);
24      createLimits(_sw, _sh);
25      _world.gravity = new phx.Vector(0,0.9);
26    }
27    function createLimits(w:Float, h:Float):Void {
28      // Creamos los límites del mundo: makeBox(Ancho, Alto, X, Y)
29      // La X e Y de la caja tiene origen en esquina superior izqda.
30      var s = Shape.makeBox(w,200,0,h-20);
31      s.material.restitution = 0.2;  // Establecemos propiedades
32      s.material.density = 9;        // en el material...
33      _world.addStaticShape(s);
34      _world.addStaticShape(Shape.makeBox(w,200,0,-200));
35      _world.addStaticShape(Shape.makeBox(200,h,-200,0));
36      _world.addStaticShape(Shape.makeBox(200,h,w,0));
37    }
38    // Update ======================================================
39    function onEnterFrame(e:Event):Void {
40      var now = Lib.getTimer();
41      var secondsSinceClick = (now - _previousTime)/1000.0;
42      _world.step(1,20);
43      if (_ball != null) {       // Si hay bola (usuario hizo click)
44        _ball.update(_sw, _sh);  // Actualizamos la bola!
45        if (secondsSinceClick > 2) {   // Si han pasado más de 2 seg
46      // Comprobamos si la bola está tocando con el objeto
47      // que define la base de la papelera...
48      if (_ball.checkCollision(_bin.getIdBase()))
49        _scoreBoard.update(1);
50      _ball.destroy(); _ball = null;
51      generateRandomPaperPosition();  // Nueva posición!!
52      _arrow.setBasePos(_spx, _spy);  // Actualizar flecha
53        }
54      }
55      // Solo actualizamos la flecha si no hay bola activa...
56      else { _arrow.update();  _layerArrow.render(); }
57      _layerSky.render(); _layerPaper.render(); _layerBin.render();
58      _fpsText.update();
59    }
60    // Events ======================================================
61    function onMouseClick(e:MouseEvent):Void {
62      if (_ball == null) // Si no hay pelota activa, creamos una
63        _ball = new PhPaperBall("ball", _layerPaper, _world,
64                    _spx, _spy, _arrow.getVector());
65      _previousTime = Lib.getTimer();
66    }
67    function onMouseMove(event:MouseEvent):Void {
68      _arrow.setArrowPos(event.localX, event.localY);  }
69  }
```

Tras ejecutar el ejemplo, vemos que el juego genera posiciones aleatorias de la pelota de papel. Tras el lanzamiento de la pelota, dejamos que la simulación física actúe durante un intervalo de tiempo y generamos una nueva posición. El intervalo de tiempo transcurrido se calcula en base a la variable miembro *_previousTime* (ver línea 2). Las posiciones aleatorias de la bola se almacenan en las variables *_spx* y *_spy* mediante el método auxiliar *generateRandomPaperPosition* (definido en las líneas 10–12).

Los objetos de tipo *PhPaperBall* y *PhPaperBin* son clases hijas de la nueva implementación de la clase *PhSprite* (ver líneas 5–6), y serán descritas en las siguientes subsecciones. La creación de la papelera se realiza en la función *CreateBasket*, que instancia el objeto de tipo *PhPaperBin* en una posición aleatoria de la pantalla.

La variable *_arrow* utiliza la clase auxiliar *VectorArrow* para dibujar la flecha (ver Figura 3.27). El objeto se instancia en el método auxiliar *createArrow* (ver líneas 13–15).

El bucle principal (descrito en las líneas 39–54) se encarga de comprobar si el objeto *_ball* existe (línea 43). Se crea una instancia de la bola en la función de *callback onMouseClick* (definida en las líneas 61–66), empleando el constructor de la clase *PhPaperBall*.

Si la bola existe, se actualiza su posición, empleando el método *update* de la clase (línea 44). Si han pasado más de 2 segundos desde su lanzamiento (ver líneas 45 y 40–41), comprobamos si la pelota está colisionando con la base de la papelera. Si es así, (ver línea 49), actualizamos la puntuación. Tanto si hubo colisión como si no fue así, destruimos la bola actual y creamos una nueva (líneas 50–52).

Hemos introducido una modificación en la creación de los límites del mundo con respecto del ejemplo anterior. En este caso, estamos definiendo propiedades específicos del material (ver líneas 30–32).

La clase **Shape** mantiene una variable *material* de la clase *Material*. En esta variable se pueden establecer ciertas propiedades básicas de los materiales:

- **restitution**. El *Coeficiente de Restitución* mide la cantidad de velocidad que mantiene un cuerpo cuando colisiona con otro cuerpo (rebote). Es un valor entre 1 y 0. Valores cercanos a 1 implican mayor rebote.
- **density**. La densidad se mide en kg/m^3, y se utiliza para calcular la masa del cuerpo.
- **friction**. La fricción mide cuánto se desliza una forma sobre una superficie. Este valor habitualmente se especifica entre 0 (no hay fricción) y 1.

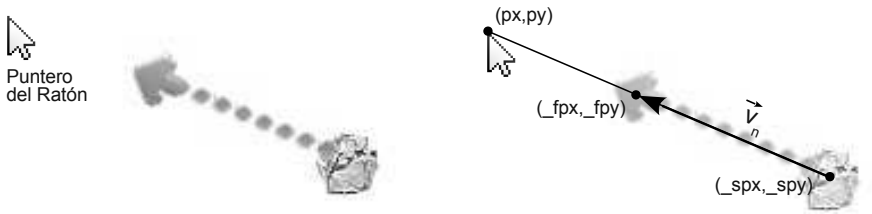


Figura 3.27: Las variables miembro *_fpx*, *_fpy* se calculan en base a la posición del ratón *px,py*. Obtenemos el vector v normalizado a una determinada longitud, especificado en la variable miembro *_maxLength*.

La tabla 3.2 muestra los valores que toman estas variables para algunos materiales.

Material	**Densidad**	**Fricción**	**Restitución**
Metal	7.85	0.2	0.2
Piedra	2.4	0.5	0.1
Madera	0.53	0.4	0.15
Cristal	2.5	0.1	0.2
Goma	1.5	0.8	0.4
Hielo	0.92	0.01	0.1
Tela	0.03	0.6	0.1
Esponja	0.018	0.9	0.05
Aire	0.001	0.9	0.0

Cuadro 3.2: Definición de las propiedades de algunos materiales.

A continuación estudiaremos los aspectos más relevantes de la implementación de la clase de utilidad *VectorArrow*. Como muestra la Figura 3.27, las variables miembro (*_spx, _spy*) y (*_fpx, _fpy*) definen las coordenadas iniciales y finales del vector (ver líneas 5–6). La posición incial vendrá determinada por la posición de la bola (calculada aleatoriamente en la clase principal del juego). Como veremos, la posición final vendrá determinada por la línea que forma el puntero del ratón y la posición de la bola. La clase, además, permite especificar el número de puntos que se dibujarán como parte del vector *_nPoints* o la longitud máxima de dibujado *_maxLength* (líneas 3–4).

Listado 3.53: Clase auxiliar de Vector Arrow (Fragmento).

```
1  class VectorArrow {
2    var _vSprite:Array<TileClip>;     // Sprites a dibujar...
3    var _nPoints:Int;                 // Número de puntos a desplegar
4    var _maxLength:Int;               // Longitud máxima del vector
5    var _spx:Float; var _spy:Float;   // Posición inicial del vector
6    var _fpx:Float; var _fpy:Float;   // Posición final del vector
7    // Constructor ====================================================
8    public function new(l: TileLayer, idBase:String, idDot:String,
9                idArrow, px:Float, py:Float, nPoints:Int,
10               length:Int):Void {
11     _root = l; _nPoints = nPoints; _spx = px; _spy = py;
12     _maxLength = length;  // Longitud máxima del vector
13     _vSprite = new Array<TileClip>();
14     var s:TileClip;
15     for (i in 1..._nPoints+1) {  // Añadir elementos
16       if (i==1) s = new TileClip(l, idBase);
17       else if (i==nPoints) {s = new TileClip(l, idArrow);
18     s.scale = 2;}
19       else s = new TileClip(l, idDot);
20       s.x = _spx; s.y = _spy; // Inicialmente todos en esta posición
21       _vSprite.push(s); _root.addChild(s);
22     }
23   }
24   // Clases auxiliares ==============================================
25   public function getVector():phx.Vector {
26     var v = new Point(_fpx - _spx, _fpy - _spy);
27     v.normalize(v.length * 0.25);
28     return new phx.Vector(v.x, v.y);
29   }
30   public inline static function radToDeg(rad:Float):Float
31     { return 180 / Math.PI * rad;  }
32   // Update =========================================================
33   public function update():Void {
34     var v = new Point(_fpx - _spx, _fpy - _spy);
35     v.normalize(v.length / _nPoints);
36     for (i in 0..._nPoints) {         // Calculamos nueva posición
37       _vSprite[i].x = _spx + v.x * i; // escalando el vector v
38       _vSprite[i].y = _spy + v.y * i;
39       if (i == _nPoints -1) {  // Actualizar rotación de la flecha
40         v.normalize(1); var angle = Math.acos(v.x);
41         if (v.y < 0) angle = 2*Math.PI - angle;
42         _vSprite[i].rotation = angle;
43       }
44     }
45   }
46   public function setArrowPos(px:Float, py:Float):Void {
47     var v = new Point(px - _spx, py - _spy);
48     if (v.length > _maxLength) {
49       v.normalize(_maxLength);
50       _fpx = _spx + v.x; _fpy = _spy + v.y;  }
51     else { _fpx = px; _fpy = py; }
52   }
53   public function setBasePos(px:Float, py:Float):Void
54     { _spx = px; _spy = py;  }
55 }
```

El constructor de la clase (líneas 8–22) crea un array de *Sprites* que serán dibujados para representar el vector. En la base se posiciona una instancia de la bola de papel (línea 16), en el otro extremo una flecha (línea 17) y como elementos intermedios Sprites de puntos (línea 18). Estos sprites serán reposicionados cada vez que el usuario mueva el ratón. La función de *callback* que se ejecuta cuando hay un cambio en la posición del ratón es *setArrowPos*, definida en las líneas 46–52.

Esta función calcula la posición final del array de Sprites en base a la línea que une la posición del ratón con la posición inicial de la bola de papel (calculada aleatoriamente). El vector v (línea 47) se normaliza a la dimensión máxima especificada en el constructor de la clase (ver Figura 3.27). El punto final se calcula simplemente sumando ese vector a la posición inicial de la bola de papel (línea 50).

El método *update* (definido en las líneas 32–45) reposiciona los sprites del array, obteniendo posiciones intermedias mediante una sencilla interpolación lineal. El ángulo del sprite final se obtiene empleando el producto escalar del vector V con la rotación incial de la flecha (especificada mediante el vector (1,0)).

Como hemos comentado anteriormente, la clase auxiliar *PhSprite* ha sido convenientemente rediseñada en este ejemplo para facilitar la incorporación de nuevas formas de colisión. En concreto, el método *createShape* (definido en la línea 8 del siguiente listado) debe ser sobreescrito por cada subclase de *PhSprite*. Este método puede ser empleado tanto por subclases que implementen objetos dinámicos (como el caso de *PhPaperBall*) como por objetos estáticos (la papelera, implementada en *PhPaperBin* es un objeto de colisión estático). Esta clase de utilidad emplea un atributo *_isDynamic* (línea 5) que indica el tipo de comportamiento del objeto.

Únicamente en el caso de ser un objeto dinámico, será necesario actualizar su posición y rotación en el método *update* (ver líneas 37–39).

Listado 3.54: Clase auxiliar de PhSprite V.2 (Fragmento).

```
1  class PhSprite extends TileSprite {
2    var _root:TileLayer;      // Capa de dibujado del objeto
3    var _world:World;         // Mundo de simulación en Physaxe
4    var _body:Body;           // Cuerpo físico de simulación
5    var _isDynamic:Bool;      // El objeto es dinámico o estático?
6    // Este método debe ser sobreescrito...
7    // Si es una StaticShape, puede añadirlas al mundo y devolver null
8    function createShape():Shape { return null;  }
9    // Constructor ==============================================
10   public function new(id:String, l:TileLayer, w:World, px:Float,
11               py:Float, isDynamic:Bool, ?vect:phx.Vector):Void {
12     super(l, id); _root = l; _world = w; _isDynamic = isDynamic;
13     x = px; y = py;
14     _root.addChild(this);
```

```
15      if (_isDynamic) {
16        _body = new Body(x,y);
17        _body.addShape(createShape());
18        _body.w = Std.random(100)/1000.0;   // Veloc. Angular
19        _body.v = vect;
20        _body.updatePhysics();
21        _world.addBody(_body);
22      }
23      else {
24        // El objeto es estático, puede añadir la forma al mundo
25        // en el propio método 'createShape'
26        var s = createShape();
27        if (s!=null) _world.addStaticShape(s);
28      }
29    }
30    // Destructor ====================================================
31    public function destroy():Void {
32      if (_isDynamic) _world.removeBody(_body);
33      _root.removeChild(this);
34    }
35    // Update ========================================================
36    public function update(w:Int, h:Int):Bool {
37      if (_isDynamic) {
38        x = _body.x;  y = _body.y;  rotation = _body.a;
39        _body.updatePhysics();
40      }
41      return true;
42    }
43  }
```

En base a esta implementación de la clase *PhSprite* se definen dos clases hijas específicas para el objeto dinámico de la bola de papel (clase *PhPaperBall*) y para la papelera (clase *PhPaperBin*). La implementación de estas clases hijas de *PhSprite* es muy sencilla, y básicamente se encargan de implementar los métodos específicos que definen su geometría de colisión y su comportamiento.

La clase *PhPaperBall* define un polígono convexo en la función *createShape* (líneas 3-10). El convenio requerido por *Physaxe* en la definición de los vértices es que éstos deben ser indicados por orden en el sentido contrario al giro de las agujas del reloj (ver Figura 3.28). Las coordenadas pueden especificarse además según un desplazamiento inicial. Si no se indica nada, se tienen que indicar respecto del centro del Sprite. Esto resulta molesto, ya que la mayoría de los paquetes gráficos miden el desplazamiento con respecto de la esquina superior izquierda. Si conocemos el ancho y el alto del sprite, podemos indicar ese desplazamiento como último parámetro al constructor de la clase *Polygon* (ver línea 9). En el caso de la bola de papel, el Sprite tiene un tamaño de 32x32 píxeles. Como el (0,0) se indicaría en el centro del Sprite, para reposicionar el origen en la esquina superior izquierda, hay que indicar (-16,-16) como vector de *offset*.

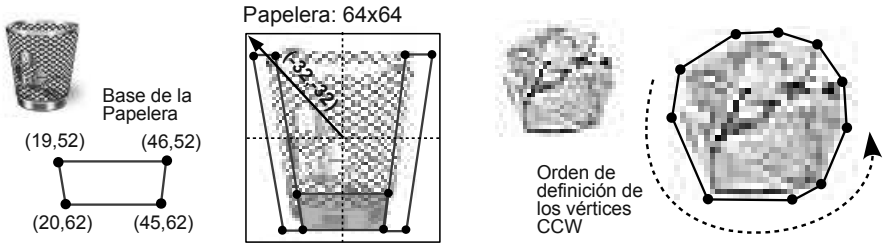


Figura 3.28: Definición de los vértices de los objetos del Mini-Juego, descritos en el método *createShape* de *PhPaperBin* y *PhPaperBall*. En la figura de la izquierda se describen las coordenadas relativas de una de las partes que definen la papelera, así como el vector de *Offset* (-32,-32) que indica el nuevo origen para especificar las coordenadas. A la derecha se representan los vértices empleados en la bola de papel, así como el orden de definción de los mismos.

De forma análoga, en la Figura 3.28 (izquierda) se indica que el offset en el caso de la papelera debe ser de (-32,-32) porque el tamaño del Sprite es de 64x64 píxeles.

Listado 3.55: Clase PhPaperBall (Fragmento).

```
1  class PhPaperBall extends PhSprite {
2    // Create Shape ================================================
3    override function createShape():Shape {
4      return (new phx.Polygon ([new phx.Vector(1,12),
5         new phx.Vector(1,17),  new phx.Vector(7,30),
6         new phx.Vector(20,30), new phx.Vector(25,29),
7         new phx.Vector(29,19), new phx.Vector(28,12),
8         new phx.Vector(17,2),  new phx.Vector(7,6)],
9         new phx.Vector(-16,-16)));
10   }
11   // Constructor =================================================
12   public function new(id:String, l:TileLayer, w:World, px:Float,
13                 py:Float, vect:phx.Vector):Void {
14     super(id, l, w, px, py, true, vect);   }
15   // Check Collision =============================================
16   public function checkCollision(idBase:Int):Bool {
17     for (a in _body.arbiters.iterator())
18       if ((a.s1.id == idBase) || (a.s2.id == idBase)) return true;
19     return false;
20   }
21   // Update ======================================================
22   override public function update(w:Int, h:Int):Bool {
23     x = _body.x;  y = _body.y;  rotation = _body.a;
24     _body.updatePhysics();
25     return true;  }
26 }
```

Para comprobar si la bola colisiona con la base de la papelera se ha creado un método auxiliar *checkCollision* (ver líneas 17–20). La clase *Body* incorpora una lista de *árbitros* con referencias a todos los objetos que colisionan entre sí (por parejas). En el caso de la línea 18 se recorren todos estos árbitros y se comprueba si alguno de los dos objetos que están colisionando tiene el mismo identificador que el objeto pasado como argumento a la función. Esto nos permite, desde la clase principal del juego pasarle como identificador el correspondiente a la base de la papelera (ver Figura 3.28).

Documentación de Physaxe. Desafortunadamente, no existe una documentación completa de las clases de la biblioteca Physaxe. Sin embargo, el código fuente de la misma es muy claro, y en la mayoría de los casos, basta con mirar la implementación de las clases. En tu distribución de GNU/Linux, la implementación de la versión 1.2 puedes encontrarla en `/usr/lib/haxe/lib/physaxe/1,2/phx/`.

Por último, la implementación de la clase *PhPaperBin* añade tres formas de colisión estáticas en el método *createShape* (ver líneas 4–21). Como desde el propio método se añaden las tres formas al mundo, es necesario devolver *null* en la línea 21 (de otra forma, el constructor de la clase *PhSprite* se encargaría de añadir la forma devuelta al invocar a ese método).

Coordenadas Globales. Recordemos que las coordenadas de los objetos de colisión estáticos deben ser especificadas de forma global. Por esta razón, a las coordenadas específicas de cada vértice de la papelera hay que añadirle las coordenadas globales de su centro (indicadas como variables miembro de la clase Sprite, en x e y).

Cuando se añade el objeto base de la papelera (líneas 16–20), se guarda el identificador como variable miembro de la clase *_idBase*. Este identificador puede consultarse mediante la llamada al método público *getIdBase*, definido en la línea 24. Esto nos permite proporcionar dicho identificador al objeto de la clase *PhPaperBall* y comprobar si existe colisión.

Listado 3.56: Clase PhPaperBin (Fragmento).

