

Desarrollo de Videojuegos Técnicas Avanzadas



```
0:  
.cfi_startproc  
xorl %eax, %eax  
testl %esi, %esi  
pushq %rbx  
.cfi_def_cfa_offset 16  
.cfi_offset 3, -16  
j .L2  
movq %r8, %rcx  
shl $2, %r8  
shl %r8  
andl $3, %r8d  
cmpl %esi, %r8d  
cmova %esi, %r8d  
xorl %edx, %edx  
testl %r8d, %r8d  
movl %r8d, %ebx  
je .L11  
.p2align 4,,10  
.p2align 3
```

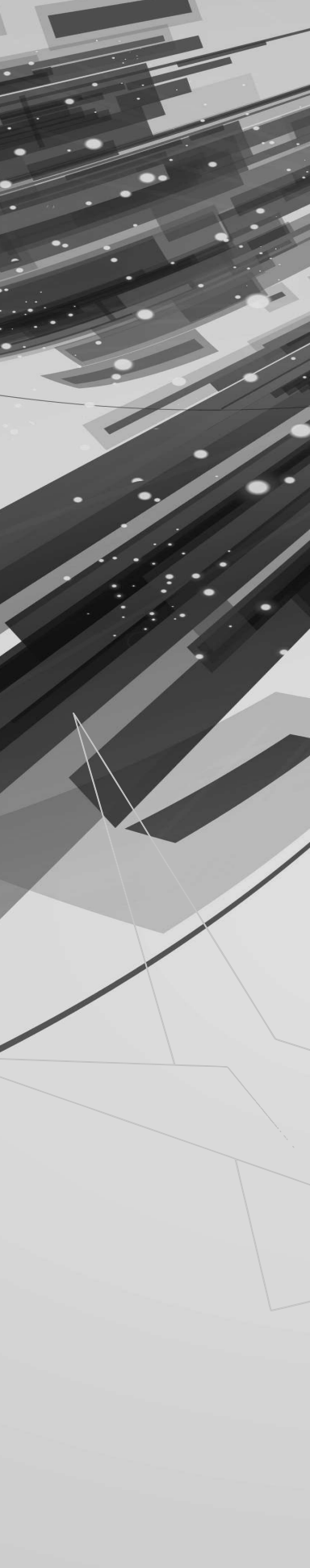
Francisco Moya Fernández
Carlos González Morcillo
David Villa Alises
Sergjo Pérez Camacho
Miguel A. Redondo Duque
César Mora Castro
Félix J. Villanueva Molina
Miguel García Corchero

3

Desarrollo de Videojuegos Técnicas Avanzadas

```
0:  
.cfi startproc  
xorl %eax, %eax  
testl %esi, %esi  
pushq %rbx  
.cfi def cfa offset 16  
.cfi_offset 3, -16  
jmp .L2  
movq %rcx  
movq %r8d  
movq %r8  
andl $3, %r8d  
cmpl %esi, %r8d  
movq %esi, %r8d  
xorl %edx, %edx  
testl %r8d, %r8d  
movl %r8d, %ebx  
je .L11  
.p2align 4,,10  
.p2align 3
```

Francisco Moya Fernández
Carlos González Morcillo
David Villa Alises
Sergio Pérez Camacho
Miguel A. Redondo Duque
César Mora Castro
Félix J. Villanueva Molina
Miguel García Corchero



Título: Desarrollo de Videojuegos: Técnicas Avanzadas
Autores: Francisco Moya Fernández, Carlos González Morcillo, David Villa Alises, Sergio Pérez Camacho, Miguel A. Redondo Duque, César Mora Castro, Félix J. Villanueva Molina, Miguel García Corchero
ISBN: 978-84-686-1059-7 (de la Edición Física, a la venta en www.bubok.es)
ISBN (C): 978-84-686-1056-6 (ISBN de la Colección)
Publica: Bubok (*Edición Física*) LibroVirtual.org (*Edición electrónica*)
Edita: David Vallejo Fernández y Carlos González Morcillo
Diseño: Carlos González Morcillo

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



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 licenciadore. 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



Francisco Moya (2003, Doctor Ingeniero en Telecomunicación, Universidad Politécnica de Madrid). Desde 1999 trabaja como profesor de la Escuela Superior de Informática de la Universidad de Castilla la Mancha, desde 2008 como Profesor Contratado Doctor. Sus actuales líneas de investigación incluyen los sistemas distribuidos heterogéneos, la automatización del diseño electrónico y sus aplicaciones en la construcción de servicios a gran escala y en el diseño de sistemas en chip. Desde 2007 es también Debian Developer.



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 Villa (2009, Doctor Ingeniero Informático, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en materias relacionadas con las redes de computadores y sistemas distribuidos desde el 2002. Sus intereses profesionales se centran en los sistemas empujados en red, los sistemas ubicuos y las redes heterogéneas y virtuales. Es experto en métodos de desarrollo ágiles y en los lenguajes C++ y Python. Colabora con el proyecto Debian como maintainer de paquetes oficiales.



Sergio Pérez (2011, Ingeniero en Informática, Universidad de Castilla-La Mancha) trabaja como ingeniero consultor diseñando software de redes para Ericsson R&D. Sus intereses principales son GNU/Linux, las redes, los videojuegos y la realidad aumentada.



Miguel Ángel Redondo (2002, Doctor en Ingeniería Informática, Universidad de Castilla – La Mancha) es Profesor Titular de Universidad en la Escuela Superior de Informática de la UCLM en Ciudad Real, impartiendo docencia en asignaturas relacionadas con Interacción Persona-Computador y Sistemas Operativos. Su actividad investigadora se centra en la innovación y aplicación de técnicas de Ingeniería del Software al desarrollo de sistemas avanzados de Interacción Persona-Computador y al desarrollo de sistemas de e-Learning.



César Mora (2011, Ingeniero en Informática, Universidad de Castilla-La Mancha - UCLM) Tecnólogo en el grupo de investigación Oreto, desarrollando proyectos relacionados con Informática Gráfica, Visión Artificial y Realidad Aumentada.



Félix J. Villanueva (2009, Doctor en Ingeniería Informática, Universidad de Castilla-La Mancha) es contratado doctor e imparte docencia en el área de tecnología y arquitectura de computadores. Las asignaturas que imparte se centran en el campo de las redes de computadores con una experiencia docente de más de diez años. Sus principales campos de investigación en la actualidad son redes inalámbricas de sensores, entornos inteligentes y sistemas empotrados.



Miguel García es desarrollador independiente de Videojuegos en plataformas iOS, Android, Mac OS X, GNU/Linux y MS Windows y socio fundador de Atomic Flavor.

Prefacio

Este libro forma parte de una colección de cuatro volúmenes dedicados al Desarrollo de Videojuegos. Con un perfil principalmente técnico, estos cuatro libros cubren los aspectos esenciales en programación de videojuegos:

1. **Arquitectura del Motor.** En este primer libro se estudian los aspectos esenciales del diseño de un motor de videojuegos, así como las técnicas básicas de programación y patrones de diseño.
2. **Programación Gráfica.** El segundo libro de la colección se centra en los algoritmos y técnicas de representación gráfica y optimizaciones en sistemas de despliegue interactivo.
3. **Técnicas Avanzadas.** En este tercer volumen se recogen ciertos aspectos avanzados, como estructuras de datos específicas, técnicas de validación y pruebas o simulación física.
4. **Desarrollo de Componentes.** El último libro está dedicado a ciertos componentes específicos del motor, como la Inteligencia Artificial, Networking, Sonido y Multimedia o técnicas avanzadas de Interacción.

Sobre este libro...

Este libro que tienes en tus manos es una ampliación y revisión de los apuntes del *Curso de Experto en Desarrollo de Videojuegos*, impartido en la Escuela Superior de Informática de Ciudad Real de la Universidad de Castilla-La Mancha. Puedes obtener más información sobre el curso, así como los resultados de los trabajos creados por los alumnos en la web del mismo: <http://www.esi.uclm.es/videojuegos>.

La versión electrónica de este manual (y del resto de libros de la colección) puede descargarse desde la web anterior. El libro «físico» puede adquirirse desde la página web de la editorial online *Bubok* en <http://www.bubok.es>.



Requisitos previos

Este libro tiene un público objetivo con un perfil principalmente técnico. Al igual que el curso del que surgió, está orientado a la capacitación de profesionales de la programación de videojuegos. De esta forma, 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 es capaz de desarrollar programas de nivel medio en C y C++. Aunque se describen algunos aspectos clave de C++ a modo de resumen, es recomendable refrescar los conceptos básicos con alguno de los libros recogidos en la bibliografía del curso. De igual modo, se asume que el lector tiene conocimientos de estructuras de datos y algoritmia. El libro está orientado principalmente para titulados o estudiantes de últimos cursos de Ingeniería en Informática.

Programas y código fuente

El código de los ejemplos del libro pueden descargarse en la siguiente página web: <http://www.esi.uclm.es/videojuegos>. Salvo que se especifique explícitamente otra licencia, todos los ejemplos del libro se distribuyen bajo GPLv3.

Agradecimientos

Los autores del libro quieren agradecer en primer lugar a los alumnos de la primera edición del *Curso de Experto en Desarrollo de Videojuegos* por su participación en el mismo y el excelente ambiente en las clases, las cuestiones planteadas y la pasión demostrada en el desarrollo de todos los trabajos.

De igual modo, se quiere reflejar el agradecimiento especial al personal de administración y servicios de la Escuela Superior de Informática, por su soporte, predisposición y ayuda en todos los caprichosos requisitos que planteábamos a lo largo del curso.

Por otra parte, este agradecimiento también se hace extensivo a la Escuela de Informática de Ciudad Real y al Departamento de Tecnologías y Sistema de Información de la Universidad de Castilla-La Mancha.

Finalmente, los autores desean agradecer su participación a los colaboradores de esta primera edición: *Indra Software Labs*, la asociación de desarrolladores de videojuegos *Stratos* y a *Libro Virtual*.



Resumen

El objetivo de este módulo, titulado «Técnicas Avanzadas de Desarrollo» dentro del *Curso de Experto en Desarrollo de Videojuegos*, es profundizar en aspectos de desarrollo más avanzados que complementen el resto de contenidos de dicho curso y permitan explorar soluciones más eficientes en el contexto del desarrollo de videojuegos.

En este módulo se introducen aspectos básicos de jugabilidad y se describen algunas metodologías de desarrollo de videojuegos. Así mismo, también se estudian los fundamentos básicos de la validación y pruebas en este proceso de desarrollo. Por otra parte, en este módulo también se estudia un aspecto esencial en el desarrollo de videojuegos: la simulación física.

No obstante, uno de los componentes más importantes del presente módulo está relacionado con aspectos avanzados del lenguaje de programación C++, como por ejemplo el estudio en profundidad de la biblioteca SDL, y las optimizaciones.

Finalmente, el presente módulo se complementa con el estudio de la gestión de *widgets* y el estudio de la plataforma de desarrollo de videojuegos *Unity*, especialmente ideada para el desarrollo de juegos en plataformas móviles.



Índice general

1. Aspectos de Jugabilidad y Metodologías de Desarrollo	1
1.1. Jugabilidad y Experiencia del Jugador	1
1.1.1. Introducción	1
1.1.2. Caracterización de la Jugabilidad	3
1.1.3. Facetas de la Jugabilidad	4
1.1.4. Calidad de un juego en base a la Jugabilidad	8
1.2. Metodologías de Producción y de Desarrollo	11
1.2.1. Pre-Producción	12
1.2.2. Producción	15
1.2.3. Post-Producción	17
1.3. Metodologías Alternativas	17
1.3.1. Proceso Unificado del Juego	18
1.3.2. Desarrollo Incremental	18
1.3.3. Desarrollo Ágil y Scrum	19
1.3.4. Desarrollo Centrado en el Jugador	19
2. C++ Avanzado	23
2.1. Programación genérica	23
2.1.1. Algoritmos	24
2.1.2. Predicados	27
2.1.3. Functors	28
2.1.4. Adaptadores	30
2.1.5. Algoritmos idempotentes	32

2.1.6. Algoritmos de transformación	35
2.1.7. Algoritmos de ordenación	39
2.1.8. Algoritmos numéricos	42
2.1.9. Ejemplo: inventario de armas	42
2.2. Aspectos avanzados de la STL	46
2.2.1. Eficiencia	46
2.2.2. Semántica de copia	50
2.2.3. Extendiendo la STL	52
2.2.4. Allocators	55
2.3. Estructuras de datos no lineales	58
2.3.1. Árboles binarios	59
2.3.2. Recorrido de árboles	75
2.3.3. <i>Quadtree</i> y <i>octree</i>	78
2.4. Patrones de diseño avanzados	81
2.4.1. Forma canónica ortodoxa	81
2.4.2. Smart pointers	82
2.4.3. Handle-body	88
2.4.4. Command	91
2.4.5. Curiously recurring template pattern	95
2.4.6. Acceptor/Connector	100
2.5. C++11: Novedades del nuevo estándar	104
2.5.1. Compilando con g++	104
2.5.2. Cambios en el núcleo del lenguaje	105
2.5.3. Cambios en la biblioteca de C++	118

3. Técnicas específicas 123

3.1. Plugins	123
3.1.1. Entendiendo las bibliotecas dinámicas	124
3.1.2. Plugins con <code>libdl</code>	127
3.1.3. Plugins con <code>Glib</code> <code>gmodule</code>	133
3.1.4. Carga dinámica desde Python	135
3.1.5. Plugins como objetos mediante el patrón <i>Factory Method</i>	135
3.1.6. Plugins multi-plataforma	138
3.2. Serialización de objetos	140
3.2.1. <i>Streams</i>	140

3.2.2. Serialización y Dependencias entre objetos	146
3.2.3. Serialización con Boost	155
3.3. C++ y scripting	159
3.3.1. Consideraciones de diseño	160
3.3.2. Invocando Python desde C++ de forma nativa	161
3.3.3. Librería boost	162
3.3.4. Herramienta SWIG	168
3.3.5. Conclusiones	169
4. Optimización	171
4.1. Perfilado de programas	173
4.1.1. El perfilador de Linux <i>perf</i>	175
4.1.2. Obteniendo ayuda	177
4.1.3. Estadísticas y registro de eventos	177
4.1.4. Multiplexación y escalado	178
4.1.5. Métricas por hilo, por proceso o por CPU	180
4.1.6. Muestreo de eventos	180
4.1.7. Otras opciones de <i>perf</i>	185
4.1.8. Otros perfiladores	185
4.2. Optimizaciones del compilador	187
4.2.1. Variables registro	188
4.2.2. Código estático y funciones <i>inline</i>	189
4.2.3. Eliminación de copias	195
4.2.4. Volatile	196
4.3. Conclusiones	197
5. Validación y pruebas	199
5.1. Programación defensiva	199
5.1.1. Sobrecarga	201
5.2. Desarrollo ágil	202
5.3. TDD	203
5.3.1. Las pruebas primero	203
5.3.2. rojo, verde, refactorizar	204
5.4. Tipos de pruebas	204
5.5. Pruebas unitarias con google-tests	206
5.6. Dobles de prueba	208

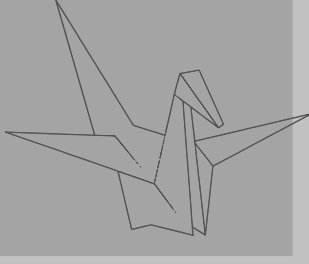
5.7. Dobles de prueba con google-mock	210
5.8. Limitaciones	215
6. Simulación física	217
6.1. Introducción	217
6.1.1. Algunos Motores de Simulación	218
6.1.2. Aspectos destacables	219
6.1.3. Conceptos Básicos	221
6.2. Sistema de Detección de Colisiones	222
6.2.1. Formas de Colisión	222
6.2.2. Optimizaciones	225
6.2.3. Preguntando al sistema...	226
6.3. Dinámica del Cuerpo Rígido	227
6.4. Restricciones	228
6.5. Introducción a Bullet	229
6.5.1. Pipeline de Físicas de Cuerpo Rígido	230
6.5.2. Hola Mundo en Bullet	232
6.6. Integración manual en Ogre	240
6.7. Hola Mundo en OgreBullet	243
6.8. RayQueries	247
6.9. TriangleMeshCollisionShape	250
6.10 Detección de colisiones	251
6.11 Restricción de Vehículo	253
6.12 Determinismo	257
6.13 Escala de los Objetos	259
6.14 Serialización	260
7. Gestión de Widgets	261
7.1. Interfaces de usuario en videojuegos	261
7.1.1. Menú	264
7.1.2. HUD	265
7.2. Introducción CEGUI	265
7.2.1. Inicialización	266
7.2.2. El Sistema de Dimensión Unificado	268
7.2.3. Detección de eventos de entrada	269
7.3. Primera aplicación	271

7.4. Tipos de <i>Widgets</i>	274
7.5. <i>Layouts</i>	275
7.6. Ejemplo de interfaz	276
7.7. Editores de <i>layouts</i> gráficos	279
7.8. Scripts en detalle	280
7.8.1. <i>Scheme</i>	280
7.8.2. <i>Font</i>	281
7.8.3. <i>Imageset</i>	281
7.8.4. <i>LookNFeel</i>	282
7.9. Cámara de Ogre en un <i>Window</i>	282
7.10 Formateo de texto	285
7.10.1 Introducción	285
7.10.2 Color	286
7.10.3 Formato	286
7.10.4 Insertar imágenes	286
7.10.5 Alineamiento vertical	287
7.10.6 <i>Padding</i>	287
7.10.7 Ejemplo de texto formateado	288
7.11 Características avanzadas	289

8. Plataformas Móviles 291

8.1. Método de trabajo con un motor de videojuegos	291
8.1.1. Generación de contenido externo al motor	291
8.1.2. Generación de contenido interno al motor	292
8.2. Creación de escenas	292
8.3. Creación de <i>prefabs</i>	296
8.4. Programación de scripts	296
8.4.1. Algunos scripts básicos	298
8.4.2. <i>Triggers</i>	299
8.4.3. Invocación de métodos retardada	299
8.4.4. Comunicación entre diferentes scripts	299
8.4.5. Control del flujo general de la partida	301
8.4.6. Programación de enemigos	303
8.4.7. Programación del control del jugador	305
8.4.8. Programación del <i>interface</i>	307

8.5. Optimización	309
8.5.1. <i>Light mapping</i>	310
8.5.2. <i>Occlusion culling</i>	310
8.6. Resultado final	311



Aspectos de Jugabilidad y Metodologías de Desarrollo

Miguel Ángel Redondo Duque

En este capítulo se introducen aspectos básicos relativos al concepto de jugabilidad, como por ejemplo aquellos vinculados a su caracterización en el ámbito del desarrollo de videojuegos o las facetas más relevantes de los mismos, haciendo especial hincapié en la parte relativa a su calidad.

Por otra parte, en este capítulo también se discuten los fundamentos de las metodologías de desarrollo para videojuegos, estableciendo las principales fases y las actividades desarrolladas en ellas.

1.1. Jugabilidad y Experiencia del Jugador

1.1.1. Introducción

En el desarrollo de sistemas interactivos es fundamental contar con la participación del usuario. Por ello, se plantea lo que vienen a denominarse métodos de **Diseño Centrado en el Usuario** que se aplican, al menos, al desarrollo software que soporta directamente la interacción con el usuario. En otras palabras, es fundamental contar con su participación para que tengamos la garantía de que se le va a proporcionar buenas experiencias de uso. El software para videojuegos se puede considerar como un caso particular de sistema interactivo

por lo que requiere un planteamiento similar en este sentido aunque, en este ámbito, los términos y conceptos que se emplean para este propósito varían ligeramente.

En el desarrollo de videojuegos es importante tener siempre presente que hay que lograr que el jugador sienta las mejores experiencias (entretenimiento, diversión, et.) posibles durante su utilización. El incremento de estas experiencias revierte directamente en el éxito del videojuego. Así pues, es conveniente conocer las propiedades que caracterizan dichas experiencias, poder medirlas durante el proceso de desarrollo y así asegurar su éxito y calidad. En adelante, nos referiremos a esto como **Experiencia del Jugador**.

La Experiencia del Jugador suele medirse utilizándose el concepto de **Jugabilidad** como propiedad característica de un videojuego, aunque su caracterización y forma de medirla no es algo plenamente formalizado e implantado en la industria del desarrollo de videojuegos.

Como paso previo para entender los conceptos de Jugabilidad y, por extensión, de Experiencia del Jugador, conviene repasar un concepto fundamental en el ámbito de los sistemas interactivos que es la Usabilidad. La usabilidad se refiere a la capacidad de un software de ser comprendido, aprendido, usado y ser satisfactorio para el usuario, en condiciones específicas de uso o la eficiencia y satisfacción con la que un producto permite alcanzar objetivos específicos a usuarios específicos en un contexto de uso específico. Para entender y medir la Usabilidad, se han identificado una serie de propiedades como son: efectividad, eficiencia, satisfacción, aprendizaje y seguridad [4] [34] [27] [13]. Éstas son las propiedades que son objeto de medición y, a partir de ellas, se puede valorar el grado de Usabilidad de un sistema.

El desarrollo de software usable redundante directamente en reducción de costes de producción, optimización del mantenimiento e incremento de la calidad final del producto. Además, las propiedades que caracterizan la Usabilidad influyen muy directamente en el uso que los usuarios hacen, contribuyendo incrementar su satisfacción, su productividad en la realización de tareas y reduciendo su nivel de estrés. En definitiva, la Usabilidad puede considerarse como un reflejo de la Experiencia del Usuario en un sistema interactivo que soporta la realización de una serie de tareas específicas para lograr un objetivo bien definido.

Según Nielsen Norman Group se define la *Experiencia del Usuario como la sensación, sentimiento, respuesta emocional, valoración y satisfacción del usuario respecto a un producto, resultado del proceso de interacción con el producto y de la interacción con su proveedor* [28]. En este sentido, cabe destacar la importancia de juegan diversos conceptos como la utilidad, la usabilidad, la deseabilidad, la accesibilidad, facilidad de uso, lo valioso del producto y lo creíble que pueda ser para el usuario. La Experiencia de Usuario está estrechamente relacionado con el contexto de uso del sistema interactivo, el contenido manipulado y los usuarios que lo usan. Lo que significa que variando alguno de estos elementos, el resultado puede ser totalmente diferente e incluso opuesto.

La relación que existe entre Experiencia de Usuario y Usabilidad puede considerarse equivalente a la que existe entre Experiencia del Jugador y Jugabilidad, aunque no se trata de una simple traslación del dominio de aplicación. Así lo vamos a considerar para explicar cómo se puede caracterizar la Jugabilidad y que en base a su medición se obtenga una valoración de la Experiencia del Jugador. Además, se apuntarán algunas ideas metodológicas orientadas a lograr mejores desarrollos de videojuegos, desde el punto de vista de la Jugabilidad.

1.1.2. Caracterización de la Jugabilidad

La Jugabilidad extiende el concepto de usabilidad, pero no se reduce únicamente la idea de Usabilidad en el caso particular de los videojuegos. Tampoco sería correcto reducirla únicamente al grado de diversión de un juego. Para diferenciar claramente este concepto que es un tanto difuso, lo adecuado es representarlo por un conjunto de atributos o propiedades que lo caracterizan. Estos atributos podrán ser medidos y valorados, para así comparar y extraer conclusiones objetivas. Este trabajo fue realizado por [15] que define la Jugabilidad como *el conjunto de propiedades que describen la experiencia del jugador ante un sistema de juego determinado, cuyo principal objetivo es divertir y entretener “de forma satisfactoria y creíble”, ya sea solo o en compañía.*

Es importante remarcar los conceptos de satisfacción y credibilidad. El primero es común a cualquier sistema interactivo. Sin embargo, la credibilidad dependerá del grado en el que se pueda lograr que los jugadores se impliquen en el juego.

Hay que significar que los atributos y propiedades que se utilizan para caracterizar la Jugabilidad y la Experiencia del Jugador, en muchos casos ya se han utilizado para caracterizar la Usabilidad, pero en los videojuegos presentan matices distintos. Por ejemplo, el “Aprendizaje” en un videojuego puede ser elevado, lo que puede provocar que el jugador se vea satisfecho ante el reto que supone aprender a jugarlo y, posteriormente, desarrollar lo aprendido dentro del juego.

Un ejemplo lo tenemos en el videojuego *Prince of Persia*, donde es difícil aprender a controlar nuestro personaje a través de un mundo virtual, lo que supone un reto en los primeros compases del juego. Sin embargo, en cualquier otro sistema interactivo podría suponer motivo suficiente de rechazo. Por otro lado, la “Efectividad” en un juego no busca la rapidez por completar una tarea, pues entra dentro de la naturaleza del videojuego que el usuario esté jugando el máximo tiempo posible y son muchos los ejemplos que podríamos citar.

Los atributos a los que hacemos referencia para caracterizar la Jugabilidad son los siguientes:

- **Satisfacción.** Agrado o complacencia del jugador ante el videojuego y el proceso de jugarlo.
- **Aprendizaje.** Facilidad para comprender y dominar el sistema y la mecánica del videojuego. Más adelante se indica cómo estos conceptos se definen en lo que se denomina Gameplay y que se construye durante el proceso de desarrollo del juego.
- **Efectividad.** Tiempo y recursos necesarios para ofrecer diversión al jugador mientras éste logra los objetivos propuestos en el videojuego y alcanza su meta final.
- **Inmersión.** Capacidad para creerse lo que se juega e integrarse en el mundo virtual mostrado en el juego.
- **Motivación.** Característica del videojuego que mueve a la persona a realizar determinadas acciones y a persistir en ellas para su culminación.
- **Emoción.** Impulso involuntario originado como respuesta a los estímulos del videojuego, que induce sentimientos y que desencadena conductas de reacción automática.
- **Socialización.** Atributos que hacen apreciar el videojuego de distinta manera al jugarlo en compañía (multijugador), ya sea de manera competitiva, colaborativa o cooperativa.

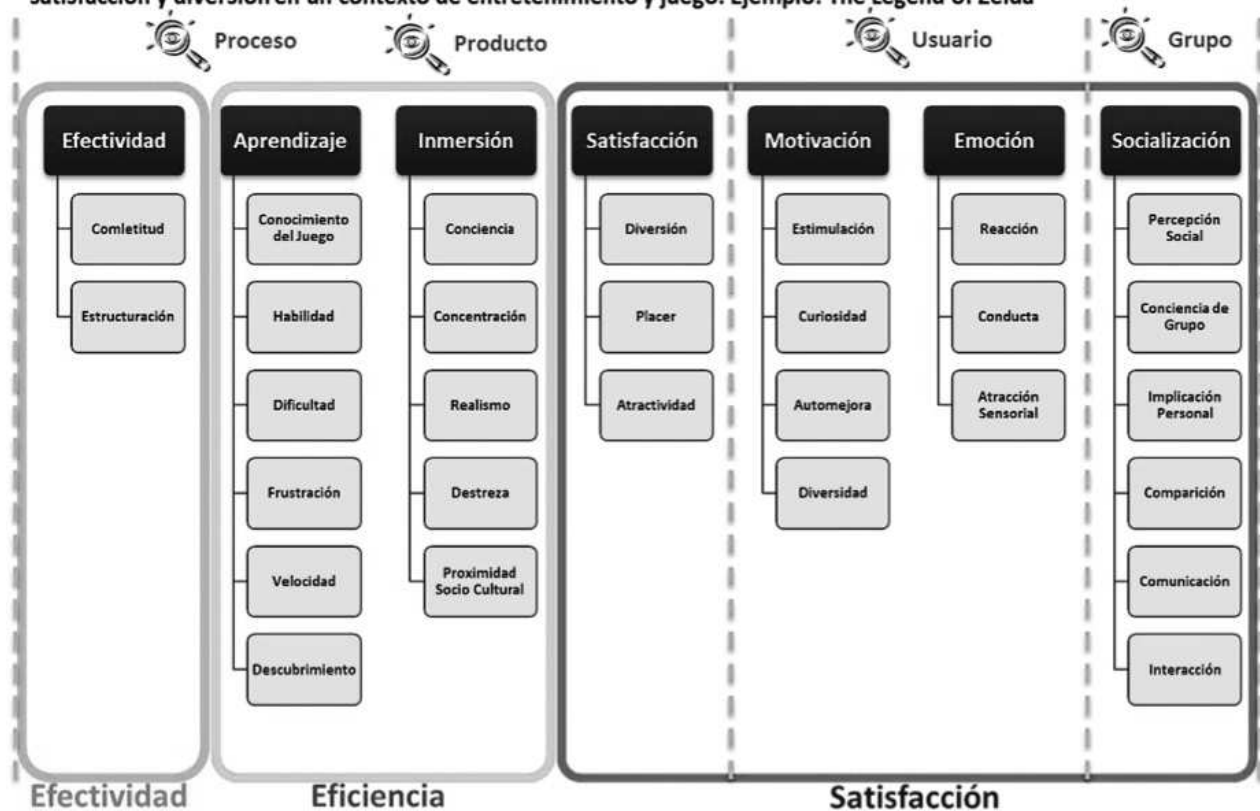
La figura 1.1 muestra como estos atributos y algunos otros más pueden estar relacionados con el concepto de Usabilidad tal y como se recoge en las normas ISO/IEC-9241. Hay algunos atributos que están relacionados con el videojuego (producto) y otros se vinculan al proceso de desarrollo del juego (desarrollo), algunos hacen referencia a su influencia sobre el jugador/es (usuarios o grupos de usuarios).