```
1  class PhPaperBin extends PhSprite {
2    var _idBase:Int;
3    // Create Shape ==================================================
4    override function createShape():Shape {
5      // Borde izquierdo de la papelera
6      _world.addStaticShape(
7         new phx.Polygon ([new phx.Vector(6+x,9+y),
8            new phx.Vector(14+x,62+y), new phx.Vector(20+x,62+y),
9        new phx.Vector(12+x,9+y)], new phx.Vector(-32,-32)));
10     // Borde derecho de la papelera
11     _world.addStaticShape(
12        new phx.Polygon ([new phx.Vector(52+x,9+y),
13       new phx.Vector(45+x,62+y), new phx.Vector(51+x,62+y),
14       new phx.Vector(58+x,9+y)], new phx.Vector(-32,-32)));
15     // Base de la papelera
16     var saux = new phx.Polygon ([new phx.Vector(19+x,52+y),
17       new phx.Vector(20+x,62+y), new phx.Vector(45+x,62+y),
18       new phx.Vector(46+x,52+y)],new phx.Vector(-32,-32));
19     _idBase = saux.id;
20     _world.addStaticShape(saux);
21     return null;
22   }
23   // getIdBase =====================================================
24   public function getIdBase():Int {return _idBase;}
25   // Constructor ===================================================
26   public function new(id:String, l:TileLayer, w:World, px:Float,
27               py:Float):Void {
28     super(id, l, w, px, py, false);
29   }
30 }
```

3.5. Inteligencia Artificial

3.5.1. Introducción

La Inteligencia Artificial (IA) es un elemento fundamental para dotar de realismo a un videojuego. Uno de los retos principales que se plantean a la hora de integrar comportamientos inteligentes es alcanzar un equilibrio entre la **sensación de inteligencia** y el tiempo de cómputo empleado por el subsistema de IA. Dicho equilibrio es esencial en el caso de los videojuegos, como exponente más representativo de las aplicaciones gráficas en tiempo real.

Este planteamiento gira, generalmente, en torno a la generación de soluciones que, sin ser óptimas, proporcionen una cierta sensación de inteligencia. En concreto, dichas soluciones deberían tener como meta que el jugador se enfrente a un reto que sea factible de manera que suponga un estímulo emocional y que consiga *engancharlo* al juego.

Figura 3.29: El robot-humanoide *Asimo*, creado por *Honda* en el año 2000, es uno de los exponentes más reconocidos de la aplicación de técnicas de IA sobre un prototipo físico real. Sin embargo, todavía queda mucho por recorrer hasta que los robots puedan acercarse de manera significativa a la forma de actuar de una persona, tanto desde el punto de vista físico como racional.

En los últimos años, la IA ha pasado de ser un componente secundario en el proceso de desarrollo de videojuegos a convertirse en uno de los aspectos más importantes. Actualmente, lograr un alto nivel de IA en un juego sigue siendo **uno de los retos** más emocionantes y complejos y, en ocasiones, sirve para diferenciar un juego normal de uno realmente deseado por los jugadores.

Tal es su importancia, que las grandes desarrolladores de videojuegos mantienen en su plantilla a ingenieros especializados en la parte de IA, donde los lenguajes de *scripting*, como LUA o Python, y la comunicación con el resto de programadores del juego resulta esencial.

En esta sección se ofrece una introducción general de la IA aplicada al desarrollo de videojuegos, haciendo hincapié en conceptos como el de **agente** y en herramientas como las **máquinas de estados**. Posteriormente se discuten dos casos de estudio concretos. El primero se aborda desde el punto de vista teórico y en él se estudia cómo implementar el módulo de IA del Tetris. El segundo se aborda desde una perspectiva práctica y en él se hace uso de NME para llevar a cabo una implementación del famoso *tres en raya*. En concreto, en este último caso se discute una implementación del **algoritmo minimax** para simular el comportamiento de la máquina como adversario virtual.

3.5.2. Aplicando el Test de Turing

La Inteligencia Artificial es un área fascinante y relativamente moderna de la Informática que gira en torno a la construcción de programas inteligentes. Existen diversas interpretaciones para el término *inteligente* (vea [12] para una discusión en profundidad), las cuales se diferencian en función de la similaritud con conceptos importantes como **racionalidad** y **razonamiento**.

Figura 3.30: Alan Turing (1912-1954), matemático, científico, criptógrafo y filósofo inglés, es considerado uno de los Padres de la Computación y uno de los precursores de la Informática Moderna.

De cualquier modo, una constante en el campo de la IA es la relación entre un programa de ordenador y el comportamiento del ser humano. Tradicionalmente, la IA se ha entendido como la intención de crear programas que actuasen como lo haría una persona ante una situación concreta en un contexto determinado.

Hace más de medio siglo, en 1950, Alan Turing propuso la denominada **Prueba de Turing**, basada en la incapacidad de una persona de distinguir entre hombre o máquina a la hora de evaluar un programa de ordenador. En concreto, un programa pasaría el test si un evaluador humano no fuera capaz de distinguir si las respuestas a una serie de preguntas formuladas eran o no de una persona. Hoy en día, esta prueba sigue siendo un reto muy exigente ya que, para superarlo, un programa tendría que ser capaz de procesar lenguaje natural, representar el conocimiento, razonar de manera automática y aprender.

Además de todas estas funcionalidades, esta prueba implica la necesidad de interactuar con el ser humano, por lo que es prácticamente imprescindible integrar técnicas de visión por computador y de robótica para superar la *Prueba Global de Turing*. Todas estas disciplinas cubren gran parte del campo de la IA, por lo que Turing merece un gran reconocimiento por plantear un problema que hoy en día sigue siendo un reto muy importante para la comunidad científica.

En el ámbito del **desarrollo de videojuegos**, la Prueba de Turing se podría utilizar para evaluar la IA de un juego. Básicamente, sería posible aplicar esta prueba a los *Non-Player Characters* (NPCs) con el objetivo de averiguar si el jugador humano es capaz de saber si son realmente *bots* o podrían confundirse con jugadores reales.

Aunque actualmente existen juegos que tienen un grado de IA muy sofisticado, en términos generales es relativamente fácil distinguir entre

NPC y jugador real. Incluso en juegos tan trabajados desde el punto de vista computacional como el ajedrez, en ocasiones las decisiones tomadas por la máquina delatan su naturaleza.

Desafortunadamente, los desarrolladores de videojuegos están condicionados por el tiempo, es decir, los videojuegos son aplicaciones gráficas en tiempo real que han de generar una determinada tasa de *frames* o imágenes por segundo. En otras palabras, este aspecto tan crítico hace que a veces el tiempo de cómputo dedicado al sistema de IA se vea reducido. La buena noticia es que, generalmente, el módulo responsable de la IA no se tiene que actualizar con la misma frecuencia, tan exigente, que el motor de *rendering*.

Aunque esta limitación se irá solventando con el incremento en las prestaciones hardware de las estaciones de juego, hoy en día es un gran condicionante que afecta a los recursos dedicados al módulo de IA. Una de las consecuencias de esta limitación es que dicho módulo se basa en proporcionar una ***ilusión de inteligencia***, es decir, está basado en un esquema que busca un equilibro entre garantizar la simplicidad computacional y proporcionar al jugador un verdadero reto.

3.5.3. Ilusión de inteligencia

De una manera casi inevitable, el *componente inteligente* de un juego está vinculado a la dificultad o al reto que al jugador se le plantea. Sin embargo, y debido a la naturaleza cognitiva de dicho componente, esta cuestión es totalmente subjetiva. Por lo tanto, gran parte de los desarrolladores opta por intentar que el jugador se sienta inmerso en lo que se podría denominar *ilusión de inteligencia*.

Por ejemplo, en el videojuego *Metal Gear Solid*, desarrollado por Konami y lanzado para PlayStation™en 1998, los enemigos empezaban a mirar de un lado a otro y a decir frases del tipo *¿Quién anda ahí?* si el personaje principal, Solid Snake, se dejaba ver mínimamente o hacía algún ruido en las inmediaciones de los NPCs. En este juego, el espionaje y la infiltración predominaban sobre la acción, por lo que este tipo de elementos generaban una cierta sensación de IA, aunque en realidad su implementación fuera sencilla.

Un caso más general está representado por modificar el **estado de los NPCs**, típicamente incrementando su nivel de *stamina* o vida. De este modo, el jugador puede tener la sensación de que el enemigo es más inteligente porque cuesta más abatirlo. Otra posible alternativa consiste en proporcionar más habilidades al enemigo, por ejemplo haciendo que se mueva más rápido o que dispare con mayor velocidad.

En [2], el autor discute el caso de la IA del juego *Halo*, desarrollado por *Bungie Studios* y publicado por *Microsoft Games Studio* en 2001, y

cómo los desarrolladores consiguieron *engañar* a los *testers* del juego. En concreto, los desarrolladores asociaban el nivel de IA con la altitud de los puntos de impacto sobre los NPCs. Así, los jugadores percibían un grado bajo de IA cuando este nivel no era elevado, es decir, cuando los impactos en la parte inferior del NPC eran relevantes para acabar con el mismo. Sin embargo, al incrementar dicho nivel y forzar a los jugadores a apuntar a partes más elevadas del NPC, éstos percibían que el juego tenía una IA más elevada.

3.5.4. ¿NPCs o Agentes?

En gran parte de la bibliografía del desarrollo de videojuegos, especialmente en la relativa a la IA, el concepto de *agent* (agente) se utiliza para referirse a las distintas entidades virtuales de un juego que, de alguna manera u otra, tienen asociadas un **comportamiento**. Dicho comportamiento puede ser trivial y basarse, por ejemplo, en un esquema totalmente preestablecido o realmente complejo y basarse en un esquema que gire entorno al aprendizaje. De cualquier modo, estas dos alternativas comparten la idea de mostrar algún tipo de inteligencia, aunque sea mínima.

Tradicionalmente, el **concepto de agente** en el ámbito de la IA se ha definido como cualquier entidad capaz de percibir lo que ocurre en el medio o contexto en el que habita, mediante la ayuda de sensores, y actuar en consencuancia, mediante la ayuda de actuadores, generando normalmente algún tipo de cambio en dicho medio o contexto. La figura 3.31 muestra esta idea tan general. Note cómo de nuevo la idea de la Prueba de Turing vuelve a acompañar al concepto de agente.

El concepto de agente se ha ligado a ciertas propiedades que se pueden trasladar perfectamente al ámbito del desarrollo de videojuegos y que se enumeran a continuación:

- **Autonomía**, de manera que un agente actúa sin la intervención directa de terceras partes. Por ejemplo, un personaje de un juego de rol tendrá sus propios deseos, de manera independiente al resto.
- **Habilidad social**, los agentes interactúan entre sí y se comunican para alcanzar un objetivo común. Por ejemplo, los NPCs de un *shooter* se comunicarán para cubrir el mayor número de entradas a un edificio.
- **Reactividad**, de manera que un agente actúa en función de las percepciones del entorno. Por ejemplo, un enemigo reaccionará, normalmente, atacando si es atacado.

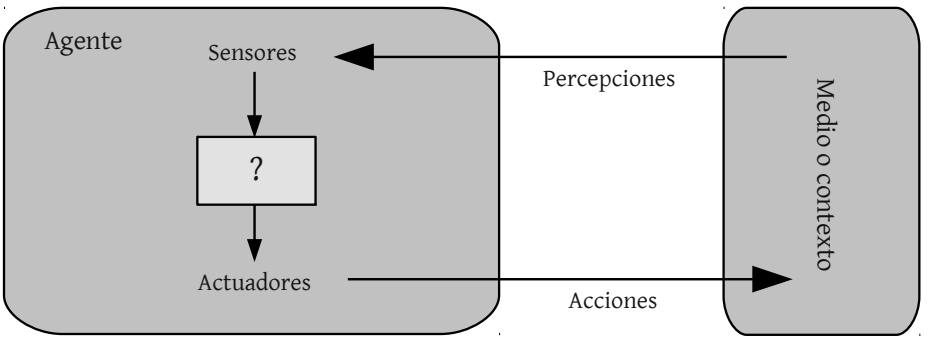


Figura 3.31: Visión abstracta del concepto de agente.

- **Proactividad**, de manera que un agente puede tomar la iniciativa en lugar de ser puramente reactivo. Por ejemplo, un enemigo feroz atacará incluso cuando no haya sido previamente atacado.

De manera adicional a estas propiedades, los conceptos de razonamiento y aprendizaje forman parte esencial del núcleo de un agente. Actualmente, existen juegos que basan parte de la IA de los NPCs en esquemas de aprendizaje y los usan para comportarse de manera similar a un jugador real. Este aprendizaje puede basar en la detección de patrones de comportamiento de dicho jugador.

Un ejemplo típico son los juegos deportivos. En este contexto, algunos juegos de fútbol pueden desarrollar patrones similares a los observados en el jugador real que los maneja. Por ejemplo, si la *máquina* detecta que el jugador real carga su juego por la parte central del terreno de juego, entonces podría contrarrestarlo atacando por las bandas, con el objetivo de desestabilizar al rival.

> Aunque los agentes se pueden implementar haciendo uso de una filosofía basada en el diseño orientado a objetos, informalmente se suele afirmar que *los objetos lo hacen gratis, mientras que los agentes lo hacen porque quieren hacerlo.*

Comúnmente, los agentes basan su modelo de funcionamiento interno en una **máquina de estados**, tal y como se muestra de manera gráfica en la figura 3.32. Este esquema se ha utilizado durante muchos años como herramienta principal para proporcionar esa *ilusión de inteligencia* por parte del desarrollador de IA. De hecho, aunque en algunos

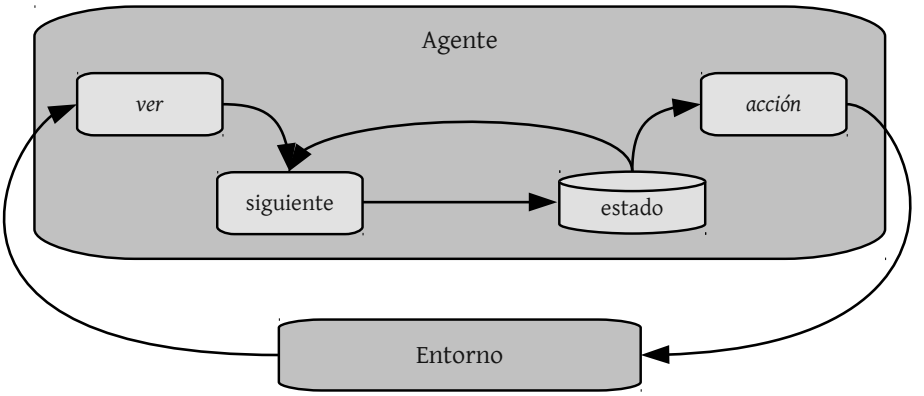


Figura 3.32: Visión abstracta del funcionamiento interno de un agente.

proyectos se planteen arquitecturas mucho más sofisticadas, en la práctica la mayor parte de ellas girarán en torno a la idea que se discute en la siguiente sección.

3.5.5. Diseño de agentes basado en estados

Una máquina de estados define el comportamiento que especifica las secuencias de estados por las que atraviesa un objeto durante su ciclo de ejecución en respuesta a una serie de eventos, junto con las respuestas a dichos eventos. En esencia, una máquina de estados permite descomponer el comportamiento general de un agente en *pedazos* o subestados más manejables. La figura 3.33 muestra un ejemplo concreto de máquinas de estados, utilizada para definir el comportamiento de un NPC en base a una serie de estados y las transiciones entre los mismos.

Como el lector ya habrá supuesto, los conceptos más importantes de una máquina de estados son dos: los estados y las transiciones. Por una parte, un **estado** define una condición o una situación durante la vida del agente, la cual satisface alguna condición o bien está vinculada a la realización de una acción o a la espera de un evento. Por otra parte, una **transición** define una relación entre dos estados, indicando lo que ha de ocurrir para pasar de un estado a otro. Los cambios de estado se producen cuando la transición se *dispara*, es decir, cuando se cumple la condición que permite pasar de un estado a otro.

Aunque la idea general de las máquinas de estado es tremendamente sencilla, su popularidad en el área de los videojuegos es enorme debido a los siguientes factores [2]:

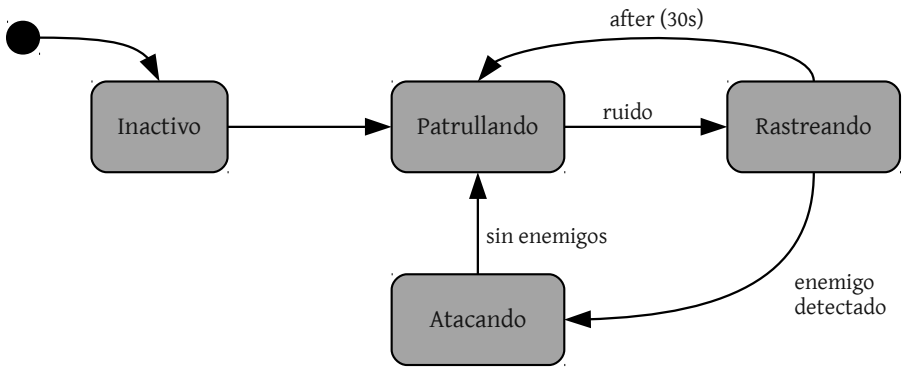


Figura 3.33: Máquina de estados que define el comportamiento de un NPC.

- Son **fáciles** de implementar y muy **rápidas**. Aunque existen diversas alternativas, todas ellas tienen una complejidad baja.

- Su **depuración** es sencilla, ya que se basan en el principio de descomposición en subestados que sean manejables.

- Tienen una **mínima sobrecarga computacional**, ya que giran en torno a un esquema *if-then-else*.

- Son muy **intuitivas**, ya que se asemejan al modelo de razonamiento del ser humano.

- Son muy **flexibles**, debido a que permiten la integración de nuevos estados sin tener un impacto significativo en el resto y posibilitan la combinación de otras técnicas clásicas de IA, como la lógica difusa o las redes neuronales.

Una máquina de estados se puede implementar utilizando distintas aproximaciones, como por ejemplo la que se estudió en la sección 3.1.4, para dar soporte al sistema de gestión de estados. En este contexto, es bastante común hacer uso del polimorfismo para manejar distintos estados que hereden de uno más general, proporcionando distintas implementaciones en función de dicha variedad de estados. En esencia, la idea consiste en mantener la interfaz pero concretando la implementación de cada estado. Normalmente, también se suele hacer uso del patrón *singleton* para manejar una única instancia de cada estado.

3.5.6. Búsqueda entre adversarios

En el ámbito de la IA muchos problemas se pueden resolver a través de una estrategia de búsqueda sobre las distintas soluciones existentes. De este modo, el razonamiento se puede reducir a llevar a cabo un proceso de búsqueda. Por ejemplo, un algoritmo de planificación se puede plantear como una búsqueda a través de los árboles y sub-árboles de objetivos para obtener un camino hasta el objetivo deseado (definido por un *test objetivo-meta*).

Para definir los árboles de búsqueda, es necesario diferenciar entre el concepto de **Estado** y **Nodo**. Un **Estado** nos representa una situación real; una fotografía de la situación del mundo en un determinado momento. El **Nodo** por su parte es una estructura de datos que es parte del árbol de búsqueda y, como parte de su información contiene:

- **Estado**. Representación interna del estado. Suele ser una referencia (puntero) a un conjunto de estados posibles. Varios nodos pueden compartir el mismo estado. Por ejemplo, en un problema de búsqueda de un camino entre dos ciudades de España, es posible que varios nodos pasen por la misma ciudad (*estado*).
- **Padre**. Referencia al nodo padre que, mediante una determinada acción, genera el nodo actual.
- **Hijos**. Referencias a los nodos hijo que se generan aplicando una determinada acción. A partir de un nodo, aplicando una acción (que tendrá asociado un coste), llegamos a un nodo hijo.
- **Profundidad**. Indica la profundidad en el árbol en la que se encuentra el nodo.
- **Costo del camino**. Indica el coste acumulado desde el nodo inicial (raíz del árbol de búsqueda) hasta el nodo actual.

De este modo, el pseudo-código de construcción de un árbol de búsqueda, se muestra en el listado 1. La única diferencia entre los diferentes métodos de búsqueda se basa en la estrategia de expansión del árbol. En realidad, el árbol de búsqueda es el mismo, pero se construirá en un determinado orden (podando ciertas ramas) según la estrategia utilizada.

Dependiendo de cada problema concreto, se creará el nodo raíz del árbol con el *Estado Inicial* del problema. Si el problema es la búsqueda del camino óptimo para viajar desde Madrid hasta Córdoba pasando por las capitales de provincia, la definición parcial del árbol de búsqueda puede verse en la Figura 3.34. En esa figura, el Nodo raíz contiene el *Estado Inicial* (Madrid). A partir de ahí se expanden los nodos hijo de

Algoritmo 1 Pseudocódigo del Árbol de Búsqueda

```
Función ÁrbolBusqueda(Problema,Estrategia):return Solución o Fallo
Inicia el árbol con Estado Inicial del Problema
while (Hay Candidatos para Expandir) do
    Elegir una hoja para expandir según Estrategia
    if Nodo.Estado cumple Test-Objetivo then return Solución
    elseExpande Nodo y añade hijos al árbol de búsqueda
    end if
end while
return Fallo // No quedan Candidatos a Expandir
```

primer nivel de profundidad. El camino hasta ese nodo tendrá asociado un coste (por ejemplo, el número de kilómetros entre ciudades). Vemos que el estado en diferentes nodos puede estar repetido (como en el caso de *Ciudad Real* o *Cuenca*).

Como hemos comentado anteriormente, la *Estrategia* define el orden de expansión de los nodos del árbol. Diferentes estrategias ofrecen diferentes niveles aproximaciones a la construcción del árbol de búsqueda y puede dar lugar a soluciones que no sean óptimas (como vemos, el algortimo general estudiado en el listado 1, devuelve como solución el camino hasta el primer nodo cuyo estado cumple el *Test-Objetivo*). A continuación veremos algunas de los estrategias clásicas empleadas en la creación del árbol.

Una **solución** en problemas de búsqueda puede definirse como una **secuencia de acciones** que dirigen al sistema desde un **estado inicial** hasta un estado que cumpla un determinado **test objetivo**.

Una **Estrategia** queda definida por el orden de expansión de los nodos hoja del árbol de búsqueda (denominada *Frontera*, que está formada por los nodos que aún no han sido expandidos). Cada Estrategia se puede evaluar en base a cuatro parámetros:

1. **Completitud**. Si existe solución, ¿la *estrategia* la encuentra?
2. **Optimalidad**. De entre las soluciones posibles, ¿la *estrategia* garantiza que encuentra la mejor solución (o solución óptima en base a la función de coste definida)?
3. **Complejidad Temporal**. ¿Cuál es el orden de complejidad temporal (número de nodos a analizar para construir la solución)?.

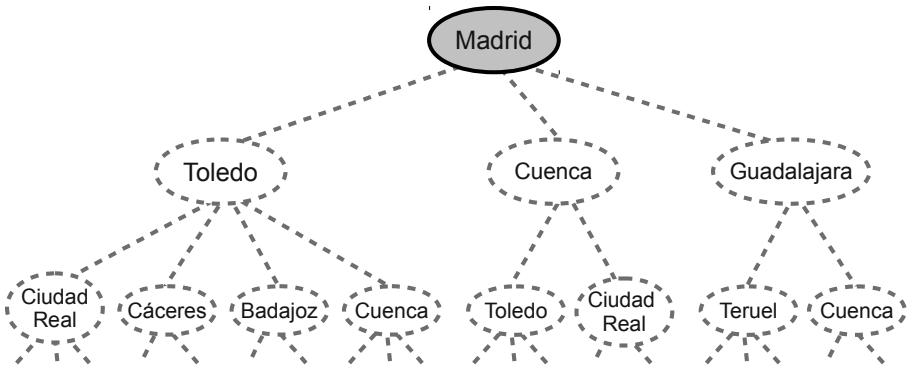


Figura 3.34: Definición parcial del árbol de búsqueda en el problema de viajar de Madrid a Córdoba pasando por capitales de provincia.

4. **Complejidad Espacial**. ¿Cuántos nodos tienen que mantenerse en memoria para conseguir construir la solución?.

Las estrategias de expansión pueden ser categorizadas en base a la información empleada en la construcción del árbol de búsqueda. En una primera taxonomía, podemos distinguir dos grandes familias de técnicas:

- **Estrategias Uniformes**. También denominadas técnicas de *Búsqueda a Ciegas*, donde no se dispone de información adicional; únicamente contamos con la información expresada en la propia definición del problema. De entre las estrategias de búsqueda a ciegas más empleadas se encuentran la búsqueda *en Anchura* y la búsqueda *en Profundidad*.
- **Búsqueda Informada**. Utilizan información adicional para dar una valoración sobre la *apariencia* de cada nodo, aportando información extra que puede utilizarse para *guiar* la construcción del árbol de búsqueda. Algunas de las estrategias más empleadas de búsqueda informada se encuentran los algoritmos *Voraces* y el *A**.

La **Búsqueda en Anchura** (ver Figura 3.35) expande el nodo más antiguo de la *Frontera*. Así, la *Frontera* se maneja como una cola FIFO. Esta estrategia es completa, no da una solución óptima y tiene una gran complejidad espacial (mantiene todos los nodos en memoria).

La **Búsqueda en Profundidad** (ver Figura 3.36) expande el nodo más moderno de la *Frontera*, manteniendo una estrutura tipo Pila. Aunque

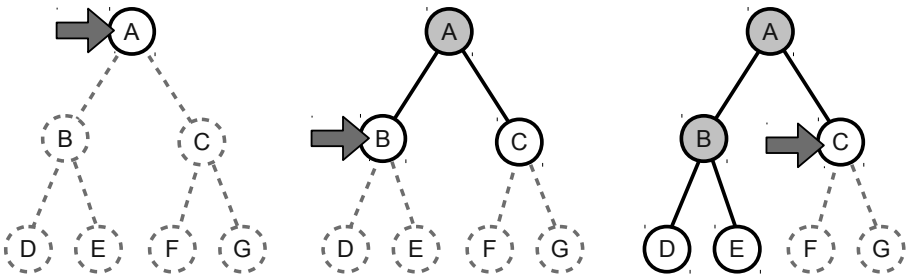


Figura 3.35: Ejemplo de Estrategia de Expansión del árbol en Anchura.

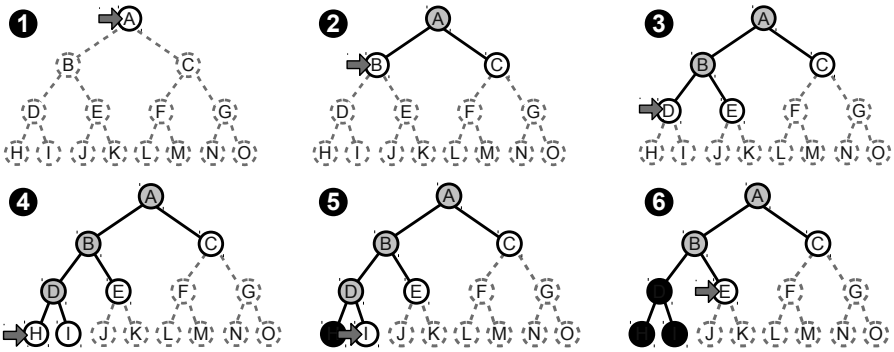


Figura 3.36: Algunas etapas de expansión en la Búsqueda en Profundidad. Los nodos marcados en color negro (por ejemplo, en el paso 5 y 6) son liberados de la memoria, por lo que la complejidad espacial de esta estrategia es muy baja.

tiene una baja complejidad espacial, cuenta con una gran complejidad temporal, es igualmente una estrategia completa aunque no óptima.

Las **Técnicas de Búsqueda Informada** se basan en la idea de utilizar una función de evaluación de cada nodo $f(n)$. Esta función nos resume la *apariencia* de cada nodo de ser mejor que el resto, de modo que expandimos los nodos de la frontera que parecen ser más prometedores de llevar a una solución.

Los algoritmos **Voraces** ordenan los nodos de la frontera por orden decreciente de la valoración $f(n)$. En el ejemplo de realizar una búsqueda del mejor camino entre Madrid y Córdoba pasando por las capitales de provincia española, podríamos utilizar como función f la distancia en línea recta desde cada capital al destino. Así, en cada nodo tendríamos una estimación de *cómo de bueno* parece ese nodo. La Figura 3.37

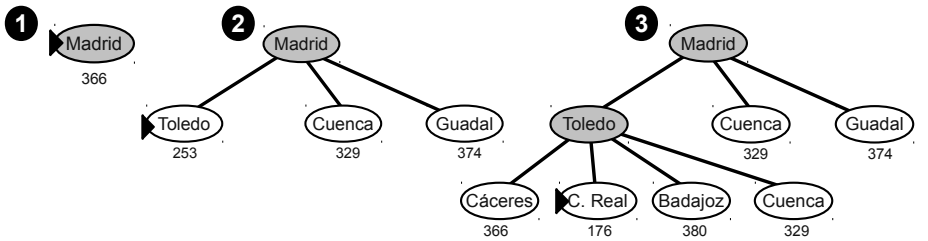


Figura 3.37: Ejemplo de Expansión mediante estrategia Voraz (las distancias entre capitales de provincia son ficticias).

muestra un ejemplo de expansión con valores de distancia ficticios. Vemos que siempre se elige como nodo a expandir aquel de la frontera que tenga menor valor de la función f, expandiendo siempre el nodo que parece estar más cerca del objetivo. Este tipo de búsqueda no es completa, puede quedarse *atascada* en bucles, y puede no dar la solución óptima. Sin embargo, eligiendo una buena función f puede ser una aproximación rápida a ciertos problemas de búsqueda.

Un método de búsqueda ampliamente utilizado en el mundo de los videojuegos es el **algoritmo A*** (que se suele leer como *A Asterisco* o *A Estrella*). La idea base de este método es tratar de evitar expandir nodos que son caros hasta el momento actual. Define una función de evaluación para cada nodo que suma dos términos $f(n) + g(n)$. La función $f(n)$ que indica el coste estimado total del camino que llega al objetivo pasando por n. Por su parte, $g(n)$ mide el coste del camino para llegar a n. Si en la definición de la función $f(n)$ aseguramos que siempre damos un valor menor o igual que el coste real (es decir, la función es *optimista*), se puede demostrar formalmente que A* siempre dará la solución óptima.

En las Figuras 3.38 y 3.39 se describe un ejemplo de uso de A*. En la Figura 3.38 definimos un mapa con ciertas poblaciones por donde batalló Don Quijote. El grafo muestra el coste (ficticio) en Kilómetros entre las poblaciones. Vamos a suponer que queremos viajar desde Ciudad Real al bello paraje de *Las Pedroñeras*. Para ello, definimos como función $f(n)$ el coste en línea recta desde cada estado al destino (recogido en la tabla de la derecha).

La Figura 3.39 resume la aplicación del algoritmo A* en diferentes etapas. Con la definición de mapa anterior, podemos comprobar que el resultado objenido en el paso 6 es óptimo. En cada nodo representaremos el coste total como la suma del coste acumulado $g(n)$ más el coste estimado (en línea recta) hasta el destino $f(n)$.

Así, la evaluación del nodo inicial (paso 1), el coste total es $366 = g(n)0 + f(n)366$. Elegimos ese nodo (el único en la frontera) y expandimos

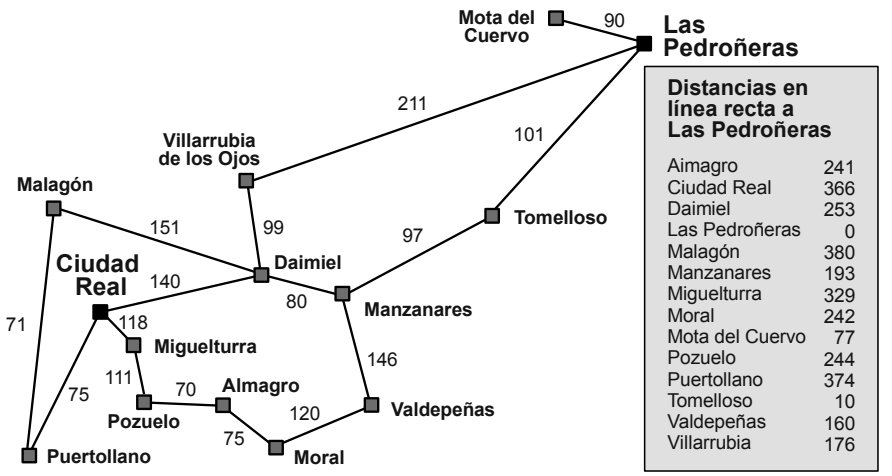


Figura 3.38: Descripción de costes de viajar por la tierra de Don Quijote (distancias ficticias). Ejemplo basado en caso de estudio de [12].

todos sus hijos. Como *Ciudad Real* está conectado con tres poblaciones, tendremos 3 hijos (paso 2). Calculamos el valor de $g(n) + f(n)$ para cada uno de ellos, y los añadimos de forma ordenada (según ese valor) en la frontera. El nodo que tiene menor valor es Daimiel (140 de coste de ir desde Ciudad Real más 253 de coste estimado - en línea recta - hasta llegar a Las Pedroñeras). Como Daimiel está conectado con cuatro poblaciones, al expandir (en 3), añadiremos cuatro nuevos nodos a la frontera de forma ordenada. Ahora la frontera tendría en total 6 nodos (con estados representados en color blanco: C. Real, Villarubia, Malagón, Manzanares, Miguelturra y Puertollano). Al ordenarlos según el valor de la suma de ambas funciones, elegimos *Manzanares* como candidato a expandir. Siguiendo el mismo procedimiento, llegamos en el paso 6 a la solución óptima de *Ciudad Real*, *Daimiel*, *Manzanares*, *Tomelloso* y *Las Pedroñeras*. Basta con recorrer el árbol desde la solución al nodo raíz para componer la ruta óptima. Cabe destacar el mínimo número de nodos *extra* que hemos tenido que expandir hasta llegar a la solución óptima.

En el contexto del desarrollo de videojuegos, las estrategias de búsqueda son realmente relevantes para implementar el módulo de IA de un juego. Sin embargo, resulta esencial considerar la naturaleza del juego cuya IA se desea implementar. Por ejemplo, algunos **juegos clásicos** como las damas, el tres en raya o el ajedrez son casos particulares en los que los algoritmos de búsqueda se pueden aplicar para evaluar cuál es el mejor movimiento en cada momento.

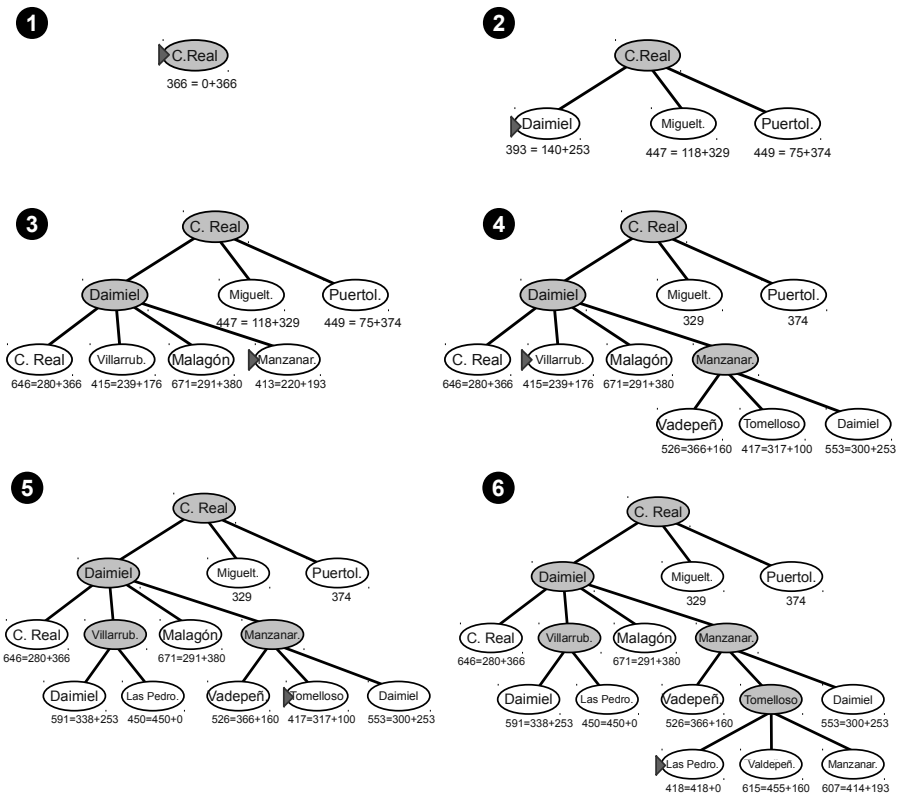


Figura 3.39: Ejemplo de Expansión mediante A* en el viaje de Ciudad Real a Las Pedroñeras.

En los juegos es necesario considerar todas las posibles acciones de réplica del contrincante. Aunque la aplicación de un algoritmo de búsqueda exhaustivo es, generalmente, muy costoso, la evolución del hardware ha posibilitado implementar soluciones de IA que derroten fácilmente a casi cualquier oponente humano. Un caso representativo es *Deep Blue*, una supercomputadora desarrollada por IBM que derrotó en una primera partida al campeón mundial de ajedrez, Gary Kaspárov, en 1997. La implementación del módulo de IA de *Deep Blue* estaba basada en la fuerza bruta, evaluando unos 200 millones de posiciones por segundo[26].

[26]En realidad, la implementación era algo más sofisticada, contando con tablas específicas de jugadas para la finalización y movimientos de inicio de partida clásicos.

En la actualidad, podemos asegurar que prácticamente cualquier juego *clásico* puede ser jugado de una forma muy satisfactoria por una máquina. En las Damas, el sistema *Chinook* ganó en 1994 al campeón mundial *Marion Tinsley* empleando una Base de Datos de jugadas perfectas cuando hay 8 piezas o menos en el tablero (unas 400.000 millones de posiciones). En el Othello (también conocido como Reversi), hace muchos años que los campeones humanos no quieren jugar contra los ordenadores porque son demasiado buenos. Por su parte, en el juego de estrategia para dos jugadores chino Go, los campeones humanos no quieren jugar contra ordenadores porque son demasiado malos[27].

Este gran número de posiciones es consecuencia de la generación del árbol de búsqueda a través del **algoritmo MiniMax**, el cual se discutirá a continuación.

La **estrategia MiniMax** es perfecta para juegos deterministas (donde no interviene el azar). La idea básica es elegir como transiciones entre nodos las acciones que maximizan el valor para el jugador *MAX* y que minimizan para el jugador *MIN*.

La consideración general se basa en tener un juego con los dos jugadores descritos anteriormente. *MAX* mueve primero y, después, mueven por turnos hasta que el juego finaliza. Cuando el juego termina, el ganador recibe una bonificación y el perdedor una penalización. Así, un juego se puede definir como una serie de problemas de búsqueda con los siguientes elementos [12]:

- El **estado inicial**, que define la posición inicial del tablero de juego y el jugador al que le toca mover.
- Una **función sucesor**, que permite obtener una serie de pares <movimiento, estado>, reflejando un movimiento legal y el estado resultante.
- Un **test terminal**, que permite conocer cuándo ha terminado el juego. Los estados en los que el test terminal devuelve un valor lógico verdadero se denominan estados terminales y representan los nodos hoja del árbol de búsqueda.
- Una **función de utilidad** o función objetivo que asigna un valor numérico a los estados terminales (nodos hoja del árbol). Por ejemplo, en el caso de las tres en raya, esta función devuelve un valor +1 a una situación ganadora, un valor de -1 a una situación perdedora y un valor de 0 a una situación de empate (ver Figura 3.40).

[27] No está todo inventado, afortunadamente quedan muchos retos abiertos para seguir trabajando en técnicas de Inteligencia Artificial.

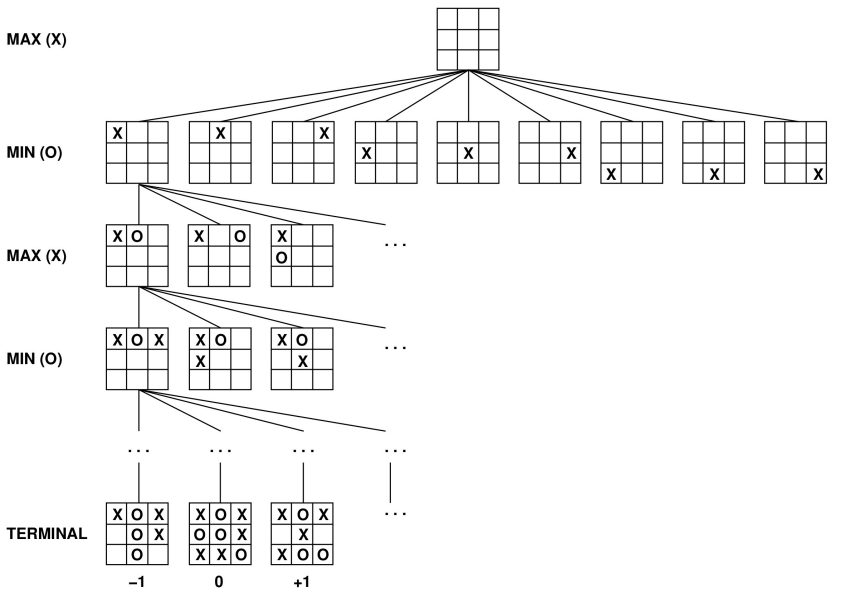


Figura 3.40: Árbol parcial de búsqueda para el juego tres en raya (tic-tac-toe). El nodo superior representa el estado inicial. MAX mueve en primer lugar, colocando una X en una de las casillas vacías. El juego continúa con movimientos alternativos de MIN y MAX hasta alcanzar nodos terminales.

En este contexto, la generación de descendientes mediante la función sucesor a partir del estado inicial permite obtener el **árbol de búsqueda**. La Figura 3.40 muestra una parte del árbol de búsqueda para el caso particular del tres en raya[28]. Note cómo el jugador MAX tiene 9 opciones posibles a partir del estado inicial. Después, el juego alterna entre la colocación de una ficha por parte de MIN (O) y MAX (X), hasta llegar a un estado terminal. En este estado, se evalúa si alguno de los dos jugadores ha ganado o si el tablero está completo. El número asociado a cada estado terminal representa la salida de la función de utilidad para cada uno de ellos desde el punto de vista de MAX. Así, los valores altos son buenos para MAX y negativos para MIN. Este planteamiento, basado en estudiar el árbol de búsqueda, permite obtener el mejor movimiento para MAX. El listado 2 muestra el pseudocódigo de la implementación general de Minimax.

[28]Puede consultar las reglas en http://es.wikipedia.org/wiki/Tres_en_línea

Algoritmo 2 Pseudocódigo del algoritmo Minimax