1.1.3. Facetas de la Jugabilidad

Uno de los objetivos, una vez definida la Jugabilidad, es poder medirla o cuantificarla. Este proceso es costoso debido a la cantidad de objetivos no funcionales que afectan a la Experiencia del Jugador. Como plantea [15], una buena estrategia es la de considerar una representación de la Jugabilidad basada en facetas de la misma.

La organización en facetas puede considerarse una subdivisión lógica de la Jugabilidad global en jugabilidades un poco más específicas. Cada una de estas facetas facilitará la identificación y medición de las propiedades introducidas anteriormente. Además, así será más fácil relacionar la Jugabilidad con los elementos particulares de un videojuego. A continuación se enumeran algunas facetas que podrían considerarse particulares. Sin embargo, esta lista no es una lista cerrada y, en algún juego particular, podría aparecer y proponerse alguna otra faceta que fuese objeto de consideración:

Jugabilidad: El grado en el cual usuarios específicos pueden alcanzar metas especificadas con efectividad, eficiencia, satisfacción y diversión en un contexto de entretenimiento y juego. Ejemplo: The Legend of Zelda



Usabilidad (ISO 9241-11): La medida en que un producto puede ser usado por usuarios específicos para lograr los objetivos especificados con efectividad, eficiencia y satisfacción en un contexto de uso. Ejemplo: Procesador de Textos.

Figura 1.1: Relación entre atributos de Usabilidad y de Jugabilidad

- **Jugabilidad Intrínseca.** Se trata de la Jugabilidad medida en la propia naturaleza del juego y cómo se proyecta al jugador. Está ligada al diseño del Gameplay que se describe más adelante. La forma de valorarla pasa por analizar cómo se representan las reglas, los objetivos, el ritmo y las mecánicas del videojuego.
- **Jugabilidad Mecánica.** Es la Jugabilidad asociada a la calidad del videojuego como sistema software. Está ligada a lo que sería el motor del juego, haciendo hincapié en características como la fluidez de las escenas cinemáticas, la correcta iluminación, el sonido, los movimientos gráficos y el comportamiento de los personajes del juego y del entorno, sin olvidar los sistemas de comunicación en videojuegos que incorporan un modo multijugador.

- **Jugabilidad Interactiva.** Es la faceta asociada a todo lo relacionado con la interacción con el usuario, el diseño de la interfaz de usuario, los mecanismos de diálogo y los sistemas de control.
- **Jugabilidad Artística.** Está asociada a la calidad y adecuación artística y estética de todos los elementos del videojuego y a la naturaleza de éste. Entre ellos estarán la calidad gráfica y visual, los efectos sonoros, la banda sonora y las melodías del juego, la historia y la forma de narración de ésta, así como la ambientación realizada de todos estos elementos dentro del videojuego.
- **Jugabilidad Intrapersonal (o Personal).** Está relacionada con la percepción que tiene el propio usuario del videojuego y los sentimientos que le produce. Como tal, tiene un alto valor subjetivo.
- **Jugabilidad Interpersonal (o de Grupo).** Muestra las sensaciones o percepciones de los usuarios que aparecen cuando se juega en grupo, ya sea de forma competitiva, cooperativa o colaborativa. En relación a cualquier sistema interactivo con soporte para grupos, se relacionaría con lo que tiene que ver con percepción del grupo (o *awareness* de grupo).

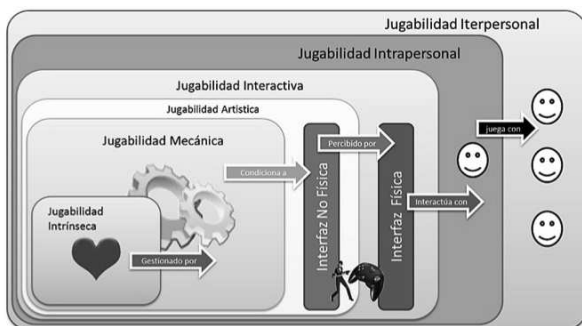


Figura 1.2: Relaciones entre las Facetas de la Jugabilidad

En [15] incluso se relacionan, a nivel interactivo, estas facetas para ilustrar cómo pueden ser las implicaciones e influencias que presentan. Esta relación se resume en la figura 1.2.

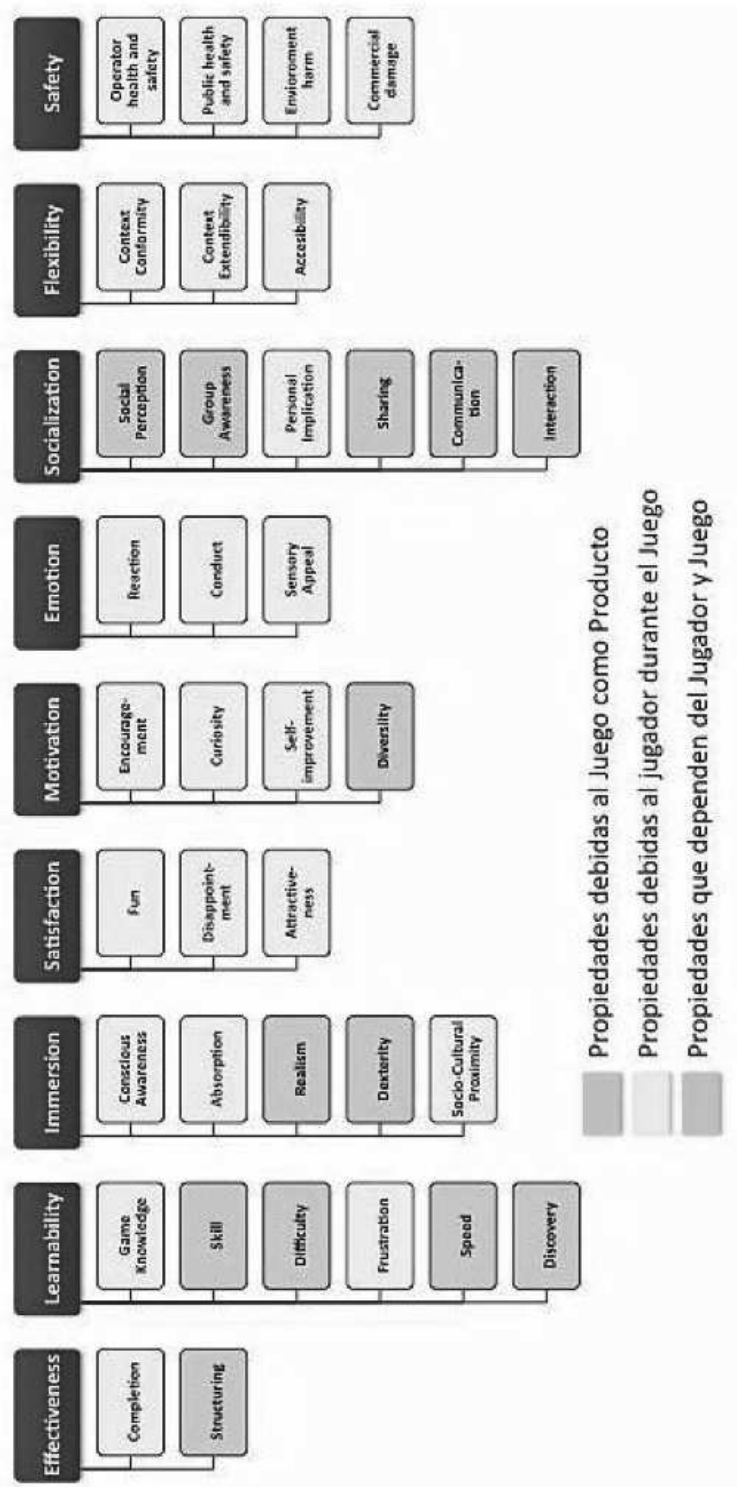


Figura 1.3: Clasificación de las propiedades de la calidad del producto y del proceso en un videojuego

1.1.4. Calidad de un juego en base a la Jugabilidad

Como ha quedado patente, el análisis de la calidad de un videojuego únicamente a partir de la Usabilidad o de la calidad de uso es insuficiente. Por esta razón, la caracterización de la Experiencia del Jugador en base a la Jugabilidad mediante una serie de propiedades, atributos y facetas proporciona un instrumento adicional. Con esto se pueden obtener medidas de la calidad de las experiencias durante el juego e incluso pueden utilizarse para extender el estándar de calidad ISO 25010 al contexto de los videojuegos.

Se puede destacar que hay una serie de propiedades de la Jugabilidad que influyen directamente en la Calidad del Producto y otras en la Calidad del Proceso de Uso y que, fundamentalmente tienen que ver con la habilidad del jugador para utilizarlo. La figura 1.3 ilustra cómo se clasifican estas propiedades.

En definitiva queda claro que la Experiencia de Juego requiere de su evaluación y esto podría llevarse a cabo en base a la Jugabilidad. Se han presentado algunos instrumentos conceptuales para ello. Sin embargo, el proceso no es fácil y menos si no se dispone de alguna herramienta que facilite la recopilación de información y el proceso de medición.

En este sentido, se desarrolló *PHET* como una herramienta que hace uso de técnicas de evaluación heurística para obtener y clasificar la información de una forma flexible y manejable. *PHET* está diseñada para facilitar la aplicación de los criterios de evaluación de la Jugabilidad anteriormente descritos, obtener resultados que permiten un análisis de la Experiencia del Jugador y de los elementos más relevantes de un videojuego en este sentido. El resultado final que proporciona es un informe de interpretación directa, visual y portable.

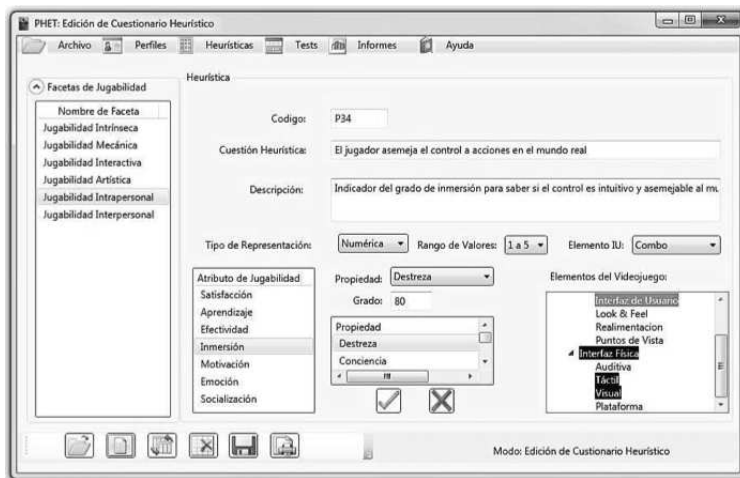


Figura 1.4: Ejemplo de creación de una cuestión de carácter heurístico

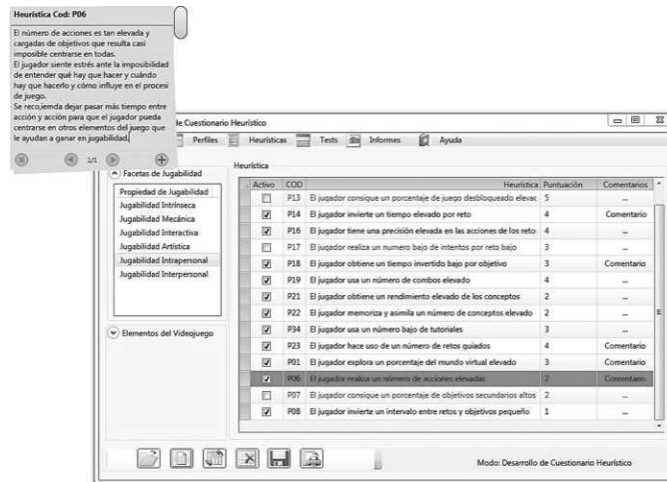


Figura 1.5: Respuesta de jugadores a los tests y cuestionarios

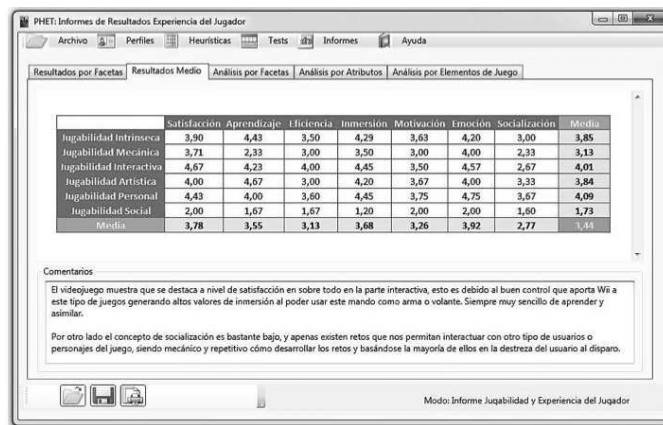


Figura 1.6: Resultados según las Facetas de la Jugabilidad

El procedimiento de uso de esta herramienta, básicamente, es la creación de heurísticas, cuestionarios y test (figura 1.4). Posteriormente la realización de estos tests por parte de los usuarios (figura 1.5) y visualización de los resultados. En este sentido, puede observarse como en la figura 1.6 se muestra una representación en base a las Facetas de la Jugabilidad y en la figura 1.7 se muestra el resultado en base a distintas propiedades.

Como se apuntó anteriormente, toda esta información se utiliza para obtener una valoración global de la Jugabilidad tal y como se muestra en la figura 1.8.

Todo esto tiene como interesante poder relacionarse con los elementos específicos del videojuego tal y como muestra la figura 1.8 que ilustra como la herramienta incorpora este soporte.

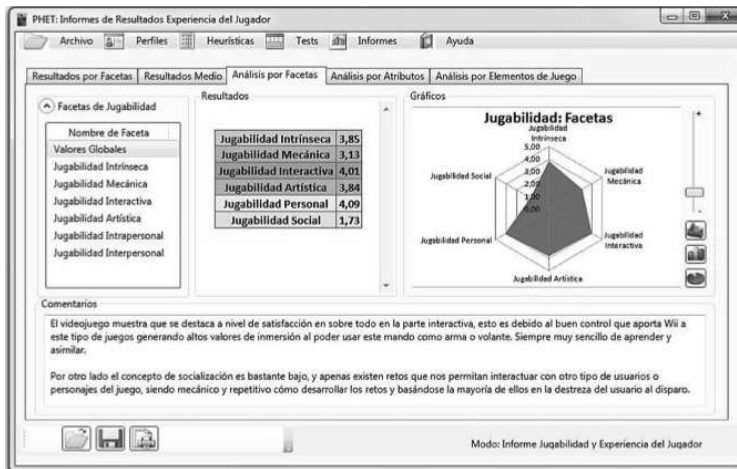


Figura 1.7: Resultados según diversas propiedades

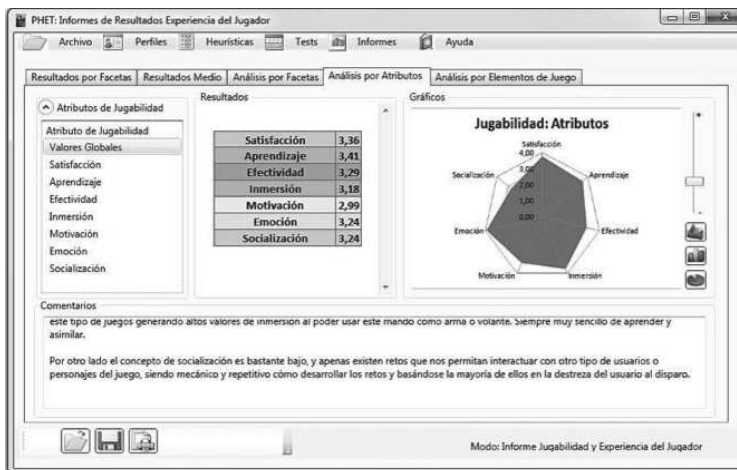


Figura 1.8: Valoración global de la Jugabilidad

1.2. Metodologías de Producción y de Desarrollo

Como en el desarrollo de cualquier producto software, para el construcción de un videojuego se requiere tener presente los principios fundamentales de la Ingeniería del Software y, especialmente, la metodología de desarrollo adecuada para el producto que se pretende construir y el contexto en el que se llevará a cabo. Sin embargo, el diseño y desarrollo de un videojuego no sólo se reduce al desarrollo técnico de un producto software sino que supone una actividad multidisciplinar que abarca desde la idea y concepción inicial hasta su versión final. Además, hay que tener presente que el desarrollo suele ser un proyecto de gran envergadura en tiempo y en dinero.

Por ejemplo, la producción de *Half-Life 2* supuso más de cuatro años de trabajo y un presupuesto final que se situó alrededor de los cincuenta millones de dólares. En estas situaciones, hay aspectos clave que requieren de una minuciosa planificación y metodología, ya que desde que se concibe un proyecto hasta que se comercializa transcurren grandes periodos de tiempo lo que en el ámbito tecnológico puede ser la causa de presentar importantes desfases y, por lo tanto, desembocar en un estrepitoso fracaso.

Así pues, se puede asegurar que la realización de un videojuego es una tarea delicada que requiere de una metodología específica. Sin embargo, las metodologías claramente establecidas para desarrollo de software no se adaptan a este proceso con garantías de calidad suficientes y no existe en este ámbito un claro planteamiento de cómo afrontar el trabajo.

No obstante, son muchos expertos los que coinciden en que el ciclo de vida del desarrollo de videojuegos se debe aproximar al del desarrollo de una película de cine, estableciendo tres fases claramente diferenciadas que son **Pre-Producción**, **Producción** y **Post-Producción**. A su vez en cada una de estas fases se identifican diversas etapas significativas y el equipo de producción se distribuye para colaborar en cada una de ellas.

El equipo de personas que suelen trabajar en un proyecto de desarrollo de un videojuego comercial de tamaño medio-alto oscila entre 40 y 140. Además, el tiempo que dura el proceso puede llegar a superar los tres años. Teniendo presente esto y, especialmente, su similitud con la producción de una película en [8] se propone una organización de referencia para el equipo de producción. Esta organización es la que aparece en la figura 1.9 y que ha sido traducida en [15].

La organización de las etapas del proceso de producción y la relación entre las mismas da lugar a un modelo de proceso que se asemeja al denominado Modelo en Cascada de Royce [31] en el que se establece la realización secuencial de una serie de etapas, impidiendo el comienzo de una nueva etapa sin la finalización de la anterior. Esta característica sacrifica de forma importante la posibilidad de paralelismo en el desarrollo de un videojuego y puede suponer una mala utilización de los recursos disponibles.

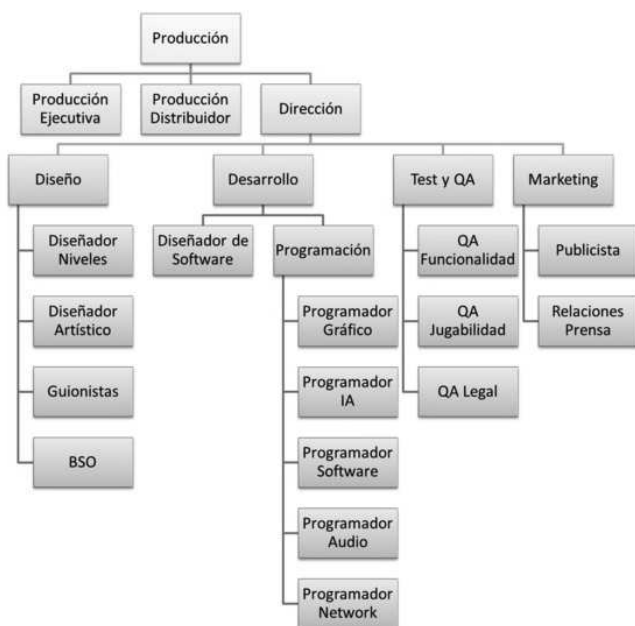


Figura 1.9: Organización de referencia de un equipo de producción de videojuegos

La distribución de las distintas etapas entre las tres fases mencionadas anteriormente tampoco está ampliamente consensuado. Predomina la idea de que la fase de Producción agrupa todo aquello que conlleva la obtención de elementos tangibles y elaborados para el juego, mientras que la fase de Pre-Producción se asocia con los procesos de obtención de elementos poco tangibles o preliminares, la cual se puede denominar Diseño Conceptual del Juego.

En cualquier caso, cabe destacar que la principal carga de trabajo se sitúa en lo que puede denominarse Diseño General del Juego y en el Diseño Técnico que es donde se aborda fundamentalmente el desarrollo del software del videojuego. Así pues, son estas etapas las que requieren mayor número de recursos y una mayor coordinación entre ellos. La figura 1.10 ilustra un posible planteamiento de organización de fases y etapas extraído de [8].

Describimos a continuación cada una de sus etapas de forma más detallada para comprender su objetivo de una forma más clara.

1.2.1. Pre-Producción

En la fase de Pre-Producción se lleva a cabo la concepción de la idea del juego, identificando los elementos fundamentales que lo caracterizarán y finalizando, si es posible, en un diseño conceptual del mismo. Esta información se organiza para dar lugar a lo que puede considerarse una primera versión del documento de diseño del juego

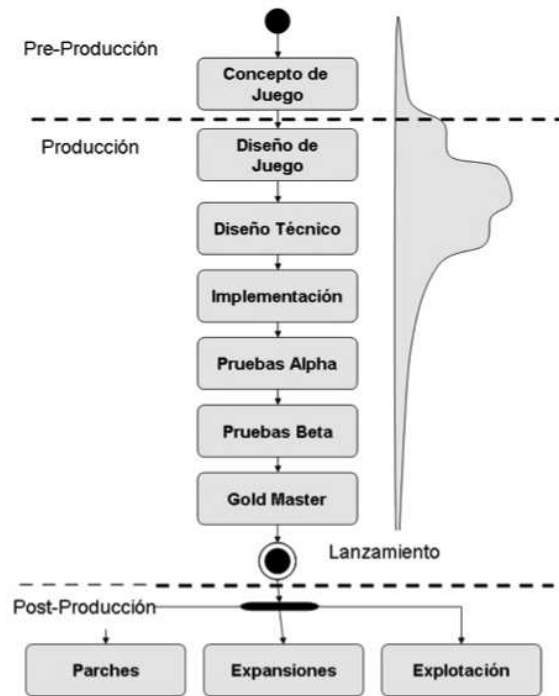


Figura 1.10: Organización de fases y etapas en la producción de un videojuego

o más conocido como **Game Design Document** (GDD). En este GDD, que debe ser elaborado por el equipo creativo del diseño de videojuegos, se debe identificar y fijar todo lo relacionado con el Diseño del Videojuego que será necesario abordar posteriormente (normalmente en la fase de Producción).

Como patrón de referencia y de acuerdo a lo establecido en [8], el GDD debe contener lo siguiente:

- **Genero.** Clasificación del juego según su naturaleza. La identificación del género al que pertenece el juego servirá para fijar una serie de características básicas para su posterior diseño.
- **Jugadores.** Modalidad de juego: individual o colectivo; multijugador o no; si los jugadores son personas o son máquinas; etc.
- **Historia.** Resumen de la historia del juego. Se realizará una primera aproximación de la trama o la historia a desarrollar durante el juego, destacando qué se quiere contar y cómo se pretende hacerlo. Esto se denomina *storyline* y *storytelling* respectivamente.
- **Bocetos.** Los bocetos son diseños preliminares, fundamentalmente, de los personajes y de los escenarios por los que se desarrollará la acción del juego.

- **Look and Feel.** A partir de los bocetos se define el aspecto gráfico y artístico del juego, colores, temas dominantes, musicalidad, técnicas de diseño 3D o 2D, posiciones de cámaras, etc.
- **Interfaz de Usuario.** Se apuntará la forma en la que el jugador interactuará con el juego y con qué mecanismos contará para ello: estilos de interacción, metáforas de interacción, paradigma de interacción, etc.
- **Objetivos.** Se fijan las metas del juego de acuerdo a la historia que se va a desarrollar.
- **Reglas.** Se establece qué acciones podrá desarrollar el jugador y cómo podrá hacerlo.
- **Características.** Se recogen las características principales de cada personaje del juego y de los elementos que intervienen durante su historia.
- **Gameplay.** Este es un concepto poco preciso y de muy amplio alcance, siendo ligeramente diferente su aplicación a cada tipo de juego. En esencia se trata de la naturaleza general del videojuego y de la interactividad que soportará. Es decir, los aspectos fundamentales que caracterizan la forma en la que se va a jugar, las cosas que el jugador va a poder hacer en el juego, la forma en la que el entorno del juego reaccionará a las acciones del jugador, mediadas por los correspondientes personajes, etc. Estos aspectos se describirán sin detallar en exceso a nivel de gráficos, sonido o de la propia historia.
- **Diseño de Niveles.** Se describen los niveles de dificultad que presentará el juego indicando cuántos será y cómo serán, así como los retos a los que el jugador se enfrentará en cada uno de ellos. En algunos casos, estos niveles también pueden estar asociados a etapas o fases del juego.
- **Requerimientos técnicos.** Se definen los requerimientos técnicos de máquina y dispositivos que requerirá el videojuego para su utilización.
- **Marketing.** Esta es una parte esencial en cualquier producto, pero especialmente en el caso de un videojuego todavía más. Muchos videojuegos con fuertes inversiones han sido prácticamente un fracaso por no abordar este aspecto desde las primeras fases de desarrollo. Por lo tanto, es necesario plantear, desde esta fase, la líneas maestras por las que se va a regir la generación de marketing y publicidad del producto.
- **Presupuesto.** Se realizará una primera aproximación al presupuesto que soportará el proyecto de desarrollo del videojuego.

Como se ha indicado anteriormente, esta primera versión del GDD será el punto de partida para iniciar la fase de Producción, pero cabe insistir sobre la importancia de uno de sus elementos: se trata del

Gameplay. Dado el carácter un tanto difuso de este concepto, consideremos como ejemplo el caso particular del conocido y clásico juego “*Space Invaders*”.

En este juego indicaríamos que se debe poder mover una nave alrededor del cuadrante inferior de la pantalla y disparar a una serie de enemigos que aparecen por la parte superior de la pantalla y que desaparecen cuando son alcanzados por los disparos. Estos enemigos tratan de atacarnos con sus disparos y presionándonos mediante la reducción de nuestro espacio de movimientos e intentando chocar contra nuestra nave.

El Gameplay tiene una implicación importantísima en la calidad final del juego y, por lo extensión, en la Jugabilidad del mismo. Luego los esfuerzos destinados a su análisis y planteamiento revertirán directamente en las propiedades que caracterizan la Jugabilidad. No obstante y para profundizar en más detalle sobre este aspecto, se recomienda consultar los siguientes libros: “*Rules of Play: Game Design Fundamentals*” [32] y “*Game Design: Theory and Practice*” [30].

1.2.2. Producción

La fase de Producción es la fase donde se concentra el trabajo principal, en volumen y en número de participantes, del proceso de desarrollo del videojuego, especialmente en lo que se denomina Diseño del Juego y Diseño Técnico. Hay que significar que este curso está orientado, fundamentalmente, a las tareas y técnicas relacionadas con el Diseño Técnico, pero no queremos dejar de situarlo en el contexto del proceso global que requiere llevarse a cabo para concebir y desarrollar un producto de estas características.

Siguiendo lo presentado en la figura 1.10, las etapas principales que se identifican en esta fase son las siguientes:

- **Diseño de Juego.** Esta es una etapa fundamental en la que se describen con alto nivel de detalle todos los elementos que formarán parte del juego. Principalmente, lo que se hace es refinar lo contemplado en el GDD para obtener su versión definitiva, diseñando en profundidad todos sus aspectos anteriormente especificados. Así se obtiene lo que se denomina Documento Técnico de Diseño (DTD) junto con la Biblia de la Historia, la Biblia del Arte y la primera versión del Motor del Juego.