```
función Decisión-Minimax (estado) devuelve una acción
variables de entrada: estado // estado actual del juego
v ← Max-Valor(estado)
devolver la acción de Sucesores(estado) con valor v

función Max-Valor (estado) devuelve un valor utilidad
if Test-Terminal (estado) then devolver Utilidad(estado)
v ← −∞
while (s en Sucesores (estado)) do
    v ← Máximo (v, Min-Valor(s))
end while
devolver v

función Min-Valor (estado) devuelve un valor utilidad
if Test-Terminal (estado) then devolver Utilidad(estado)
v ← ∞
while (s en Sucesores (estado)) do
    v ← Mínimo (v, Max-Valor(s))
end while
devolver v
```

La función principal *Decisión-Minimax* se encarga de devolver la acción a aplicar ante un determinado *estado* de entrada. La construcción del árbol de juego se realiza empleando dos funciones auxiliares que se llamarán alternativamente: *Max-Valor* que trata de maximizar el valor para el jugador MAX, y *Min-valor* que trata de obtener la jugada que más beneficia al jugador MIN. Comenzamos construyendo el árbol para MAX, pasando ese estado como inicial.

En cada llamada, la función *Max-Valor* primero comprueba si el estado es terminal (fin de partida). En ese caso, ejecuta la función de utilidad sobre el estado y devuelve una valoración (en el caso de las tres en raya, si gana el jugador MAX, devolverá +1). Si no es terminal, estudiamos para todos los sucesores de ese estado, cuál es la valoración más beneficiosa para el jugador MAX. Es decir, estudiamos de entre todos los hijos (expandiendo con *Min-valor*), y nos quedamos con el que nos aporta mayor beneficio.

La implementación de *Min-Valor* es similar, pero eligiendo como mejor valor el que devuelva el mínimo de entre los hijos. En el ejemplo de la Figura 3.41, vemos que el nodo raíz se quedaría con la mayor valoración de sus hijos (que son nodos MIN). A su vez, cada nodo MIN se ha quedado con el mínimo de sus hijos (que son nodos MAX). Los nodos hoja (tipo MAX en este caso) obtienen su valoración mediante la ejecución de la función de utilidad.

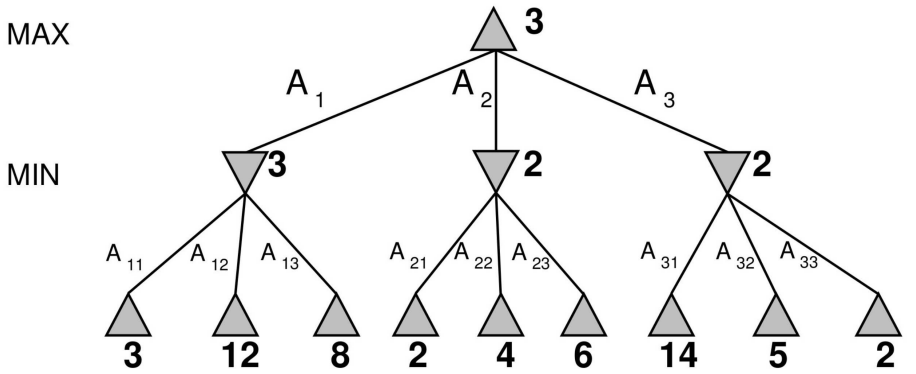


Figura 3.41: Ejemplo de árbol Minimax. La raíz es un nodo Max. Los nodos hoja incluyen el valor de la función utilidad.

Este algoritmo es completo (siempre que el árbol de sea finito), y ofrece una solución óptima. El principal problema es de complejidad [29]. Ante este problema relativo al crecimiento en el número de nodos exponencial, existen aproximaciones que *podan* el árbol de juego, tomando decisiones sin explorar todos los nodos. La poda $\alpha - \beta$ es una técnica de implementación clásica empleada en el algoritmo Minimax, que no afecta a la calidad del resultado final (sigue siendo óptimo). Puede verse como una forma simple de *meta-razonamiento*, donde se estudian qué cálculos son relevantes. La eficacia de la poda depende de la elección ordenada de los sucesores.

3.5.7. Caso de estudio. Un Tetris *inteligente*

En esta sección se discute cómo afrontar el módulo de IA del Tetris en el modo ***Human VS CPU***, es decir, cómo se puede diseñar el comportamiento de la máquina cuando se enfrenta a un jugador real.

Tradicionalmente, el modo *versus* del Tetris se juega a pantalla dividida, al igual que el modo de dos jugadores. En la parte izquierda juega el jugador real y en la derecha lo hace la máquina. En esencia, el perdedor es aquél que es incapaz de colocar una ficha debido a que ha ocupado la práctica totalidad del tablero sin *limpiar* líneas.

Para llevar a cabo la gestión de la dificultad de este modo de juego o, desde otro punto de vista, modelar la habilidad de la máquina, se pueden plantear diversas alternativas. Por ejemplo, si se desea modelar

[29] Por ejemplo, para el caso del ajedrez hablamos de una complejidad temporal de $O(35^100)$

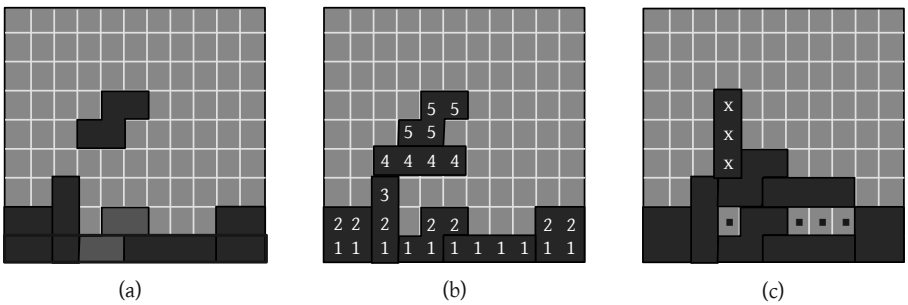


Figura 3.42: Planteando un módulo de IA para el Tetris. **a)** *Limpieza* de una línea, **b)** Cuantificando la altura del montón de fichas, **c)** Problema de huecos perdidos (los puntos representan los huecos mientras que las 'x' representan los potenciales bloqueos).

un nivel de complejidad elevado, entonces bastaría con incrementar la **velocidad** de caída de las fichas. En este caso, la máquina no se vería afectada ya que tendría tiempo más que suficiente para colocar la siguiente ficha. Sin embargo, el jugador humano sí que se vería afectado significativamente.

Otra posibilidad para ajustar el nivel de dificultad consistiría en que el jugador y la máquina recibieran distintos **tipos de fichas**, computando cuáles pueden ser más adecuadas para completar una línea y, así, reducir la altura del montón de fichas. También sería posible establecer un ***handicap***, basado en introducir deliberadamente piezas en la pantalla del jugador real.

No obstante, la implementación de todas estas alternativas es trivial. El **verdadero reto** está en modelar la IA de la máquina, es decir, en diseñar e implementar el comportamiento de la máquina a la hora de ir colocando fichas.

La solución inicial planteada en esta sección consiste en **asignar una puntuación** a cada una de las posible colocaciones de una ficha. Note que las fichas se pueden rotar y, al mismo tiempo, se pueden colocar en distintas posiciones del tablero. El objetivo perseguido por el módulo de IA será el de colocar una ficha allí donde obtenga una mejor puntuación. El siguiente paso es, por lo tanto, pensar en cómo calcular dicha puntuación.

Para ello, una opción bastante directa consiste en distinguir qué aspectos resultan fundamentales para ganar o perder una partida. En principio, el módulo de IA debería evitar formar torres de fichas de gran altura, ya que lo aproximarían a perder la partida de forma inminente,

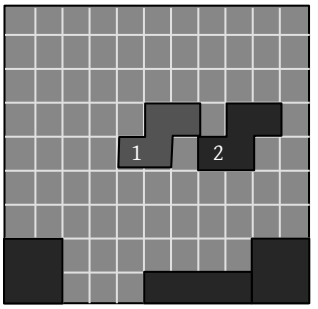


Figura 3.43: Idealmente, la posición 2 debería ser premiada en detrimento de la posición 1, ya que facilita la colocación de futuras piezas. Por lo tanto, esta primera opción debería tener una puntuación superior a la última cuando se realice el cálculo de la *utilidad* de la pieza.

tal y como muestra la figura 3.42.b. Este factor debería suponer una penalización para la puntuación asociada a colocar una ficha. Por el contrario, la máquina debería *limpiar* líneas siempre que fuera posible, con el objetivo de obtener puntos y evitar que el montón de fichas siga creciendo. Este factor representaría una bonificación.

Además, idealmente el módulo de IA debería evitar generar espacios que no se puedan aprovechar por las fichas siguientes, es decir, debería evitar que se perdieran huecos, tal y como se refleja en la figura 3.42.c. Evidentemente, la generación de huecos perdidos es, a veces, inevitable. Si se produce esta situación, el módulo de IA debería evitar la colocación de fichas sobre dichos huecos, con el objetivo de liberarlos cuanto antes y, en consecuencia, seguir *limpiando* líneas.

Estos cuatro factores se pueden ponderar para construir una **primera aproximación** de la fórmula que se podría utilizar para obtener la puntación asociada a la colocación de una ficha:

$$P = w_1 * salt + w_2 * nclears + w_3 * nh + w_4 * nb \tag{3.1}$$

donde

- $salt$ representa la suma de las alturas asociada a la pieza a colocar en una determinada posición,
- $nclears$ representa el número de líneas *limpiadas*,
- nh representa el número de huecos generados por la colocación de la ficha en una determinada posición,
- nb representa el número de bloqueos como consecuencia de la potencial colocación de la ficha.
- w_i representa el peso asociado al factor i.

Evidentemente, $nclears$ tendrá asociado un peso positivo, mientras que el resto de factores tendrán asociados pesos negativos, ya que representan penalizaciones a la hora de colocar una ficha.

Un inconveniente inmediato de esta primera aproximación es que no se contempla, de manera explícita, la construcción de **bloques consistentes** por parte del módulo de IA de la máquina. Es decir, sería necesario incluir la lógica necesaria para premiar el acoplamiento entre fichas de manera que se premiara de alguna forma la *colocación lógica* de las mismas. La figura 3.43 muestra un ejemplo gráfico de esta problemática, cuya solución resulta fundamental para dotar a la máquina de un comportamiento *inteligente*.

3.5.8. Caso de estudio. Tres en raya (tic-tac-toe) con NME

En esta sección se discuten los aspectos relacionados con el módulo de IA de un juego tres en raya o tic-tac-toe. Básicamente, la implementación de dicho módulo es una variación del **algoritmo minimax** discutido en la sección 3.5.6. Este juego de dos jugadores es un ejemplo sencillo, ya que es fácil averiguar que el juego perfecto siempre finaliza en empate, independientemente de quién comience, que permite ilustrar adecuadamente la implementación del módulo de IA.

La clase Board

Antes de discutir la implementación del algoritmo minimax es importante discutir la clase *board* o tablero, la cual almacena el **estado de juego** y permite reflejar la evolución del mismo. Esta clase, tal y como muestra el siguiente listado de código, contiene un array de elementos de tipo *TPlayer* (línea 13) que permite representar el estado de cada casilla. Dicho estado queda definido por el tipo enumerado *TPlayer* (línea 5), el cual alberga los valores *Empty* (casilla vacía), *Player_O* (jugador humano) y *Player_X* (máquina).

En esta solución se ha optado por utilizar un array unidimensional en lugar de uno bidimensional debido a que simplifica la implementación de la lógica del juego. De este modo, la longitud de este array será de 9, es decir, mantiene casillas desde la número 0 hasta la número 8.

Listado 3.57: Clase Board. Variables miembro y constructor.

```
1  // Enumeración para las opciones disponibles en una casilla.
2  // Empty ---->Casilla vacía.
3  // Player O -->Casilla con O.
4  // Player X -->Casilla con X.
5  enum TPlayer { Empty; Player_O; Player_X; }
6
7  class Board {
8
9    // Estado del tablero.
10   // - X O
11   // X X O
12   // - O -
13   private var _pieces:Array<TPlayer>;
14
15   public function new () {
16     _pieces = new Array<TPlayer>();
17     for (i in 0...9)
18       _pieces[i] = Empty;
19   }
20
21   // ...
22
23 }
```

De este modo, las acciones de **efectuar y deshacer movimientos** son triviales, como se expone a continuación.

Listado 3.58: Clase Board. Funciones makeMove() y undoMove().

```
1  // No comprueba si el movimiento es válido.
2  // move representa la casilla [0..8].
3  // player representa al jugador.
4  public function makeMove (move:Int, player:TPlayer) : Void {
5    _pieces[move] = player;
6  }
7
8  // Deshace un movimiento previamente realizado.
9  public function undoMove (move:Int) : Void {
10   _pieces[move] = Empty;
11 }
```

La clase *Board* también contiene la lógica necesaria para detectar si un jugador ha ganado. Para ello, la función *getWinner()* comprueba si alguna de las filas, columnas o diagonales contiene todas sus casillas del mismo tipo (siempre que no estén vacías).

Listado 3.59: Clase Board. Función getWinner().

```
1  // Devuelve, si existe, el ganador de una partida
2  // considerando el estado actual del tablero.
3  // Si no hay ganador, devuelve Empty.
4  public function getWinner () : TPlayer {
5    var winningStates:Array<Array<Int>> =
6      [
7        [0, 1, 2], [3, 4, 5], [6, 7, 8],
8        [0, 3, 6], [1, 4, 7], [2, 5, 8],
9        [0, 4, 8], [2, 4, 6],
10      ];
11
12   for (st in winningStates) {
13     var aux:Array<TPlayer> = [_pieces[st[0]],
14                   _pieces[st[1]],
15                   _pieces[st[2]]];
16     if (_pieces[st[0]] != Empty && allEqual(aux))
17       return _pieces[st[0]];
18   }
19   return Empty;
20 }
```

Por otra parte, otras dos funciones interesantes son las que permiten obtener la lista de movimientos válidos (función *getValidMoves()* en líneas 3-10) y comprobar si el juego ha terminado (función *isGameOver()* en líneas 15-17), respectivamente. La primera de ellas es trivial, ya que devuelve una lista con los números de las casillas que están vacías, es decir, sin ficha. La segunda comprueba dos casos: i) si no existen más movimientos válidos o ii) si ya hay un ganador.

Listado 3.60: Clase Board. Funciones getValidMoves() y isGameOver().

```
1  // Devuelve una lista que contiene las casillas
2  // con los movimientos posibles.
3  public function getValidMoves () : Array<Int> {
4    var validMoves:Array<Int> = new Array<Int>();
5    for (i in 0...9)
6      if (_pieces[i] == Empty)
7        validMoves.push(i);
8
9    return validMoves;
10 }
11
12 // Comprueba si el juego ha terminado...
13 // ¿Existe algún movimiento válido?
14 // ¿Hay un ganador?
15 public function isGameOver () : Bool {
16   return (getValidMoves().length == 0 || getWinner() != Empty);
17 }
```

La clase *Board* se utiliza tanto en la clase *Minimax*, la cual se discute a continuación, con en la clase *TicTacToe*, la cual es la clase principal

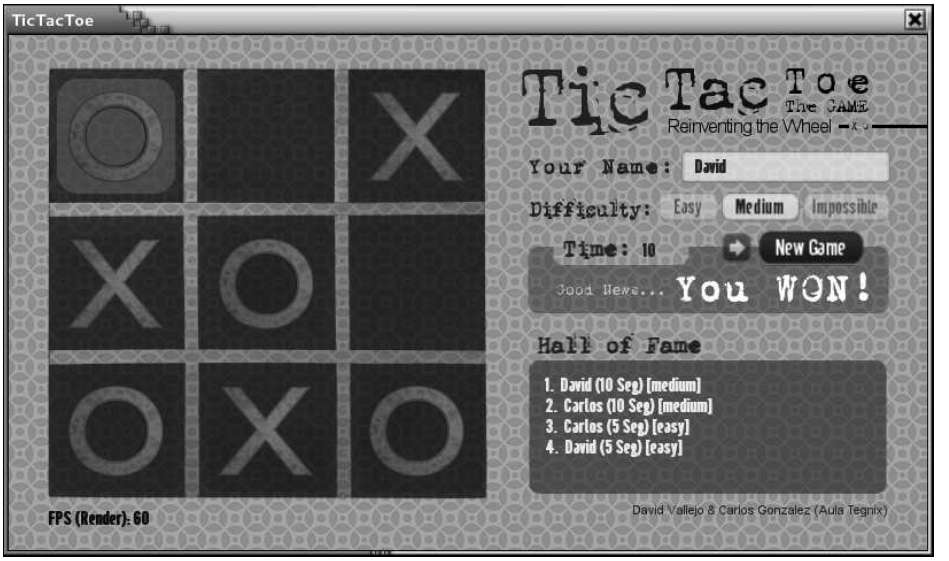


Figura 3.44: Captura de pantalla del juego Tic-Tac-Toe.

del juego y que se encarga de la gestión de toda la parte gráfica y de eventos.

La clase Minimax

La clase *Minimax* encapsula la funcionalidad asociada a una variante del algoritmo minimax, denominada negamax, que simplifica la lógica de dicho algoritmo. En particular, esta variante de búsqueda está basada en la propiedad *zero-sum*, la cual se basa en la siguiente propiedad:

$$max(a, b) = -min(-a, -b)$$

es decir, el valor de una posición para el jugador A es la negación del valor para el jugador B. De este modo, el jugador al que le toca mover busca un movimiento que maximice la negación del valor resultante de dicho movimiento. Esta posición sucesora debe ser, por definición, evaluada por el oponente. Este planteamiento garantiza la independencia con respecto al jugador que mueve, por lo que la implementación sirve de manera independiente al jugador al que le toca jugar en cada momento.

La simplificación del **algoritmo negamax** requiere que el jugador A seleccione el movimiento con el sucesor de máximo valor mientras que el jugador B elija el movimiento con el sucesor de menor valor.

El siguiente listado de código muestra la implementación de la función *buildTreeRec()*. Está función tiene un **planteamiento recursivo** y su principal responsabilidad es **construir el árbol de búsqueda** hasta una profundidad *depth* (pasada como parámetro) y devolver la mejor puntuación calculada.

Antes de pasar a discutir dicho código, es importante resaltar cómo se modela la **dificultad de la máquina** cuando un jugador se enfrenta a la misma. Actualmente, existen tres niveles de dificultad: *easy*, *medium* e *impossible*. Estos se modelan a través del nivel de recursión del algoritmo minimax. Actualmente, *easy* se modela con un nivel de recursión igual a 1, *medium* con un nivel de 3 y *impossible* con un nivel de 6. Como el lector podrá comprobar, no es posible vencer a la máquina en nivel de dificultad *impossible*.

En primer lugar, esta función contempla al principio el cambio de turno en las líneas 8–10 para que cuando, posteriormente, se realice una llamada recursiva (línea 31), entonces se llame con el jugador contrario. Así mismo, al principio se controlan los casos base de la recursividad, basados en detectar si existe algún ganador o si el juego quedó en empate (previamente se controla el nivel máximo de profundidad en la línea 23):

- El jugador actual gana y la función devuelve +INFINITO (línea 13).
- El oponente gana y la función devuelve -INFINITO (línea 14).
- Hay un empate y la función devuelve 0 (línea 15).

A continuación, la implementación planteada recupera la lista de posibles movimientos, denominada *movelist*, para el estado actual en la línea 18 y maneja una lista paralela, denominada *salist*, para almacenar la evaluación de cada uno de ellos, como se aprecia en la línea 22. En la variable *alpha* de la línea 20 se almacena la mejor puntuación y, por ese motivo, se inicializa a -INFINITO.

Como se ha mencionado anteriormente, el algoritmo minimax es un algoritmo recursivo. Precisamente, las siguientes líneas de código evalúan, de manera recursiva, las **opciones existentes** en la lista de posibles movimientos. El bucle for de las líneas 45–57 permite llevar a cabo tal tarea. La idea es sencilla, simular el movimiento en la línea 49 y cambiar el turno en la línea 51, efectuando una llamada recursiva y negando el signo del resultado obtenido, como ya se ha discutido en esta sección.

Listado 3.61: Clase Minimax. Función buildTreeRec().

```
1  // Construye recursivamente el árbol hasta una profundidad depth.
2  // Devuelve la mejor puntuación.
3  public function buildTreeRec (board:Board, currentPlayer:TPlayer,
4                    depth:Int) :Int {
5    // Para en profundidad maxDepth.
6    if (depth > _maxDepth) return 0;
7
8    // Para el cambio de turno...
9    var otherPlayer:TPlayer;
10   if (currentPlayer == Player_O) otherPlayer = Player_X;
11   else                           otherPlayer = Player_O;
12
13   var winner:TPlayer = board.getWinner();
14   if (winner == currentPlayer)   return INFINITY;
15   if (winner == otherPlayer)     return -INFINITY;
16   if (board.isGameOver())        return 0;
17
18   // Lista de posibles movimientos.
19   var movelist:Array<Int> = board.getValidMoves();
20   // Mejor puntuación.
21   var alpha:Int = -INFINITY;
22   // Lista con resultados (paralela a movelist)
23   var salist:Array<Int> = new Array<Int>();
24
25   // Obtiene los mejores resultados y asocia el mejor movimiento.
26   for (i in movelist) {
27     // Copia para evaluar el movimiento i.
28     var board_copy:Board = board.copy();
29     // Movimiento potencial.
30     board_copy.makeMove(i, currentPlayer);
31     // Cambio de turno (y de signo).
32     var subalpha:Int = -buildTreeRec(board_copy,
33                       otherPlayer,
34                       depth + 1);
35     if (alpha < subalpha) alpha = subalpha;
36
37     // Añade resultado después de la evaluación.
38     if (depth == 0) salist.push(subalpha);
39   }
40
41   // Si llega a profundidad 0 y se han explorado todos los
42   // sub-árboles, estudia la lista de mejores movimientos
43   // y se elige uno de ellos al azar para jugar.
44   if (depth == 0) {
45     var candidates:Array<Int> = new Array<Int>();
46     for (i in 0...salist.length)
47       if (salist[i] == alpha)
48     candidates.push(movelist[i]);
49     _bestMove = candidates[Std.random(candidates.length)];
50   }
51
52   // Devuelve la mejor puntuación.
53   // En bestMove se almacena la mejor opción.
54   return alpha;
55 }
```

Note el cambio de turno mediante la variable *otherPlayer* y la actualización del nivel de profundidad con la expresión *depth + 1*. A la vuelta de la recursividad se comparan los valores resultantes de evaluar los situaciones posibles, de manera que cada jugador se quede con la mejor (almacena su valor en *alpha*).

Finalmente, cuando se llega a profundidad 0 (líneas 62-68), habiendo estudiado los sub-árboles, se obtiene la lista con los **mejores movimientos** y se elige uno de ellos al azar. A continuación, se actualiza la variable miembro *_bestMove* (línea 67), para la siguiente jugada, y se devuelve la mejor puntuación (línea 72).

El siguiente listado muestra la llamada inicial a *buildTreeRec()*. Note cómo esta función devuelve el mejor movimiento de los realmente evaluados (línea 5).

Listado 3.62: Clase Minimax. Función buildTree().

```
1 public function buildTree (board:Board, currentPlayer:TPlayer) : Int
      {
2   _bestMove = -1;
3   // Llamada a la función recursiva.
4   var alpha:Int = buildTreeRec(board, currentPlayer, 0);
5   return _bestMove;
6 }
```

La llamada desde el **código de Tic-Tac-Toe** se realiza a través de la función *makeMovement()* de la clase *TicTacToe*. La generación del árbol de búsqueda se realiza a través de la llamada *buildTree()* en la línea 2. Ésta llamada sólo se realiza a la hora de efectuar el movimiento inicial (ver valor por defecto del parámetro *move* en la línea 1).

Listado 3.63: Clase TicTacToe. Función makeMovement().

```
1 function makeMovement(player:TPlayer, move:Int=-9):Void {
2   if (move == -9) move = _miniMax.buildTree(_board, player);
3   var moves:Array<Int> = _board.getValidMoves();
4   if (Lambda.has(moves, move)) {
5     _board.makeMove(move, player);
6     updateTab(move, player);
7     changeTurn();
8   }
9 }
```

3.6. Networking

3.6.1. Introducción

La evolución de Internet como sistema global de comunicaciones ha tenido un **impacto muy significativo en la creación de videojuegos**. De hecho, hoy en día se puede afirmar que la mayoría de los juegos que se distribuyen incorporan algún tipo de funcionalidad en red o características *online* que permiten, entre otras cosas, jugar con amigos al mismo juego desde distintas ubicaciones físicas. En concreto, Internet ha permitido, entre otros aspectos, integrar en el mundo de los videojuegos posibilidades como las siguientes:

- Aparición de videojuegos con modos de juego exclusivos para múltiples jugadores en red, como por ejemplo *World of Warcraft*™.[30]
- Distribución masiva de juegos, considerando especialmente el caso de las plataformas móviles, como por ejemplo los *smartphones*.
- Actualización automática de juegos para garantizar el contenido más actual o la correción de *bugs* existentes.
- Aparición de nuevos modelos de negocio basados en la distribución de juegos gratuitos y la posterior monetización a través de sistemas de publicidad.

Evidentemente, el juego en red introduce una **complejidad adicional** a la hora de desarrollar un videojuego. Si, por ejemplo, dos amigos juegan *online* al mismo juego, entonces los dos deberían tener una representación lo más similar posible del estado del juego para poder jugar de una manera adecuada. En otras palabras, es desable minimizar el tiempo que transcurre desde que el jugador A interactúa con el juego (por ejemplo, moviendo su personaje) hasta que el jugador B observa el resultado de dicha interacción (por ejemplo, observar al personaje en la posición actual).

En esta sección se introducen algunas consideraciones de diseño especialmente relevantes y se discute el uso de **sockets** como herramienta básica de comunicación en juego en red. Posteriormente, se estudia una posible solución de integración de funcionalidad básica *online* con **Haxe y NME**.

[30] http://es.wikipedia.org/wiki/World_of_Warcraft

3.6.2. Consideraciones iniciales de diseño

Desde el punto de vista de la red, los juegos multi-jugador que se desarrollan en tiempo real, esto es, varios jugadores jugando de forma simultánea, son los más exigentes. Esto se debe a que su diseño y desarrollo tiene que considerar diversas cuestiones:

- **Sincronización**. La identificación de las acciones que ocurren en juegos de tiempo real requiere de una gran eficiencia para proporcionar al usuario una buena experiencia. En este contexto, se debe diseñar el sistema de transmisión y sincronización así como los turnos de red para que el juego sea capaz de evolucionar sin problemas.

- **Gestión de actualizaciones**. La identificación de las actualizaciones de información y la dispersión de dicha información es esencial desde el punto de vista del jugador para que, en cada momento, todos las partes involucradas tengan la información necesaria y actualizada del resto.

- **Determinismo**. La consistencia es crítica para asegurar que todas las partes del videojuego disponen de la misma información.

Generalmente, el **diseño del modo multi-jugador** de un juego se aborda desde el principio [3]. En otras palabras, el modo *online* de un juego no suele integrarse como un añadido más en las últimas etapas de desarrollo, sino que representa un aspecto crucial que se aborda desde el principio. De hecho, una práctica bastante extendida consiste en abordar el modo *single player* o de un único jugador (*offline*) como un caso particular de la versión multi-jugador del juego.

En este contexto, se debe identificar, desde las primeras fases del diseño del juego, qué información se va a distribuir para que todas las partes involucradas tengan la información necesaria que garantice la correcta evolución del juego.

Al más bajo nivel, es necesario diseñar un protocolo de comunicaciones que, típicamente mediante el uso de sockets, permita transmitir toda la información necesaria en el momento oportuno. Este protocolo debe definir idealmente los siguientes aspectos:

- **Sintaxis**: qué información y cómo se estructura la información a transmitir. Esta especificación define la estructura de los mensajes a transmitir y recibir, su longitud, los campos de los mensajes, etc. El resultado de esta fase de diseño debe ser una serie de estructuras a transmitir y recibir a través de un punto de comunicación.

- **Semántica**: qué significa la información transmitida y cómo interpretarla una vez recibida. La interpretación de la información se realiza mediante el *parsing* o procesamiento de los mensajes transmitidos e interpretando la información recibida.
- **Temporización**: el modelo de temporización expresa la secuencia de mensajes que se deben recibir y transmitir en función de los mensajes recibidos y enviados con anterioridad, teniendo en cuenta la información que se necesite transmitir. La temporización y la semántica generalmente se traducen en una máquina de estados cuyas transiciones vienen determinadas por los mensajes recibidos, los mensajes transmitidos y la información contenida en los mismos.

El protocolo de un videojuego debe especificar la sintaxis, semántica y temporización.

Un aspecto determinante en el diseño del modo multi-jugador consiste en definir **qué información es necesaria transmitir**. Esta pregunta es específica del videojuego a desarrollar y, por lo tanto, la respuesta variará de un juego a otro. Sin embargo, de forma genérica, se han de identificar los siguientes aspectos:

- Aquella información vinculada al estado de una entidad, como por ejemplo su posición en el espacio 3D y su orientación.
- Información relativa a la lógica del juego, siempre y cuando sea necesario su transmisión en red.
- Aquellos eventos que un jugador puede realizar y que afectan al resto de jugadores y al estado del propio juego.

La eficiencia en la transmisión y procesamiento de la información de red, así como el objetivo de minimizar la información transmitida, deben guiar el diseño y la implementación de la parte de *networking*.

De forma paralela al diseño del protocolo de comunicación de un juego, es necesario decidir la **estructura lógica** de la parte de *networking*. Generalmente existen las siguientes alternativas viables:

- **Arquitectura P2P** (*peer-to-peer*): en este modelo cada usuario comparte la información con todos los usuarios integrados en una partida en curso.
- **Arquitectura cliente-servidor**: en este modelo todos los usuarios envían la información a un servidor central que se encarga de redistribuirla.

A continuación se discute el caso particular de los sockets TCP/IP como herramienta básica para enviar y recibir información.

3.6.3. Sockets TCP/IP

La programación con sockets es la programación en red de más bajo nivel que un desarrollador de videojuegos generalmente realizará. Por encima de los sockets básicos, que se discutirán a continuación, es posible utilizar bibliotecas y *middlewares* de comunicaciones de más alto nivel. Sin embargo, estas herramientas aportan un nivel más alto de abstracción aumentando, a priori, la productividad y la calidad del código a costa, en algunas ocasiones, de una pérdida de eficiencia.

Conceptualmente, un socket es un **punto de comunicación** con un proceso que permite comunicarse con él utilizando, en función del tipo de socket utilizado, una serie de protocolos de comunicación. En función del rol que asuma el proceso, se pueden distinguir dos tipos de programas:

- **Cliente**, entidad que solicita a un servidor un servicio.
- **Servidor**, entidad atiende peticiones de los clientes.

En el **contexto de NME** y el lenguaje de programación Haxe existen diversas opciones desde el punto de vista de la implementación de un sistema cliente/servidor basado en sockets. Un ejemplo muy interesante está representado por la clase *ThreadServer*[31], la cual se puede utilizar en Haxe para generar la parte servidora en Neko.

Neko es un lenguaje de programación y un entorno de ejecución creado por Nicolas Cannase y que está soportado por Haxe. En otras palabras, Haxe puede generar *bytecode* para Neko. De hecho, cuando NME dio sus primeros pasos, fue diseñado para proporcionar soporte gráfico, sonido y otra funcionalidad multimedia para la máquina virtual de Neko. Sin embargo, el rendimiento no fue el esperado inicialmente y, a partir de ahí, surgió C++ como *target* por defecto para aplicaciones de escritorio desarrolladas con Haxe.

[31] http://haxe.org/doc/neko/threadserver

La **clase ThreadServer** se puede utilizar para crear, de una manera sencilla, servidores multi-hilo en los que cada hilo puede manejar un número arbitrario de peticiones o sockets. Sin embargo, el código necesario para el tratamiento de dichas peticiones se encapsula en un único hilo, simplificando el mismo y evitando así la integración de mecanismos de sincronización para evitar inconsistencias.

Por otra parte, esta clase se puede parametrizar para trabajar con distintos tipos de datos, típicamente asociados a las estructuras *cliente* y *mensaje*. El siguiente listado de código muestra las estructuras *Client* (línea 1-3) y *Message* (líneas 5-7), las cuales encapsulan, respectivamente, el identificador del cliente y el contenido del mensaje.

Listado 3.64: Clase Server. Tipos de datos.

```
1 typedef Client = {
2   var id : Int;
3 }
4
5 typedef Message = {
6   var str : String;
7 }
```

Para poder utilizar esta clase, al menos es necesario sobreescribir las funciones *clientConnected()*, *readClientMessage()* y *clientMessage*, además de definir las estructuras *Client* y *Message*. Todas estas funciones están afectadas por el modificador *dynamic* con el objetivo de garantizar la interoperabilidad con distintas plataformas.

El siguiente listado muestra una posible implementación de la función *clientConnected()*. Básicamente, esta función genera un identificador aleatorio para el cliente conectado y muestra información de la conexión.

Listado 3.65: Clase Server. Función clientConnected().

```
1 class Server extends ThreadServer<Client, Message> {
2
3   // Crea un nuevo cliente.
4   override function clientConnected ( s : Socket ) : Client {
5     var num = Std.random(100);
6     Lib.println("Cliente " + num + " desde " + s.peer());
7     return { id: num };
8   }
9
10  // Restos de funciones...
11
12 }
```

A continuación, la función *readClientMessage()* es la responsable de devolver una dupla formada por el contenido del mensaje y el tamaño del mismo. Note cómo en las líneas 8-11 se utiliza un bucle para leer el contenido del mensaje de manera incremental.

Listado 3.66: Clase Server. Función readClientMessage().

```
1  // Lee el mensaje de un cliente concreto.
2  override function readClientMessage (c:Client,
3                      buf:Bytes,
4                      pos:Int, len:Int) {
5    // ¿Mensaje completo?
6    var complete = false;
7    var cpos = pos;
8    while (cpos < (pos+len) && !complete) {
9      complete = (buf.get(cpos) == 46);
10     cpos++;
11   }
12
13   // Si no llega un mensaje completo, readClientMessage retorna.
14   if( !complete )
15     return null;
16
17   // Devuelve el mensaje completo.
18   var msg:String = buf.readString(pos, cpos-pos);
19   return {msg: {str: msg}, bytes: cpos-pos};
20 }
```

Finalmente, la función *clientMessage()* es la responsable de gestionar el mensaje recibido por parte del servidor. En el siguiente listado se muestra una implementación trivial en la que simplemente se imprime dicho mensaje.

Listado 3.67: Clase Server. Función clientMessage().

```
1 // Gestiona el mensaje del cliente.
2 override function clientMessage ( c : Client, msg : Message ) {
3   Lib.println(c.id + " envio: " + msg.str);
4 }
```

Como se introdujo anteriormente, los sockets representan puntos de comunicación entre pares. La información básica para establecer estos puntos de comunicación consiste en definir la dirección del host y el puerto de escucha. En el siguiente listado se muestra la instanciación de un objeto de la clase *Server*, la cual hereda de *ThreadServer*. Note cómo es necesario invocar a la función *run()* para indicar dicha información.

Listado 3.68: Clase Server. Función main().

```
1 public static function main() {
2   var server:Server = new Server();
3   server.run("localhost", 1234);
4 }
```

En el lado del cliente, el código es mucho más sencillo ya que sólo es necesario invocar la funcionalidad de la parte servidora.

Listado 3.69: Clase Cliente.

```
1  class Client {
2
3    public static function main () {
4        Lib.println("Abriendo socket...");
5        var sock = new Socket();
6        sock.connect(new Host("localhost"), 1234);
7
8        Lib.println("Enviando mensajes...");   Sys.sleep(.1);
9        sock.write("Mensaje de test.");        Sys.sleep(.3);
10       sock.write("Otro mensaje de test.");   Sys.sleep(.3);
11       sock.write("Y otro mensaje de test.");
12
13       sock.close(); Lib.println("Cliente finalizado.");
14   }
15
16 }
```

Para compilar el código fuente, tanto del cliente como del servidor, para Neko es necesario ejecutar los siguientes comandos:

```
$ haxe -neko server.n -main Server
$ haxe -neko client.n -main Client
```

Para ejecutar el servidor, es necesario ejecutar el siguiente comando:

```
$ neko server.n
```

Del mismo modo, para ejecutar el cliente, es necesario ejecutar el siguiente comando:

```
$ neko client.n
```

Una posible salida del servidor es la siguiente:

```
Cliente 13 desde { host => 127.0.0.1, port => 56216 }
13 envio: Mensaje de test.
13 envio: Otro mensaje de test.
13 envio: Y otro mensaje de test.
Cliente 13 desconectado
```

Una posible salida del cliente es la siguiente:

```
Abriendo socket...
Enviando mensajes...
Cliente finalizado.
```

3.6.4. Gestión on-line de récords en Tic-Tac-Toe

En esta sección se discute la integración de un servidor de registro de récords para el juego de tres en raya o tic-tac-toe. El principal objetivo perseguido es el de estudiar cómo se puede **integrar funcionalidad on-line** mediante el uso de de sockets. El término *récord* se ha utilizado en este caso de estudio para reflejar la situación en la que el jugador gana a la máquina, de manera independiente a la dificultad y tiempo empleado. En este contexto, el servidor será responsable de manejar, como se discutirá más adelante, una lista ordenada con los récords registrados.

Por una parte, se ha implementado un servidor utilizando Haxe y la clase *ThreadServer* proporcionada en la API de Neko. Este servidor es una versión adaptada del servidor discutido en la sección anterior.

Por otra parte, en el propio juego se ha incluido un **módulo de comunicaciones** que tiene como principal responsabilidad tanto el envío de récords al servidor como la obtención de las mejores puntuaciones. Dichas puntuaciones aparecen en la ventana principal del propio juego.

Servidor de récords

Como se ha comentado anteriormente, el servidor de récords es una versión modificada del servidor de la sección 3.6.3, prestando especial atención a los tipos de mensajes necesarios para comunicar el juego con el servidor y viceversa.

El siguiente listado de código muestra la implementación de la función *clientMessage()*, responsable de procesar el mensaje recibido de un cliente. En función de su contenido se distingues dos casos posibles:

- Mensaje de **solicitud de récords**, identificado por el servidor de manera sencilla si el propio mensaje comienza por el token *request* (líneas 14–18). En este caso, el mensaje recibido tendrá el formato $request : host : puerto$ para que el servidor pueda extraer tanto el host como el puerto del cliente y establecer un socket para enviar el listado de récords (ver función *sendRecords()* en la línea 17).
- Mensaje de **registro de un récord**, el cual contiene la información del nuevo récord de acuerdo al formato $jugador : dificultad : tiempo$ (líneas 20–24).

Listado 3.70: Clase Server. Función clientMessage().

```
1  class Server extends ThreadServer<Client, Message> {
2
3    private var _records:Array<String>;
4
5    // Resto de código...
6
7    // Gestiona el mensaje del cliente.
8    override function clientMessage (c : Client, msg : Message) : Void
         {
9      var msg_rec = msg.str.substr(0, msg.str.length - 1);
10     var tokens = msg_rec.split(":");
11
12     // Petición de récords.
13     // msg ->request:host:port
14     if (tokens[0] == "request") {
15       Lib.println(c.id + " solicita el registro de rcords.");
16       // Envía records actualizados al cliente que envió el suyo.
17       sendRecords(tokens[1], Std.parseInt(tokens[2]));
18     }
19     // Registro de un nuevo récord.
20     else {
21       Lib.println(c.id + " registra un nuevo record: " + msg.str);
22       // Añade el récord.
23       addRecord(msg.str.substr(0, msg.str.length - 1));
24     }
25   }
26
27 }
```

La **gestión de récords** se delega en las funciones del siguiente listado. La primera de ellas, *addRecord()* (líneas 2-5), añade el récord recibido como argumento a la lista de récords para, posteriormente, reordenarla. El criterio de ordenación se define en la función *sort()* (línea 4) y basa en:

- Primero los récords con un nivel de dificultad más elevado, es decir, *impossible*, *medium* y *easy*.
- Si el nivel de dificultad es el mismo para dos récords, entonces aparecerá primero el que tenga un tiempo inferior (se derrotó a la máquina en un tiempo menor).

La figura 3.45 muestra una captura de pantalla de la interfaz gráfica del juego en la que se resalta la parte de **visualización de récords**. Como se puede apreciar, el número máximo de récords mostrados está limitado, actualmente, a 5. En este contexto, el servidor maneja una lista con todos los récords registrados, pero sólo enviará a un cliente los n primeros (actualmente 5).

Listado 3.71: Clase Server. Gestión de récords.

```
1  // Añade un nuevo record.
2  public function addRecord (newRecord:String) : Void {
3    _records.push(newRecord);
4    _records.sort(sort);
5  }
6
7  // Envía los récords al cliente.
8  public function sendRecords (host:String, port:Int) : Void {
9    var sock = new Socket();
10   sock.connect(new Host(host), port);
11
12   Lib.println("Enviando records..."); Sys.sleep(.1);
13   sock.write(getUpdatedRecords());
14
15   sock.close();
16 }
17
18 // Devuelve un String con los 5 mejores récords.
19 public function getUpdatedRecords (numRecords:Int = 5) : String {
20   return buildRecordsStr(_records.slice(0, numRecords));
21 }
```

Por otra parte, el **envío de récords** al cliente está gestionado por la función *sendRecords()* (líneas 8-16). En esencia, esta función establece un socket con el cliente del juego y le envía una representación textual del registro actual de récords (ver función *getUpdatedRecords()* en las líneas 19-21).

Antes de discutir la parte de comunicaciones del cliente, es decir, del jugador, es importante resaltar que el servidor ha de estar arrancado para registrar y enviar récords. Para ello, simplemente es necesario compilarlo y ejecutarlo mediante los siguientes comandos:

```
$ haxe -neko server.n -main Server
$ neko server.n
```

La clase NetworkManager

Para centralizar la comunicación de la parte cliente, es decir, del juego, se ha diseñado la clase *NetworkManager*. Básicamente, está clase representa el único **punto de comunicación** centralizado para interactuar con el servidor. Para ello, esta clase implementa el patrón *Singleton*.

El siguiente listado muestra tanto las variables miembro (líneas 4-14) como el **método constructor** (líneas 16-29) de la clase *NetworkManager*. Las variables miembro comprenden la información básica de comunicación, tanto de la parte cliente como del servidor, y los sockets que se utilizan como puntos básicos de envío y recepción de información.

Listado 3.72: Clase NetworkManager. Variables miembro y constructor.