Fundamentalmente, se debe trabajar en tres líneas de trabajo que vienen a caracterizar lo que se denomina diseño del juego y son las siguientes:

- Diseño Artístico que incluye:
 - La Biblia de la Historia donde se recogen todas las historias de los personajes y del mundo donde se desarrolla el juego así como el argumento completo del juego.
 - Biblia del Arte que incluye:

- ◊ Elementos sonoros del juego, es decir, voces, efectos, música, ambiente, etc. Incluso se empieza a trabajar en lo que debe dar lugar al Motor de Sonido.
- ◊ Visualización gráfica de los elementos con los que interactuarán los jugadores.
- ◊ Elementos gráficos como los modelos en 3D, las cámaras, las luces, los sprites, los tiles, etc. De igual manera que en el caso del sonido, esto sirve de punto de partida para comenzar a trabajar en lo que se denomina Motor Gráfico.
- Diseño de la Mecánica del Juego, en el que se trabaja en los aspectos que se enumeran a continuación:
 - Cómo se va a interactuar en el juego, cuáles son las reglas que lo rigen y cómo es la comunicación que tendrá lugar en caso de tratarse de un juego on-line.
 - Se debe diseñar el comportamiento, habilidades y otros detalles significativos de los personajes y del mundo que les rodea.
 - Se empieza a trabajar en el diseño del motor de Inteligencia Artificial (IA) que pueda requerir y en todo lo asociado con esto.
 - Se diseña lo que se denomina el Motor Físico con el objetivo de generar los aspectos físicos del juego como explosiones, disparos, etc.
- Motor del Juego que hace referencia a una serie de rutinas que permiten la representación de todos los elementos funcionales del juego. En síntesis puede decirse que agrupa todo lo relacionado con el Motor Gráfico, el Motor de Sonido, el Gestor de IA, el Motor Físico y todo el resto de gestores que pueden ser necesario para manejar el universo completo del videojuego.
- **Diseño Técnico.** Ésta se trata de la etapa que directamente está relacionada al desarrollo del software del juego y con lo se aborda en profundidad como contenido técnico esencial de este curso. Es aquí donde se describe cómo será implementado el juego. Para ello se hace uso de notaciones como UML y se plantea y decide la metodología de desarrollo software más apropiada según las características y, sobretodo, envergadura del producto software que se pretende implementar.

Es importante tener una descripción conceptual y precisa que permita ver el funcionamiento del software desde puntos de vistas estructurales, dinámicos, de interacción y de despliegue. En definitiva, se trata de un proyecto de desarrollo de software completo que debe incluir también una planificación de tareas a realizar, una asignación a los miembros del equipo de desarrolladores. Esto incluye la identificación de hitos importantes, las fechas de entrega y el análisis de riesgos.
- **Implementación.** En esta etapa debe abordarse la implementación de los elementos software del proyecto que se describieron

en la etapa anterior, utilizando para ello métodos, técnicas y herramientas como las que se trabajan a lo largo de este curso. Es posible que se detecten algunos errores del diseño inicial y que se requieran revisiones. En muchos casos, esta etapa y la anterior son repetidas de forma iterativa o se someten a ciclos iterativos. Esto, en muchos casos viene determinado por la metodología de desarrollo software que se emplea y que, como se ha apuntado anteriormente, depende de muchos factores como la envergadura del proyecto, los recursos disponibles, etc.

Generalmente, en este momento se suelen construir demos reducidas del juego que son objeto de publicación, contribuyendo así a materializar la campaña de marketing y publicidad que tan esencial es para lograr el éxito comercial del producto.

- **Pruebas Alpha.** Estas pruebas se abordan cuando tenemos ya partes del producto software terminado. También se suelen denominan pruebas Code Complete. Mediante las mismas, el producto se somete a diversas pruebas que realizan pequeños equipos que han estado llevando a cabo el proceso de diseño y desarrollo del juego. El objetivo de las mismas es buscar pequeños errores y refinar algunos aspectos. Uno de los aspectos más importantes que se valoran en esta etapa es la Jugabilidad del juego a través de diversas propiedades y facetas como se describió anteriormente.
- **Pruebas Beta.** En las pruebas Beta o también denominadas Content Complete se finaliza todo lo relacionado con contenidos como el decorado de las misiones, los gráficos, los textos en diferentes idiomas, doblaje del sonido, etc. Además, se trabaja para asegurar que los contenidos incluidos en el juego se ajustan a las leyes vigentes y a la ética establecida en aquellos países donde se pretende comercializar el juego. Estas pruebas son llevadas a cabo por personas ajenas al equipo de desarrollo.
- **Gold Master.** Esta etapa aborda una prueba definitiva con el producto final que se publicará y que se producirá. Obviamente, incluye todo el contenido artístico, técnico y documental (es decir, los manuales de usuario). En este momento, la publicidad debe ser la mayor posible, incluyéndose la realización de reportajes, artículos, etc.

1.2.3. Post-Producción

La fase de Post-Producción, en la que no nos vamos a detener ya que se aleja bastante del contenido tratado en el curso, aborda fundamentalmente la explotación y el mantenimiento del juego como si de cualquier otro producto software se tratase.

1.3. Metodologías Alternativas

El método descrito anteriormente prácticamente es un caso particular de aplicación del Modelo de Proceso en Cascada, que conlleva la

finalización de una etapa antes de poder abordar la siguiente. En el caso del desarrollo de software, esto condiciona bastante lo relacionado con las etapas de pruebas, cuya realización se retrasa en exceso quedando situada casi al final del desarrollo. En ese momento, depurar y solucionar cualquier problema, si es que es posible, puede resultar excesivamente costoso en tiempo y, en consecuencia, en dinero.

Precisamente, en el área del desarrollo de sistemas interactivos, está claramente establecido que las pruebas, sobretodo de Usabilidad, deben hacerse desde las primeras fases, incluso cuando los prototipos están únicamente a nivel de bocetos y en papel. Así pues, eso entra firmemente en contradicción con el hecho de que un videojuego se considere como un caso particular de sistema interactivo.

Por otro lado, la necesidad de evaluar lo antes posible las propiedades relacionadas con la Jugabilidad y la Experiencia del Jugador requieren plantear variaciones a la metodología de producción y desarrollo anteriormente presentada. Por esta razón, se describen a continuación algunos otros métodos alternativos que se utilizan en la industria del desarrollo de software de videojuegos.

1.3.1. Proceso Unificado del Juego

Tomando como punto de partida el Proceso Unificado de Desarrollo (o RUP) de IBM, en [14] se plantea la metodología denominada Proceso Unificado del Juego (o *Game Unified Process*, GUP). Este método se caracteriza por incentivar la comunicación entre los equipos de trabajo que abordan cada etapa del desarrollo, la documentación estricta de cada paso y por abordar el proceso de desarrollo de una forma iterativa y en ciclos muy cortos. Se puede considerar como una versión ágil de la metodología RUP particularizada para el desarrollo de software de videojuegos.

Además, este método propone la utilización del paradigma de Programación Extrema [7] como instrumento para agilizar el desarrollo del software del videojuego. Por lo tanto, esto es especialmente aplicable a lo que serían las etapas de Diseño del Juego, Diseño Técnico, Implementación y Pruebas.

1.3.2. Desarrollo Incremental

Otro método que puede ser adecuado, si se pretende potenciar la realización de pruebas en las fases tempranas y obtener realimentación, es el *Desarrollo Incremental* de Sikora [35]. Básicamente, se introduce la idea de disponer de un equipo de “jugadores” dentro del equipo de desarrolladores encargados de las pruebas. Estos “jugadores” siempre realizan una subetapa de pruebas en cada etapa antes de validar los resultados y asumir las tareas de la siguiente etapa.

1.3.3. Desarrollo Ágil y Scrum

Una de las metodologías que mejores resultados está produciendo recientemente en la industria del software de videojuegos es la propuesta por Clinton Keith dentro de su estudio de desarrollo *High Moon* [21]. Como ejemplo de caso de éxito en el que se ha aplicado esta metodología, cabe mencionar *DarkWatch*.

Esta metodología plantea la utilización de procesos ágiles de desarrollo de software, unido a los pilares básico de la metodología de desarrollo de productos Scrum [37].

El objetivo principal del método de Keith es hacer un diseño centrado en el jugador y en los resultados del proceso de desarrollo en cada una de sus fases. Así, se resalta la importancia de obtener la opinión del usuario en cada momento, por lo que intenta involucrar al equipo de pruebas lo antes posible. De esta forma, se facilitará la posibilidad detectar y solucionar a tiempo todos los posibles errores y se podrá analizar la Jugabilidad en cada momento para ir mejorándola continuamente, del mismo modo que se hace para el caso particular de la Usabilidad en un sistema interactivo.

Esta metodología requiere de la realización de importantes esfuerzos iniciales para lograr obtener prototipos básicos pero jugables y, por lo tanto, evaluables. Con estos prototipos se inicia un proceso iterativo en el que el equipo de pruebas lo utiliza y proporciona realimentación orientada a la mejora, especialmente de la Jugabilidad pero también de otros detalles que pueden caracterizar el producto final. Información mucho más detallada de cómo aplicar esta metodología puede encontrarse en el libro “*Agile Game Development with Scrum*” de [22].

1.3.4. Desarrollo Centrado en el Jugador

En esta subsección se va a describir la propuesta de [15] que está inspirada directamente en los principios fundamentales del *Diseño Centrado en el Usuario* (DCU) y de las metodologías de desarrollo software que se han derivado de los mismos.

La idea fundamental del Diseño Centrado en el Usuario, como ya se ha apuntado anteriormente, es la involucrar al usuario y hacerlo al principio de cualquier proceso de desarrollo, ya que muchos de los problemas del software se deben a una carencia en las fases iniciales del desarrollo, concretamente en las fases de elicitación y de análisis de requisitos. Esto ya ha sido contemplado en diversos estándares que plantean ciclos de vida del proceso que incluyen modelos de madurez para la Usabilidad como pilar fundamental que garantizar el éxito del producto en cuanto a la Experiencia del Usuario.

De la misma forma que el DCU es necesario para el desarrollo de aplicaciones que cubran los requisitos del usuario de forma adecuada, el Diseño Centrado en el Jugador es especialmente importante para considerar la diversidad y subjetividad de los perfiles de jugadores existentes. Además, esto contribuye directamente a la reducción de la proliferación de productos que requieren numerosos “parches” incluso

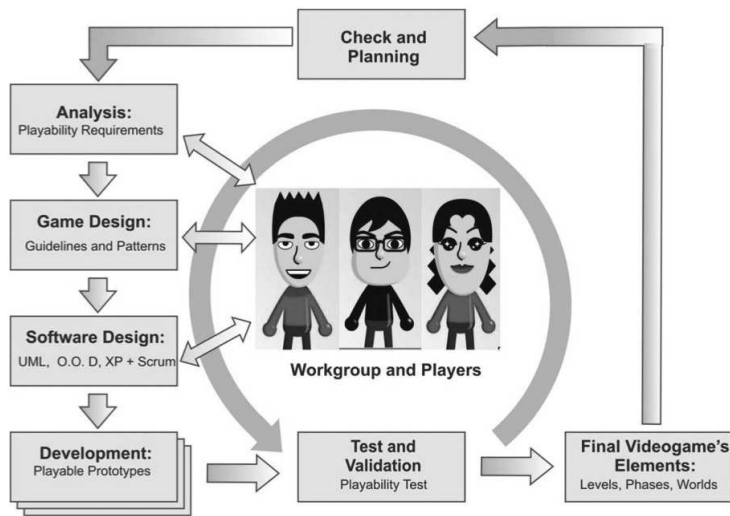


Figura 1.11: Método de Diseño Centrado en el Jugador de [15]

desde los primeros meses de vida en el mercado.

En este sentido, [15] propone un método inspirado en la metodología *PPIu+a* propuesta en [16] para Ingeniería de la Usabilidad y que se resume en la figura 1.11. Para facilitar su comprensión puede utilizarse la figura 1.12 en la que se relaciona y compara esta metodología.

En las fuentes citadas pueden encontrar muchos más detalles sobre las fases más destacables que son las de análisis, diseño, desarrollo y evaluación de elementos jugables. Especialmente, se plantea un patrón a seguir para la obtención de requisitos de Jugabilidad con ejemplos de aplicación, se proponen una serie de guías de estilo para llevar a cabo un diseño que fomente la Jugabilidad, se muestra cómo aplicar Scrum y programación extrema para la construcción de prototipos jugables y se describe cómo evaluar la Jugabilidad de los prototipos para obtener conclusiones sobre la experiencia del jugador.

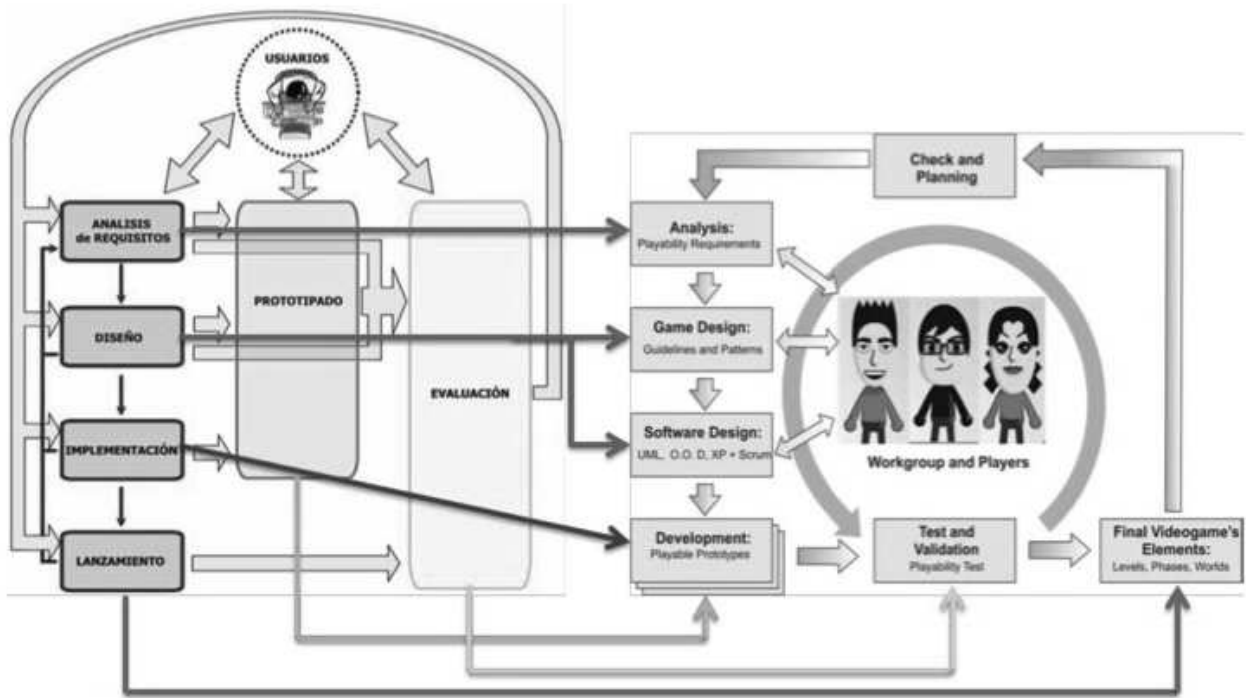
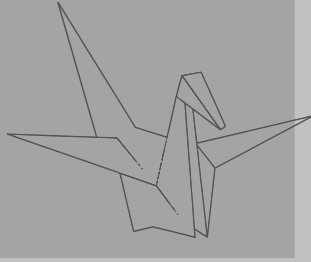


Figura 1.12: Comparación entre el método de Diseño Centrado en el Jugador y el de Diseño Centrado en el Usuario de MPIu+a



Capítulo 2

C++ Avanzado

Francisco Moya Fernández
David Villa Alises
Sergio Pérez Camacho



Figura 2.1: Alexander Stepanov, padre de la programación genérica y la librería STL

2.1. Programación genérica

La programación genérica es un paradigma de programación que trata de conseguir un mayor grado de reutilización tanto de las estructuras de datos como de los algoritmos, evitando así la duplicidad de código. Para conseguirlo, los algoritmos deben escribirse evitando asociar los detalles a tipos de datos concretos. Por ejemplo, en un algoritmo de ordenación, la operación que compara dos elementos cualesquiera se delega a una entidad ajena al algoritmo: un operador de comparación.

Hoy en día, prácticamente todos los lenguajes de programación importantes disponen o han adoptado características de programación genérica (tal como los llamados «genéricos» en Java o C#).

El diseño de la librería STL pretende proporcionar herramientas básicas de programación genérica. No es casualidad que la creación de STL y las ideas tras el paradigma de la programación genérica fueran desarrolladas por los mismos autores, especialmente Alexander Stepanov y David Musser [26]. Y de ahí el interés por separar las estructuras de datos (los contenedores) de los algoritmos. Como veremos, los otros dos componentes de la STL (iteradores y functors) sirven también al mismo propósito: posibilitan la interacción entre contenedores y algoritmos, a la vez que permiten un acoplamiento mínimo.

Es interesante indicar que la disociación entre los datos y los algoritmos que los manejan contradice en cierta medida los principios de la programación orientada a objetos. En la POO las operaciones relativas a un tipo de dato concreto se ofrecen como métodos de dicha clase. El polimorfismo por herencia¹ permite en la práctica utilizar un algoritmo definido como un método de la superclase con instancias de sus subclases. Sin embargo, esto no se considera programación genérica pues la implementación del algoritmo normalmente depende al menos de la superclase de la jerarquía.

En STL los algoritmos están implementados normalmente como funciones (no métodos) y por supuesto no tienen estado, algo que por definición es ajeno a la POO. A pesar de ello, en el diseño de la librería están muy presentes los principios de orientación a objetos.

2.1.1. Algoritmos

Para conseguir estructuras de datos genéricas, los contenedores se implementan como plantillas —como ya se discutió en capítulos anteriores— de modo que el tipo de dato concreto que han de almacenar se especifica en el momento de la creación de la instancia.

Aunque es posible implementar algoritmos sencillos del mismo modo —parametrizando el tipo de dato— STL utiliza un mecanismo mucho más potente: los iteradores. Los iteradores permiten desacoplar tanto el tipo de dato como el modo en que se organizan y almacenan los datos en el contenedor.

Lógicamente, para que un algoritmo pueda hacer su trabajo tiene que asumir que tanto los elementos del contenedor como los iteradores tienen ciertas propiedades, o siendo más precisos, un conjunto de métodos con un comportamiento predecible. Por ejemplo, para poder comparar dos colecciones de elementos, deben ser comparables dos a dos para determinar su *igualdad* —sin entrar en qué significa eso realmente. Así pues, el algoritmo `equal()` espera que los elementos soporten en «modelo» *EqualityComparable*, que implica que tienen sobrecargado el método `operator==()`, además de cumplir éste ciertas condiciones como reflexividad, simetría, transitividad, etc.

Escribiendo un algoritmo genérico

El mejor modo de comprender en qué consiste la «genericidad» de un algoritmo es crear uno desde cero. Escribamos nuestra propia versión del algoritmo genérico `count()` (uno de los más sencillos). Este algoritmo sirve para contar el número de ocurrencias de un elemento en una secuencia. Como una primera aproximación veamos cómo hacerlo para un array de enteros. Podría ser algo como:

algoritmos «escalares»

Aunque la mayoría de los algoritmos de la STL manejan secuencias delimitadas por dos iteradores, también hay algunos que utilizan datos escalares, tales como `min()`, `max()`, `power()` o `swap()` que pueden resultar útiles para componer algoritmos más complejos.

¹también llamado polimorfismo «de subclase» o «de inclusión», en contraposición con el «polimorfismo paramétrico»

Listado 2.1: Escribiendo un algoritmo genérico: my_count () (1/4)

```
1 int my_count1(const int* sequence, int size, int value) {
2     int retval = 0;
3     for (int i=0; i < size; ++i)
4         if (sequence[i] == value)
5             retval++;
6
7     return retval;
8 }
9
10 void test_my_count1() {
11     const int size = 5;
12     const int value = 1;
13     int numbers[] = {1, 2, 3, 1, 2};
14
15     assert(my_count1(numbers, size, value) == 2);
16 }
```

Destacar el especificador `const` en el parámetro `sequence` (línea 1). Le indica al compilador que esta función no modificará el contenido del array. De ese modo es más general; se podrá aplicar a cualquier array (sea constante o no).



Recuerda, en las funciones, aquellos parámetros que no impliquen copia (puntero o referencia) deberían ser constantes si la función efectivamente no va a modificarlos.

En la siguiente versión vamos a cambiar la forma de iterar sobre el array. En lugar de emplear un índice vamos a utilizar un puntero que se desplaza a través del array. Esta versión mantiene el prototipo, es decir, se invoca de la misma forma.

Listado 2.2: Escribiendo un algoritmo genérico: my_count () (2/4)

```
1 int my_count2(const int* first, int size, int value) {
2     int retval = 0;
3     for (const int* it=first; it < first + size; ++it)
4         if (*it == value)
5             retval++;
6
7     return retval;
8 }
```

Dos cuestiones a destacar:

- Utiliza aritmética de punteros. Es decir, la dirección del puntero `it` (línea 3) no se incrementa de uno en uno, sino que depende del tamaño del tipo `int`.
- El valor consultado en el array se obtiene de-referenciando el puntero (`*it` en la línea 4).

A continuación la función cambia para imitar la signatura habitual de STL. En lugar de pasar un puntero al comienzo y un tamaño, se le pasan punteros al comienzo y al final-más-uno.

Listado 2.3: Escribiendo un algoritmo genérico: my_count () (3/4)

```

1 int my_count3(const int* first, const int* last, int value) {
2     int retval = 0;
3     for (const int* it=first; it < last; ++it)
4         if (*it == value)
5             retval++;
6
7     return retval;
8 }
9
10 void test_my_count3() {
11     const int size = 5;
12     const int value = 1;
13     int numbers[] = {1, 2, 3, 1, 2};
14
15     assert(my_count3(numbers, numbers+size, value) == 2);
16 }

```

Se puede apreciar como el criterio del final-mas-uno simplifica la invocación, puesto que el valor correcto se consigue con `numbers+size` (línea 15) y la condición de parada es también más simple (`it<last`) en la línea 3.

Por último, veamos como queda la función cambiando los punteros por iteradores. Es fácil comprobar como resultan funcionalmente equivalentes, hasta el punto de que la función se puede utilizar también con un contenedor `vector`. También se ha convertido la función en una plantilla, de modo que se podrá utilizar con cualquier tipo de dato, a condición de que sus elementos soporten la operación de comprobación de igualdad:

Listado 2.4: Escribiendo un algoritmo genérico: my_count () (4/4)

```

1 template <typename Iter, typename T>
2 int my_count4(Iter first, Iter last, T value) {
3     int retval = 0;
4     for (Iter it=first; it < last; ++it)
5         if (*it == value)
6             retval++;
7
8     return retval;
9 }
10
11 void test_my_count4_numbers() {
12     const int size = 5;
13     const int value = 1;
14     int numbers[] = {1, 2, 3, 1, 2};
15     vector<int> numbers_vector(numbers, numbers + size);
16
17     assert(my_count4(numbers, numbers+size, value) == 2);
18     assert(my_count4(numbers_vector.begin(), numbers_vector.end(),
19                     value) == 2);
20 }
21

```

```

22 void test_my_count4_letters() {
23     const int size = 5;
24     const int value = 'a';
25     char letters[] = {'a', 'b', 'c', 'a', 'b'};
26     vector<char> letters_vector(letters, letters + size);
27
28     assert(my_count4(letters, letters+size, value) == 2);
29     assert(my_count4(letters_vector.begin(), letters_vector.end(),
30                     value) == 2);
31 }

```

Esta última versión es bastante similar a la implementación habitual del algoritmo `count()` estándar con la salvedad de que éste último realiza algunas comprobaciones para asegurar que los iteradores son válidos.

Comprobamos que nuestras funciones de prueba funcionan exactamente igual utilizando el algoritmo `count()` estándar²:

Listado 2.5: El algoritmo `count()` estándar se comporta igual

```

1 void test_count_numbers() {
2     const int size = 5;
3     const int value = 1;
4     int numbers[] = {1, 2, 3, 1, 2};
5     vector<int> numbers_vector(numbers, numbers + size);
6
7     assert(count(numbers, numbers+size, value) == 2);
8     assert(count(numbers_vector.begin(), numbers_vector.end(),
9                 value) == 2);
10 }
11
12 void test_count_letters() {
13     const int size = 5;
14     const int value = 'a';
15     char letters[] = {'a', 'b', 'c', 'a', 'b'};
16     vector<char> letters_vector(letters, letters + size);
17
18     assert(count(letters, letters+size, value) == 2);
19     assert(count(letters_vector.begin(), letters_vector.end(),
20                 value) == 2);
21 }

```

Lógica de predicados

En lógica, un **predicado** es una expresión que se puede evaluar como cierta o falsa en función del valor de sus variables de entrada. En programación, y en particular en la STL, un predicado es una función (en el sentido amplio) que acepta un valor del mismo tipo que los elementos de la secuencia sobre la que se usa y devuelve un valor booleano.

2.1.2. Predicados

En el algoritmo `count()`, el criterio para contar es la igualdad con el elemento proporcionado. Eso limita mucho sus posibilidades porque puede haber muchos otros motivos por los que sea necesario contar elementos de una secuencia: esferas de color rojo, enemigos con nivel mayor al del jugador, armas sin munición, etc.

Por ese motivo, muchos algoritmos de la STL tienen una versión alternativa que permite especificar un parámetro adicional llamado *predicado*. El algoritmo invocará el predicado para averiguar si se cumple la condición indicada por el programador y así determinar cómo debe proceder con cada elemento de la secuencia.

²Para utilizar los algoritmos estándar se debe incluir el fichero `<algorithm>`.

En C/C++, para que una función pueda invocar a otra (en este caso, el algoritmo al predicado) se le debe pasar como parámetro un puntero a función. Veamos la definición de un predicado (`not_equal_2`) que, como habrá imaginado, será cierto para valores distintos a 2:

Listado 2.6: Predicado `not_equal_2()`

```
1 bool not_equal_2(int n) {
2     return n != 2;
3 }
4
5 void test_not_equal_2() {
6     const int size = 5;
7     int numbers[] = {1, 2, 3, 1, 2};
8
9     assert(count_if(numbers, numbers+size, not_equal_2) == 3);
10 }
```

Igual que con cualquier otro tipo de dato, cuando se pasa un *puntero a función* como argumento, el parámetro de la función que lo acepta debe estar declarado con ese mismo tipo. Concretamente el tipo del predicado `not_equal_2` sería algo como:

Listado 2.7: Tipo para un predicado que acepta un argumento entero

```
1 bool (*)(int);
```

El algoritmo `count_if()` lo acepta sin problema. Eso se debe a que, como ya hemos dicho, los algoritmos son funciones-plantilla y dado que la secuencia es un array de enteros, asume que el valor que acepta el predicado debe ser también un entero, es decir, el algoritmo determina automáticamente la signatura del predicado.

Aunque funciona, resulta bastante limitado. No hay forma de modificar el comportamiento del predicado en tiempo de ejecución. Es decir, si queremos contar los elementos distintos de 3 en lugar de 2 habría que escribir otro predicado diferente. Eso es porque el único argumento que puede tener el predicado es el elemento de la secuencia que el algoritmo le pasará cuando lo invoque³, y no hay modo de darle información adicional de forma limpia.

2.1.3. Functors

Existe sin embargo una solución elegante para conseguir «predicados configurables». Consiste en declarar una clase que sobrecargue el operador de invocación —método `operator()()`— que permite utilizar las instancias de esa clase como si fueran funciones. Las clases que permiten este comportamiento se denominan «functors»⁴. Veamos como implementar un predicado `not_equal()` como un functor:

³En la terminología de STL se denomina «predicado unario»

⁴«functor» se traduce a veces como «objeto-función».

Listado 2.8: Predicado not_equal() para enteros (como functor)

```

1 class not_equal {
2     const int _ref;
3
4 public:
5     not_equal(int ref) : _ref(ref) {}
6
7     bool operator()(int value) {
8         return value != _ref;
9     }
10 };

```

Y dos pruebas que demuestran su uso:

Listado 2.9: Ejemplos de uso de not_equal()

```

1 void test_not_equal_functor() {
2     not_equal not_equal_2(2);
3
4     assert(not_equal_2(0));
5     assert(not not_equal_2(2));
6 }
7
8 void test_not_equal_count_if() {
9     const int size = 5;
10    int numbers[] = {1, 2, 3, 1, 2};
11
12    assert(count_if(numbers, numbers+size, not_equal(2)) == 3);
13 }

```

Para disponer de un predicado lo más flexible posible deberíamos implementarlo como una clase plantilla de modo que sirva no solo para enteros:

Listado 2.10: Predicado not_equal() genérico como functor

```

1 template <typename _Arg>
2 class not_equal {
3     const _Arg _ref;
4
5 public:
6     not_equal(_Arg ref) : _ref(ref) {}
7
8     bool operator()(_Arg value) const {
9         return value != _ref;
10    }
11 };

```

Pero los predicados no son la única utilidad interesante de los functors. Los predicados son una particularización de las «funciones» (u operadores). Los operadores pueden devolver cualquier tipo, no sólo booleanos. La STL clasifica los operadores en 3 categorías básicas:

Generador Una función que no acepta argumentos.