```
1  class NetworkManager {
2
3    // Variable estática para implementar Singleton.
4    private static var _instance:NetworkManager;
5
6    private var _server:String;            // Servidor de récords.
7    private var _port:Int;                 // Puerto del servidor.
8    private var _client:String;            // Host del cliente.
9    private var _clientPort:Int;           // Puerto del cliente.
10   private var _socketServer:Socket;      // Socket del server.
11   private var _socketClient:Socket;      // Socket de escucha.
12   private var _socketIncoming:Socket;    // Socket de peticiones.
13   private var _networkingThread:Thread;  // Hilo gestión récords.
14   private var _recordsStr:String;        // String con los récords.
15
16   private function new () {
17     _server = "localhost";
18     _port = 2048;
19     _client = "localhost";
20     _clientPort = 1024;
21     _socketServer = new Socket();
22     _socketClient = new Socket();
23     _socketClient.bind(new Host(_client), _clientPort);
24     _socketClient.listen(100);
25     _recordsStr = "";
26
27     // Hilo para atender peticiones del servidor de récords.
28     _networkingThread = Thread.create(networkingThread);
29   }
30
31   // ...
32
33 }
```

En este punto en concreto, resulta esencial discutir cómo la parte del cliente y la del servidor interactúan. Por una parte, el servidor puede recibir información mediante un socket específico (línea 21) que, por defecto, reside en el puerto 2048 (línea 18). Esto es necesario para que un cliente pueda enviar un nuevo récord o solicitar el listado de mejores récords.

Sin embargo, la parte cliente, es decir, el propio juego, también tiene la **necesidad de recibir información del servidor**. En concreto, necesita esta funcionalidad para recibir el listado de récords. Esta situación tiene dos consecuencias importantes:

1. El cliente también utiliza un socket para recibir información del servidor. En las líneas 22-24 se muestra el código para instanciar un socket que reciba información por el puerto 1024.

2. Debido a que los sockets utilizados son bloqueantes, se utiliza un **hilo de control adicional** para atender la recepción de dicha in-

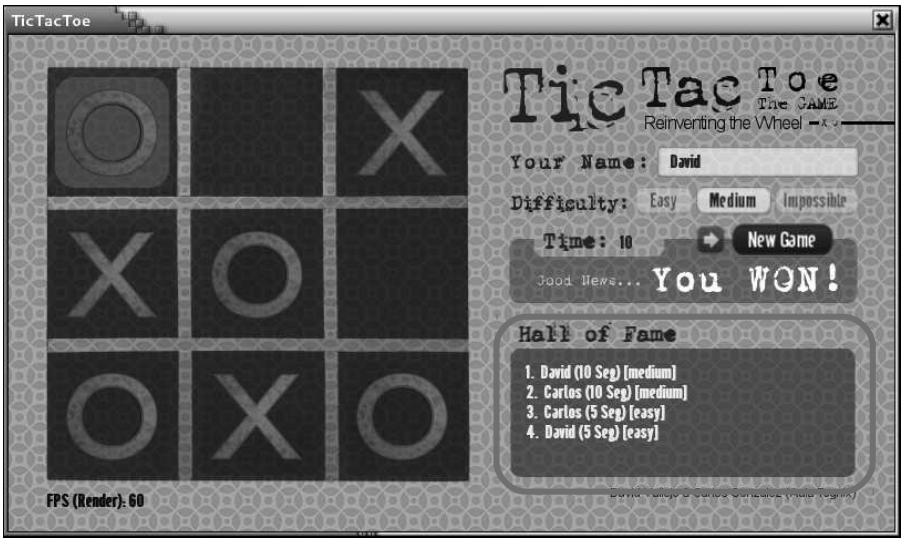


Figura 3.45: Captura de pantalla del juego Tic-Tac-Toe. En el recuadro rojo se enmarca la parte de visualización de récords.

formación. La línea 28 muestra cómo se crea un hilo de la clase *cpp.vm.Thread*[32], cuya funcionalidad está integrada en la función *networkingThread()*.

Precisamente, el siguiente listado muestra la implementación de esta función. Note cómo el hilo se bloquea mediante la función *waitForRead()* de la clase *Socket* hasta que reciba información del servidor. Cuando recibe la lista de récords, la lee (línea 11) y actualiza la variable que los contiene.

[32]Se ha optado por utilizar la clase Thread del paquete cpp debido a la inexistencia de una clase más general.

Listado 3.73: Clase NetworkManager. Función networkingThread().

```
1  // Función con el código del hilo auxiliar
2  // para manejar las conexiones entrantes del servidor.
3  private function networkingThread() {
4    while (true) {
5      try {
6        // Acepta un nuevo cliente.
7        _socketIncoming = _socketClient.accept();
8        // Se bloquea a la espera de datos...
9        _socketIncoming.waitForRead();
10       // Lectura de récords.
11       _recordsStr = _socketIncoming.input.readLine();
12     }
13     // Captura excepción cuando llega a fin de buffer.
14     catch (exception : Eof) {
15       _socketIncoming.shutdown(true, true);
16     }
17   }
18 }
```

Por otra parte, la **solicitud de récords** al servidor se realiza a través de la función *getRecordsFromServer()* de la clase *NetworkManager*. Básicamente, esta función genera un mensaje específico para formalizar la petición (línea 4), concretando el host y el puerto del cliente, y lo envía mediante la escritura en el socket al servidor (línea 5).

Listado 3.74: NetworkManager. Función getRecordsFromServer().

```
1  // Solicita la lista de récords al servidor.
2  public function getRecordsFromServer () {
3    try {
4      var msg:String = "request:" + _client + ":" + _clientPort + ".";
5      _socketServer.write(msg);
6    }
7    catch (unknown : Dynamic) {
8      trace("Error de conexion con " + _server);
9    }
10 }
```

Finalmente, el siguiente listado muestra la implementación de la función *sendRecordToServer()*, cuya responsabilidad es, como su nombre indica, **enviar un nuevo récord** al servidor. Para ello, en esta función se construye el mensaje que contiene el récord (líneas 6–9) y, de nuevo, se envía a través del socket que conecta con el servidor (línea 12). Note cómo después de enviar el récord al servidor se obtiene la lista actualizada del servidor (línea 19) para poder mostrarla en la interfaz gráfica.

Integrando la funcionalidad online en Tic-Tac-Toe

La integración de la funcionalidad online desde la lógica de Tic-Tac-Toe se simplifica enormemente gracias a la existencia de la clase *NetworkManager*. En esencia, sólo es necesario **invocar a la función adecuada** cuando se requiera enviar un nuevo récord. Por ejemplo, el siguiente listado de código muestra un fragmento de código que permite enviar dicho récord cuando el jugador humano gana una partida.

Listado 3.75: Envío de un nuevo récord a través del NetworkManager.

```
1 // Envío del nuevo récord.
2 if (_board.getWinner() == Player_O) {
3   _newRecord = true;
4   NetworkManager.getInstance()
5     .sendRecordToServer(_name.text,
6             getDifficulty(),
7             Std.int((Lib.getTimer() - _newGameTime) / 1000));
8  }
```

A la hora de renderizar los récords registrados en la interfaz gráfica, sólo es necesario acceder a la copia actualizada que maneja el *NetworkManager*, como se puede apreciar en el siguiente listado.

Listado 3.76: Obtención de récords a través del NetworkManager.

```
1 // Obtiene los récords del NetworkManager,
2 // el cual los recuperó previamente del servidor.
3 var recordsServer = NetworkManager.getInstance().getRecords();
4
5 // Separación de récords.
6 var records:Array<String> = recordsServer.split(";");
7 var i:Int = 1;
8 // Muestra los 5 primeros registros.
9 for (r in records) {
10   var aux:Array<String> = r.split(":");
11   // Rellena String auxiliar con la info del récord i.
12   ++i;
13 }
```

Apéndice A

Despliegue, Distribución y Gestión de Publicidad

A lo largo de este libro hemos aprendido a desarrollar un juego gracias a NME. En este anexo discutiremos el despliegue, distribución y publicación de dicho juego y, en cierta medida, la remuneración por nuestro desarrollo. Es cierto que podemos realizar el juego para nosotros, sin distribuirlo, pero siempre es interesante conocer dónde distribuir nuestro juego dependiendo de su plataforma. Para dicho fin estudiaremos una serie de páginas webs que servirán para publicar los juegos que desarrollemos, tanto para la plataforma *Flash* como para *Android*. Del mismo modo, se explicará cómo añadir las bibliotecas externas que nos otorgan dichas páginas, las cuales añaden funcionalidad y permiten incluso gestionar tablas de récords.

A continuación, se explicará un sistema de distribución que puede generar ganancias en forma de publicidad, aportada por una serie de empresas: *Mochi Media* y *Kongregate*, las cuales aportan la publicidad para Flash. Este tipo de distribución, para juegos amateur, es el más rentable, y permite que todos los usuarios tengan nuestro juego y lo prueben, creando así un nombre de desarrollador o de empresa que servirá para darnos a conocer. Ademas, para la plataforma Android, la distribución en Google Play de los juegos puede generar beneficios si optamos por cobrar por nuestro juego.

Figura A.1: Adobe Flex es un término que agrupa una serie de tecnologías publicadas desde Marzo de 2004 por Macromedia para dar soporte al despliegue y desarrollo de Aplicaciones Enriquecidas de Internet, basadas en su plataforma propietaria Flash.

A.1. Despliegue en Flash

Este apartado se centra en el despliegue en Flash. Flash es la principal plataforma para la que se generan juegos 2D y, aunque últimamente los smartphones están en auge, la facilidad que tiene Flash para distribuirse por Internet hace de ésta una buena plataforma para el desarrollo. Existen un gran número de páginas web que se dedican a alojar juegos en Flash, aportando información a los desarrolladores, como el número de visitas, votos de los usuarios y comentarios. Si estas páginas se usan de forma correcta se puede llevar un buen control de nuestro juego, conocer opiniones y saber si realmente se juega o no.

Antes de empezar a explicar a fondo estas páginas web hay un punto que necesitamos aprender, el uso de bibliotecas externas Flash en NME.

A.1.1. Usando bibliotecas externas en Flash

En ocasiones necesitaremos implementar una función para nuestro juego que requiera usar cierta parte de la API de ActionScript que no tenga implementado NME, o bien usar una API externa que encontremos por Internet. Para estos casos podremos importar la biblioteca a nuestro proyecto y usarla.

En ambos casos, el caso de que sea una biblioteca propia o en el caso de que sea una descargada de Internet, necesitaremos el archivo con extensión swc o swf. Si no se dispone de dicho archivo se puede compilar el ActionScript (el código) gracias al Adobe Flex SDK [1]. Este SDK nos suministra una serie de aplicaciones para la compilación del código ActionScript. La aplicación que nosotros usaremos es **compc**, que en realidad es compilador de ActionScript.

[1] http://www.adobe.com/devnet/flex/flex-sdk-download.htm

Instalación FlexSDK en GNU/Linux

Para usar el compilador de swf del FlexSDK en GNU/Linux necesitaremos tener instalado el Java Development Kit. En la sección 2.1.3 de este libro ya se discutió cómo instalar Java 7. Una vez instalado, sólo hace falta descargar de el paquete .zip con el SDK[2] y, a continuación, realizar los siguientes pasos para su instalación[3].

```
$ sudo mkdir /opt/flex
$ sudo mv Downloads/flex_sdk_4.6.zip /opt/flex/
$ sudo unzip /opt/flex/flex_sdk_4.6.zip
$ sudo chmod -R a+rx /opt/flex/
```

Realizado esto, tenemos que añadir el directorio a nuestro PATH para poder usar las aplicaciones.

```
$ echo 'export PATH=/opt/flex/bin:$PATH' >> ~/.bashrc
```

Si arrancamos un nuevo proceso bash, ya podremos usar compc para compilar.

Compilación del código

Este compilador, ya sea en GNU/Linux o en Windows[4], se puede usar por línea de órdenes para generar archivos swc mediante el comando[5]:

```
compc -source-path "Directorio raiz del proyecto"
-output "Directorio raiz del proyecto\salida.swc"
-include-classes "lista de clases a compilar"
```

La lista de clases a compilar por lo general coincidirá con la lista de archivos *.as* menos su extensión. En caso de que formen parte de un paquete (*package*) será necesario establecer la ruta completa a la clase.

```
paquete.clase
```

Realizado este paso ya tendremos nuestro archivo swc.

El siguiente paso consiste en generar la interfaz para haxe, realizando de forma sencilla siguiendo estos pasos:

[2] http://www.adobe.com/devnet/flex/flex-sdk-download.html

[3] La instalación en Windows es muy sencilla, ya que sólo tenemos que descomprimir el archivo .zip donde queramos.

[4] Compc en Windows seria Compc.exe.

[5] Cuando hablamos de la raíz nos referimos al directorio donde empieza nuestro proyecto. Dichos proyectos pueden contener subdirectorios, por lo que la ruta de éstos sería RAIZ/subdirectorio.

1. Descomprimir el archivo .swc. Podemos cambiar la extensión de nuestro archivo o usar el programa *7zip* para descomprimirlo directamente. Este archivo contendrá un catalog.xml y un library.zip.
2. Renombrar library.zip al nombre que queramos asignar a nuestra biblioteca. La primera letra ha de ser mayúscula.
3. Ejecutar el siguiente comando.

```
haxe -swf Biblioteca.swf --no-output
-swf-lib Biblioteca.swf --gen-hx-classes
```

4. Automáticamente se generará un directorio llamado *hxclasses*, dentro del cual encontraremos el directorio o archivos generados de la biblioteca. Éstos son los archivos importantes.

Realizados estos pasos, sólo necesitamos copiar la interfaz generada, es decir, los archivos importantes, y copiarlos en nuestro proyecto NME. Una vez copiados necesitamos configurar el archivo .nmml para que el compilador añada la biblioteca a la compilación:

```
<compilerflag name="-swf-lib Source/operacion/Operacion.swf"/>
```

e importar la interfaz generada en los archivos donde la queramos usar.

```
import operacion.Operacion;
```

En este punto, resulta interesante destacar que existe un entorno llamado **FlashDevelop** muy recomendado por los desarrolladores que usan NME. Este entorno se puede instalar tanto para Windows [6] como para GNU/Linux[7], aunque en GNU/Linux es necesario usar Wine para emularlo. FlashDevelop proporciona una serie de herramientas, entre las cuales se puede encontrar un depurador y un perfilador para archivos SWF de gran uso para gestionar memoria. También se pueden añadir herramientas tales como un compilador de swc, **ExportSWC** [8] de gran utilidad, ya que aligera la compilación de nuestras bibliotecas externas.

[6] http://www.gemfruit.com/getting-started-with-haxe-nme-and-flashdevelop
[7] http://camboris.blogspot.com.es/2011/05/instanlando-flashdevelop-en-ubuntu.html
[8] http://www.flashdevelop.org/community/viewtopic.php?t=2987

Figura A.2: Adobe ActionScript es el lenguaje de programación de la Plataforma Adobe Flash. La versión más extendida actualmente es ActionScript 3.0 y es utilizada en las últimas versiones de Adobe Flash y Flex. Desde la versión 2 de Flex viene incluido ActionScript 3.

A.1.2. Caso de estudio. Operaciones con una biblioteca externa.

Después de explicar todo lo anterior, vamos a ponerlo en práctica con un ejemplo muy sencillo. En este ejemplo usaremos un poco de código ActionScript, muy similar a Haxe.

En primer lugar, será necesario crear un directorio llamado *operacion* y dentro crear, con cualquier editor de texto, un archivo llamado *Operacion.as*. En ActionScript es importante que la primera letra de las clases y archivos sea mayúscula.

A continuación se muestra el contenido de dicho archivo:

Listado A.1: Archivo Operacion.as.

```
1  package operacion
2  {
3    public class Operacion {
4      private var _a:int; private var _b:int;
5      public function Operacion(a:int,b:int ) {
6        _a = a; _b = b;
7      }
8      public function suma():int {
9        return _a + _b;
10     }
11     public function resta():int {
12       return _a - _b;
13     }
14   }
15 }
```

Este código es muy simple. Se declaran unas variables, un objeto y unas funciones. ActionScript, al contrario que Haxe, obliga al programador a especificar el valor de retorno de funciones, como podemos ver en las líneas 8 y 11. En concreto, se ha incluido un **:int**, pero en el caso

de que la la función no devolviese nada, sería necesario añadir **:void** (en Haxe no es necesario).

Con el código ya escrito y el archivo guardado, podemos compilarlo y generar el archivo **.swc**. Esto, como ya se comentó, requiere el uso del SDK de Flex. Usaremos **compc**, el compilador que nos proporciona el SDK. Nos posicionamos en el directorio raíz del proyecto, donde tenemos un directorio llamado *operacion*, dentro del cual está nuestro archivo a compilar.

```
compc.exe -source-path ''RAIZ'' -output ''RAIZ\Operacion.swc''
-include-classes operacion.Operacion
```

Este comando permite generar en nuestro directorio raíz un archivo llamado *Operacion.swc*. Un archivo .swc básicamente es un paquete y su extensión (.swc) puede ser cambiada por .zip y descomprimirse para ver lo que contiene. Si se dispone del programa *7zip* instalado en el sistema se puede optar por descomprimir directamente el archivo .swc. Estos archivos, por regla general, contienen un *library.swf* y un *catalog.xml*. En nuestro caso necesitamos renombrar el archivo *library.swf* a *Operacion.swf*.

Con el archivo ya renombrado pasamos a generar la interfaz para Haxe. Como ya se explicó anteriormente, sólo es necesario ejecutar el siguiente comando.

```
haxe -swf Operacion.swf --no-output
-swf-lib Operacion.swf --gen-hx-classes
```

Este comando creará un directorio con los archivos de Haxe llamada *hxclasses*. En dicho directorio tendremos un subdirectorio llamado *operacion* y, dentro de él, *Operacion.hx*. Dicho archivo es una interfaz para nuestra biblioteca externa.

Ahora ya podemos usar nuestra biblioteca en nuestro proyecto de NME. Para ello es necesario copiar el directorio *operacion* dentro de nuestro proyecto NME. En el caso de este ejemplo, ha sido copiado dentro del directorio *Source* que es el que contiene nuestro código. Además de *operacion*, necesitamos copiar el archivo *Operacion.swf* dentro de dicho directorio, facilitando la futura localización de la biblioteca externa.

Lo primero que tenemos que hacer es informar a NME de que use la biblioteca externa. Para ello, en el archivo de configuración .nmml, añadiremos la siguiente línea.

Figura A.3: Estructura de directorios de nuestro proyecto NME.

```
<compilerflag name="-swf-lib
Source/operacion/Operacion.swf"/>
```

En segundo lugar, antes de escribir código, es necesario importar la biblioteca externa en el archivo que vaya a ser usada. Para ello, añadimos a la cabecera del archivo (NMEprueba.hx en nuestro caso) lo siguiente:

```
import operacion.Operacion;
```

Con la biblioteca ya importada, ahora sólo necesitamos crear un objeto *Operacion*, pasarle una pareja de enteros y usarlo para sumar y restar.

Listado A.2: Usando la biblioteca externa generada.

```
1 var op:Operacion = new Operacion(5, 5);
2 trace(op.suma());
3 trace(op.resta());
```

La salida de este sencillo programa es

```
NMEprueba.hx:23: 10
NMEprueba.hx:24: 0
```

Siguiendo esta receta seremos capaces de crear bibliotecas externas propias o usar las que encontremos por Internet.

Figura A.4: Mochi Media es una plataforma para juegos web con más de 140 millones de usuarios activos al mes, más de 15.000 juegos y alrededor de 40.000 páginas web donde publican sus juegos.

A.1.3. Mochi Media, publicidad y *highscores*

Una de las principales páginas de juegos Flash y que más aporta a los desarrolladores de juegos Flash es *Mochi Media*[9]. En concreto, otorga a los desarrolladores una serie de herramientas para que sean usadas en sus juegos:

1. **MochiAds**, una aplicación de publicidad de MochiMedia que otorga a los desarrolladores una forma de ganar dinero en sus juegos Flash. La publicidad puede ser mostrada como un cargador del juego o dentro del juego a petición del desarrollador, por ejemplo entre niveles.
2. **MochiCoins**, una plataforma que permite a los desarrolladores añadir contenido adicional a los usuarios que paguen. Este pago se realiza con MochiCoins asociados a la cuenta del usuario. Es otra forma de ganar dinero para los desarrolladores.
3. **MochiSocial**, que permite a los desarrolladores enviar información del juego a las redes sociales. Esta interacción incluye escribir noticias, enviar regalos y notificar logros a los amigos registrados en MochiMedia. Esto permite al desarrollador filtrar el contenido de su juego para usuarios registrados e invitados.
4. **MochiScores**, que permite al desarrollador del juego añadir una serie de tablones de récords en su juego. Estos tablones pueden otorgar medallas a los usuarios de MochiMedia.

Lo más relevante en este punto para nosotros son *MochiScores* y *MochiAds*.

Creando y configurando el perfil para nuestro juego

Una vez registrados en la web de Mochi Media (ver figura A.5), proceso muy sencillo, pasamos a crear un perfil para nuestro juego.

[9] http://www.mochimedia.com

Figura A.5: Interfaz web de Mochi Media.

La creación de un nuevo perfil se hace desde el apartado de **Dev Tool** o **dashboard**. En dicho apartado encontramos un botón llamado **Add Game**. Además, en dicho apartado podemos descargar la API, pero eso ya lo estudiaremos más adelante.

En la siguiente página nos pedirán información básica de nuestro juego, como el nombre y las dimensiones y nos preguntaran si queremos usar el sistema **Live Updates**. Por ahora, le diremos que no; más tarde explicaremos qué es este sistema que implementa Mochi Media.

Una vez confirmados los datos, llegaremos al perfil de nuestro juego. Esta sección muestra información de nuestro juego, como las sesiones iniciadas, las partidas jugadas y las impresiones de publicidad. También nos muestra nuestro identificador de juego, el **Game ID**. Este ID se usará con la API para identificar a nuestro juego en el sistema. En dicha página también podemos configurar las herramientas que nos otorga Mochi Media, como la publicidad (Ads), la distribución de nuestro juego (Distribution), los tablones de récords (Scores), y el registro o control de enlaces (Link Tracking). Este sistema permite añadir enlaces a nuestro juego y cambiarlos desde esta herramienta, sin necesidad de tocar el código del juego.

Compilando MochiAPI para Haxe

En la web de Mochi Media se puede obtener la última versión de su API. Actualmente ofrecen la versión 4.1.1[10]. En la documentación para desarrolladores podemos encontrar ejemplos de código, tanto para ActionScript 2 y 3, ejemplos que nos servirán de ayuda a la hora de implementar la publicidad y los récords en nuestro juego.

Una vez bajado la API pasamos a compilarla en swc, como ya se estudió en el apartado anterior. La API contiene una versión de código

[10]Esta ultima versión, además de toda la documentación, se puede descargar de http://www.mochimedia.com/support/dev_docs o en el apartado **Dev Tool**.

para AS2 y otra para AS3, pero nosotros nos centraremos en la versión para AS3. Además, tenemos que recordar incluir todos los archivos. El comando quedaría como se expone a continuación:

```
compc.exe -source-path . -output .Mochi.swc -include-classes
mochi.as3.Base64Decoder mochi.as3.Base64Encoder
...... mochi.as3.MochiUserData
```

Ahora sólo necesitamos crear la interfaz en Haxe con lo explicado anteriormente. Esto nos generara los archivos .hx y el .swf necesarios para poder usarlos en nuestros juegos con NME.

Creando la clase Anuncio.hx

Para nuestros juegos usaremos la clase Anuncio.hx, una clase que hereda de **Sprite** y que dibuja la publicidad en nuestro juego.

Listado A.3: La clase Anuncio en Haxe.

```
1  package;
2  import nme.display.Sprite;
3  import nme.Lib;
4  import mochi.as3.MochiServices;
5  import mochi.as3.MochiAd;
6
7  class Anuncio extends Sprite {
8    public function new(p_code:String, p_resolution:String,
9                p_callback:Void -> Void) {
10     super();
11     Lib.current.addChild(this);
12     MochiServices.connect(p_code, root);
13     MochiAd.showPreGameAd( { id:p_code, res:p_resolution,
14       clip:root, ad_finished:p_callback} );
15   }
16 }
```

Esta clase recibe nuestro **Game ID** y la resolución de nuestro juego y una función que usaremos para avisar al juego del fin del anuncio. La función **MochiServices.connect** es la encargada de identificar nuestro juego en el sistema. La función **showPreGameAd** es la función encargada de cargar los anuncios al principio del juego. Dicha función recibe como argumento un objeto, que en haxe se define entre . Entre las variables del objeto que estamos enviando necesitamos enviar el Sprite donde el anuncio será dibujado.

Estas son las variables que podemos añadir al objeto:

1. **clip:** es una referencia al MovieClip donde será añadida la publicidad.

2. **color:** por defecto es 0xFF8A00, y define el color de la barra de carga.
3. **background:** por defecto es 0xFFFFC9, y define el color del fondo del anuncio.
4. **no_bg:** por defecto a false. Si se pone a true desactiva el fondo del anuncio.
5. **outline:** por defecto es 0xD58B3C, y define el color de borde de la barra de carga.
6. **no_progress_bar:** por defecto a false. Si se pone a true desactiva la barra de carga.
7. **skip:** por defecto a false. Si se pone a true desactiva el anuncio.
8. **fadeout_time:** es el tiempo, en milisegundos, que tarda en desaparecer el anuncio una vez terminado. Por defecto a 250.
9. **ad_started:** es la función que será llamada cuando el anuncio empiece.
10. **ad_finished:** es la función que será llamada al finalizar el anuncio. También es llamada si el anuncio no se carga.
11. **ad_failed:** es la función que será llamada si no se puede mostrar la publicidad, esto suele ocurrir si el usuario tiene algún software de bloqueo instalado. Si esta función es llamada, se llamara antes que ad_finished.
12. **ad_loaded:** es la función llamada cuando se carga el anuncio.
13. **ad_skipped:** es la función llamada cuando se omite el anuncio debido a un bloqueo del dominio. De ser llamada, sería llamada antes que ad_finished.
14. **ad_progress:** es una función que envía un numero entre 0 y 100 y será llamada cuando el anuncio alcance dicho porcentaje.

No es necesario configurar todas estas variables. La más importante es **ad_finished**, que llamara a la función que le enviemos y que usaremos para cargar nuestro juego después de finalizar los anuncios. También podemos usar ad_progress y cargar nuestro juego cuando lleguemos a cierto porcentaje del anuncio mostrado, pero esta llamada puede dar problemas en caso de que el anuncio no sea cargado correctamente.

Como se discutió en la sección 2.2, NME, dispone de la directiva **define** que pueden modificar la compilación de forma condicional.

Añadiendo el anuncio al juego

Con el anuncio ya creado, podemos pasar a cargarlo en nuestro juego. Un anuncio **PreGame** se ejecutará en la intro del juego, en nuestro primer estado. Normalmente va en la función inicializadora o **init()**. Para ello, añadimos a la pantalla actual el anuncio con esta simple línea:

```
Lib.current.addChild(new Anuncio("Game ID",
                                 "resolucion",
                                 callback(FinAnuncio)));
```

Tanto **Game ID** como **resolución** son 2 variables de tipo String, mientras que la resolución está definida como "Ancho x Largo" (ejemplo: "800x600"). La función **FinAnuncio** es una función que no recibe argumentos y que no devuelve nada y que se usará como función callback cuando el anuncio termine:

```
function FinAnuncio():Void {
    ....
}
```

Sería adecuado eliminar el anuncio una vez mostrado, por lo que si el **Anuncio** es guardado en una variable **_anuncio:Anuncio**, entonces se podría eliminar más tarde con:

```
Lib.current.removeChild(_anuncio);
```

Configurando los anuncios en la web

Una vez compilada la biblioteca externa de Mochi pasamos a configurar los anuncios. En el perfil que hemos creado, en la web de **Overview** activamos los anuncios pulsando en el botón del apartado **Ads** (ver figura A.6).

Esto nos llevará a una pagina donde podremos configurar los anuncios que mostramos. Nada más crear el perfil del juego no podremos empezar a ganar dinero con los anuncios; primero hace falta realizar una serie de pasos. Estos pasos se ven marcados en el primer párrafo de la web. Básicamente son los siguientes:

1. **Test your game:** este apartado nos confirma si ha sido insertado el anuncio bien en nuestro juego. Si ya hemos añadido la publicidad a nuestro juego y lo hemos testeado, este apartado tendría que aparecer como **Success!**

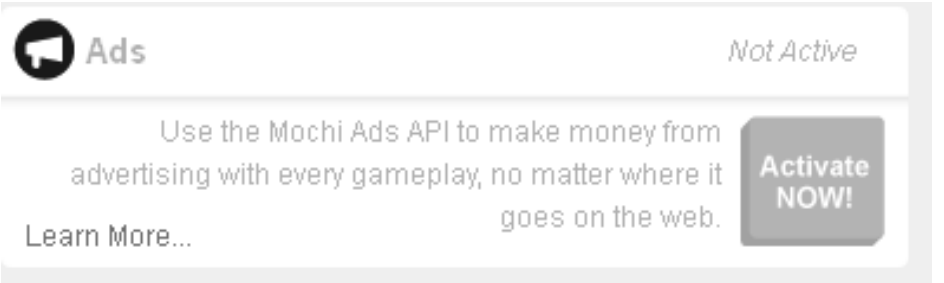


Figura A.6: Activación de anuncios Mochi Media.

2. **Fill out your game profile:** para poder publicar un juego y empezar a ganar dinero con él hace falta rellenar una serie de datos sobre del juego. Por ahora, podemos rellenar la información que nos piden dándole al enlace que nos proporcionan (game profile). Esta información se usará para publicitar nuestro juego y mostrar información. En dicho formulario nos pedirán el tipo de juego, una serie de capturas de pantalla, instrucciones, si usamos teclas para poder jugar y otras cosas. Esta información se puede guardar dándole a **save** sin necesidad de subir el juego por ahora.

3. **Your game must be reviewed and approved:** Mochi necesita revisar y aprobar nuestro juego. Este proceso suele tardar menos de 1 día.

Lo importante para configurar los anuncios es el **Ads Settings**. Nos preguntaran qué tipo de anuncios queremos mostrar, si los de Mochi, si unos propios o si lo queremos desactivar. Además, nos preguntaran si queremos mostrar anuncios cuando se visualicen los tablones de récords. Esto viene determinado por el propio desarrollador. También podemos filtrar qué dominios no queremos que sean publicitados, lo cual es adecuado para juegos publicitarios donde no quedaría bien mostrar publicidad de un producto de la competencia.

Configurando la tabla de récords en la web

Al contrario que ocurre con los anuncios, es necesario configurar primero el tablón de récords para poder usarlo. Con el objetivo de visualizar los tablones necesitamos un identificador que nos da la web cuando creamos uno (ver figura A.7).

Al igual que con los anuncios, pasamos a activar los tablones en el apartado **Scores**. Este enlace nos llevará a una web que nos informa de los tablones y nos invita a crear uno nuevo. Una vez en el formulario de

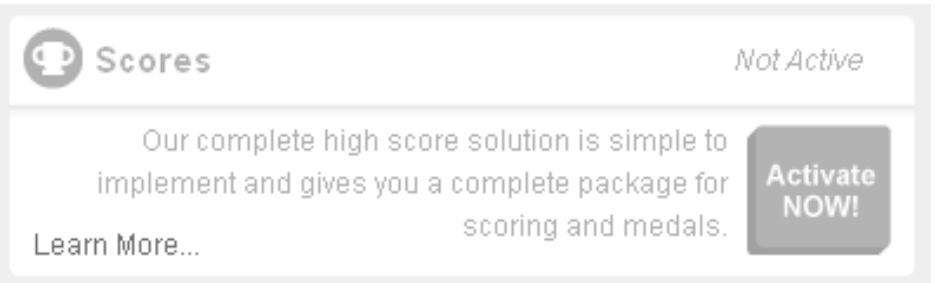


Figura A.7: Captura de pantalla del tablón de récord.

creación del tablón, sólo nos faltará rellenar los datos que nos piden: el nombre del tablón, una pequeña descripción, el tipo del récord, número o tiempo, en qué caso es mejor, si el mayor o el menor récord, y qué etiqueta le queremos dar a ese récord (por ejemplo, goles). También nos preguntarán si queremos que ese tablón incluya logros o no. Si optamos por añadir el logro, nos pedirán tres valores que se usarán para otorgar medallas a los jugadores. Por último, podemos configurar el color de fondo del tablón y qué apartado de tiempo se mostrará al ser cargado (por defecto esta seleccionado en récords diarios).

Una vez creado, se mostrarán los récords del tablón. En la parte derecha podremos seleccionar entre los distintos tablones. Con nuestro tablón seleccionado, usamos el enlace **actionscript code** y buscaremos el **Leaderboard ID**. Este ID se usará para cargar nuestro tablón. En este apartado nos muestran un código, as2 o as3, que en nuestro caso no usaremos para nada, pero podemos observar que usan una función para *encriptar* el ID del tablón. Esto puede ser necesario en caso de encontrar irregularidades en los récords. Con dicho ID ya podemos pasar a añadir los tablones al juego.

Creando la clase LeaderBoard.hx

Al igual que hicimos con el anuncio, vamos a crear una clase *LeaderBoard* que se encargará de crear el tablón de récords. Esta clase, al igual que la clase *Anuncio*, heredará de **Sprite**, por lo que podrá ser añadida a la escena con **addChild** cuando queramos.

Listado A.4: La clase LeaderBoard en Haxe.

```
1  package ;
2  import nme.display.Sprite;
3  import mochi.as3.MochiScores;
4  import mochi.as3.MochiServices;
5  import nme.Lib;
6
7  class LeaderBoard extends Sprite {
8    public function new (p_code:String, p_codeBoard:String,
9                 ?p_score:Int=0, ?p_name:String,
10                p_callback:Void -> Void,
11                p_callbackError:Void -> Void) {
12     super();
13     Lib.current.addChild(this);
14     MochiServices.connect(p_code, root);
15     if (p_score == 0) {
16       MochiScores.showLeaderboard({boardID: p_codeBoard,
17         onClose:p_callback});
18     }
19     else {
20       MochiScores.showLeaderboard({boardID: p_codeBoard,
21         score:p_score, onClose:p_callback});
22     }
23   }
24 }
```

Esta clase carga dos tipos de LeaderBoard, dependiendo de si la clase recibe una puntuación o no. En caso de recibir 0, sólo mostraría los récords pero no los actualizaría.

Esta clase, al igual que *Anuncio*, se conecta al servicio de Mochi Media, pero lo más importante de este código es la función **MochiScores.showLeaderboard()** que, al igual que **showPreGameAd()**, recibe un objeto que puede contener una serie de variables. Todas estas variables están descritas en la documentación del desarrollador en la web de Mochi Media[11]. Nosotros usaremos las siguientes:

1. **score:** la puntuación del jugador; si no recibe esta variable directamente mostrara el tablón de récords.

2. **name:** si no recibe el nombre lo pedirá mediante un formulario. En caso de estar identificado en la red Mochi, usará nuestro nickname.

3. **boardID:** es el identificador que nos dan al registrar un tablón, es decir, el **Leaderboard ID**.

4. **onDisplay:** función llamada al cargar los récords.

5. **onClose:** función llamada al cerrar los récords.

[11] http://www.mochimedia.com/support/dev_docs#Other_Leaderboard_Options

6. **onError:** función llamada si ocurre un error en la carga. Esta función tiene por defecto *onClose*, por lo que con registrar sólo la función *onClose* nos serviría.

Si nuestro juego obtiene de alguna manera el nombre del jugador, sería adecuado enviarlo con la variable **name**, aunque el sistema de Mochi, si no recibe un nombre, lo pedirá antes de enviar el récord.

En nuestro caso, sólo usaremos score, boardID y onClose. **onClose** es necesario para poder informar al juego de que el tablón ha sido cerrado y poder reiniciar el juego. Es aconsejable parar la ejecución del juego, ya sea cargando un estado nuevo de juego que se use para la visualización del tablón, o haciendo que la función *onEnterFrame* no se ejecute mientras el tablón está activo.

Cargando el tablón en nuestro juego.

Los tablones se suelen cargar a petición del jugador, añadiendo un botón para actualizar el récord, o cuando el juego se termina o se llega a cierto nivel. Así, la línea de llamada al tablón no especifica en un lugar concreto del código. Sea donde fuere, la forma de llamar al tablón será la siguiente:

```
Lib.current.addChild(_leaderBoard =
new LeaderBoard("Game ID", "leaderboard ID",
                puntos, callback(juegoNuevo)));
```

siendo *juegoNuevo*

```
function juegoNuevo():Void {
... código para reiniciar el juego
}
```

Con esto ya tenemos añadidos los récords. Nuestro juego puede contener infinidad de tablones. Por ejemplo, podemos crear uno para cada nivel de juego, o uno por cada nivel de dificultad, pero es necesario recordar que por cada tablón de nuestro juego tenemos que crear un tablón en la web.

Subiendo el juego a la web: *Lives Updates*

La web de Mochi otorga una gran herramienta para los desarrolladores, los **Live Updates**. Esto es una forma de subir nuestro juego a la web y, mediante un proceso que realiza Mochi, por el cual encapsula

nuestro juego dentro de otro swf, nos otorga seguridad y facilidad de actualización. Un usuario podrá descargar nuestro juego y mantenerlo siempre actualizado mediante esta herramienta. Además otorga la publicidad de forma fácil, ya que no sería necesario añadir el código para la publicidad **preGame** dado que el cargador de **Live Updates** ya lo trae por defecto.

Para subir nuestro juego iremos al apartado **Live updates & Game Files** y, al igual que hicimos con la publicidad y con los récords, pulsaremos en el enlace **Activate Now**[12]. En dicha web, y si ya hemos actualizado los datos de nuestro juego, sólo tendremos que decidir si usamos **Live Updates** o no y subir nuestro juego. Una vez seleccionado, nos preguntará si queremos o no distribuirlo. Si optamos por no distribuirlo y cambiamos de idea, siempre podremos distribuirlo en el mismo apartado dándole a editar la distribución.

Para los juegos con Live Updates es necesario añadir la siguiente línea de código en un lugar visible del juego. Es recomendable añadirlo en la función **main()**.

```
public var _mochiads_game_id:String = "GAME ID";
```

La red Mochi dispone de más de 40.000 páginas donde distribuir los juegos. Gracias al control de las estadísticas podemos observar desde qué parte del mundo se juega a nuestro juego.

Una vez aprobado nuestro juego, en menos de 24 horas, dispondremos de una serie de webs donde nuestro juego estará publicado y ya podremos empezar a ganar dinero con la publicidad.

A.2. Caso de estudio. Implementando MochiAPI en OfficeBasket

En este caso de estudio implementaremos la publicidad y los récords en el código del OfficeBasket discutido en la sección 3.4.7. Note que se han realizado una serie de cambios en el código para que la introducción de anuncios y récords se adapte de una forma más adecuada.

Inicialmente, añadiremos las dos clases ya explicadas con anterioridad: *Anuncios* y *LeaderBoard*, encargadas de mostrar la publicidad y los tablones en el juego, respectivamente. Ambas clases serán añadidas

[12]Si obtenemos un aviso de *The game cannot be found or does not load at the URL provided. Please provide the full URL for us to access your game*, sólo tendremos que añadir la dirección **Hosted game url**, obtenida al subir el juego a Mochi, al formulario **Game Profile** en el apartado **Approval URL**.

en el directorio *util* (utilidades), por lo que tendrán que formar parte de dicho paquete. Es necesario modificar el paquete de cada clase con la siguiente línea de código:

```
package util;
```

A.2.1. Añadiendo un final al juego y un tablón de récords

Para añadir un final al juego se ha puesto un límite de lanzamientos, fijado a 10. Una vez agotados los lanzamientos se reiniciará el juego. Se ha modificado el texto mostrado por *ScoreBoard* para que muestre las canastas de 10, como se muestra en el siguiente listado.

Listado A.5: Función update() en ScoreBoard.hx.

```
1 public function update (delta:Int = 0) : Void {
2   _score += delta;
3   htmlText = Std.string(_score+"/10");
4 }
```

Para controlar las canastas y las bolas lanzadas lo primero que se ha modificado ha sido la función onMouseClick para registrar el número de bolas lanzadas.

Listado A.6: Función onMouseClick() en OfficeBasket.hx.

```
1 function onMouseClick(e:MouseEvent):Void {
2   // Si no hay pelota activa, creamos una.
3   if (_ball == null){
4     _ball = new PhPaperBall("ball", _layerPaper, _world,
5                 _spx, _spy, _arrow.getVector());
6     _lanzadas++;
7   }
8   _previousTime = Lib.getTimer();
9 }
```

Podemos observar que se realiza un incremento de **_lanzadas** cuando generamos una bola nueva. También se ha modificado el método **onEnterFrame** de la clase *OfficeBasket*.

Listado A.7: Función onEnterFrame() en OfficeBasket.hx (I).

```
1 // Sólo actualizamos la flecha si no hay bola activa...
2 else {
3   if (_lanzadas >= 10) {
4     // .........
5   }
6   else{
7     _arrow.update();
8     _layerArrow.render();
9   }
```

Este fragmento de código está incluido dentro de la función **onEnterFrame** se corresponde con el bloque **else**. Este bloque se ejecuta si la variable **_ball == null**. Una vez centrados en el código, ya sólo tenemos que añadir la comprobación de **_lanzadas**, variable de clase creada con visibilidad privada. Si hemos lanzado 10 bolas de papel, ejecutaremos el código incluido en el bloque. En caso de no ser verdadera la comprobación, actualizaremos la flecha y la dibujaremos.

En este punto, el lector observará que esta parte del código es un buen lugar para colocar nuestro tablón de récords[13][14].

Listado A.8: Función onEnterFrame() en OfficeBasket.hx (II).

```
1 // Sólo actualizamos la flecha si no hay bola activa...
2 else {
3   if (_lanzadas >= 10) {
4     #if flash
5       // Quitamos los listeners para que no nos molesten
6       // durante la visualización del tablón.
7       removeListeners();
8     // llamamos al scoreboard.
9     Lib.current.addChild(_leaderBoard =
10              new LeaderBoard("cac05607f49a2ef6",
11                      "Leaderboard ID",
12                      _scoreBoard.score,
13                      callback(juegoNuevo)));
14    #else
15      juegoNuevo();
16    #end
17  }
18  else{
19    _arrow.update();
20    _layerArrow.render();
21  }
22 }
```

[13]En el perfil del juego se ha configurado un tablón de récords numérico, donde el mayor registro se queda guardado.

[14]Se ha ocultado el Leaderboard ID por cuestiones de seguridad.

Note cómo se han usado los **defines** de precompilación para definir qué código se compilará para la plataforma flash y qué código se compilara para otras plataformas.

Por otra parte, la función **removeListeners** ha sido añadida al código de OfficeBasket.hx. Al igual que disponemos de addListeners, hemos creado un removeListeners:

Listado A.9: Función removeListeners() en OfficeBasket.hx.