Unario Una función que acepta un argumento.

Binario Una función que acepta dos argumentos.

Aunque obviamente pueden definirse un operador con 3 o más argumentos, no hay ningún algoritmo estándar que los utilice. Si el functor devuelve un booleano es cuando se denomina «predicado unario» o «binario» respectivamente. Para ser un predicado debe tener al menos un argumento como hemos visto. Además se habla también de modalidades «adaptables» para las tres categorías, que se distinguen porque exponen sus tipos de argumentos y retorno como atributos de la clase. Los veremos más adelante.

Los operadores (los functors que no son predicados) se utilizan normalmente en algoritmos que realizan algún cálculo con los elementos de una secuencia. Como ejemplo, el siguiente listado multiplica los elementos del array `numbers`:

Listado 2.11: `accumulate()` multiplicando los elementos de una secuencia de enteros

```
1 void test_accumulate_multiplies() {
2     int numbers[] = {1, 2, 3, 4};
3     const int size = sizeof(numbers) / sizeof(int);
4
5     int result = accumulate(numbers, numbers+size,
6                             1, multiplies<int>());
7     assert(result == 24);
8 }
```

El algoritmo `accumulate()` aplica el functor binario especificado como último parámetro (`multiplies()` en el ejemplo) empezando por el valor inicial indicado como tercer parámetro (1) y siguiendo con los elementos del rango especificado. Corresponde con la operación $\prod_{i=1}^n i$.

Además de `multiplies()`, la librería estándar incluye muchos otros functors que se clasifican en operaciones aritméticas (grupo al que corresponde `multiplies()`), lógicas, de identidad y comparaciones. Los iremos viendo y utilizando a lo largo de esta sección.

2.1.4. Adaptadores

Es habitual que la operación que nos gustaría que ejecute el algoritmo sea un método (una función miembro) en lugar de una función estándar. Si tratamos de pasar al algoritmo un puntero a método no funcionará, porque el algoritmo no le pasará el parámetro implícito `this` que todo método necesita.

Una posible solución sería escribir un functor que invoque el método deseado, como se muestra en el siguiente listado.

Listado 2.12: Adaptador «manual» para un método

```

1 class Enemy {
2 public:
3     bool is_alive(void) const {
4         return true;
5     }
6 };
7
8 class enemy_alive {
9 public:
10    bool operator()(Enemy enemy) {
11        return enemy.is_alive();
12    }
13 };
14
15 void test_my_adapter() {
16     vector<Enemy> enemies(2);
17
18     assert(count_if(enemies.begin(), enemies.end(),
19                    enemy_alive()) == 2);
20 }

```

```
mem_fun()
```

Existe un adaptador alternativo llamado `mem_fun()` que debe utilizarse si los elementos del contenedor son punteros. Si son objetos o referencias se utiliza `mem_fun_ref()`.

Listado 2.13: Uso del adaptador `mem_fun_ref()`

```

1 void test_mem_fun_ref() {
2     vector<Enemy> enemies(2);
3
4     assert(count_if(enemies.begin(), enemies.end(),
5                    mem_fun_ref(&Enemy::is_alive)) == 2);
6 }

```

Veamos de nuevo el problema de tener una operación o predicado que requiere un argumento adicional aparte del elemento de la secuencia. En la sección 2.1.3 resolvimos el problema creando un functor (`not_equal`) que sirviera como adaptador. Bien, pues eso también lo prevé la librería y nos proporciona dos adaptadores llamados `bind1st()` y `bind2nd()` para realizar justo esa tarea, y de manera genérica. Veamos cómo —gracias a `bind2nd()`— es posible reescribir el listado 2.6 de modo que es posible especificar el valor con el que comparar (parámetro `ref`) sin tener que escribir un functor ad-hoc:

Listado 2.14: Uso del adaptador `bind2nd()`

```

1 bool not_equal(int n, int ref) {
2     return n != ref;
3 }
4
5 void test_not_equal_bind() {
6     const int size = 5;
7     int numbers[] = {1, 2, 3, 1, 2};
8
9     assert(count_if(numbers, numbers+size,
10                    bind2nd(ptr_fun(not_equal), 2)) == 3);
11 }

```

Nótese que `bind2nd()` espera un functor como primer parámetro. Como lo que tenemos es una función normal es necesario utilizar otro

adaptador llamado `ptr_fun()`, que como su nombre indica adapta un puntero a función a functor.

`bind2nd()` pasa su parámetro adicional (el 2 en este caso) como segundo parámetro en la llamada a la función `not_equal()`, es decir, la primera llamada para la secuencia del ejemplo sería `not_equal(1, 2)`. El primer argumento (el 1) es el primer elemento obtenido de la secuencia. El adaptador `bind1st()` los pasa en orden inverso, es decir, pasa el valor extra como primer parámetro y el elemento de la secuencia como segundo.

Hay otros adaptadores de menos uso:

not1() devuelve un predicado que es la negación lógica del predicado *unario* al que se aplique.

not2() devuelve un predicado que es la negación lógica del predicado *binario* al que se aplique.

compose1() devuelve un operador resultado de componer las dos funciones unarias que se le pasan como parámetros. Es decir, dadas las funciones $f(x)$ y $g(x)$ devuelve una función $f(g(x))$.

compose2() devuelve un operador resultado de componer una función binaria y dos funciones unarias que se le pasan como parámetro del siguiente modo. Dadas las funciones $f(x, y)$, $g_1(x)$ y $g_2(x)$ devuelve una función $h(x) = f(g_1(x), g_2(x))$.

2.1.5. Algoritmos idempotentes

Los algoritmos idempotentes (*non-mutating*) son aquellos que no modifican el contenedor sobre el que se aplican. Podríamos decir que son algoritmos *funcionales* desde el punto de vista de ese paradigma de programación. Nótese que aunque el algoritmo en sí no afecte al contenedor, las operaciones que se realicen con él sí pueden modificar los objetos contenidos.

`for_each()`

El algoritmo `for_each()` es el más simple y general de la STL. Es equivalente a un bucle `for` convencional en el que se ejecutara un método concreto (o una función independiente) sobre cada elemento de un rango. Veamos un ejemplo sencillo en el que se recargan todas las armas de un jugador:

Nomenclatura

Los nombres de los algoritmos siguen ciertos criterios. Como ya hemos visto, aquellos que tienen una versión acabada en el sufijo `_if` aceptan un predicado en lugar de utilizar un criterio implícito. Los que tienen el sufijo `_copy` generan su resultado en otra secuencia, en lugar de modificar la secuencia original. Y por último, los que acaban en `_n` aceptan un iterador y un entero en lugar de dos iteradores; de ese modo se pueden utilizar para insertar en «cosas» distintas de contenedores, por ejemplo flujos.

Listado 2.15: Ejemplo de uso del algoritmo `for_each()`

```
1 class Weapon {
2 public:
3     void reload() { /* some code */ }
4 };
5
6 void test_for_each() {
7     vector<Weapon> weapons(5);
8     for_each(weapons.begin(), weapons.end(),
9             mem_fun_ref(&Weapon::reload));
10 }
```

`find()` / `find_if()`

Devuelve un iterador al primer elemento del rango que coincide con el indicado (si se usa `find()`) o que cumple el predicado (si se usa `find_if()`). Devuelve el iterador `end` si no encuentra ninguna coincidencia. Un ejemplo en el que se busca el primer entero mayor que 6 que haya en el rango:

Listado 2.16: Ejemplo de `find_if()`

```
1 void test_for_each() {
2     const int size = 5;
3     const int value = 1;
4     int numbers[] = {2, 7, 12, 9, 4};
5
6     assert(find_if(numbers, numbers + size,
7                 bind2nd(greater<int>(), 6)) == numbers+1);
8 }
```

Se utiliza `greater`, un predicado binario que se cumple cuando su primer parámetro (el elemento del rango) es mayor que el segundo (el 6 que se pasa `bind2nd()`). El resultado del algoritmo es un iterador al segundo elemento (el 7) que corresponde con `numbers+1`. Hay algunos otros functors predefinidos para comparación: `equal_to()`, `not_equal_to()`⁵, `less()`, `less_equal()` y `greater_equal()`.

`count()` / `count_if()`

Como ya hemos visto en ejemplos anteriores `count()` devuelve la cantidad de elementos del rango igual al dado, o que cumple el predicado, si se usa la modalidad `count_if()`.

`mismatch()`

Dados dos rangos, devuelve un par de iteradores a los elementos de cada rango en el que las secuencias difieren. Veamos el siguiente ejemplo —extraído de la documentación de SGI⁶:

⁵Equivalente al que implementamos en el listado 2.10.

⁶<http://www.sgi.com/tech/stl/mismatch.html>

Listado 2.17: Ejemplo de uso de mismatch()

```

1 void test_mismatch() {
2   int A1[] = { 3, 1, 4, 1, 5, 9, 3};
3   int A2[] = { 3, 1, 4, 2, 8, 5, 7};
4   const int size = sizeof(A1) / sizeof(int);
5
6   pair<int*, int*> result = mismatch(A1, A1 + size, A2);
7   assert(result.first == A1 + 3);
8   assert((*result.first) == 1 and (*result.second) == 2);
9 }

```



Muchos algoritmos de transformación que manejan dos secuencias requieren solo *tres* iteradores. El tercer iterador indica el comienzo de la secuencia de salida y se asume que ambas secuencias son del mismo tamaño.

equal()

Indica si los rangos indicados son iguales. Por defecto utiliza el `operator==()`, pero opcionalmente es posible indicar un predicado binario como cuarto parámetro para determinar en qué consiste la «igualdad». Veamos un ejemplo en el que se comparan dos listas de figuras que se considerarán iguales simplemente porque coincida su color:

Listado 2.18: Ejemplo de uso de equal()

```

1 enum Color{BLACK, WHITE, RED, GREEN, BLUE};
2
3 class Shape {
4 public:
5   Color color;
6
7   Shape(void) : color(BLACK) {}
8   bool cmp(Shape other) {
9     return color == other.color;
10  }
11 };
12
13 void test_equal() {
14   const int size = 5;
15   Shape shapes1[size], shapes2[size];
16   shapes2[3].color = RED;
17
18   assert(equal(shapes1, shapes1+size, shapes2,
19               mem_fun_ref(&Shape::cmp)) == false);
20 }

```

`search()`

Localiza la posición del segundo rango en el primero. Devuelve un iterador al primer elemento. Opcionalmente acepta un predicado binario para especificar la igualdad entre dos elementos. Veamos este ejemplo extraído de la documentación de SGI.

Listado 2.19: Ejemplo de uso de `search()`

```
1 void test_search() {
2     const char s1[] = "Hello, world!";
3     const char s2[] = "world";
4     const int n1 = strlen(s1);
5     const int n2 = strlen(s2);
6
7     const char* p = search(s1, s1 + n1, s2, s2 + n2);
8     assert(p == s1 + 7);
9 }
```

El algoritmo `find_end()` (a pesar de su nombre) es similar a `search()` solo que localiza la última aparición en lugar de la primera.

El algoritmo `search_n()` también es similar. Busca una secuencia de n elementos iguales (no otro rango) que debería estar contenida en el rango indicado.

2.1.6. Algoritmos de transformación

Normalmente, en los algoritmos de transformación (*mutating algorithms*) se distingue entre el rango o secuencia de entrada y la de salida, ya que su operación implica algún tipo de modificación (inserción, eliminación, cambio, etc.) sobre los elementos de la secuencia de salida.



Es importante recordar que las secuencias de salida que se utilizan en los algoritmos de transformación deben disponer de memoria suficiente para los datos que recibirán u obtendremos comportamientos erráticos aleatorios y errores de acceso a memoria en tiempo de ejecución (SEGFault).

`copy()`

Copia los elementos de un rango en otro. No debería utilizarse para copiar una secuencia completa en otra ya que el operador de asignación que tienen todos los contenedores resulta más eficiente. Sí resulta interesante para copiar fragmentos de secuencias. Veamos un uso interesante de `copy()` para enviar a un flujo (en este caso `cout`) el contenido de una secuencia.

Listado 2.20: Ejemplo de uso de `copy()`

```
1 int main() {
2     int values[] = {1, 2, 3, 4, 5};
3     const int size = sizeof(values) / sizeof(int);
4
5     copy(values+2, values+size,
6         ostream_iterator<int>(cout, " "));
7     cout << endl;
8 }
```

La plantilla `ostream_iterator` devuelve un iterador de inserción para un tipo concreto (`int` en el ejemplo) que escribirá en el flujo (`cout`) los elementos que se le asignen, escribiendo además una cadena opcional después de cada uno (nueva línea).

Existe una variante llamada `copy_backward()` que copia desde el final y en la que se debe pasar un iterador de la secuencia de salida al que copiar el último elemento.

`swap_ranges()`

Intercambia el contenido de dos secuencias. Como es habitual, se pasan los iteradores a principio y fin de la primera secuencia y al principio de la segunda, dado que asume que los rangos deben ser del mismo tamaño. Nótese que este algoritmo modifica ambos rangos.

`transform()`

El algoritmo `transform()` es uno de los más versátiles de la librería. La versión básica (que opera sobre un único rango de entrada) aplica un operador unario a cada elemento del rango y escribe el resultado a un iterador de salida.

Existe una versión alternativa (sobrecargada) que acepta dos secuencias de entrada. En este caso, el algoritmo utiliza un operador binario al que pasa un elemento que obtiene de cada una de las secuencias de entrada, el resultado se escribe sobre el iterador de salida.

Es interesante destacar que en ambos casos, el iterador de salida puede referirse a una de las secuencias de entrada.

Veamos un ejemplo en el que se concatenan las cadenas de dos vectores y se almacenan en un tercero haciendo uso del operador `plus()`:

Listado 2.21: Ejemplo de uso de `transform()`

```
1 void test_transform() {
2     vector<string> v1, v2, result(2);
3     v1.push_back("hello ");
4     v1.push_back("bye ");
5     v2.push_back("world");
6     v2.push_back("hell");
7
8     transform(v1.begin(), v1.end(), v2.begin(),
9             result.begin(),
```

```

10         plus<string>());
11
12     assert(result[0] == "hello world");
13     assert(result[1] == "bye hell");
14 }

```

replace() / replace_if()

Dado un rango, un valor antiguo y un valor nuevo, Substituye todas las ocurrencias del valor antiguo por el nuevo en el rango. La versión `replace_if()` substituye los valores que cumplan el predicado unario especificado por el valor nuevo. Ambos utilizan un única secuencia, es decir, hacen la substitución in situ.

Existen variantes llamadas `replace_copy()` y `replace_copy_if()` respectivamente en la que se copian los elementos del rango de entrada al de salida a la vez que se hace la substitución. En este caso la secuencia original no cambia.

fill()

Dado un rango y un valor, copia dicho valor en todo el rango:

Listado 2.22: Ejemplo de uso de fill()

```

1 void test_fill() {
2     vector<float> v(10);
3     assert(count(v.begin(), v.end(), 0));
4
5     fill(v.begin(), v.end(), 2);
6     assert(count(v.begin(), v.end(), 2) == 10);
7 }

```

La variante `fill_n()` utiliza un único iterador de salida y copia sobre él n copias del valor especificado. Útil con iteradores de inserción.

generate()

En realidad es una variante de `fill()` salvo que los valores los obtiene de un operador que se le da como parámetro, en concreto un «generador», es decir, una función/functor sin parámetros que devuelve un valor:

Listado 2.23: Ejemplo de uso de generate()

```

1 class next {
2     int _last;
3 public:
4     next(int init) : _last(init) {}
5     int operator()() {
6         return _last++;
7     }

```

```

8 };
9
10 void test_generate() {
11     vector<int> v(10);
12     generate(v.begin(), v.end(), next(10));
13     assert(v[9] == 19);
14 }

```

Existe un algoritmo `generate_n()` al estilo de `copy_n()` o `fill_n()` que en lugar de dos iteradores, espera un iterador y una cantidad de elementos a generar.

`remove()`

Dado un rango y un valor, elimina todas las ocurrencias de dicho valor y retorna un iterador al nuevo último elemento. En realidad `remove()` no elimina nada, solo reordena la secuencia, dejando los elementos «eliminados» detrás del iterador que retorna.

Como en el caso de `replace()` existen alternativas análogas llamadas `replace_if()`, `replace_copy()` y `replace_copy_if()`.

`unique()`

Elimina elementos duplicados consecutivos. Dado que puede eliminar elementos in situ, retorna un iterador al nuevo último elemento de la secuencia. Existe una modalidad `unique_copy()` que copia el resultado sobre un iterador de salida dejando a la secuencia original intacta. En ambos casos existen también modalidades que aceptan un predicado binario para definir la «igualdad» entre elementos.

`reverse()`

Invierte un rango in situ. También existe una modalidad que deja la secuencia original intacta llamada `reverse_copy()`. Se ilustra con un sencillo ejemplo que invierte **parte** de una cadena y no el contenedor completo:

Listado 2.24: Ejemplo de uso de `reverse()`

```

1 void test_reverse() {
2     char word[] = "reversion";
3     const int size = strlen(word);
4
5     reverse(word + 5, word + size);
6
7     assert(strcmp(word, "revernois") == 0);
8 }

```

rotate()

Rota los elementos del rango especificado por 3 iteradores, que indican el inicio, el punto medio y el final del rango. Existe una modalidad `rotate_copy()`, que como siempre aplica el resultado a un iterador en lugar de modificar el original.

Algoritmos aleatorios

Hay tres algoritmos que tienen que ver con operaciones aleatorias sobre una secuencia:

random_shuffle() reordena de forma aleatoria los elementos del rango.

random_sample() elige aleatoriamente elementos de la secuencia de entrada y los copia en la secuencia de salida. Es interesante destacar que este algoritmo requiere 4 iteradores ya que se puede crear una secuencia de salida de tamaño arbitrario, siempre que sea menor o igual que la secuencia de entrada.

random_shuffle_n() realiza la misma operación que `random_sample()` salvo que la cantidad de elementos a generar se especifica explícitamente en lugar de usar un cuarto iterador. Eso permite utilizarlo con un iterador de inserción.

Los tres aceptan opcionalmente una función que genere números aleatorios.

partition()

Dada una secuencia y un predicado, el algoritmo reordena los elementos de modo que los que satisfacen el predicado aparecen primero y los que lo incumplen después. Devuelve un iterador al primer elemento que incumple el predicado.

La modalidad `stable_partition()` preserva el orden de los elementos en cada parte respecto al orden que tenían en la secuencia original.

2.1.7. Algoritmos de ordenación

Los algoritmos de ordenación también son de transformación, pero se clasifican como un grupo distinto dado que todos tienen que ver con la ordenación de secuencias u operaciones con secuencias ordenadas.

sort()

Ordena in situ el rango especificado por dos iteradores. La modalidad `stable_sort()` preserva el orden relativo original a costa de algo

menos de rendimiento. Veamos un ejemplo sencillo tomado del manual de SGI para ordenar un array de caracteres:

Listado 2.25: Ejemplo de uso de `sort()`

```
1 bool less_nocase(char c1, char c2) {
2     return tolower(c1) < tolower(c2);
3 }
4
5 void test_sort() {
6     char letters[] = "ZfdBeACFDdBecz";
7     const int size = strlen(letters);
8
9     sort(letters, letters+size, less_nocase);
10
11     char expected[] = "AaBbCcdDeEfFz";
12     assert(equal(letters, letters+size, expected));
13 }
```

La mayoría de los algoritmos de ordenación aceptan un predicado especial para comparación de elementos dos a dos. Es muy habitual ordenar secuencias de elementos no numéricos o por características que tienen poco que ver con la relación mayor o menor en el sentido tradicional del término.

El algoritmo `partial_sort()` ordena parcialmente una secuencia especificada por tres iteradores de modo que solo el rango correspondiente a los dos primeros estará ordenado en la secuencia resultante. Tiene una modalidad `partial_sort_copy()` que no modifica la secuencia original.

`nth_element()`

Dada una secuencia y tres iteradores, ordena la secuencia de modo que todos los elementos en el subrango por debajo del segundo iterador (`nth`) son menores que los elementos que quedan por encima. Además, el elemento apuntado por el segundo iterador es el mismo que si se hubiera realizado una ordenación completa.

Operaciones de búsqueda

A continuación se incluye una pequeña descripción de los algoritmos relacionados con búsquedas binarias:

`binary_search()` determina si el valor indicado se encuentra en la secuencia.

`lower_bound()` devuelve un iterador a la primera posición en la que es posible insertar el elemento indicado manteniendo el orden en la secuencia.

`upper_bound()` devuelve un iterador a la última posición en la que es posible insertar el elemento indicado manteniendo el orden en la secuencia.

`equal_range()` combina los dos algoritmos anteriores. Devuelve un par con los iteradores a la primera y última posición en la que es posible insertar el elemento indicado manteniendo el orden de la secuencia.

Se muestra un ejemplo de los cuatro algoritmos:

Listado 2.26: Ejemplo de uso de los algoritmos de búsqueda

```

1 int numbers[] = {0, 3, 7, 7, 10, 11, 15};
2 int size = sizeof(numbers) / sizeof(int);
3
4 void test_binary_search() {
5     assert(binary_search(numbers, numbers+size, 6) == false);
6     assert(binary_search(numbers, numbers+size, 10));
7 }
8
9 void test_bounds() {
10    assert(lower_bound(numbers, numbers+size, 6) == numbers+2);
11    assert(upper_bound(numbers, numbers+size, 8) == numbers+4);
12 }
13
14 void test_equal_range() {
15     pair<int*, int*> bounds = equal_range(numbers, numbers+size, 7);
16     assert(bounds.first == numbers+2 and bounds.second == numbers+4);
17 }

```

`merge()` combina dos secuencias, dadas por dos pares de iteradores, y crea una tercera secuencia que incluye los elementos de ambas, manteniendo el orden.

Mínimo y máximo

Los algoritmos `min_element()` y `max_element()` permiten obtener respectivamente el elemento mínimo y máximo del rango especificado. Veamos un ejemplo:

Listado 2.27: Ejemplo de uso de `max_element()` y `min_element()`

```

1 char letters[] = "ZfdBeACFDbEacz";
2 const int size = strlen(letters);
3
4 void test_min() {
5     char* result = min_element(letters, letters+size);
6     assert(*result == 'A');
7 }
8
9 void test_max() {
10    char* result = max_element(letters, letters+size);
11    assert(*result == 'z');
12 }

```

2.1.8. Algoritmos numéricos

`accumulate()`

Aplica un operador (la suma si no se especifica otro) sobre el rango especificado por dos iteradores. Debe indicarse también un valor inicial ya que el algoritmo opera sobre un valor acumulado (de ahí su nombre) y un elemento extraído de la secuencia. El listado 2.11 muestra un ejemplo de uso.

`partial_sum()`

Calcula la «suma parcial» para cada elemento de una secuencia y lo almacena sobre un iterador de salida:

Listado 2.28: Ejemplo de uso de `partial_sum()`

```
1 void test_partial_sum() {
2     const int size = 5;
3     vector<int> values(size);
4     fill(values.begin(), values.end(), 1);
5
6     partial_sum(values.begin(), values.end(), values.begin());
7
8     int expected[size] = {1, 2, 3, 4, 5};
9     assert(equal(values.begin(), values.end(), expected));
10 }
```

`adjacent_difference()`

Calcula las diferencias entre elementos consecutivos de la secuencia de entrada y los escribe sobre el iterador de salida:

Listado 2.29: Ejemplo de uso de `adjacent_difference()`

```
1 void test_adjacent_difference() {
2     int values[] = {1, 3, 0, 10, 15};
3     const int size = sizeof(values) / sizeof(int);
4     int result[size];
5
6     adjacent_difference(values, values+size, result);
7
8     int expected[size] = {1, 2, -3, 10, 5};
9     assert(equal(result, result+size, expected));
10 }
```

2.1.9. Ejemplo: inventario de armas

Veamos un programa concreto que ilustra como sacar partido de las algoritmos genéricos. Se trata del típico inventario de armas habitual en cualquier videojuego tipo «shooter».

Lo primero es definir una clase para describir el comportamiento y atributos de cada arma (clase `Weapon*`). El único atributo es la munición disponible. Tiene otras dos propiedades (accesibles a través de métodos virtuales) que indican la potencia del disparo y la cantidad máxima de munición que permite:

Listado 2.30: Inventario de armas: Clase `Weapon`

```

1 class Weapon {
2     int ammo;
3
4     protected:
5         virtual int power(void) const = 0;
6         virtual int max_ammo(void) const = 0;
7
8     public:
9         Weapon(int ammo=0) : ammo(ammo) { }
10
11        void shoot(void) {
12            if (ammo > 0) ammo--;
13        }
14
15        bool is_empty(void) {
16            return ammo == 0;
17        }
18
19        int get_ammo(void) {
20            return ammo;
21        }
22
23        void add_ammo(int amount) {
24            ammo = min(ammo + amount, max_ammo());
25        }
26
27        void add_ammo(Weapon* other) {
28            add_ammo(other->ammo);
29        }
30
31        int less_powerful_than(Weapon* other) const {
32            return power() < other->power();
33        }
34
35        bool same_weapon_as(Weapon* other) {
36            return typeid(*this) == typeid(*other);
37        }
38 };

```

Los métodos `shoot()`, `is_empty()` y `get_ammo()` son auto-explicativos. El método `add_ammo()` está sobrecargado. La primera versión (línea 23) añade al arma la cantidad especificada de balas respetando el límite. Para esto se utiliza el algoritmo `min()`.

El método `less_powerful_than()` compara esta instancia de arma con otra para decidir cuál es la más potente, y por último, el método `same_weapon_as()` indica si el arma es del mismo tipo utilizando RTTI.

El siguiente listado muestra tres especializaciones de la clase `Weapon` que únicamente especializan los métodos privados `power()` y `max_ammo()` para cada uno de los tipos `Pistol`, `Shotgun` y `RPG`.

Listado 2.31: Especializaciones de Weapon

```

1 class Pistol : public Weapon {
2     virtual int power(void) const    { return 1; };
3     virtual int max_ammo(void) const { return 50; };
4
5 public:
6     Pistol(int ammo=0) : Weapon(ammo) {}
7 };
8
9 class Shotgun : public Weapon {
10    virtual int power(void) const    { return 10; };
11    virtual int max_ammo(void) const { return 100; };
12
13 public:
14    Shotgun(int ammo=0) : Weapon(ammo) {}
15 };
16
17 class RPG : public Weapon {
18    virtual int power(void) const    { return 100; };
19    virtual int max_ammo(void) const { return 5; };
20
21 public:
22    RPG(int ammo=0) : Weapon(ammo) {}
23 };

```

Veamos por último la clase `Inventory` que representaría la colección de armas que lleva el jugador.

Listado 2.32: Inventario de armas: Clase `Inventory`

```

1 class Inventory : public vector<Weapon*> {
2 public:
3     typedef typename Inventory::const_iterator WeaponIter;
4     typedef vector<Weapon*> WeaponVector;
5     class WeaponNotFound {};
6     ~Inventory();
7
8     void add(Weapon* weapon) {
9         WeaponIter it =
10            find_if(begin(), end(),
11                  bind2nd(mem_fun(&Weapon::same_weapon_as), weapon));
12
13         if (it != end()) {
14             (*it)->add_ammo(weapon);
15             delete weapon;
16             return;
17         }
18
19         push_back(weapon);
20     }
21
22     WeaponVector weapons_with_ammo(void) {
23         WeaponVector retval;
24
25         remove_copy_if(begin(), end(), back_inserter(retval),
26                       mem_fun(&Weapon::is_empty));
27
28         if (retval.begin() == retval.end())
29             throw Inventory::WeaponNotFound();
30
31         return retval;
32     }

```

```
33
34  Weapon* more_powerful_weapon(void) {
35      WeaponVector weapons = weapons_with_ammo();
36
37      sort(weapons.begin(), weapons.end(),
38          mem_fun(&Weapon::less_powerful_than));
39
40      return *(weapons.end()-1);
41  }
```

Algunos detalles interesantes de esta clase:

- Inventory «es-un» contenedor de punteros a `Weapon`, concretamente un vector (`vector<Weapon*>`) como se puede apreciar en la línea 1.
- La línea 3 define el tipo `WeaponIter` como alias del tipo del iterador para recorrer el contenedor.
- En la línea 4, la clase `WeaponNotFound` se utiliza como excepción en las búsquedas de armas, como veremos a continuación.

El método `add()` se utiliza para añadir un nuevo arma al inventario, pero contempla específicamente el caso —habitual en los *shooters*— en el que coger un arma que ya tiene el jugador implica únicamente coger su munición y desechar el arma. Para ello, utiliza el algoritmo `find_if()` para recorrer el propio contenedor especificando como predicado el método `Weapon::same_weapon_as()`. Nótese el uso de los adaptadores `mem_fun()` (por tratarse de un método) y de `bind2nd()` para pasar a dicho método la instancia del arma a buscar. Si se encuentra un arma del mismo tipo (líneas 12–16) se añade su munición al arma existente usando el iterador devuelto por `find_if()` y se elimina (línea 14). En otro caso se añade la nueva arma al inventario (línea 18).

Por otra parte, el método `more_powerful_weapon()` (líneas 34–41) implementa una funcionalidad también muy habitual en ese tipo de juegos: cambiar al arma más potente disponible. En este contexto, invoca `weapons_with_ammo()` (líneas 22–32) para obtener las armas con munición. Utiliza el algoritmo `remove_copy_if()` para crear un vector de punteros (mediante la función `back_inserter()`), evitando copiar las vacías (línea 26).

Ordena el vector resultante usando `sort()` y utilizando como predicado el método `less_powerful_than()` que vimos antes. Por último, el método retorna un puntero al último arma (línea 40). Nótese que el '*' en esa línea es la de-referencia del iterador (que apunta a un puntero).

Para acabar, se muestra el destructor de la clase, que se encarga de liberar los punteros que almacena:

Listado 2.33: Inventario de armas: Destructor

```

1  template<class T>
2  T* deleter(T* x) {
3      delete x;
4      return 0;
5  }
6      sort(weapons.begin(), weapons.end(),
7           mem_fun(&Weapon::less_powerful_than));
8
9      return *(weapons.end()-1);
10 }
```

Aquí se utiliza el functor (`deleter`) con el algoritmo `transform()` para liberar cada puntero. La razón de usar `transform()` en lugar de `for_each()` es eliminar las direcciones de los punteros que dejan de ser válidos en el contenedor. Después se borra todo el contenedor usando su método `clear()`.

2.2. Aspectos avanzados de la STL

En esta sección veremos cómo explotar el potencial de la librería STL más allá del mero uso de sus contenedores y algoritmos.

2.2.1. Eficiencia

La eficiencia es sin duda alguna uno de los objetivos principales de la librería STL. Esto es así hasta el punto de que se obvian muchas comprobaciones que harían su uso más seguro y productivo. El principio de diseño aplicado aquí es:

Es factible construir *decoradores* que añadan comprobaciones adicionales a la versión eficiente. Sin embargo no es posible construir una versión eficiente a partir de una segura que realiza dichas comprobaciones.

Algunas de estas comprobaciones incluyen la dereferencia de iteradores nulos, invalidados o fuera de los límites del contenedor, como se muestra en el siguiente listado.

Para subsanar esta situación el programador puede optar entre utilizar una implementación que incorpore medidas de seguridad —con la consiguiente reducción de eficiencia— o bien especializar los contenedores en clases propias y controlar específicamente las operaciones susceptibles de ocasionar problemas.

En cualquier caso el programador debería tener muy presente que este tipo de decisiones ad hoc (eliminar comprobaciones) forman parte de la fase de optimización y sólo deberían considerarse cuando se detecten problemas de rendimiento. En general, tal como dice Ken Beck, «La optimización prematura es un lastre». Es costosa (en tiempo y recursos) y produce normalmente código más sucio, difícil de leer y mantener, y por tanto, de inferior calidad.

80/20

Estadísticamente el 80% del tiempo de ejecución de un programa es debido únicamente al 20% de su código. Eso significa que mejorando ese 20% se pueden conseguir importantes mejoras. Por ese motivo, la optimización del programa (si se necesita) debería ocurrir únicamente cuando se haya identificado dicho código por medio de herramientas de perfilado y un adecuado análisis de los flujos de ejecución. Preocuparse por optimizar una función lenta que solo se invoca en el arranque de un servidor que se ejecuta durante días es perjudicial. Supone un gasto de recursos y tiempo que probablemente producirá código menos legible y mantenible.

Listado 2.34: Situaciones no controladas en el uso de iteradores

```
1 void test_lost_iterator() {
2     vector<int>::iterator it;
3     int i = *it; // probably a SEGFAULT
4 }
5
6 void test_invalidated_iterator() {
7     vector<int> v1;
8     v1.push_back(1);
9     vector<int>::iterator it = v1.begin();
10    v1.clear();
11
12    int i = *it; // probably a SEGFAULT
13 }
14
15 void test_outbound_iterator() {
16     vector<int> v1;
17     vector<int>::iterator it = v1.end();
18
19     int i = *it; // probably a SEGFAULT
20 }
```

Sin embargo, existen otro tipo de decisiones que el programador puede tomar cuando utiliza la STL, que tienen un gran impacto en la eficiencia del resultado y que no afectan en absoluto a la legibilidad y mantenimiento del código. Estas decisiones tienen que ver con la elección del contenedor o algoritmo adecuado para cada problema concreto. Esto requiere conocer con cierto detalle el funcionamiento y diseño de los mismos.

Elegir el contenedor adecuado

A continuación se listan los aspectos más relevantes que se deberían tener en cuenta al elegir un contenedor, considerando las operaciones que se realizarán sobre él:

- Tamaño medio del contenedor.

En general, la eficiencia –en cuanto a tiempo de acceso– solo es significativa para grandes cantidades de elementos. Para menos de 100 elementos (seguramente muchos más considerando las computadoras o consolas actuales) es muy probable que la diferencia entre un contenedor con tiempo de acceso lineal y uno logarítmico sea imperceptible. Si lo previsible es que el número de elementos sea relativamente pequeño o no se conoce bien a priori la opción más adecuada es `vector`.

- Inserción de elementos en los dos extremos de la secuencia.

Si necesita añadir al comienzo con cierta frecuencia (>10%) debería elegir un contenedor que implemente esta operación de forma eficiente como `deque`.

- Inserción y borrado en posiciones intermedias.

El contenedor más adecuado en este caso es `list`. Al estar implementado como una lista doblemente enlazada, la operación de

inserción o borrado implica poco más que actualizar dos punteros.

- **Contenedores ordenados.**

Algunos contenedores, como `set` y `multiset`, aceptan un operador de ordenación en el momento de su instanciación. Después de cualquier operación de inserción o borrado el contenedor quedará ordenado. Esto es órdenes de magnitud más rápido que utilizar un algoritmo de ordenación cuando se necesite ordenarlo.

Otro aspecto a tener en cuenta es la distinción entre contenedores basados en bloques (como `vector`, `deque` o `string`) y los basados en nodos (como `list`, `set`, `map`, etc.). Los contenedores basados en nodos almacenan cada elemento como unidades independientes y se relacionan con los demás a través de punteros. Esto tiene varias implicaciones interesantes:

- Si se obtiene un iterador a un nodo, sigue siendo válido durante toda la vida del elemento. Sin embargo, en los basados en bloque los iteradores pueden quedar invalidados si se realoja el contenedor.
- Ocupan más memoria por cada elemento almacenado, debido a que se requiere información adicional para mantener la estructura: árbol o lista enlazada.

Elegir el algoritmo adecuado

Aunque los algoritmos de STL están diseñados para ser eficientes (incluso el estándar determina la complejidad ciclométrica máxima permitida) ciertas operaciones sobre grandes colecciones de elementos pueden implicar tiempos de cómputo muy considerables. Para reducir el número de operaciones a ejecutar es importante considerar los condicionantes específicos de cada problema.

Uno de los detalles más simples a tener en cuenta es la forma en la que se especifica la entrada al algoritmo. En la mayoría de ellos la secuencia queda determinada por el iterador de inicio y el de fin. Lo interesante de esta interfaz es que darle al algoritmo parte del contenedor es tan sencillo como dárselo completo. Se pueden dar innumerables situaciones en las que es perfectamente válido aplicar cualquiera de los algoritmos genéricos que hemos visto a una pequeña parte del contenedor. Copiar, buscar, reemplazar o borrar en los n primeros o últimos elementos puede servir para lograr el objetivo ahorrando muchas operaciones innecesarias.

Otra forma de ahorrar cómputo es utilizar algoritmos que hacen solo parte del trabajo (pero suficiente en muchos casos), en particular los de ordenación y búsqueda como `partial_sort()`, `nth_element()`, `lower_bound()`, etc.

Por ejemplo, una mejora bastante evidente que se puede hacer a nuestro *inventario de armas* (ver listado 2.32) es cambiar el algoritmo `sort()` por `max_element()` en el método `more_powerful_weapon()`.

Listado 2.35: Modificación del inventario de armas

```
1  Weapon* more_powerful_weapon(void) {
2      WeaponVector weapons = weapons_with_ammo();
3
4      return *max_element(weapons.begin(), weapons.end(),
5                          mem_fun(&Weapon::less_powerful_than));
6  }
```

Aunque no es previsible que sea un contenedor con muchos elementos, buscar el máximo (que es la verdadera intención del método) es mucho más rápido que ordenar la colección y elegir el último.

¿Algoritmos versus métodos del contenedor?

Utilizar los algoritmos genéricos de STL facilita –obviamente– escribir código (o nuevos algoritmos) que pueden operar sobre cualquier contenedor. Lamentablemente, como no podía ser menos, la generalidad suele ir en detrimento de la eficiencia. El algoritmo genérico desconoce intencionadamente los detalles de implementación de cada contenedor, y eso implica que no puede (ni debe) aprovecharlos para trabajar del modo más eficiente posible. Resumiendo, para el algoritmo genérico es más importante ser genérico que eficiente.

En aquellos casos en los que la eficiencia sea más importante que la generalidad (y eso también hay que pensarlo con calma) puede ser más adecuado utilizar los métodos del contenedor en lugar de sus algoritmos funcionalmente equivalentes. Veamos el siguiente listado:

Listado 2.36: Algoritmo genérico vs. método del contenedor

```
1  void test_algorithm_vs_method(void) {
2      int orig[] = {1, 2, 3, 4, 5};
3      const int SIZE = sizeof(orig) / sizeof(int);
4      vector<int> v1, v2;
5
6      copy(orig, orig + SIZE, back_inserter(v1));
7
8      v2.insert(v2.begin(), orig, orig + SIZE);
9
10     assert(equal(v1.begin(), v1.end(), v2.begin()));
11 }
```

Las líneas 6 y 8 realizan la misma operación: añadir a un `vector` el contenido del array `orig`, creando elementos nuevos (los vectores están vacíos). Sin embargo, la versión con `insert()` (línea 8) es más eficiente que `copy()`, ya que realiza menos copias de los elementos.

Del mismo modo, aunque parece más evidente, utilizar métodos en los que se pueden especificar rangos es siempre más eficiente que utilizar sus equivalentes en los que sólo se proporciona un elemento (muchos métodos están sobrecargados para soportar ambos casos).

El libro «Effective STL» [25] de Scott Meyers explica muchas otras «reglas» concretas en las que el uso adecuado de STL puede aumentar notablemente la eficiencia del programa.

2.2.2. Semántica de copia

Una cuestión que a menudo confunde a los programadores novatos en la semántica de copia de STL. Significa que los contenedores almacenan copias de los elementos añadidos, y del mismo modo, devuelven copias cuando se extraen. El siguiente listado ilustra este hecho.

Listado 2.37: Semántica de copia de la STL

```

1 class Counter {
2     int value;
3 public:
4     Counter(void) : value(0) {}
5     void inc(void) { ++value; }
6     int get(void) { return value; }
7 };
8
9 void test_copy_semantics(void) {
10    vector<Counter> counters;
11    Counter c1;
12    counters.push_back(c1);
13    Counter c2 = counters[0];
14
15    counters[0].inc();
16
17    assert(c1.get() == 0);
18    assert(counters[0].get() == 1);
19    assert(c2.get() == 0);
20 }
```

Esto tiene graves implicaciones en la eficiencia de las operaciones que se realizan sobre el contenedor. Todos los algoritmos que impliquen añadir, mover y eliminar elementos dentro de la secuencia (la práctica totalidad de ellos) realizan copias, al menos cuando se trata de contenedores basados en bloque.

El siguiente listado es un «decorador» bastante rudimentario para `string` que imprime información cada vez que una instancia es creada, copiada o destruida.

Listado 2.38: Semántica de copia de la STL

```

1 class String {
2     string value;
3     string desc;
4 public:
5     String(string init) : value(init), desc(init) {
6         cout << "created: " << desc << endl;
7     }
8     String(const String& other) {
9         value = other.value;
10        desc = "copy of " + other.desc;
11        cout << desc << endl;
12    }
13    ~String() {
14        cout << "destroyed: " << desc << endl;
15    }
16    bool operator<(const String& other) const {
17        return value < other.value;
18    }
19 }
```

```

18 }
19 friend ostream&
20 operator<<(ostream& out, const String& str) {
21     out << str.value;
22     return out;
23 }
24 };
25
26 void test_copy_semantics(void) {
27     vector<String> names;
28     names.push_back(String("foo"));
29     names.push_back(String("bar"));
30     names.push_back(String("buzz"));
31     cout << "-- init ready" << endl;
32
33     sort(names.begin(), names.end());
34     cout << "-- sort complete" << endl;
35     String il = names.front();
36     cout << "-- end" << endl;
37 }

```

El resultado al ejecutarlo puede resultar sorprendente:

```

created: foo
copy of foo
destroyed: foo
created: bar
copy of bar
copy of copy of foo
destroyed: copy of foo
destroyed: bar
created: buzz
copy of buzz
copy of copy of copy of foo
copy of copy of bar
destroyed: copy of copy of foo
destroyed: copy of bar
destroyed: buzz
-- init ready
copy of copy of copy of bar
destroyed: copy of copy of copy of bar
copy of copy of buzz
destroyed: copy of copy of buzz
-- sort complete
copy of copy of copy of copy of bar
-- end
destroyed: copy of copy of copy of copy of bar
destroyed: copy of copy of copy of bar
destroyed: copy of copy of buzz
destroyed: copy of copy of copy of foo

```

Como se puede comprobar, las 6 primeras copias corresponden a las inserciones (`push_back()`). El vector reubica todo el contenido cada vez que tiene que ampliar la memoria necesaria, y eso le obliga a copiar en la nueva ubicación los elementos que ya tenía. El algoritmo `sort()` reordena el vector usando solo 2 copias. La asignación implica una copia más. Por último se destruyen los tres objetos que almacena el contenedor y la variable local.

Este ejemplo demuestra la importancia de que nuestras clases dispongan de un constructor de copia correcto y eficiente. Incluso así, muchos programadores prefieren utilizar los contenedores para alma-

cenar punteros en lugar de copias de los objetos, dado que los punteros son simples enteros, su copia es simple, directa y mucho más eficiente. Sin embargo, almacenar punteros es siempre más arriesgado y complica el proceso de limpieza. Si no se tiene cuidado, puede quedar memoria sin liberar, algo difícil de localizar y depurar. Los contenedores no liberan (`delete()`) los punteros que contienen al destruirse. Debe hacerlo el programador explícitamente (ver listado 2.33).

Un punto intermedio entre la eficiencia de almacenar punteros y la seguridad de almacenar copias es utilizar *smart pointers* (aunque **nunca** deben ser `auto_ptr`). Para profundizar en este asunto vea «Implementing Reference Semantics» [20].

2.2.3. Extendiendo la STL

La librería STL está específicamente diseñada para que se pueda extender y adaptar de forma sencilla y eficiente. En esta sección veremos cómo crear o adaptar nuestros propios contenedores, functors y allocators. Ya vimos como crear un algoritmo en la sección 2.1.1.

Creando un contenedor

Dependiendo del modo en que se puede utilizar, los contenedores se clasifican por modelos. A menudo, soportar un modelo implica la existencia de métodos concretos. Los siguientes son los modelos más importantes:

Forward container

Son aquellos que se organizan con un orden bien definido, que no puede cambiar en usos sucesivos. La característica más interesante es que se puede crear más de un iterador válido al mismo tiempo.

Reversible container

Puede ser iterado de principio a fin y viceversa.

Random-access container

Es posible acceder a cualquier elemento del contenedor empleando el mismo tiempo independientemente de su posición.

Front insertion sequence

Permite añadir elementos al comienzo.

Back insertion sequence

Permite añadir elementos al final.

Associative container

Aquellos que permiten acceder a los elementos en función de valores clave en lugar de posiciones.

Cada tipo de contenedor determina qué tipo de iteradores pueden utilizarse para recorrerlo y por tanto qué algoritmos pueden utilizarse con él.

Para ilustrar cuáles son las operaciones que debe soportar un contenedor se incluye a continuación la implementación de `carray`. Se trata de una adaptación (*wrapper*) para utilizar un array C de tamaño constante, ofreciendo la interfaz habitual de un contenedor. En concreto se trata de una modificación de la clase `carray` propuesta inicialmente por Bjarne Stroustrup en su libro «The C++ Programming Language» [36] y que aparece en [20].

Listado 2.39: `carray`: Wrapper STL para un array C

```

1  template<class T, size_t thesize>
2  class carray {
3
4  private:
5      T v[thesize];
6
7  public:
8      typedef T value_type;
9      typedef T* iterator;
10     typedef const T* const_iterator;
11     typedef T& reference;
12     typedef const T& const_reference;
13     typedef size_t size_type;
14     typedef ptrdiff_t difference_type;
15
16     // iteradores
17     iterator begin() { return v; }
18     const_iterator begin() const { return v; }
19     iterator end() { return v+thesize; }
20     const_iterator end() const { return v+thesize; }
21
22     // acceso directo
23     reference operator[](size_t i) { return v[i]; }
24     const_reference operator[](size_t i) const { return v[i]; }
25
26     // size
27     size_type size() const { return thesize; }
28     size_type max_size() const { return thesize; }
29
30     // conversión a array
31     T* as_array() { return v; }
32 };

```

El siguiente listado muestra una prueba de su uso. Como los iteradores de `carray` son realmente punteros ordinarios⁷, este contenedor soporta los modelos *forward* y *reverse container* además de *random access* ya que también dispone del operador de indexación.

⁷No es extraño encontrar implementaciones de contenedores (como `vector`) perfectamente afines al estándar que utilizan punteros convencionales como iteradores

Listado 2.40: carray: Ejemplo de uso de carray

```
1 void test_carray() {
2   carray<int, 5> array;
3
4   for (unsigned i=0; i<array.size(); ++i)
5     array[i] = i+1;
6
7   reverse(array.begin(), array.end());
8
9   transform(array.begin(), array.end(),
10            array.begin(), negate<int>());
11
12  int expected[] = {-5, -4, -3, -2, -1};
13
14  assert(equal(array.begin(), array.end(), expected));
15 }
```

Functor adaptables

Los adaptadores que incorpora la librería (`ptr_fun()`, `mem_fun()`, etc.) ofrecen suficiente flexibilidad como para aprovechar los algoritmos genéricos utilizando predicados u operadores implementados como métodos o funciones. Aún así, en muchos casos puede ser conveniente escribir functors específicos (ver sección 2.1.3).

Como vimos en la sección 2.1.4 existen adaptadores (`bind1st()`, `not()` o `compose()`, etc.) que necesitan conocer el tipo de retorno o de los argumentos del operador que se le pasa. Estos adaptadores requieren un tipo especial de functor, llamado *functor adaptable*, que contiene las definiciones de esos tipos (como `typedefs`). Ese es el motivo por el que no se puede pasar una función convencional a estos adaptadores. Es necesario usar `ptr_fun()` para «convertir» la función convencional en un *functor adaptable*.

Del mismo modo que los predicados y operadores, STL considera los tipos de functors adaptables correspondientes. Así pues:

- Los Generadores adaptables deberán tener un campo con la definición anidada para su tipo de retorno llamada `result_type`.
- Las Funciones unarias adaptables, además del tipo de retorno, deben especificar además el tipo de su único argumento, con el campo `argument_type`. En el caso de los Predicados adaptables se asume que el tipo de retorno es siempre booleano.
- Las Funciones binarias adaptables, además del tipo de retorno, deben especificar el tipo de sus dos argumentos en los campos `first_argument_type` y `second_argument_type`. Del mismo modo, los Predicados binarios no necesitan especificar el tipo de retorno porque se asume que debe ser booleano.

Veamos la implementación del `ptr_fun()` de g++-4.6 para funciones unarias, que demuestra la utilidad de los functor adaptables:

Listado 2.41: Implementación de `ptr_fun()`

```

1 template<typename _Arg, typename _Result>
2 inline pointer_to_unary_function<_Arg, _Result>
3 ptr_fun(_Result (*__x)(_Arg)) {
4     return pointer_to_unary_function<_Arg, _Result>(__x);
5 }

```

Vemos que `ptr_fun()` es una función-plantilla que se instancia (línea 1) con el tipo del argumento (`_Arg`) y el tipo de retorno (`_Result`). La función devuelve una instancia del functor `pointer_to_unary_function` (línea 2) instanciada con los mismos tipos. Y el argumento de la función es un puntero a otra función (línea 4) que obviamente devuelve y acepta un parámetro de los tipos indicados en la plantilla. En resumen, `ptr_fun()` es una factoría que crea instancias del functor unario adaptable `pointer_to_unary_function`.

Para facilitar la creación de functor adaptables, STL ofrece plantillas⁸ que permiten definir los tipos anidados anteriores para los tipos `unary_function` y `binary_function`. Veamos cómo convertir nuestro functor `not_equal` (ver listado 2.10) en un predicado unario adaptable:

Listado 2.42: Predicado `not_equal()` adaptable

```

1 template <typename _Arg>
2 class not_equal : public unary_function<_Arg, bool> {
3     const _Arg _ref;
4
5 public:
6     not_equal(_Arg ref) : _ref(ref) {}
7
8     bool operator()(_Arg value) const {
9         return value != _ref;
10    }
11 };

```

2.2.4. Allocators

Los contenedores ocultan el manejo de la memoria requerida para almacenar los elementos que almacenan. Aunque en la gran mayoría de las situaciones el comportamiento por defecto es el más adecuado, pueden darse situaciones en las que el programador necesita más control sobre el modo en que se pide y libera la memoria. Algunos de esos motivos pueden ser:

- Realizar una reserva contigua, reserva perezosa, cacheado, etc.
- Registrar todas las operaciones de petición y liberación de memoria para determinar cuando ocurren y qué parte del programa es la responsable.

⁸En el fichero de cabecera `<functional>`

- Las características de la arquitectura concreta en la que se ejecuta el programa permiten un manejo más rápido o eficiente de la memoria si se realiza de un modo específico.
- La aplicación permite compartir memoria entre contenedores.
- Hacer una inicialización especial de la memoria o alguna operación de limpieza adicional.

Para lograrlo la STL proporciona una nueva abstracción: el *allocator*. Todos los contenedores estándar utilizan por defecto un tipo de allocator concreto y permiten especificar una alternativo en el momento de su creación, como un parámetro de la plantilla.

Usar un allocator alternativo

Como sabemos, todos los contenedores de STL son plantillas que se instancian con el tipo de dato que van a contener. Sin embargo, tienen un segundo parámetro: el *allocator* que debe aplicar. Veamos las primeras líneas de la definición de `vector`.

Listado 2.43: Definición del contenedor `vector`

```
1 template<typename _Tp, typename _Alloc = std::allocator<_Tp> >
2 class vector : protected _Vector_base<_Tp, _Alloc>
3 {
4     typedef typename _Alloc::value_type _Alloc_value_type;
```

Ese parámetro de la plantilla (`_Alloc`) es opcional porque la definición proporciona un valor por defecto (`std::allocator`). El *allocator* también es una plantilla que se instancia con el tipo de elementos del contenedor.

Si se desea utilizar un *allocator* basta con indicarlo al instanciar el contenedor:

Listado 2.44: Especificando un allocator alternativo

```
1 vector<int, custom_alloc> v;
```

Creando un allocator

El *allocator* es una clase que encapsula las operaciones de petición (método `allocate()`) y liberación (método `deallocate()`) de una cantidad de elementos de un tipo concreto. La signatura de estos métodos es:

Listado 2.45: Métodos básicos del *allocator*

```
1 pointer allocate(size_type n, const void* hint=0);
2 void deallocate(pointer p, size_type n);
```


Crear un allocator no es una tarea sencilla. Lo aconsejable es buscar una librería que proporcione allocators con la funcionalidad deseada, por ejemplo el `pool_alloc` de Boost. Para entender cómo crear un allocator, sin tener que manejar la complejidad que conlleva diseñar y manipular un modelo de memoria especial, se muestra a continuación un wrapper rudimentario para los operadores `new()` y `delete()` estándar. Es una modificación del que propone [20] en la sección 15.4.

Listado 2.46: Un allocator básico con `new` y `delete`

```
1 template <class T>
2 class custom_alloc {
3 public:
4     typedef T value_type;
5     typedef T* pointer;
6     typedef const T* const_pointer;
7     typedef T& reference;
8     typedef const T& const_reference;
9     typedef size_t size_type;
10    typedef ptrdiff_t difference_type;
11
12
13    template <typename U>
14    struct rebind {
15        typedef custom_alloc<U> other;
16    };
17
18    custom_alloc() {}
19
20    custom_alloc(const custom_alloc&) {}
21
22    template <typename U>
23    custom_alloc(const custom_alloc<U>&) {}
24
25    pointer address(reference value) const {
26        return &value;
27    }
28
29    const_pointer address(const_reference value) const {
30        return &value;
31    }
32
33    size_type max_size() const {
34        return numeric_limits<size_t>::max() / sizeof(T);
35    }
36
37    pointer allocate(size_type n, const void* hint=0) {
38        return (pointer) (::operator new(n * sizeof(T)));
39    }
40
41    void deallocate(pointer p, size_type num) {
42        delete p;
43    }
44
45    void construct(pointer p, const T& value) {
46        new (p) T(value);
47    }
48
49    void destroy(pointer p) {
50        p->~T();
51    }
```

Las líneas 4 a 15 definen una serie de tipos anidados que todo allocator debe tener:

value_type

El tipo del dato del objeto que almacena.

reference y const_reference

El tipo de las referencia a los objetos.

pointer y const_pointer

El tipo de los punteros a los objetos.

size_type

El tipo que representa los tamaños (en bytes) de los objetos.

difference_type

El tipo que representa la diferencia entre dos objetos.

Las líneas 18 a 23 contienen el constructor por defecto, el constructor de copia un constructor que acepta una instancia del mismo allocator para otro tipo. Todos ellos están vacíos porque este allocator no tiene estado.

El método polimórfico `address()` (líneas 25 a 31) devuelve la dirección del objeto. El método `max_size()` devuelve el mayor valor que se puede almacenar para el tipo concreto.

Por último, los métodos `allocate()` y `deallocate()` sirven para pedir y liberar memoria para el objeto. Los métodos `construct()` y `destroy()` construyen y destruyen los objetos.

2.3. Estructuras de datos no lineales

Cualquier programa donde la eficiencia sea importante, y es el caso de la mayoría de los videojuegos, necesitan estructuras de datos específicas. Hay varios motivos para ello:

- Hasta ahora hemos estudiado fundamentalmente la STL, que oculta la estructura real de los contenedores ofreciendo un aspecto de estructura lineal. Así, por ejemplo, los objetos de tipo `map` o `set` se representan realmente mediante árboles, aunque el programador está completamente aislado de ese detalle de implementación. Solo podemos anticipar la estructura subyacente mediante indicadores indirectos, como la complejidad de las operaciones o la estabilidad de los iteradores.
- Algunas estructuras, como es el caso de los grafos, no tienen una representación lineal evidente y se pueden recorrer de distintas formas. Por tanto debe existir un número variable de iteradores.
- Las estructuras de la STL están diseñadas para uso general. El diseñador no puede anticipar en qué condiciones se van a usar por lo que toma las decisiones apropiadas para el mayor número

de casos posible. Conociendo los detalles (gestión de memoria, algoritmos que se van a aplicar) se pueden obtener rendimientos muy superiores con mínimas modificaciones sobre la estructura subyacente.

Como ya hemos puntualizado en capítulos anteriores, es muy importante no optimizar de manera prematura. Para ilustrar este aspecto veamos el siguiente ejemplo tomado de [10], capítulo 11.

Anti-optimizaciones

Con los compiladores actuales es muy difícil implementar código equivalente a la STL más eficiente. Algunos ejemplos de [10] hoy en día son completamente diferentes.

Listado 2.47: Dos formas de sumar enteros

```

1 int myIntegerSum(int* a, int size) {
2   int sum=0;
3   int* begin = a;
4   int* end = a + size;
5   for (int* p = begin; p != end; ++p)
6     sum += *p;
7   return sum;
8 }
9
10 int stlIntegerSum(int* a, int size) {
11   return accumulate(a, a+size, 0);
12 }
```

En dicho libro se argumentaba que la función `myIntegerSum()` es casi cuatro veces más rápida que `stlIntegerSum()`. Y probablemente era verdad en el año 1999. Sin embargo hoy en día, empleando GNU g++ 4.6.2 o clang++ 3.0 el resultado es prácticamente idéntico, con una muy ligera ventaja hacia la versión basada en la STL.