```
function removeListeners():Void {
  Lib.current.stage.removeEventListener(MouseEvent.CLICK,
                    onMouseClick);
  Lib.current.stage.removeEventListener(Event.ENTER_FRAME,
                    onEnterFrame);
  Lib.current.stage.removeEventListener(MouseEvent.MOUSE_MOVE,
                    onMouseMove);
}
```

El código anterior es sencillo; lo único que hace es eliminar, gracias a **removeEventListeners**, los eventos que ya no queremos registrar. Esto se realiza con la intención de que el juego no haga cálculos innecesarios mientras esta mostrando el tablón de anuncios.

Una vez realizada dicha función, procedemos a cargar los anuncios como ya se explicó con anterioridad:

```
Lib.current.addChild(
  _leaderBoard = new LeaderBoard("cac05607f49a2ef6",
                                 "Leaderboard ID",
                                 _scoreBoard.score,
                                 callback(juegoNuevo)));
```

La variable **_leaderBoard** es una variable de clase, que usaremos para eliminar posteriormente el tablón. Se envía la puntuación lograda en el juego, obtenida mediante la variable **score** de **_scoreBoard**. El código de *_scoreBoard* ha sido ligeramente modificado, añadiendo métodos *getters* y *setters* y una función **reset**:

Listado A.10: Función reset() en ScoreBoard.hx.

```
public function reset () : Void {
  _score = 0;
  update();
}
```

Usaremos esta función para reiniciar la puntuación del juego.

La función *juegoNuevo*, como ya se explicó, se usa para reiniciar el juego. En el caso de OfficeBasket se usará este código:

Listado A.11: Función juegoNuevo() en ScoreBoard.hx.

```
1  function juegoNuevo () : Void {
2    _lanzadas = 0;
3    _scoreBoard.reset();
4    _arrow.update();
5    _layerArrow.render();
6    #if flash
7      // Cargar listeners para registrar eventos.
8      addListeners();
9    Lib.current.removeChild(_leaderBoard);
10   _leaderBoard = null;
11   #end
12 }
```

En primer lugar, reiniciamos los valores de las variables, actualizamos la flecha y la dibujamos. Lo siguiente consiste en volver a añadir los *listeners* con **addListeners**, quitar el sprite del tablón de récords de la escena, con **removeChild**, y establecer la variable a null.

Todas estas modificaciones hacen que nuestro juego tenga un final y un tablón de récords que mostrará las canastas encestadas.

A.2.2. Añadiendo publicidad al juego

Dado que el juego no dispone de introducción, y básicamente es la misma escena siempre, se ha optado por no cargar la escena hasta que la publicidad sea mostrada. Esto se ha realizado adaptando el código de **init()** y creando una función **carga()** que será llamada cuando la publicidad finalice.

Listado A.12: Función init() en ScoreBoard.hx.

```
1  function init () : Void {
2    #if flash
3      Lib.current.addChild(_anuncio = new Anuncio("cac05607f49a2ef6",
4                           "800x448",
5                           callback(finAnuncio)));
6    #else
7      carga();
8    #end
9  }
```

Por otra parte, la carga del anuncio se hace de forma sencilla, como ya se explico anteriormente. La función **finAnuncio()** será enviada como

parte de la variable **ad_finished**, por lo que será ejecutada al finalizar el anuncio.

Listado A.13: Función finAnuncio() en ScoreBoard.hx.

```
1 function finAnuncio () : Void {
2   trace("fin");
3   Lib.current.removeChild(_anuncio);
4   _anuncio = null;
5   carga();
6 }
```

Como se puede apreciar en el listado anterior, eliminamos el anuncio de la escena y lo establecemos a *null*, como ya hicimos con el tablón de récords. Así mismo, la función *carga* contiene todo lo que contenía nuestro antiguo **init()**:

Listado A.14: Función carga() en ScoreBoard.hx.

```
 1 function carga () :Void {
 2   loadResources();                  // Carga de recursos gráficos
 3   generateRandomPaperPosition();    // Genera posición para papel
 4   createPhysicWorld();              // Creación del mundo físico
 5   createArrow(_spx, _spy);
 6   createBasket(_sw, _sh);
 7   addListeners();
 8   _previousTime = Lib.getTimer();   // Inicializa el tiempo
 9   _lanzadas = 0;
10 }
```

Podemos ver que se ha añadido la inicialización de la variable *_lanzadas* a 0.

Una vez adaptado el código añadiremos al archivo .nmml la siguiente línea y podemos pasar a compilar para la plataforma Flash[15].

```
<compilerflag name="-swf-lib src/mochi/as3/Mochi.swf"/>
```

Una vez compilado el juego, éste se puede subir a la web de Mochi como ya se discutió en el apartado anterior[16]. Para esta distribución se ha optado por no usar el sistema Live Updates que nos otorga Mochi. Con la demo teníamos la intención de mostrar el uso de la publicidad, por lo que sería lo opuesto a usar Live Updates, ya que este sistema nos incluye la publicidad inicial de forma automática.

[15]Recordad que la biblioteca MochiAPI está en la raíz del código (directorio src).
[16]Se puede ver el resultado obtenido en http://www.mochimedia.com/games/play/officebasket

A.3. Kongregate. Gestión de logros

Otra web que publica juegos Flash es Kongregate[17], la cual dispone de una serie de herramientas que el desarrollador puede usar en sus trabajos. En esencia, Kongregate es una web que permite almacenar juegos online creados con Adobe Flash, HTML 5/JavaScript, Shockwave, Java y Unity3D. Otorga una API a los desarrolladores Flash y Unity que pueden integrar en sus juegos y que permite a los usuarios subir récords y lograr medallas.

Kongregate añade de forma automática su publicidad a los juegos publicados en su web. Tiene un trato con Mochi Media por el cual Mochi Media bloquea la publicidad de su sistema siempre que el juego esté almacenado en Kongregate. Las ganancias por publicidad en Kongregate se ven modificadas según una serie de opciones. En concreto, el desarrollador gana un 25 % de la publicidad obtenida por su juego, como base, un 10 % adicional si el juego implementa la **stats & high score API** de Kongregate, y un 15 % adicional si es un juego exclusivo de Kongregate.

Todos los juegos subidos a la web entran en dos concursos: un concurso semanal y otro concurso mensual. En dichos concursos, sólo entran los juegos lanzados esa semana o mes. Si un juego obtiene más de una cantidad de votos en su primera semana de publicación, entonces entra en un concurso por el cual puede ganar entre 250$ y 150$, dependiendo del puesto en el que quede. Si tiene los votos necesarios, entrará en el concurso mensual, un top 10 con los mejores juegos de ese mes, en el cual puede ganar entre 2500$ y 250$.

Además, si el desarrollador finalmente gana alguno de los concursos o se queda bien posicionado en el concurso, nuestro juego tendrá un gran número de posibilidades de obtener medallas para los usuarios, hecho que incrementará nuestras visitas. En definitiva, este concurso es un gran aliciente para los desarrolladores de juegos.

A.3.1. Instalación y uso con NME

Existe un proyecto que se encarga de gestionar todo el uso de la API de Kongregate y que se puede descargar mediante *haxelib*:

```
haxelib install kong.hx
```

El sistema de Kongregate es distinto al de Mochi Media, ya que Kongregate no entrega al desarrollador el código de su API. Sin embargo, si permite al desarrollador conectarse a la API y así poder usar sus funcio-

[17] http://www.kongregate.com

nes. Este proceso de conexión se realiza gracias a la biblioteca instalada.

Para usar *kong.hx* es necesario añadirlo al archivo **.nmml** para su posterior uso:

```
<haxelib name="kong.hx" />
```

Una vez cargada la biblioteca es necesario importarla donde vaya a ser usada mediante las siguientes instrucciones:

```
import kong.Kongregate;
import kong.KongregateApi;
```

Una vez realizado todo esto, lo más importante es pedir lo antes posible la API de Kongregate. Esto se realiza con la siguiente llamada:

```
Kongregate.loadApi(kongregateLoad);
```

LoadApi es una función que tiene como argumento una función *callback* que será llamada cuando la API sea cargada, tal y como se muestra en el siguiente listado de código.

Listado A.15: Función LoadApi.

```
1 public var kongApi:KongregateApi;
2
3 // ...
4
5 private function kongregateLoad (api:KongregateApi) : Void {
6   kongApi = api;
7   kongApi.services.connect();
8 }
```

Como podemos observar, **kongregateLoad** recibe una **KongregateApi** y lo que hace es conectarse a dicha API, además de guardarla para su posterior uso.

Una vez realizados estos primeros pasos, podremos usar la API recibida para enviar estadísticas y récords a Kongregate. La diferencia de estos datos, si son estadísticas o son récords, viene dada al crearse el registro en la web. Este proceso se explicará más adelante, cuando se explique la subida del juego.

La forma de subir datos se realiza a través de la siguiente instrucción:

```
kongApi.stats.submit("NombreRegistro", Dato);
```

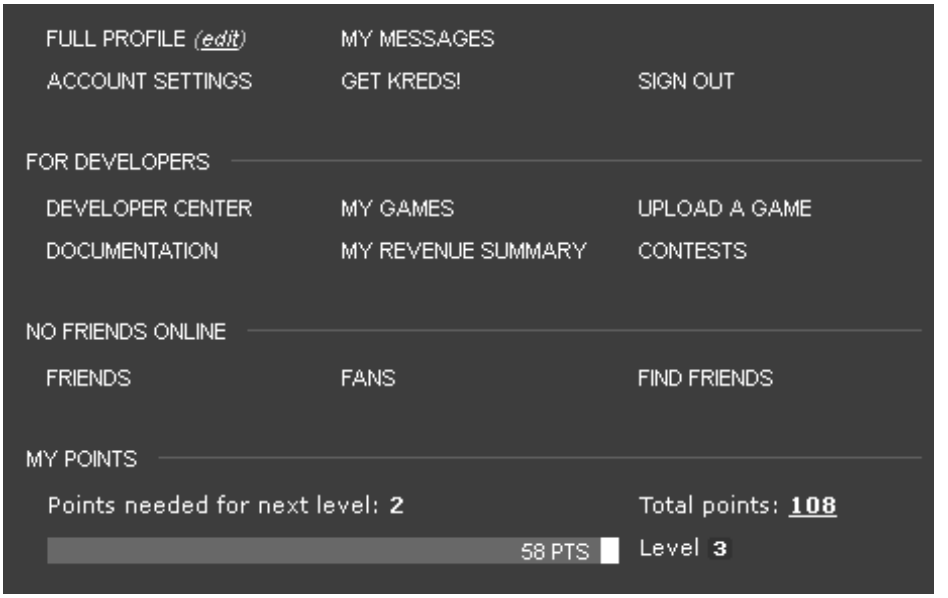


Figura A.8: Captura de pantalla de las opciones de Kongregate.

Es así de simple ya que cuando queramos, podemos subir un dato. Estos datos pueden ser usados para crear logros, por lo que se recomienda subir los datos de forma continua, y no esperar hasta el final. Hay que tener en cuenta que el usuario puede dejar la partida cuando quiera, por lo que sería adecuado guardar el progreso gradualmente[18].

A.3.2. Carga del juego en Kongregate

Una vez configuradas las estadísticas que queremos subir a Kongregate, el siguiente paso consiste en subir el juego. Este proceso se realiza desde el apartado **Upload a Game** de nuestro perfil (ver figura A.8).

En el siguiente formulario se nos pedirá cierta información, como el titulo del juego, una descripción, instrucciones y otros datos. Una vez rellenados estos datos, pasaremos a subir el juego y sus capturas de pantalla, y la parte importante: podremos configurar las estadísticas en la sección **Statistics Api**, usando el enlace *Add Statistic*.

En el formulario que se acaba de crear, al pulsar **Add Statistics**, nos pedirán el nombre del registro a guardar y la descripción, además de la siguiente información:

[18]Podemos encontrar más información de Kong.hx en el repositorio[19] del proyecto.

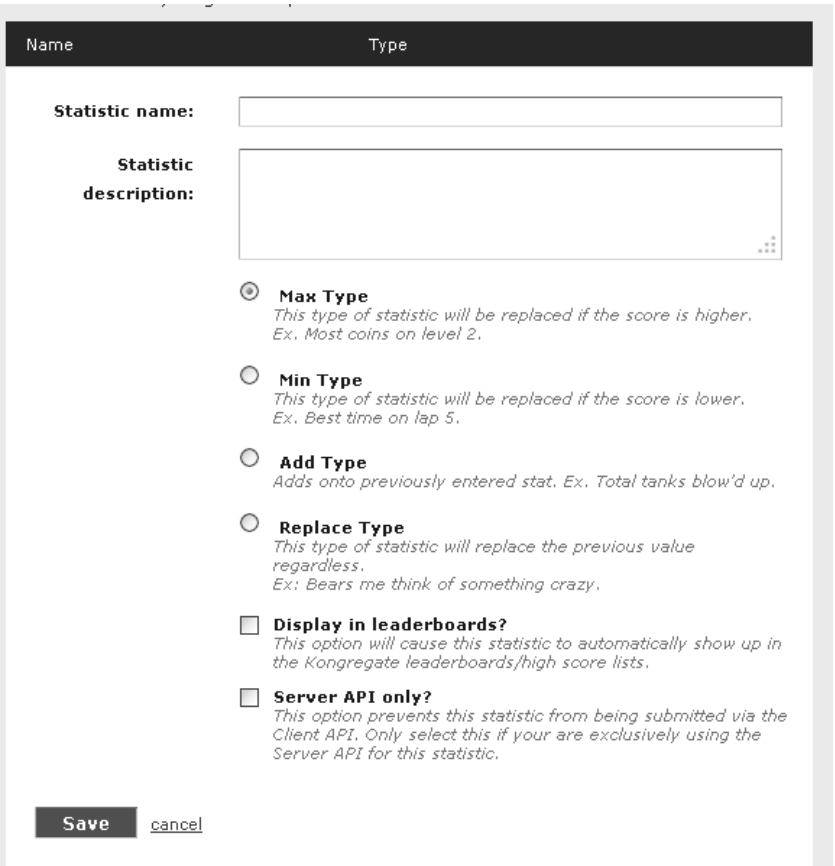


Figura A.9: Captura de pantalla de las estadísticas de Kongregate.

1. **Max Type**: sólo se guardará si la variable nueva es superior a la vieja. Ex. puntuación.
2. **Min Type**: sólo se guardará si la variable nueva es inferior a la vieja. Ex. tiempo.
3. **Add Type**: se añadirá el valor recibido al ya poseído.
4. **Replace**: se reemplazará el valor.
5. **Display in leaderboards?**: si seleccionamos esta opción, el registro que vamos a crear se usará para crear un tablón de récords.
6. **Server API only?**: esta opción en nuestro caso no se usará.

Una vez rellenado estos datos se hace click en **save** y se guarda, pasando así a añadir otra variable, hasta tener todas las usadas por nuestro juego (ver figura A.9).

A.4. Despliegue en Android. Google play

Una de las posibilidades que nos ofrece Haxe y NME es la compilación para la plataforma Android de forma rápida y sencilla. Este proceso ya se ha explicado con anterioridad en el libro, pero en este anexo se completa con una parte importante: la firma de la aplicación y su distribución en Google Play.

A.4.1. Firmando la APK

Para publicar una aplicación en Google Play, es necesario firmarla con una clave privada de desarrollador. La firma se usa para controlar que las versiones del producto vengan del mismo desarrollador.

Para generar esta clave, sólo tenemos que realizar una serie de pasos muy sencillos. Este proceso se realiza gracias a una de las aplicaciones que instala Java en nuestra máquina: la aplicación **keytool**[20].

```
keytool -genkey -v -keystore archivo.keystore
-alias UnALiasParaRecordarLaKey
-keyalg RSA -keysize 2048 -validity 10000
```

Este comando crea un *archivo.keystore* válido para 10000 días. El tiempo de validez de dicha clave se puede usar para crear aplicaciones con cierta duración. Una vez la fecha sea alcanzada, el certificado no será válido y no se podrá instalar la aplicación.

Una vez creado el archivo, sólo necesitaremos indicarle a nuestro proyecto que añada el certificado al proceso de compilación. Abriremos el archivo .nmml y añadiremos esta línea:

```
<certificate path="ruta/al/certificado/archivo.keystore"
 alias="ElAliasDeCreacion" />
```

Una vez añadida dicha línea, pasamos a compilar y nuestro proyecto será compilado y firmado con nuestra clave privada.

[20] Si en Windows no se encuentra la keytool, se puede buscar en el directorio de instalación de Java.

A.4.2. Subiendo la APK a Google Play. Distribución

Para subir nuestro juego a Google Play necesitaremos una cuenta en la consola de desarrollador[21]. Una vez registrados, y tras pagar los 25$ y rellenar los datos pertinentes, dispondremos de una cuenta activa[22]. A continuación, podremos subir nuestros juegos.

Para añadir un juego nuevo iremos a la pestaña **Todas las Aplicaciones** y pulsaremos el botón **Añadir Nueva Aplicación**, activando un formulario para configurar nuestra aplicación (idioma del juego y su nombre). Una vez introducidos estos datos, podemos optar por subir la APK ya firmada o por rellenar los datos del juego:

- **Rellenar los datos**: esta opción nos llevará a la ventana de **Entrada en Play Store**. Básicamente, es un formulario para rellenar con datos sobre el juego y capturas de pantalla. No es un proceso complejo y está muy bien explicado en la web. Es posible añadir traducciones a este formulario pulsando en el botón **Añadir traducciones**. Esto creará otra página con el mismo formulario pero que tendremos que rellenar en el idioma seleccionado.
- **Subir la APK**: en el caso de optar por subir primero la APK, navegaremos hacia el apartado APK. En esta página podemos seleccionar tres tipos de subida: **Producción**, **Beta Testing**, o **Alpha Testing**. Si optamos por usar Beta Testing o Alpha Testing, sólo los usuarios añadidos podrán descargar nuestra aplicación. Una vez subida la aplicación, podemos comprobar para cuántos dispositivos esta disponible. Este número se puede modificar excluyendo los dispositivos que nosotros deseemos.

Una vez seleccionada una de las 2 opciones, será necesario realizar la otra cambiando de ventana. Por último, necesitaremos configurar la distribución en **Precios y distribución**, un apartado donde seleccionaremos la lista de países donde queremos distribuir nuestro juego, seleccionando si nuestro juego es gratuito o de pago. Para poder configurar una cuenta de pago, necesitaremos tener enlazada una cuenta Google Wallet. Por último, será necesario confirmar los 2 últimos apartados obligatorios, **Directrices para contenido** y **Leyes de exportación de EE.UU**. La opción de **Excluir marketing** es opcional y no obligatoria, a no ser que realmente no queramos promocionar nuestro juego.

Con todo este proceso realizado, sólo necesitaremos confirmar la publicación de nuestro juego y, pasadas unas horas, nuestro juego ya estará disponible en Google Play.

[21] https://play.google.com/apps/publish/

[22] El pago es único. Sólo se paga una vez para activar la cuenta. Es posible que solicite cuenta en Google Wallet

Bibliografía

[1] T. Akenine-Moller, E. Haines, and N. Hoffman. *Real-Time Rendering (3rd Edition)*. AK Peters, 2008.

[2] M. Buckland. *Programming Game AI by Example*. Wordware Game Developer's Library, 2004.

[3] M.J. Dickheiser. *C++ for Game Programmers (2nd Edition)*. Charles River Media, 2007.

[4] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissided. *Design Patterns*. Addison Wesley, 2008.

[6] J Gregory. *Game Engine Architecture*. AK Peters, 2009.

[7] R. Ierusalimschy. *Programming in LUA (2nd Edition)*. Lua.org, 2006.

[8] G. Junker. *Pro OGRE 3D Programming*. Apress, 2006.

[9] S.D. Meyers. *Effective STL: 50 specific ways to improve your use of the standard template library*. Addison-Wesley professional computing series. Addison-Wesley, 2001.

[10] I. Millington and J. Funge. *Artificial Intelligence for Games*. Morgan Kaufmann, 2009.

[11] R.J. Rost and J.M. Kessenich. *OpenGL Shading Language*. Addison-Wesley Professional, 2006.

[12] S.J. Russell and Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 1998.

[13] H. Schildt. *C++ from the Ground Up (3rd Edition)*. McGraw-Hill Osborne, 2003.

[14] A. Silberschatz, P.B. Galvin, and G. Gagne. *Fundamentos de Sistemas Operativos*. McGraw Hill, 2006.

[15] R.M. Stallman and GCC Developer Community. *Using GCC: The GNU Compiler Collection Reference Manual*. GNU Press, 2003.

[16] R.M. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. GNU Press, 2002.

[17] B. Stroustrup. *El lenguaje de programaci'on C++*. Pearson Education, 2001.

[18] G. Weiss. *Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.

Este libro fue maquetado mediante el sistema de composición de textos LaTeX utilizando software del proyecto GNU.

Ciudad Real, 15 de Julio de 2013