2.3.1. Árboles binarios

Las estructuras arborescentes se encuentran entre las más utilizadas en la programación de todo tipo de aplicaciones. Ya hemos visto en el módulo 2 algunas de sus aplicaciones para el mezclado de animaciones (*Priority Blend Tree*), o para indexar el espacio (*BSP Tree*, *quatree*, *octree*, *BBT*). Estudiaremos su funcionamiento en este capítulo, pero el desarrollo de videojuegos no se limita a los gráficos, por lo que otro tipo de árboles más generales pueden resultar también necesarios.

Los árboles se utilizan con frecuencia como mecanismo eficiente de búsqueda. Para este fin implementan un rico conjunto de operaciones: búsqueda de un elemento, mínimo o máximo, predecesor o sucesor de un elemento y las clásicas operaciones de inserción y borrado. Se pueden emplear como un diccionario o como una cola con prioridad.

Todas estas operaciones están presentes en los contenedores ordenados de la STL, singularmente `set`, `multiset`, `map` y `multimap`. No debe extrañar por tanto que en todos ellos se emplea una variante de árbol binario denominada *red-black tree*.

Un nodo de árbol contiene habitualmente un atributo `key` que se emplea para compararlo con otros nodos y además mantiene un conjunto de punteros a otros nodos que mantienen su relación con el

resto de la estructura. Así, por ejemplo, los nodos de árboles binarios mantienen un atributo `parent` que apunta al nodo padre, y un par de punteros `left` y `right` que apuntan al hijo por la izquierda y por la derecha respectivamente. A su vez cada hijo puede tener otros nodos hijos, por lo que realmente cada nodo cuenta con dos subárboles (izquierdo y derecho).



Las operaciones básicas de los árboles se ejecutan en un tiempo proporcional a la altura del árbol. Eso implica $O(\log n)$ en el caso peor si está correctamente balanceado, pero $O(n)$ si no lo está.

Árboles de búsqueda binaria

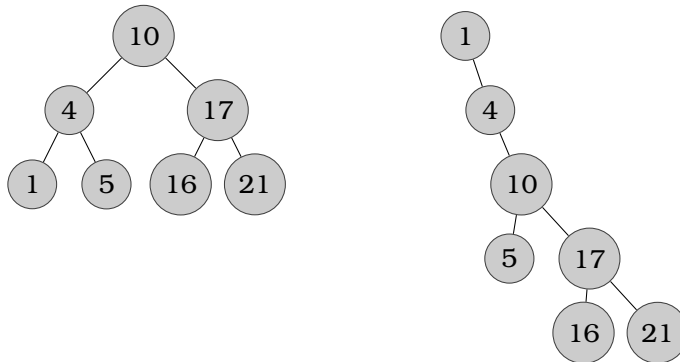


Figura 2.2: Dos árboles de búsqueda binaria. Ambos contienen los mismos elementos pero el de la izquierda es mucho más eficiente.

Los árboles de búsqueda binaria se definen por la siguiente propiedad:

Todos los nodos del subárbol izquierdo de un nodo tienen una clave menor o igual a la de dicho nodo. Análogamente, la clave de un nodo es siempre menor o igual que la de cualquier otro nodo del subárbol derecho.

Por tratarse del primer tipo de árboles expondremos con cierto detalle su implementación. Como en cualquier árbol necesitamos modelar los nodos del árbol, que corresponden a una simple estructura:

Listado 2.48: Estructura de un nodo de árbol de búsqueda binaria

```
1 template <typename KeyType>
2 struct Node {
3     typedef Node<KeyType> NodeType;
4
5     KeyType key;
6     NodeType* parent;
7     NodeType* left;
8     NodeType* right;
```

Sobre esta misma estructura es posible definir la mayoría de las operaciones de un árbol. Por ejemplo, el elemento más pequeño podría definirse como un método estático de esta manera:

Listado 2.49: Búsqueda del elemento mínimo en un árbol de búsqueda binaria

```
1 static NodeType* minimum(NodeType* x) {
2     if (x == 0) return x;
3     if (x->left != 0) return minimum(x->left);
4     return x;
5 }
```

Para obtener el mínimo basta recorrer todos los subárboles de la izquierda y análogamente para encontrar el máximo hay que recorrer todos los subárboles de la derecha hasta llegar a un nodo sin subárbol derecho.

Listado 2.50: Búsqueda del elemento máximo en un árbol de búsqueda binaria

```
1 static NodeType* maximum(NodeType* x) {
2     if (x == 0) return x;
3     if (x->right != 0) return maximum(x->right);
4     return x;
5 }
```

El motivo de utilizar métodos estáticos en lugar de métodos normales es poder invocarlos para el nodo nulo. Los métodos de clase invocados sobre un objeto nulo tienen un comportamiento indefinido.



Nuestra implementación del método estático `minimum()` es recursiva. Con frecuencia se argumenta que una implementación iterativa es más eficiente porque no crea un número indefinido de marcos de pila. Realmente eso depende del tipo de recursión. Cuando el compilador puede detectar recursión por la cola, es decir, cuando tras la llamada recursiva no quedan operaciones pendientes de realizar, el compilador puede optimizar el código y eliminar completamente la llamada recursiva.

Las instancias de `Node` no tienen por qué ser visibles directamente al programador, al igual que los contenedores tipo `set` de la STL. Por ejemplo, esto puede lograrse utilizando un *namespace* privado.

La búsqueda de un elemento también puede plantearse con un algoritmo recursivo aprovechando la propiedad que define a los árboles de búsqueda binaria:

Listado 2.51: Búsqueda de una clave en un árbol de búsqueda binaria

```

1  static NodeType* search(NodeType* x, KeyType k) {
2      if (x == 0 || x->key == k) return x;
3      else if (k < x->key) return search(x->left, k);
4      else return search(x->right, k);
5  }
```

También pueden implementarse directamente los métodos `successor()` y `predecesor()` para encontrar el nodo siguiente o anterior a uno dado según el orden de las claves:

Listado 2.52: Búsqueda del sucesor de un nodo en un árbol de búsqueda binaria

```

1  static NodeType* successor(NodeType* x) {
2      if (x->right != 0) return minimum(x->right);
3      NodeType* parent = x->parent;
4      while (parent != 0 && x == parent->right) {
5          x = parent;
6          parent = x->parent;
7      }
8  }
```

Si hay un subárbol a la derecha del nodo entonces es el mínimo de ese subárbol (en la figura 2.2 izquierda el sucesor de 10 es 16). Si no lo hay entonces tendremos que subir hasta el primer padre que tiene al nodo como subárbol izquierdo (en la figura 2.2 izquierda el sucesor de 5 es 10).

Se propone como ejercicio la implementación de la búsqueda del predecesor de un nodo determinado.

El resto de las operaciones básicas sobre un árbol (inserción y borrado de elementos) requiere de una estructura que actúa como fachada de los nodos del árbol.

Listado 2.53: Estructura de un árbol de búsqueda binaria

```

1  template <class KeyType>
2  struct Tree {
3      typedef Node<KeyType> NodeType;
4
5      NodeType* root;
```

El atributo `root` mantiene cuál es el nodo raíz del árbol. Los métodos de inserción y borrado deben actualizarlo adecuadamente.

Listado 2.54: Inserción en un árbol de búsqueda binaria

```

1  void insert(NodeType* z) {
2      NodeType* y = 0;
3      NodeType* x = root;
4      while (x != 0) {
5          y = x;
6          if (z->key < x->key)
7              x = x->left;
8          else
9              x = x->right;
10     }
11     z->parent = y;
12     if (y == 0) root = z;
13     else if (z->key < y->key)
14         y->left = z;
15     else
16         y->right = z;
17 }

```

Básicamente replica el procedimiento de búsqueda para encontrar el hueco donde debe insertar el elemento, manteniendo el padre del elemento actual para poder recuperar el punto adecuado al llegar a un nodo nulo.

El procedimiento más complejo es el de borrado de un nodo. De acuerdo a [12] se identifican los cuatro casos que muestra la figura 2.3. Un caso no representado es el caso trivial en el que el nodo a borrar no tenga hijos. En ese caso basta con modificar el nodo padre para que el hijo correspondiente sea el objeto nulo. Los dos primeros casos de la figura corresponden al borrado de un nodo con un solo hijo, en cuyo caso el hijo pasa a ocupar el lugar del nodo a borrar. El tercer caso corresponde al caso en que el hijo derecho no tenga hijo izquierdo o el hijo izquierdo no tenga hijo derecho, en cuyo caso se puede realizar la misma operación que en los casos anteriores enlazando adecuadamente las dos ramas. El cuarto caso corresponde al caso general, con dos hijos no nulos. En ese caso buscamos un sucesor del subárbol izquierdo que no tenga hijo izquierdo, que pasa a reemplazar al nodo, reajustando el resto para mantener la condición de árbol de búsqueda binaria.

Con el objetivo de facilitar el movimiento de subárboles definimos el método `transplant()`. El subárbol con raíz `u` se reemplaza con el subárbol con raíz `v`.

Listado 2.55: Transplantado de subárboles en un árbol de búsqueda binaria

```

1  void transplant(NodeType* u, NodeType* v) {
2      if (u->parent == 0)
3          root = v;
4      else if (u == u->parent->left)
5          u->parent->left = v;
6      else
7          u->parent->right = v;
8      if (v != 0)
9          v->parent = u->parent;
10 }

```

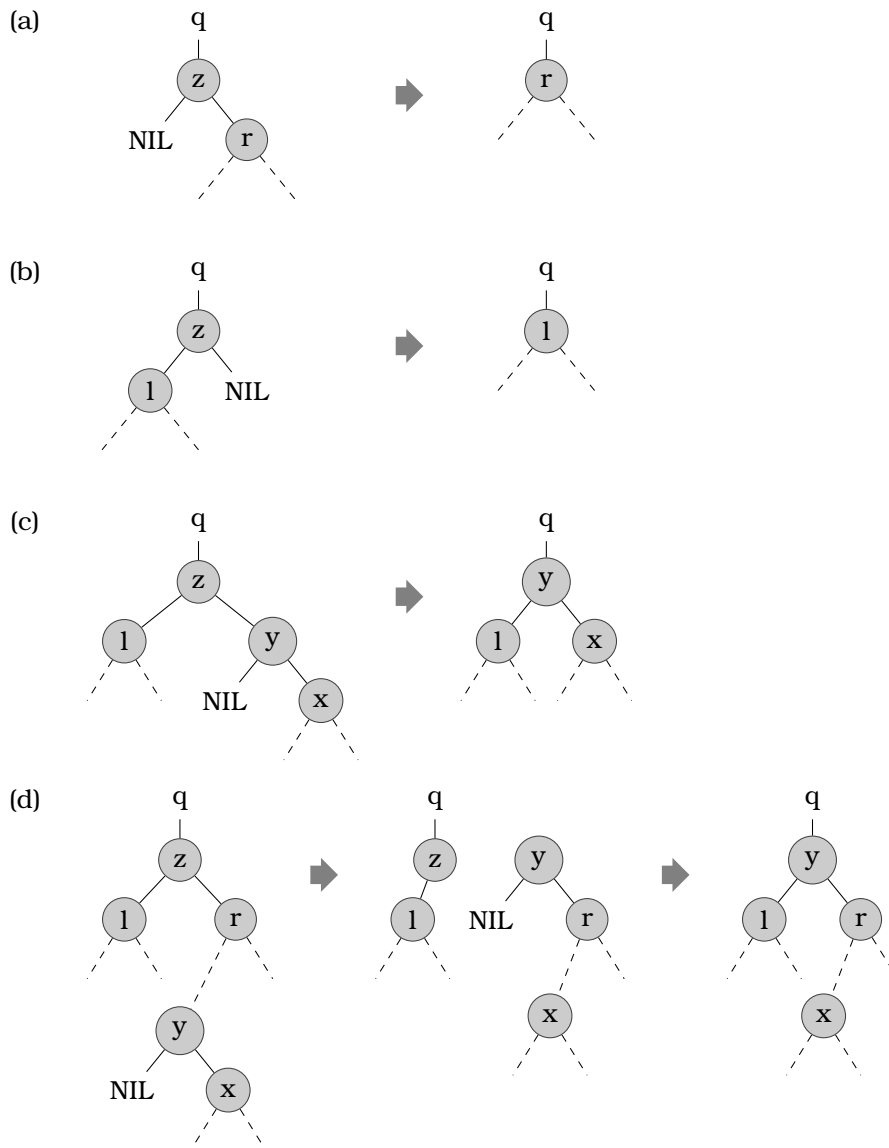


Figura 2.3: Casos posibles según [12] en el borrado de un nodo en un árbol de búsqueda binaria

Nótese que no alteramos el nodo padre de *v* ni los hijos de *v*. La responsabilidad de actualizarlos corresponde al que llama a `transplant()`.

Empleando este procedimiento auxiliar es muy sencilla la implementación de `remove()`.

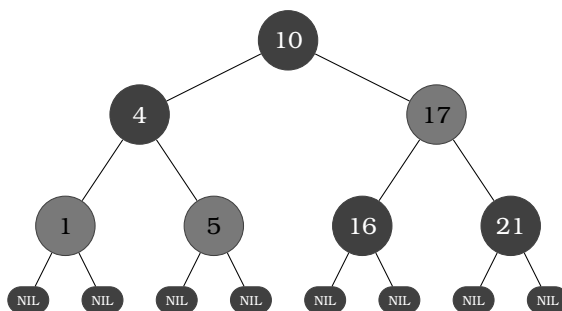


Figura 2.4: Ejemplo de árbol rojo-negro. Los nodos hoja no se representarán en el resto del texto.

Listado 2.56: Borrado en un árbol de búsqueda binaria

```

1  void remove(NodeType* z) {
2      if (z->left == 0)
3          transplant(z, z->right);
4      else if (z->right == 0)
5          transplant(z, z->left);
6      else {
7          NodeType* y = NodeType::minimum(z->right);
8          if (y->parent != z) {
9              transplant(y, y->right);
10             y->right = z->right;
11             y->right->parent = y;
12         }
13         transplant(z, y);
14         y->left = z->left;
15         y->left->parent = y;
16     }
17 }

```

Todos los procedimientos básicos (`minimum()`, `maximum()`, `search()`, `predecesor()`, `sucesor()`, `insert()` y `remove()`) se ejecutan en tiempo $O(h)$ donde h es la altura del árbol. Si el árbol está equilibrado esto implica $O(\log n)$.

Red-black trees

La eficiencia de un árbol de búsqueda binaria depende enormemente del orden en que se introduzcan los elementos. Pueden ser muy eficientes o en el caso peor degenerar a una simple lista doblemente enlazada. Para resolver este problema se han propuesto multitud de esquemas que garantizan que el árbol siempre está equilibrado complicando ligeramente la inserción y borrado.

Los árboles rojo-negro son un caso de estos árboles de búsqueda binaria balanceados. Cada nodo almacena un bit extra, el color, que puede ser rojo o negro. En cada camino simple desde el nodo raíz a una hoja se restringen los colores de manera que nunca pueda ser un camino más del doble de largo que otro cualquiera:

1. Cada nodo es rojo o negro.
2. El nodo raíz es negro.
3. Las hojas del árbol (objetos nulos) son negras.
4. Los hijos de un nodo rojo son negros.
5. Los caminos desde un nodo a todas sus hojas descendientes contienen el mismo número de nodos negros.

Podemos simplificar los algoritmos eliminando la necesidad de comprobar si es un nodo nulo antes de indexar un elemento sin más que utilizar un nodo especial que usamos como centinela. La estructura del nodo podría ser algo así:

Listado 2.57: Definición de un nodo de un árbol rojo-negro.

```

1  template <typename KeyType>
2  struct Node {
3      typedef Node<KeyType> NodeType;
4      enum Color { Red = false, Black = true };
5
6      KeyType key;
7      NodeType* parent;
8      NodeType* left;
9      NodeType* right;
10     Color color;
11
12     Node() {
13         left = right = parent = nil();
14     }
15
16     static NodeType* nil() {
17         if (!_nil)
18             _nil = new Node(Black);
19         return &_nil;
20     }

```

Las operaciones `maximum()`, `minimum()`, `search()`, `successor()` y `predecesor()` son completamente análogas a las de los árboles de búsqueda binaria tradicionales, salvo que ahora está garantizado que se ejecutan en tiempo $O(\log n)$. Por ejemplo, la función `maximum()` sería:

Listado 2.58: Búsqueda del mayor elemento en un árbol rojo-negro.

```

1  static NodeType* maximum(NodeType* x) {
2      if (x->right != NodeType::nil()) return maximum(x->right);
3      return x;
4  }

```

Nótese que ya no es necesario comprobar si `x` es nulo antes de indexar su miembro `right`, puesto que para representar al nodo nulo usamos un centinela perfectamente válido.

En cambio las operaciones de inserción y borrado deben ser modificadas para garantizar que se mantienen las propiedades de árbol rojo-negro. Para ello nos apoyaremos en dos funciones auxiliares: `rotate_left()` y `rotate_right()`:

Listado 2.59: Rotación a la izquierda en un árbol de búsqueda binaria

```

1  void rotate_left(NodeType* x) {
2      NodeType* y = x->right;
3      x->right = y->left;
4      if (y->left != NodeType::nil())
5          y->left->parent = x;
6      y->parent = x->parent;
7      if (x->parent == NodeType::nil())
8          root = y;
9      else if (x == x->parent->left)
10         x->parent->left = y;
11     else
12         x->parent->right = y;
13     y->left = x;
14     x->parent = y;
15 }

```

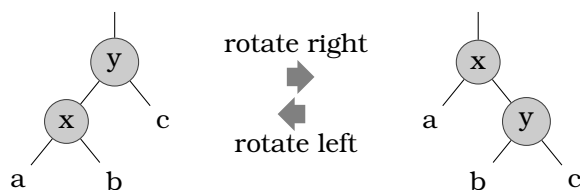


Figura 2.5: Operación de rotación a la derecha o a la izquierda en un árbol de búsqueda binaria

La operación dual `rotate_right()` puede implementarse simplemente intercambiando en el algoritmo anterior `x` por `y`, y `left` por `right`.

La inserción puede ahora realizarse de una forma muy parecida al caso general asumiendo que el color del nodo a insertar es rojo y después arreglando el árbol con rotaciones y cambios de color.

Listado 2.60: Inserción en un árbol rojo-negro

```

1  void insert(NodeType* z) {
2      NodeType* y = NodeType::nil();
3      NodeType* x = root;
4      while (x != NodeType::nil()) {
5          y = x;
6          if (z->key < x->key)
7              x = x->left;
8          else
9              x = x->right;
10     }
11     z->parent = y;
12     if (y == NodeType::nil())
13         root = z;

```

```

14     else if (z->key < y->key)
15         y->left = z;
16     else
17         y->right = z;
18     z->left = NodeType::nil();
19     z->right = NodeType::nil();
20     z->color = Node::Color::Red;
21     insert_fixup(z);
22 }

```

Al asumir el color rojo podemos haber violado las reglas de los árboles rojo-negro. Por esta razón llamamos a la función `insert_fixup()` que garantiza el cumplimiento de las reglas tras la inserción:

Listado 2.61: Reparación tras la inserción en árbol rojo-negro

```

1  void insert_fixup(NodeType* z) {
2      while (z->parent->color == Node::Color::Red) {
3          if (z->parent == z->parent->parent->left) {
4              NodeType* y = z->parent->parent->right;
5              if (y->color == Node::Color::Red) {
6                  z->parent->color = Node::Color::Black;
7                  y->color = Node::Color::Black;
8                  z->parent->parent->color = Node::Color::Red;
9                  z = z->parent->parent;
10             }
11             else {
12                 if (z == z->parent->right) {
13                     z = z->parent;
14                     rotate_left(z);
15                 }
16                 z->parent->color = Node::Color::Black;
17                 z->parent->parent->color = Node::Color::Red;
18                 rotate_right(z->parent->parent);
19             }
20         }
21         else {
22             NodeType* y = z->parent->parent->left;
23             if (y->color == Node::Color::Red) {
24                 z->parent->color = Node::Color::Black;
25                 y->color = Node::Color::Black;
26                 z->parent->parent->color = Node::Color::Red;
27                 z = z->parent->parent;
28             }
29             else {
30                 if (z == z->parent->left) {
31                     z = z->parent;
32                     rotate_right(z);
33                 }
34                 z->parent->color = Node::Color::Black;
35                 z->parent->parent->color = Node::Color::Red;
36                 rotate_left(z->parent->parent);
37             }
38         }
39     }
40     root->color = Node::Color::Black;
41 }

```

La inserción de un nodo rojo puede violar la regla 2 (el nodo raíz queda como rojo en el caso de un árbol vacío) o la regla 4 (el nodo insertado pasa a ser hijo de un nodo rojo). Este último caso es el que

se contempla en el bucle de la función `insert_fixup()`. Cada una de las dos ramas del `if` sigue la estrategia dual, dependiendo de si el padre es un hijo derecho o izquierdo. Basta estudiar el funcionamiento de una rama, dado que la otra es idéntica pero intercambiando `right` y `left`. Básicamente se identifican tres casos.

- El primero corresponde a las líneas [6] a [9]. Es el caso en que el nodo a insertar pasa a ser hijo de un nodo rojo cuyo hermano también es rojo (e.g. figura 2.6.a). En este caso el nodo padre y el nodo tío se pasan a negro mientras que el abuelo se pasa a rojo (para mantener el número de nodos negros en todos los caminos). Al cambiar a rojo el nodo abuelo es posible que se haya vuelto a violar alguna regla, y por eso se vuelven a comprobar los casos.
- Otra posibilidad es que el nodo tío sea negro y además el nodo insertado sea hijo derecho (e.g. figura 2.6.b). En ese caso se realiza una rotación a la izquierda para reducirlo al caso siguiente y se aplica lo correspondiente al último caso.
- El último caso corresponde a que el nodo tío sea negro y el nodo insertado sea hijo izquierdo (e.g. figura 2.6.c). En ese caso se colorea el padre como negro, y el abuelo como rojo, y se rota a la derecha el abuelo. Este método deja un árbol correcto.

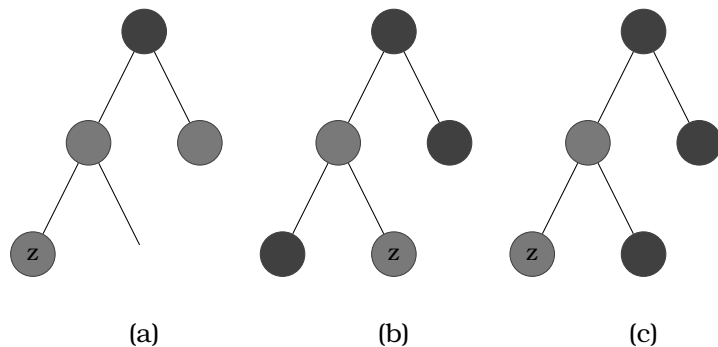


Figura 2.6: Casos contemplados en la función `insert_fixup()`.

El borrado también se apoya en la función `transplant()` que es muy similar al caso de los árboles de búsqueda binaria.

Listado 2.62: Transplantado de subárboles en árbol rojo-negro

```

1  void transplant(NodeType* u, NodeType* v) {
2      if (u->parent == NodeType::nil())
3          root = v;
4      else if (u == u->parent->left)
5          u->parent->left = v;
6      else
7          u->parent->right = v;
8      v->parent = u->parent;
9  }
```

Con este procedimiento auxiliar el borrado de un nodo queda relativamente similar al caso de árboles de búsqueda binaria.

Listado 2.63: Borrado de un nodo en árboles rojo-negro

```

1  void remove(NodeType* z) {
2      NodeType* y = z;
3      NodeType* x;
4      NodeType::Color y_orig_color = y->color;
5      if (z->left == NodeType::nil()) {
6          x = z->right;
7          transplant(z, z->right);
8      }
9      else if (z->right == NodeType::nil()) {
10         x = z->left;
11         transplant(z, z->left);
12     }
13     else {
14         y = Node::minimum(z->right);
15         y_orig_color = y->color;
16         x = y->right;
17         if (y->parent == z) {
18             x->parent = y;
19         }
20         else {
21             transplant(y, y->right);
22             y->right = z->right;
23             y->right->parent = y;
24         }
25         transplant(z, y);
26         y->left = z->left;
27         y->left->parent = y;
28         y->color = z->color;
29     }
30     if (y_orig_color == Node::Color::Black)
31         rb_remove_fixup(x);
32 }

```

El nodo *y* corresponde al nodo que va a eliminarse o moverse dentro del árbol. Será el propio *z* si tiene menos de dos hijos o el nodo *y* de los casos c y d en la figura 2.3. Mantenemos la variable *y_orig_color* con el color que tenía ese nodo que se ha eliminado o movido dentro del árbol. Solo si es negro puede plantear problemas de violación de reglas, porque el número de nodos negros por cada rama puede variar. Para arreglar los problemas potenciales se utiliza una función análoga a la utilizada en la inserción de nuevos nodos.

Listado 2.64: Reparación tras borrar un nodo en árboles rojo-negro

```

1  void remove_fixup(NodeType* x) {
2      while (x != root && x->color == Node::Color::Black) {
3          if (x == x->parent->left) {
4              NodeType* w = x->parent->right;
5              if (w->color == Node::Color::Red) {
6                  w->color = Node::Color::Black;
7                  x->parent->color = Node::Color::Red;
8                  rotate_left(x->parent);
9                  w = x->parent->right;
10             }
11             if (w->left->color == Node::Color::Black)

```

```

12         && w->right->color == Node::Color::Black) {
13             w->color = Node::Color::Red;
14             x = x->parent;
15         }
16         else {
17             if (w->right->color == Node::Color::Black) {
18                 w->left->color = Node::Color::Black;
19                 w->color = Node::Color::Red;
20                 rotate_right(w);
21                 w = x->parent->right;
22             }
23             w->color = x->parent->color;
24             x->parent->color = Node::Color::Black;
25             w->right->color = Node::Color::Black;
26             rotate_left(x->parent);
27             x = root;
28         }
29     }
30     else {
31         NodeType* w = x->parent->left;
32         if (w->color == Node::Color::Red) {
33             w->color = Node::Color::Black;
34             x->parent->color = Node::Color::Red;
35             rotate_right(x->parent);
36             w = x->parent->left;
37         }
38         if (w->right->color == Node::Color::Black
39             && w->left->color == Node::Color::Black) {
40             w->color = Node::Color::Red;
41             x = x->parent;
42         }
43         else {
44             if (w->left->color == Node::Color::Black) {
45                 w->right->color = Node::Color::Black;
46                 w->color = Node::Color::Red;
47                 rotate_left(w);
48                 w = x->parent->left;
49             }
50             w->color = x->parent->color;
51             x->parent->color = Node::Color::Black;
52             w->left->color = Node::Color::Black;
53             rotate_right(x->parent);
54             x = root;
55         }
56     }
57 }
58 x->color = Node::Color::Black;
59 }

```

Nuevamente se trata de un código dual. En el *if* más exterior se distinguen los casos de borrar un hijo derecho o izquierdo. En ambas ramas se encuentra el mismo código intercambiando *left* por *right*. Por tanto basta analizar la primera de ellas.

Se distinguen cuatro casos:

- El hermano *w* es rojo. En ese caso forzosamente los hijos de *w* deben ser negros. Por tanto se puede intercambiar los colores del hermano y del padre y hacer una rotación a la izquierda sin violar nuevas reglas. De esta forma el nuevo hermano será forzosamente negro, por lo que este caso se transforma en alguno de los posteriores.

- El hermano w es negro y los dos hijos de w son negros. En ese caso cambiamos el color del hermano a rojo. De esta forma se equilibra el número de negros por cada rama, pero puede generar una violación de reglas en el nodo padre, que se tratará en la siguiente iteración del bucle.
- El hermano w es negro y el hijo izquierdo de w es rojo. En ese caso intercambiamos los colores de w y su hijo izquierdo y hacemos una rotación a la derecha. De esta forma hemos reducido este caso al siguiente.
- El hermano w es negro y el hijo derecho de w es rojo. En este caso cambiando colores en los nodos que muestra la figura 2.7.d y rotando a la izquierda se obtiene un árbol correcto que compensa el número de negros en cada rama.

AVL trees

Los árboles AVL son otra forma de árbol balanceado en el que se utiliza la altura del árbol como criterio de balanceo. Solo puede haber una diferencia de 1 entre la altura de dos ramas. Es por tanto un criterio más estricto que los *red-black trees*, lo que lo hace menos eficiente en las inserciones y borrados pero más eficiente en las lecturas.

Cada nodo tiene información adicional con la altura del árbol en ese punto. En realidad tan solo es necesario almacenar el *factor de equilibrio* que es simplemente la diferencia entre las alturas del subárbol izquierdo y el derecho. La ventaja de esta última alternativa es que es un número mucho más reducido (siempre comprendido en el rango -2 a +2) por lo que puede almacenarse en solo 3 bits.

Para insertar elementos en un árbol AVL se utiliza un procedimiento similar a cualquier inserción en árboles de búsqueda binaria, con dos diferencias:

- La inserción debe computar el factor de equilibrio en los nodos afectados.
- Finalmente hay que equilibrar el árbol si es necesario.

El equilibrado se realiza con rotaciones siguiendo el procedimiento representado en la figura 2.8. Es importante destacar que las propias funciones de rotación alteran los factores de equilibrio de los nodos involucrados (nodos x e y en la figura 2.5).

Árboles balanceados

Los *red-black trees* son más eficientes en `insert()` y `remove()`, pero los *AVL trees* son más eficientes en `search()`.



Las operaciones `insert()`, `rotate_right()`, `rotate_left()` y `remove()` sobre árboles AVL deben recalculan el factor de equilibrio en los nodos afectados. Además, en caso de dejar un árbol desequilibrado, las operaciones `insert()` y `remove()` deben equilibrar el árbol según el procedimiento descrito en la figura 2.8.

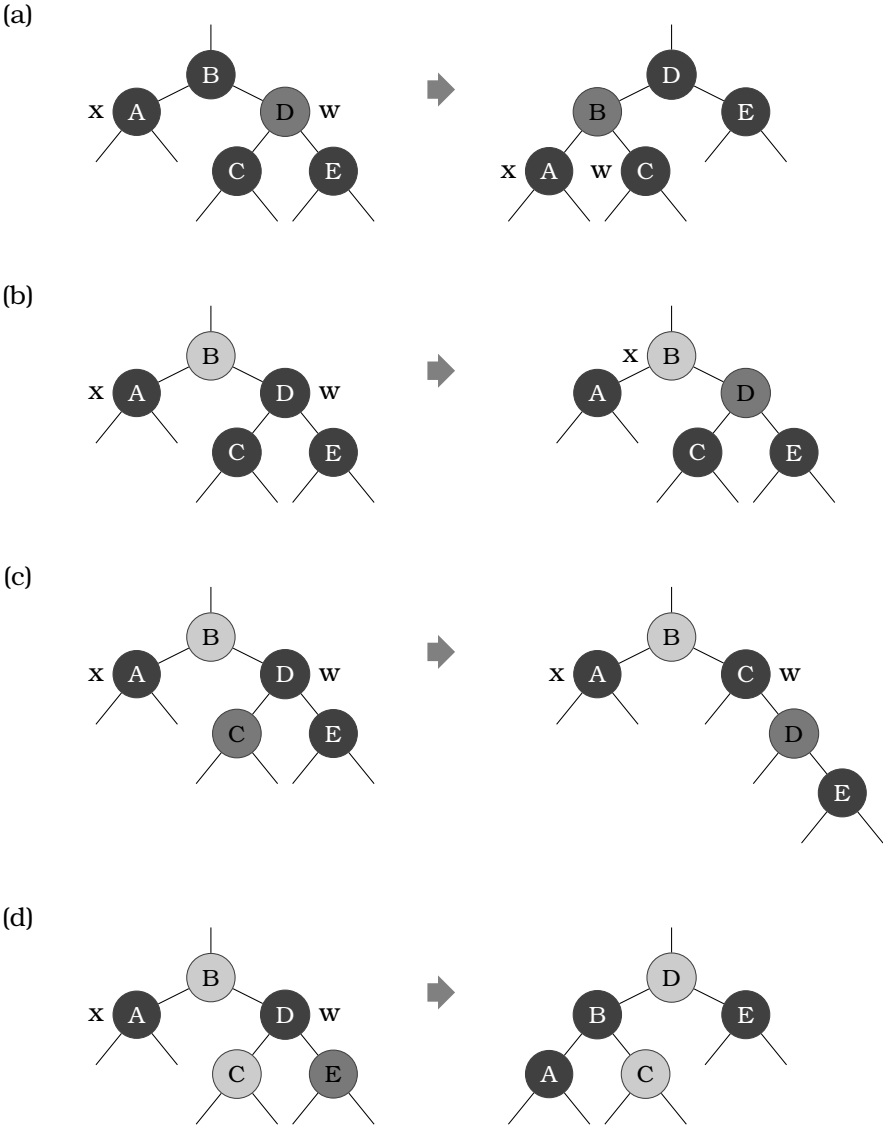


Figura 2.7: Casos contemplados en la función `remove_fixup()` según [12].

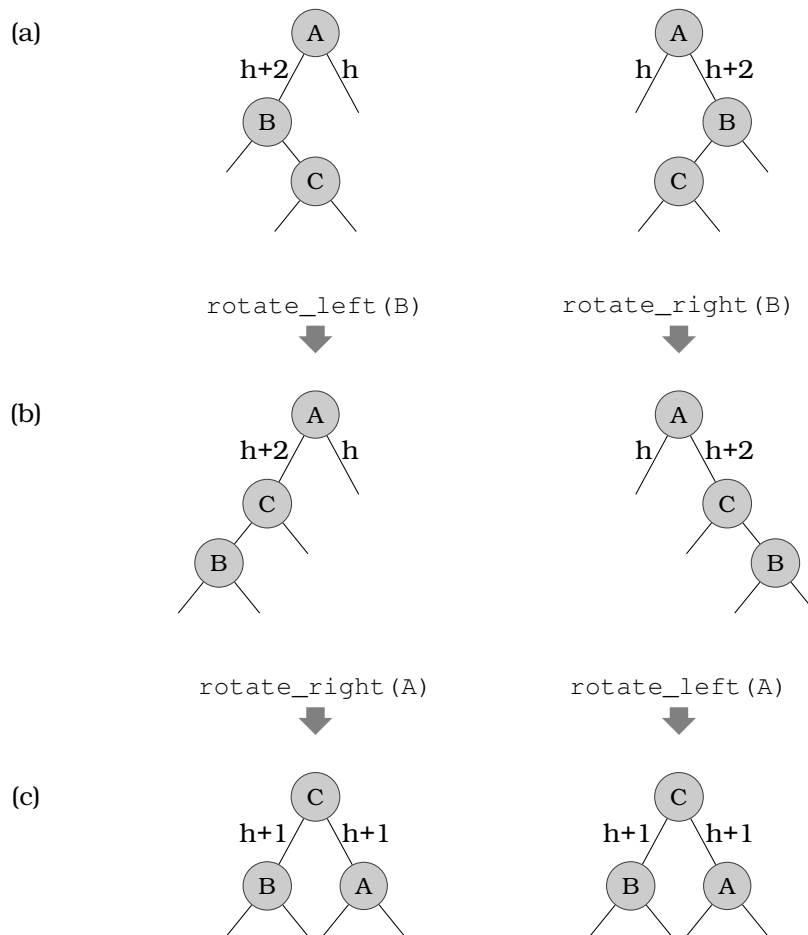


Figura 2.8: Casos contemplados en la función de equilibrado de árboles AVL.

Radix tree

Aún hay otro tipo de árboles binarios que vale la pena comentar por sus implicaciones con los videojuegos. Se trata de los *árboles de prefijos*, frecuentemente llamados *tries*⁹.

La figura 2.9 muestra un ejemplo de árbol de prefijos con un conjunto de enteros binarios. El árbol los representa en orden lexicográfico. Para cada secuencia binaria si empieza por 0 está en el subárbol izquierdo y si empieza por 1 en el subárbol derecho. Conforme se recorren las ramas del árbol se obtiene la secuencia de bits del número a buscar. Es decir, el tramo entre el nodo raíz y cualquier nodo intermedio define el prefijo por el que empieza el número a buscar. Por eso a este árbol se le llama *prefix tree* o *radix tree*.

⁹El nombre en singular es *trie*, que deriva de **tr**ieve. Por tanto la pronunciación correcta se asemeja a la de *tree*, aunque muchos autores la pronuncian como *try*.

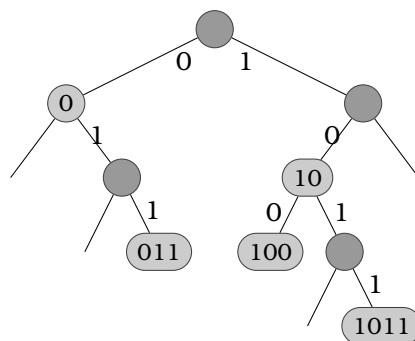


Figura 2.9: Un ejemplo de *trie* extraído de [12]. Contiene los elementos 1011, 10, 011, 100 y 0.

Un árbol de prefijos (pero no binario) se utiliza frecuentemente en los diccionarios predictivos de los teléfonos móviles. Cada subárbol corresponde a una nueva letra de la palabra a buscar.

También se puede utilizar un árbol de prefijos para indexar puntos en un segmento de longitud arbitraria. Todos los puntos en la mitad derecha del segmento están en el subárbol derecho, mientras que todos los puntos de la mitad izquierda están en el subárbol izquierdo. Cada subárbol tiene las mismas propiedades con respecto al subsegmento que representa. Es decir, el subárbol derecho es un árbol de prefijos que representa a medio segmento derecho, y así sucesivamente. El número de niveles del árbol es ajustable dependiendo de la precisión que requerimos en el posicionamiento de los puntos.

En los árboles de prefijos la posición de los nodos está prefijada a priori por el valor de la clave. Estos árboles no realizan ninguna función de equilibrado por lo que su implementación es trivial. Sin embargo estarán razonablemente equilibrados si los nodos presentes están uniformemente repartidos por el espacio de claves.

2.3.2. Recorrido de árboles

En multitud de ocasiones es necesario recorrer los elementos de un árbol en un orden determinado. Son frecuentes los recorridos *en orden*, *en preorden*, y *en postorden*.

El recorrido *en orden* sigue el orden del campo clave. Es decir, para cualquier nodo primero se visitan los nodos del subárbol izquierdo, luego el nodo y finalmente los nodos del subárbol derecho.

Listado 2.65: Recorrido *en orden* en un árbol de búsqueda binaria

```

1  template <typename Func>
2  void inorder_tree_walk(Func f) {
3      inorder_tree_walk(root, f);
4  }
5
6  template <typename Func>
```

```

7     void inorder_tree_walk(NodeType* x, Func f) {
8         if (x == 0) return;
9         inorder_tree_walk(x->left, f);
10        f(x);
11        inorder_tree_walk(x->right, f);
12    }

```

El recorrido *en preorden* visita el nodo antes de cualquiera de sus subárboles.

Listado 2.66: Recorrido *en preorden* en un árbol de búsqueda binaria

```

1     template <typename Func>
2     void preorder_tree_walk(Func f) {
3         preorder_tree_walk(root, f);
4     }
5
6     template <typename Func>
7     void preorder_tree_walk(NodeType* x, Func f) {
8         if (x == 0) return;
9         f(x);
10        preorder_tree_walk(x->left, f);
11        preorder_tree_walk(x->right, f);
12    }

```

Finalmente el recorrido *en postorden* visita el nodo después de visitar ambos subárboles.

Listado 2.67: Recorrido *en postorden* en un árbol de búsqueda binaria

```

1     template <typename Func>
2     void postorder_tree_walk(Func f) {
3         postorder_tree_walk(root, f);
4     }
5
6     template <typename Func>
7     void postorder_tree_walk(NodeType* x, Func f) {
8         if (x == 0) return;
9         postorder_tree_walk(x->left, f);
10        postorder_tree_walk(x->right, f);
11        f(x);
12    }

```

Pero para el recorrido de estructuras de datos con frecuencia es mucho mejor emplear el patrón iterador. En ese caso puede reutilizarse cualquier algoritmo de la STL.

Incluir el orden de recorrido en el iterador implica almacenar el estado necesario. Las funciones de recorrido anteriormente descritas son recursivas, por lo que el estado se almacenaba en los sucesivos marcos de pila correspondientes a cada llamada anidada. Por tanto necesitamos un contenedor con la ruta completa desde la raíz hasta el nodo actual. También tendremos que almacenar el estado de recorrido de dicho nodo, puesto que el mismo nodo es visitado en tres ocasiones, una para el subárbol izquierdo, otra para el propio nodo, y otra para el subárbol derecho.

Listado 2.68: Iterador en orden en un árbol de búsqueda binaria

```

1 class inorder_iterator : public std::iterator<std::
    input_iterator_tag,
2                                     Node<KeyType>,
3                                     ptrdiff_t,
4                                     const Node<KeyType>*,
5                                     const Node<KeyType>&>
    {
6     typedef Node<KeyType> NodeType;
7     enum IteratorState { VisitingLeft, VisitingNode, VisitingRight
    };
8     std::vector<std::pair<NodeType*, IteratorState> > _current;
9
10 public:
11     inorder_iterator(NodeType* x) {
12         _current.push_back(std::make_pair(x, VisitingLeft));
13         goToNextNode();
14     }
15
16     const NodeType& operator*() const {
17         return *_current.back().first;
18     }
19
20     const NodeType* operator->() const {
21         return _current.back().first;
22     }
23
24     bool equal(inorder_iterator<KeyType> const& rhs) const {
25         return *this == rhs;
26     }
27
28     inorder_iterator<KeyType>& operator++() {
29         goToNextNode();
30     }
31
32     inorder_iterator<KeyType> operator++(int) {
33         inorder_iterator<KeyType> ret(*this);
34         goToNextNode();
35         return ret;
36     }
37
38 private:
39     void goToNextNode();
40 };
41
42 template<typename KeyType>
43 inline bool operator==(inorder_iterator<KeyType> const& lhs,
44                       inorder_iterator<KeyType> const& rhs) {
45     return lhs.equal(rhs);
46 }

```

En el caso del iterador *en orden* la función de recorrido sería similar a la siguiente:

Listado 2.69: Función para obtener el siguiente nodo en un iterador en orden.

```

1 void
2 inorder_iterator<KeyType>::goToNextNode()
3 {
4     if (_current.empty()) return;
5 }

```

```

6     std::pair<NodeType*, IteratorState>& last = _current.back();
7
8     if (last.second == VisitingLeft) {
9         NodeType* l = last.first->left;
10        if (l == 0) last.second = VisitingNode;
11        else {
12            _current.push_back(std::make_pair(l, VisitingLeft));
13            goToNextNode();
14        }
15    }
16    else if (last.second == VisitingNode) {
17        NodeType* r = last.first->right;
18        if (r == 0) _current.pop_back();
19        else {
20            last.second = VisitingRight;
21            _current.push_back(std::make_pair(r, VisitingLeft));
22        }
23        goToNextNode();
24    }
25    else if (last.second == VisitingRight) {
26        _current.pop_back();
27        goToNextNode();
28    }
29 }

```

Se propone como ejercicio la definición de iteradores para el recorrido en preorden y postorden.

2.3.3. *Quadtree y octree*

Los árboles binarios particionan de forma eficiente un espacio de claves de una sola dimensión. Pero con pequeñas extensiones es posible particionar espacios de dos y tres dimensiones. Es decir, pueden ser usados para indexar el espacio de forma eficiente.

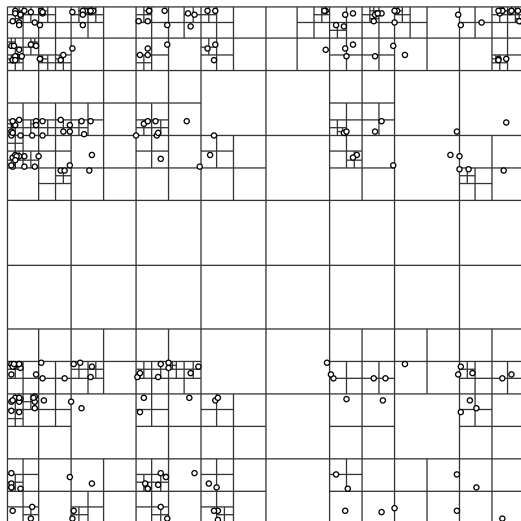


Figura 2.10: Ejemplo de *quadtree* de puntos. Fuente: Wikipedia.

Los *quadtrees* y los *octrees* son la extensión natural de los árboles binarios de prefijos (*tries*) para dos y tres dimensiones respectivamente. Un *quadtree* es un árbol cuyos nodos tienen cuatro subárboles correspondientes a los cuatro cuadrantes de un espacio bidimensional. Los nodos de los *octrees* tienen ocho subárboles correspondientes a los ocho octantes de un espacio tridimensional.

La implementación y el funcionamiento es análogo al de un árbol prefijo utilizado para indexar los puntos de un segmento. Adicionalmente, también se emplean para indexar segmentos y polígonos.

Cuando se utilizan para indexar segmentos o polígonos puede ocurrir que un mismo segmento cruce el límite de un cuadrante o un octante. En ese caso existen dos posibles soluciones:

- Hacer un recortado (*clipping*) del polígono dentro de los límites del cuadrante u octante.
- Poner el polígono en todos los cuadrantes u octantes con los que intersecta.

En este último caso es preciso disponer de alguna bandera asociada a los polígonos para no recorrerlos más veces de las necesarias.

Simon Perreault distribuye una implementación sencilla y eficiente de *octrees* en C++¹⁰. Simplificando un poco esta implementación los nodos son representados de esta forma:

Listado 2.70: Representación de nodos en un *octree*.

```

1 enum NodeType { BranchNode, LeafNode };
2
3 class Node {
4 public:
5     NodeType type() const;
6
7 private:
8     NodeType type_ : 2;
9 };
10
11 class Branch : public Node {
12 public:
13     Node& child( int x, int y, int z );
14     Node& child( int index );
15
16 private:
17     Node* children[2][2][2];
18 };
19
20 class Leaf : public Node {
21 public:
22     Leaf( const T& v );
23
24     const T& value() const;
25     T& value();
26     void setValue( const T& v );
27

```

¹⁰En el momento de editar estas notas se distribuye bajo la GPL en <http://nomis80.org/code/octree.html>.

```

28 private:
29     T value_;
30 };

```

Esta representación de árboles diferencia entre nodos hoja y nodos de ramificación. Los valores solo se almacenan en los nodos hoja y éstos no tienen la sobrecarga de los punteros a los ocho subárboles. Por contra, los nodos de ramificación no tienen sobrecarga de valores asociados, puesto que para la inserción de un elemento puede ser necesario añadir un número de nodos de ramificación sin valor alguno.

El uso básico es muy sencillo. El contenido a incluir puede ser cualquier cosa, desde simples valores (color de un punto), pasando por un *voxel* (pixel 3D) hasta polígonos o poliedros. El tipo de contenido puede ser también una referencia a un objeto gráfico. De esta forma se podría incluir el mismo elemento (e.g. un polígono) en múltiples nodos hoja. En el capítulo siguiente veremos cómo la técnica de referencias con contador puede ayudar en casos como éste.

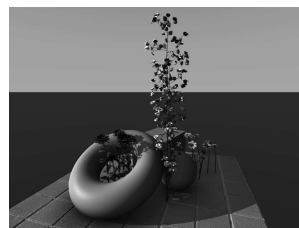


Figura 2.11: Ejemplo del motor de *Sparse Voxel Octree* de nVidia.

Listado 2.71: Representación de nodos en un *octree*.

```

1 #include "octree.h"
2
3 int main()
4 {
5     Octree<double> o(4096);
6     o(1,2,3) = 3.1416;
7     o.erase(1,2,3);
8 }

```

La línea [5] construye un *octree* de 4096 puntos de ancho en cada dimensión. Esta implementación requiere que sea una potencia de dos, siendo posible indexar $4096 \times 4096 \times 4096$ nodos.



La regularidad de los *octree* los hacen especialmente indicados para la paralelización con GPU y recientemente están teniendo cierto resurgimiento con su utilización en el renderizado de escenas con *raycasting* o incluso *raytracing* en una técnica denominada *Sparse Voxel Octree*.

La propia nVidia¹¹ ha desarrollado una biblioteca que implementa esta técnica de *sparse voxel octree* sobre GPU con CUDA, y se distribuye bajo licencia Apache. Otra excelente fuente de información es la tesis de Cyril Crassin de INRIA¹² que explica los fundamentos teóricos de GigaVoxels, también basada en *octrees*.

¹¹Ver <http://research.nvidia.com/publication/efficient-sparse-voxel-octrees>

¹²Disponible en línea en http://maverick.inria.fr/Membres/Cyril.Crassin/thesis/CCrassinThesis_EN_Web.pdf

2.4. Patrones de diseño avanzados

En el módulo 1 ya se expusieron un buen número de patrones. En esta sección completaremos la colección con algunos patrones muy utilizados en todo tipo de aplicaciones.

2.4.1. Forma canónica ortodoxa

Veamos un ejemplo de mal uso de C++ que se ve frecuentemente en programas reales:

Listado 2.72: Ejemplo de uso incorrecto de C++.

```

1 #include <vector>
2
3 struct A {
4     A() : a(new char[3000]) {}
5     ~A() { delete [] a; }
6     char* a;
7 };
8
9 int main() {
10     A var;
11     std::vector<A> v;
12     v.push_back(var);
13     return 0;
14 }
```

Si compilamos y ejecutamos este ejemplo nos llevaremos una desagradable sorpresa.

```

$ g++ bad.cc -o bad
$ ./bad
*** glibc detected *** ./bad: double free or corruption (!prev): 0
    x0000000025de010 ***
===== Backtrace: =====
...

```

¿Qué es lo que ha pasado? ¿No estamos reservando memoria en el constructor y liberando en el destructor? ¿Cómo es posible que haya corrupción de memoria? La solución al enigma es lo que no se ve en el código. Si no lo define el usuario el compilador de C++ añade automáticamente un constructor de copia que implementa la estrategia más simple, copia de todos los miembros. En particular cuando llamamos a `push_back()` creamos una copia de `var`. Esa copia recibe a su vez una copia del miembro `var.a` que es un puntero a memoria ya reservada. Cuando se termina `main()` se llama al destructor de `var` y del `vector`. Al destruir el vector se destruyen todos los elementos. En particular se destruye la copia de `var`, que a su vez libera la memoria apuntada por su miembro `a`, que apunta a la misma memoria que ya había liberado el destructor de `var`.

Antes de avanzar más en esta sección conviene formalizar un poco la estructura que debe tener una clase en C++ para no tener sorpresas.

Básicamente se trata de especificar todo lo que debe implementar una clase para poder ser usada como un tipo cualquiera:

- Pasarlo como parámetro por valor o como resultado de una función.
- Crear *arrays* y contenedores de la STL.
- Usar algoritmos de la STL sobre estos contenedores.

Para que no aparezcan sorpresas una clase no trivial debe tener como mínimo:

1. *Constructor por defecto*. Sin él sería imposible instanciar *arrays* y no funcionarían los contenedores de la STL.
2. *Constructor de copia*. Sin él no podríamos pasar argumentos por valor, ni devolverlo como resultado de una función.
3. *Operador de asignación*. Sin él no funcionaría la mayoría de los algoritmos sobre contenedores.
4. *Destructor*. Es necesario para liberar la memoria dinámica reservada. El destructor por defecto puede valer si no hay reserva explícita.

A este conjunto de reglas se le llama normalmente forma canónica ortodoxa (*orthodox canonical form*).

Además, si la clase tiene alguna función virtual, el destructor debe ser virtual. Esto es así porque si alguna función virtual es sobrecargada en clases derivadas podrían reservar memoria dinámica que habría que liberar en el destructor. Si el destructor no es virtual no se podría garantizar que se llama. Por ejemplo, porque la instancia está siendo usada a través de un puntero a la clase base.

2.4.2. Smart pointers

Los punteros inteligentes (*smart pointers*) son tipos de datos que simplifican de alguna manera la gestión de la memoria dinámica. Facilitan la gestión del ciclo de vida de las variables dinámicas para evitar los problemas frecuentemente asociados a los punteros, especialmente la liberación de la memoria.

La biblioteca estándar de C++ incorpora una plantilla denominada `auto_ptr`. Su objetivo es envolver un puntero normal de tal forma que la destrucción del puntero lleve consigo también la destrucción del objeto apuntado. Por lo demás, un `auto_ptr` se comporta como si se tratara del propio puntero.

Por ejemplo, es frecuente encontrar código como el que sigue:

Listado 2.73: Ejemplo de uso inseguro de punteros.

```
1   T* p = new T();
2
3   // cuerpo de la función
4
5   delete p;
```

Este fragmento tiene dos problemas:

- Es relativamente fácil olvidar llamar a `delete`. Conforme evoluciona el código pueden aparecer puntos de retorno que no invocan al destructor.
- En esta secuencia no es posible garantizar que el flujo del programa será secuencial. Es perfectamente posible que en medio del código de la función se eleve una excepción. En ese caso no se ejecutará el `delete`. Por supuesto siempre es posible utilizar construcciones `try/catch` pero el código cada vez se haría menos legible.

Bjarne Stroustrup inventó una técnica de aplicación general para resolver este tipo de problemas. Se llama *resource acquisition is initialization* (RAII) y básicamente consiste en encapsular las operaciones de adquisición de recursos y liberación de recursos en el constructor y destructor de una clase normal. Esto es precisamente lo que hace `auto_ptr` con respecto a la reserva de memoria dinámica. El mismo código del fragmento anterior puede reescribirse de forma segura así:

Listado 2.74: Ejemplo de uso seguro de punteros.

```
1   auto_ptr<T> p = new T();
2
3   // cuerpo de la función
```

No importa el camino que siga el programa, aunque se eleve una excepción. En el momento en que se abandone el bloque en el que se ha declarado el `auto_ptr` se invocará a su destructor, que a su vez invocará `delete`.

Como puede verse hemos ligado el tiempo de vida del objeto construido en memoria dinámica al tiempo de vida del `auto_ptr`, que suele ser una variable automática o un miembro de clase. Se dice que el `auto_ptr` posee al objeto dinámico. Pero puede ceder su posesión simplemente con una asignación o una copia a otro `auto_ptr`.

Listado 2.75: Cesión de la posesión del objeto dinámico.

```
1 auto_ptr<T> q(p);
2 auto_ptr<T> r;
3 p->f(); // error (NULL ref)
4 q->f(); // ok
5 r = q;
6 q->f(); // error (NULL ref)
7 r->f(); // ok
```

Es decir, `auto_ptr` garantiza que solo hay un objeto que posee el objeto dinámico. También permite desligar el objeto dinámico del `auto_ptr` para volver a gestionar la memoria de forma explícita.

Listado 2.76: Recuperación de la propiedad del objeto dinámico.

```
1 T* s = r.release();
2 delete s;
```

Nunca se deben usar `auto_ptr` en contenedores estándar, porque los contenedores de la STL asumen una semántica de copia incompatible con la del `auto_ptr`. La copia de un `auto_ptr` no genera dos objetos equivalentes.

Esta limitación, que no es detectada en tiempo de compilación, es una de las motivaciones de un completo rediseño de esta funcionalidad para el estándar C++ de 2011. Aún sigue soportando `auto_ptr` pero se desaconseja su uso en favor de `unique_ptr`. El nombre deriva de que, al igual que `auto_ptr`, garantiza que solo un `unique_ptr` puede estar apuntando a un mismo recurso. Sin embargo, a diferencia de `auto_ptr` no es posible copiarlos. Sin embargo existe la posibilidad de transferencia de propiedad entre `unique_ptr` utilizando la nueva semántica de movimiento del estándar C++11.

Listado 2.77: Ejemplo de uso de `unique_ptr`.

```
1 unique_ptr<T> p(new T());
2 unique_ptr<T> q;
3 q = p; // error (no copiable)
4 q = std::move(p);
```

La plantilla `unique_ptr` no tiene un constructor de copia, pero sí cuenta con un constructor de movimiento. Este nuevo constructor se aplica cuando el parámetro es un *rvalue*, es decir, una expresión del tipo de las que aparecen en el lado derecho de una asignación (de ahí el nombre, *right value*) o un valor de retorno de función, o la copia temporal de un parámetro pasado por copia (ahora se puede pasar también por movimiento). Este tipo de expresiones se caracterizan en C++ porque generan un *temporary*, una variable temporal.

La semántica de movimiento resuelve el problema de la generación inconsciente de multitud de variables temporales y la separación entre constructor de copia y constructor de movimiento permite detectar en tiempo de compilación los problemas semánticos. La copia siempre debería generar dos objetos equivalentes.

Tanto `auto_ptr` como `unique_ptr` proporcionan un método sencillo para gestionar variables en memoria dinámica casi como si se tratara de variables automáticas. Por ejemplo:

Listado 2.78: Función que reserva memoria dinámica y traspasa la propiedad al llamador. También funcionaría correctamente con `auto_ptr`.

```
1 unique_ptr<T> f() {
2     unique_ptr<T> p(new T());
3     // ...
4     return p;
5 }
```

La función `f()` devuelve memoria dinámica. Con simples punteros eso implicaba que el llamante se hacía cargo de su destrucción, de controlar su ciclo de vida. Con esta construcción ya no es necesario. Si el llamante ignora el valor de retorno éste se libera automáticamente al destruir la variable temporal correspondiente al valor de retorno. Si en cambio el llamante asigna el valor de retorno a otra variable `unique_ptr` entonces está asumiendo la propiedad y se liberará automáticamente cuando el nuevo `unique_ptr` sea destruido.



Las nuevas características de la biblioteca estándar para la gestión del ciclo de vida de la memoria dinámica están ya disponibles en los compiladores libres GCC y clang. Tan solo hay que utilizar la opción de compilación `-stdc++0x`.

Tanto con `auto_ptr` como con `unique_ptr` se persigue que la gestión de memoria dinámica sea análoga a la de las variables automáticas con semántica de copia. Sin embargo no aprovechan la posibilidad de que el mismo contenido de memoria sea utilizado desde varias variables. Es decir, para que la semántica de copia sea la natural en los punteros, que se generen dos objetos equivalentes, pero sin copiar la memoria dinámica. Para ese caso el único soporte que ofrecía C++ hasta ahora eran los punteros y las referencias. Y ya sabemos que ese es un terreno pantanoso.

La biblioteca estándar de C++11 incorpora dos nuevas plantillas para la gestión del ciclo de vida de la memoria dinámica que ya existían en la biblioteca Boost: `shared_ptr` y `weak_ptr`. Ambos cooperan para disponer de una gestión de memoria muy flexible. La plantilla `shared_ptr` implementa una técnica conocida como *conteo de referencias*.

Cuando se asigna un puntero por primera vez a un `shared_ptr` se inicializa un contador interno a 1. Este contador se almacena en memoria dinámica y es compartido por todos los `shared_ptr` que apunten al mismo objeto. Cuando se asigna este `shared_ptr` a otro `shared_ptr` o se utiliza el constructor de copia, se incrementa el contador interno. Cuando se destruye un `shared_ptr` se decrementa el

contador interno. Y finalmente cuando el contador interno llega a 0, se destruye automáticamente el objeto dinámico.

Listado 2.79: Ejemplos de uso de `shared_ptr`.

```

1  shared_ptr<T> p(new T());
2  shared_ptr<T> q;
3  {
4      q = p;
5      shared_ptr<T> r(p);
6      // ...
7  }
8  // ...

```

En la línea ① se construye un `shared_ptr` que apunta a un objeto dinámico. Esto pone el contador interno de referencias a 1. En la línea ④ se asigna este `shared_ptr` a otro. No se copia el objeto dinámico, sino solo su dirección y la del contador de referencias, que además es automáticamente incrementado (pasa a valer 2). En la línea ⑤ se utiliza el constructor de copia de otro `shared_ptr`, que nuevamente copia solo el puntero y el puntero al contador de referencias, además de incrementar su valor (pasa a valer 3). En la línea ⑦ se destruye automáticamente `r`, con lo que se decrementa el contador de referencias (vuelve a valer 2). Cuando acabe el bloque en el que se han declarado `p` y `q` se destruirán ambas variables, y con ello se decrementará dos veces el contador de referencias. Al llegar a 0 automáticamente se invocará el operador `delete` sobre el objeto dinámico.

El conteo de referencias proporciona una poderosa herramienta para simplificar la programación de aplicaciones con objetos dinámicos. Los `shared_ptr` pueden copiarse o asignarse con total libertad y con una semántica intuitiva. Pueden emplearse en contenedores de la STL y pasarlos por valor libremente como parámetros a función o como valor de retorno de una función. Sin embargo no están totalmente exentos de problemas. Considera el caso en el que `main()` tiene un `shared_ptr` apuntando a una clase `A` y ésta a su vez contiene directa o indirectamente un `shared_ptr` que vuelve a apuntar a `A`. Tendríamos un ciclo de referencias y el contador de referencias con un valor de 2. En caso de que se destruyera el `shared_ptr` inicial seguiríamos teniendo una referencia a `A`, por lo que no se destruirá.

Para romper los ciclos de referencias la biblioteca estándar incluye la plantilla `weak_ptr`. Un `weak_ptr` es otro *smart pointer* a un objeto que se utiliza en estas condiciones:

1. Solo se necesita acceso al objeto si existe.
2. Puede ser borrado por otros en cualquier momento.
3. Debe ser destruido tras su último uso.

Bjarne Stroustrup¹³ pone un ejemplo que tiene mucho que ver con la programación de videojuegos. Consideremos el caso del juego de los

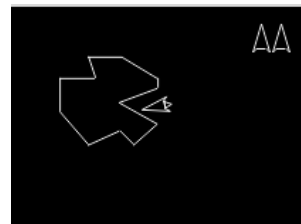


Figura 2.12: El propio creador de C++ pone como ejemplo el videojuego *asteroids* para explicar las extensiones a la biblioteca estándar.

¹³Ver http://www.research.att.com/~bs/C++0xFAQ.html#std-weak_ptr.

asteroides. Todos los asteroides son poseídos por “el juego” pero cada asteroide tiene que seguir los movimientos de los asteroides vecinos para detectar colisiones. Una colisión lleva a la destrucción de uno o más asteroides. Cada asteroide debe almacenar una lista de los asteroides vecinos. Pero el hecho de pertenecer a esa lista no mantiene al asteroide vivo. Por tanto el uso de `shared_ptr` sería inapropiado. Por otro lado un asteroide no debe ser destruido mientras otro asteroide lo examina (para calcular los efectos de una colisión, por ejemplo), pero debe llamarse al destructor en algún momento para liberar los recursos asociados. Necesitamos una lista de asteroides que *podrían* estar vivos y una forma de sujetarlos por un tiempo. Eso es justo lo que hace `weak_ptr`.

Listado 2.80: Esquema de funcionamiento del propietario de los asteroides. Usa `shared_ptr` para representar propiedad.

```

1   vector<shared_ptr<Asteroid>> va(100);
2   for (int i=0; i<va.size(); ++i) {
3       // ... calculate neighbors for new asteroid ...
4       va[i].reset(new Asteroid(weak_ptr<Asteroid>(va[neighbor])))
5       ;
6       launch(i);
7   }
8   // ...

```

El cálculo de colisiones podría tener una pinta similar a esto:

Listado 2.81: Esquema de funcionamiento de la detección de colisiones. Usa `weak_ptr` para representar la relación con los vecinos.

```

1   if (shared_ptr<Asteroid> q = p.lock()) {
2       // ... Asteroid still alive: calculate ...
3   }
4   else {
5       // ... oops: Asteroid already destroyed
6   }

```

Aunque el propietario decidiera terminar el juego y destruir todos los asteroides (simplemente destruyendo los correspondientes `shared_ptr` que representan la relación de propiedad) todo funcionaría con normalidad. Cada asteroide que se encuentra en mitad del cálculo de colisión todavía terminaría correctamente puesto que el método `lock()` proporciona un `shared_ptr` que no puede quedar invalidado.

Por último merece la pena comentar en esta sección un conjunto de reglas para escribir código correcto con *smart pointers*:

- Siempre que aparezca un operador `new` debe ser en un constructor de un *smart pointer*.
- Evitar el uso de *smart pointers* sin nombre (e.g. *temporaries*).

La primera regla impide tener punteros normales coexistiendo con los *smart pointers*. Eso solo puede generar quebraderos de cabeza, puesto que el *smart pointer* no es capaz de trazar los accesos al objeto desde los punteros normales.

La segunda regla garantiza la liberación correcta de la memoria en presencia de excepciones¹⁴. Veamos un ejemplo extraído de la documentación de Boost:

Listado 2.82: Uso de *smart pointers* en presencia de excepciones.

```
1 void f(shared_ptr<int>, int);
2 int g();
3
4 void ok() {
5     shared_ptr<int> p(new int(2));
6     f(p, g());
7 }
8
9 void bad() {
10    f(shared_ptr<int>(new int(2)), g());
11 }
```

Para entender por qué la línea 10 es peligrosa basta saber que el orden de evaluación de los argumentos no está especificado. Podría evaluarse primero el operador `new`, después llamarse a la función `g()`, y finalmente no llamarse nunca al constructor de `shared_ptr` porque `g()` eleva una excepción.



En la mayoría de las bibliotecas de relativa complejidad encontramos algún tipo de *smart pointer*. En Ogre ya hemos visto `Ogre::SharedPtr`, en ZeroC Ice hemos visto `IceUtil::Handle`, en Boost hay una amplia colección de *smart pointers* que incluye `boost::shared_ptr` y `boost::unique_ptr`. Ahora que el nuevo estándar C++ incluye conteo de referencias veremos una progresiva evolución de las bibliotecas para adoptar la versión estándar. Mientras tanto, es muy importante utilizar en cada biblioteca los mecanismos que incluye y no mezclarlos con otras bibliotecas.

2.4.3. Handle-body

Un pequeño pero muy útil patrón de diseño que seguro que ya hemos usado multitud de veces por el mero uso de bibliotecas externas es el denominado *handle-body* o Pimpl (abreviatura de *private implementation*).

Problema

Conforme evoluciona una aplicación la jerarquía de clases aumenta y las relaciones entre ellas también. El hecho de que en C++ todos los *data members* tengan que ser visibles en el archivo de cabecera hace

¹⁴Este caso ha sido descrito en detalle por Herb Sutter en <http://www.gotw.ca/gotw/056.htm>.

que se tengan que incluir archivos que responden realmente a detalles de implementación. Pero lo peor de todo es que acopla excesivamente la implementación de una clase con el uso de dicha clase. Aún cuando no se modifique la interfaz de programación de las clases, solo por el hecho de cambiar el tipo de un miembro privado es preciso recompilar todos los archivos que usan esta clase.

Solución

La forma más sencilla de implementar el patrón consiste en separar en dos clases distintas la interfaz pública de los detalles de implementación. El objeto público carece de cualquier detalle de implementación, pero contiene un miembro privado con un puntero al objeto de implementación.

Implementación

Por ejemplo, esta implementación de Sobeit Void puede encontrarse en *gamedev.net*¹⁵:

Listado 2.83: Ejemplo del patrón Pimpl (archivo de cabecera).

```
1 class MyClassImp;
2
3 class MyClass {
4 public:
5     MyClass();
6     ~MyClass();
7
8     MyClass(const MyClass& rhs );
9     MyClass& operator=(const MyClass& rhs);
10
11     void Public_Method();
12
13 private:
14     MyClassImp *pimpl_;
15 };
```

En el archivo de cabecera no se expone absolutamente nada de la implementación. La clase pública (también llamada *handle*) tan solo tiene los métodos públicos y un puntero a la clase privada (también llamada *body*) de la que solo existe una declaración anticipada. En el archivo de implementación aparece el constructor y el destructor del *handle*, que ya si tiene acceso a la implementación.

Cualquier cambio en la implementación que no afecte a la interfaz pública no requiere recompilar los clientes.

Nótese que la implementación no está completa. No se muestran las implementaciones del constructor de copia y el operador de asignación.

¹⁵En el momento de escribir este texto puede consultarse en http://www.gamedev.net/page/resources/_/technical/general-programming/the-c-pimpl-r1794.

Listado 2.84: Ejemplo del patrón Pimpl (archivo de implementación).

```

1 class MyClassImp {
2 public:
3     void    Private_Method() {}
4
5     int     private_var_;
6 };
7
8 MyClass::MyClass() : pimpl_( new MyClassImp() )
9 {
10 }
11
12 MyClass::~MyClass()
13 {
14     delete pimpl_;
15 }
16
17 void    MyClass::Public_Method()
18 {
19     pimpl_->Private_Method();
20
21     pimpl_->private_var_ = 3;
22 }

```

La semántica de copia y de asignación no corresponde propiamente al patrón *Pimpl*, pero la implementación más sencilla correspondería a igualar los tiempos de vida del *handle* y de la implementación:

Listado 2.85: Ejemplo del patrón Pimpl (constructor de copia y operador de asignación).

```

1 MyClass::MyClass( const MyClass &rhs )
2     : pimpl_( new MyClassImp(*rhs.pimpl_) )
3 {
4 }
5
6 MyClass& MyClass::operator=(const MyClass& rhs )
7 {
8     delete pimpl_;
9     pimpl_ = new MyClassImp(*rhs.pimpl_);
10    return *this;
11 }

```

Sin embargo esto puede implicar hacer muchas operaciones con el *heap* incluso en casos en los que no se hace nada con los objetos. Una optimización sencilla consiste en retrasar la construcción del objeto implementación hasta el momento en que se vaya a acceder.

Un buen compromiso entre automatización de la gestión de memoria y flexibilidad en la implementación de este patrón es la plantilla `auto_ptr` (o `unique_ptr` para C++11) de la biblioteca estándar de C++. La implementación del patrón *Pimpl* puede simplificarse aún más como recomienda Herb Sutter¹⁶:

¹⁶Por ejemplo, en http://www.gotw.ca/publications/using_auto_ptr_effectively.htm.

Listado 2.86: Ejemplo mejorado del patrón Pimpl (archivo de cabecera).

```

1 class C {
2 public:
3     C();
4     /*...*/
5 private:
6     class CImpl;
7     auto_ptr<CImpl> pimpl_;
8 };

```

La diferencia clave es la declaración del puntero a la implementación como un `auto_ptr` en la línea 7. La declaración anticipada de la clase implementación se ha metido también en la parte privada del *handle* para mejorar la ocultación.

Listado 2.87: Ejemplo mejorado del patrón Pimpl (archivo de implementación).

```

1 class C::CImpl { /*...*/ };
2
3 C::C() : pimpl_( new CImpl ) { }

```

Ahora no es necesario incluir un destructor explícitamente porque el destructor por defecto llamará a los destructores de los miembros, en particular de `pimpl_`. Y el destructor de un `auto_ptr` llama automáticamente al operador `delete` con el puntero interno.

Consideraciones

- Este patrón puede reducir drásticamente los tiempos de compilación cuando la cantidad de código es abundante. TrollTech utiliza extensivamente una variante de este patrón (*d-pointer*) en su biblioteca Qt.
- La indirección adicional implica una pequeña pérdida de rendimiento.

2.4.4. Command

El patrón *command* (se traduciría como *orden* en castellano) se utiliza frecuentemente en interfaces gráficas para el manejo de las órdenes del usuario. Consiste en encapsular las peticiones en objetos que permiten desacoplar la emisión de la orden de la recepción, tanto desde el punto de vista lógico como temporal.

Problema

Existe un gran número de situaciones en las que la sincronía inherente a la invocación directa a métodos resulta poco conveniente:

- La invocación directa solamente involucra a emisor y receptor de la orden, por lo que resulta complicado trazar la actividad del sistema en otros componentes (barras de progreso, capacidad de deshacer las órdenes ejecutadas, ayuda contextual, etc.).
- En algunas ocasiones es necesario un modelo de ejecución transaccional, o con limitaciones de orden. Así, por ejemplo si se ejecuta una acción también deben ejecutarse todas las acciones relacionadas. Y si no se ejecuta una acción deben deshacerse todas las relacionadas. Las acciones sobre un mundo virtual (e.g. un MMORPG) deben garantizar la ejecución en orden causal para todos los jugadores (la causa precede al efecto).
- En ocasiones conviene grabar y reproducir una secuencia de órdenes (e.g para la implementación de macros o simplemente para la prueba del juego).
- Muchas acciones conllevan la interacción con el usuario en forma de *wizards* o cuadros de diálogo para configurar la acción. El patrón *command* permite que el objeto orden sea creado en el momento de mostrar el *wizard*, que el usuario configure el objeto mediante la interacción con el *wizard*, y finalmente, al cerrar el *wizard* se desencadena el proceso de emisión del mensaje. De esta forma la orden no necesita nada de código de interfaz de usuario.
- La mayoría de los juegos actuales son programas multi-hilo. Las órdenes pueden ser generadas desde multitud de hilos, y el procesamiento de éstas puede corresponder a otro conjunto de hilos diferente. El patrón *command* proporciona un método sencillo para desacoplar productores y consumidores de órdenes.
- En los juegos en red necesitamos ejecutar órdenes en todos los ordenadores participantes. El patrón *command* facilita la serialización de las órdenes sin más que serializar los objetos que las representan.
- Muchos juegos añaden algún tipo de consola para interactuar directamente con el motor empleando un intérprete de órdenes. El patrón *command* permite sintetizar órdenes en el juego como si se hubieran producido en el propio juego, lo que facilita enormemente la prueba y depuración.

Solución

La figura 2.13 muestra un diagrama de clases con las entidades involucradas. El cliente es el que crea los objeto *command* concretos y los asocia con el receptor de la acción. Posteriormente, y de forma totalmente desacoplada, un invocador llamará al método `execute()` de cada objeto orden creado. Los objetos *command* concretos implementan el método `execute()`, normalmente delegando total o parcialmente sobre el receptor de la acción.



Figura 2.14: Las acciones de los personajes de un juego son perfectas para el patrón *command*.

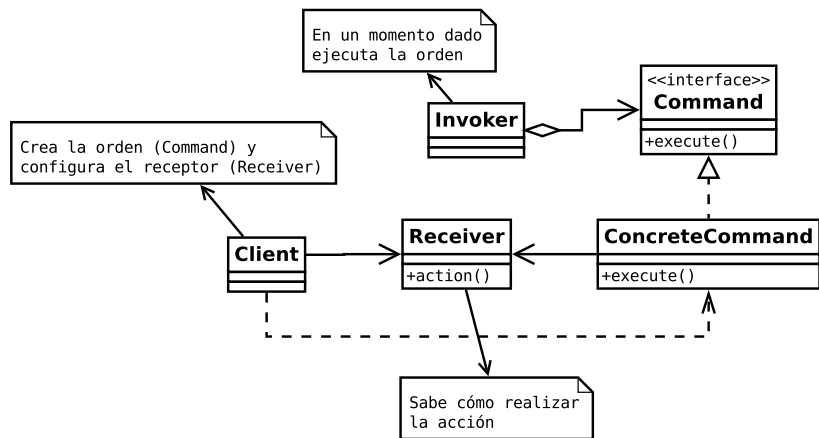


Figura 2.13: Estructura del patrón *command*.

Un ejemplo de aplicación en un videojuego podría ser el que se muestra en la figura 2.15.

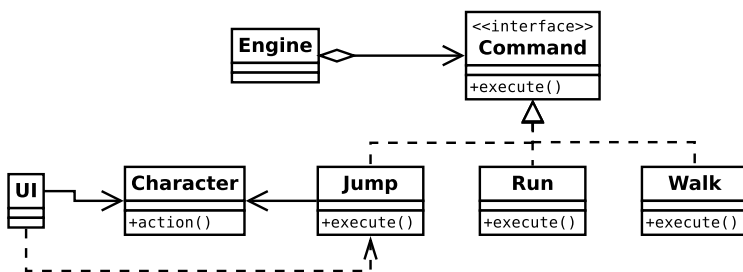


Figura 2.15: Ejemplo de aplicación del patrón *command*.

El interfaz de usuario crea las órdenes a realizar por el personaje o los personajes que están siendo controlados, así como la asociación con su personaje. Estas acciones se van procesando por el motor del juego, posiblemente en paralelo.

Implementación

En términos generales el patrón *command* permite descargar más o menos inteligencia sobre el objeto *ConcreteCommand*. Se juega entre los dos posibles extremos.

- El objeto *ConcreteCommand* no realiza ninguna función por sí mismo, sino que delega todas las acciones en el objeto *Receiver*. A este tipo de órdenes se les llama *forwarding commands*.
- El objeto *ConcreteCommand* implementa absolutamente todo, sin delegar nada en absoluto al objeto *Receiver*.

Entre estos dos extremos se encuentran las órdenes que realizan algunas funciones pero delegan otras en el receptor. En general a todo este tipo de órdenes se les denomina *active commands*.

Desde el punto de vista de la implementación hay poco que podamos añadir a una orden activa. Tienen código de aplicación específico que hay que añadir en el método `execute()`.

Sin embargo, los *forwarding commands* actúan en cierta forma como si se tratara de punteros a función. El `Invoker` invoca el método `execute()` del objeto orden y éste a su vez ejecuta un método del objeto `Receiver` al que está asociado. En [6] se describe una técnica interesante para este fin, los *generalized functors* o adaptadores polimórficos para objetos función. Se trata de una plantilla que encapsula cualquier objeto, cualquier método de ese objeto, y cualquier conjunto de argumentos para dicho método. Su ejecución se traduce en la invocación del método sobre el objeto con los argumentos almacenados. Este tipo de *functors* permiten reducir sensiblemente el trabajo que implicaría una jerarquía de órdenes concretas. Boost implementa una técnica similar en la plantilla `function`, que ha sido incorporada al nuevo estándar de C++ (en la cabecera `functional`). Por ejemplo:

Listado 2.88: Ejemplo de uso de *generalized functors*.

```

1  #include <functional>
2
3  using namespace std;
4
5  int f1(const char* s) { return 0; }
6
7  struct f2 {
8      int operator() (const char* s) { return 0; }
9  };
10
11 struct A {
12     int fa(const char* s) { return 0; }
13 };
14
15 int
16 main()
17 {
18     function<int (const char*)> f;
19
20     f = f1; f("test1");
21     f = f2(); f("test2");
22     A a;
23     auto f3 = bind1st(mem_fun(&A::fa), &a);
24     f = f3; f("test3");
25 }

```

La plantilla `function` se instancia simplemente indicando la signatura de las llamadas que encapsula. A partir de ahí se puede asignar cualquier tipo de objeto que cumpla la signatura, incluyendo funciones normales, métodos o *functors* de la STL, *functors* implementados a mano, etc.

Consideraciones

- El patrón *command* desacopla el objeto que invoca la operación del objeto que sabe cómo se realiza.
- Al contrario que la invocación directa, las órdenes son objetos normales. Pueden ser manipulados y extendidos como cualquier otro objeto.
- Las órdenes pueden ser agregadas utilizando el patrón *composite*.
- Las órdenes pueden incluir transacciones para garantizar la consistencia sin ningún tipo de precaución adicional por parte del cliente. Es el objeto *Invoker* el que debe reintentar la ejecución de órdenes que han abortado por un interbloqueo.
- Si las órdenes a realizar consisten en invocar directamente un método o una función se puede utilizar la técnica de *generalized functors* para reducir el código necesario sin necesidad de implementar una jerarquía de órdenes.

2.4.5. Curiously recurring template pattern

Este patrón fue inicialmente descrito y bautizado por James O. Coplien en [11]. Se trata de un patrón que ya se utilizaba años antes, desde los primeros tiempos de las plantillas de C++.

Problema

El patrón CRT pretende extraer funcionalidad común a varias clases, pero que requieren especialización parcial para cada una de ellas.

Solución

La solución pasa por una interesante recurrencia.

Listado 2.89: Estructura básica del patrón CRT.

```
1 template<typename T> class Base;  
2  
3 class Derived: public Base<Derived> {  
4     // ...  
5 };
```

La clase derivada hereda de una plantilla instanciada para ella misma. La clase base cuenta en su implementación con un tipo que deriva de ella misma. Por tanto la propia clase base puede llamar a funciones especializadas en la clase derivada.

Implementación

Se han propuesto multitud de casos donde puede aplicarse este patrón. Nosotros destacaremos en primer lugar su uso para implementar visitantes alternativos a los ya vistos en el módulo 1.

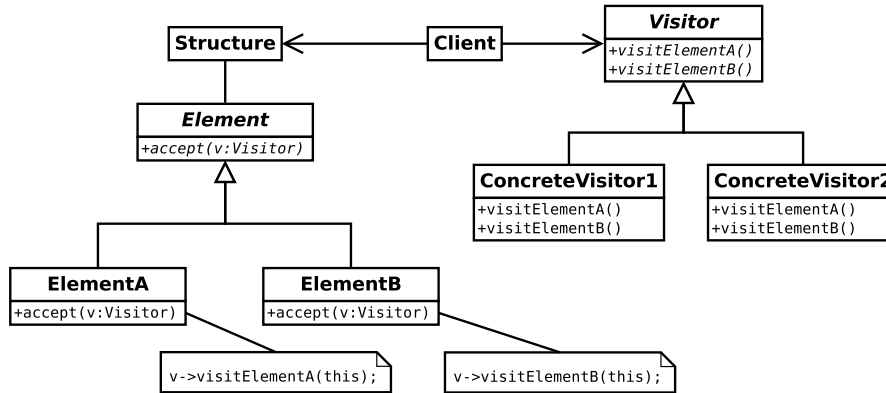


Figura 2.16: Diagrama de clases del patrón Visitor

Recordaremos brevemente la estructura del patrón visitante tal y como se contó en el módulo 1. Examinando la figura 2.16 podemos ver que:

- La clase base *Visitor* (y por tanto todas sus clases derivadas) es tremendamente dependiente de la jerarquía de objetos visitables de la izquierda. Si se implementa un nuevo tipo de elemento *ElementC* (una nueva subclase de *Element*) tendremos que añadir un nuevo método `visitElementB()` en la clase *Visitor* y con ello tendremos que reescribir todos y cada uno de las subclases de *Visitor*. Cada clase visitable tiene un método específico de visita.
- La jerarquía de elementos visitables no puede ser una estructura arbitraria, debe estar compuesta por subclases de la clase *Element* e implementar el método `accept()`.
- Si se requiere cambiar la estrategia de visita. Por ejemplo, unificar el método de visita de dos tipos de elementos, es preciso cambiar la jerarquía de objetos visitables.
- El orden de visita de los elementos agregados está marcado por la implementación concreta de las funciones `accept()` o `visitX()`. O bien se introduce el orden de recorrido en los métodos `accept()` de forma que no es fácil cambiarlo, o bien se programa a medida en los métodos `visitX()` concretos. No es fácil definir un orden de recorrido de elementos (en orden, en preorden, en postorden) común para todos las subclases de *Visitor*.

En general, se considera que el patrón *visitor* introduce un excesivo acoplamiento en el código y resulta tremendamente invasivo. Sin embargo, el patrón CRT permite aliviar gran parte de los problemas.

La jerarquía de visitables implementa el método `accept()` exclusivamente para que puedan elegir el método `visit()` correcto de la clase derivada de `Visitor`. Por eso se le llama también despachado doble. El despachado de la función virtual `accept()` selecciona la subclase de `Element` concreta y a su vez ese elemento concreto desencadena el despachado de `visitX()` que selecciona la subclase de `Visitor` concreta. El segundo despachado es esencial para cualquier recorrido. Sin embargo el primer despachado no siempre es necesario si conocemos de alguna manera el tipo a visitar. Por ejemplo, en el ejemplo del patrón *visitor* mostrado en el módulo 1 el tipo de objetos es completamente fijo. Sabemos que hay un objeto `Scene` que contiene un número variable de objetos `ObjectScene`. Otra forma de realizar este primer despachado podría ser utilizando RTTI u otro mecanismo de introspección.

En este caso en que no sea necesario el primer despachado virtual se puede lograr de una manera mucho más eficiente sin ni siquiera usar funciones virtuales, gracias al patrón CRT. Por ejemplo, el mismo ejemplo del módulo 1 quedaría así:

Listado 2.90: Visitante genérico usando el patrón CRT.

```

1  struct ObjectScene {
2      string name;
3      Point position;
4      int weight;
5  };
6
7  struct Scene {
8      template <typename Derived> friend class Visitor;
9      string name;
10     vector<ObjectScene> objects;
11 };
12
13 template <typename Derived>
14 class Visitor {
15 public:
16     void traverseObject(ObjectScene* o) {
17         getDerived().visitObject(o);
18     }
19     void traverseScene(Scene* s) {
20         getDerived().visitScene(s);
21         for (auto o : s->objects)
22             traverseObject(o);
23     }
24     void visitObject(ObjectScene* o) {}
25     void visitScene(Scene* s) {}
26 private:
27     Derived& getDerived() {
28         return *static_cast<Derived*>(this);
29     }
30 };
31
32 class NameVisitor : public Visitor<NameVisitor> {
33     vector<string> _names;
34 public:
35     void visitObject(ObjectScene* o) {
36         _names.push_back(o->name);
37     }
38     void visitScene(Scene* s) {
39         cout << "The scene '" << s->name << "' has the following

```

```

        objects:"
        << endl;
41     for (auto n : _names) cout << n << endl;
42 }
43 };
44
45 class BombVisitor : public Visitor<BombVisitor> {
46     Bomb _bomb;
47 public:
48     BombVisitor(const Bomb& bomb) : _bomb(bomb) {}
49     void visitObject(ObjectScene* o) {
50         Point new_pos = calculateNewPosition(o->position,
51                                             o->weight,
52                                             _bomb.intensity);
53         o->position = new_pos;
54     }
55 };

```

Como puede observarse, ahora no tocamos en absoluto la jerarquía de visitables (no se necesita método `accept`) y no hay ninguna función virtual involucrada. En el `Visitor` distinguimos entre las funciones de recorrido, que son comunes a cualquier otro `Visitor` y las de visita, que se especifican por cada visitante concreto. Su uso es prácticamente igual de sencillo:

Listado 2.91: Utilización del visitante basado en CRT.

```

1     Scene* scene = createScene();
2     NameVisitor nv;
3     tv.traverseScene(scene);
4     // ...
5     // bomb explosion occurs
6     BombVisitor bv(bomb);
7     bv.traverseScene(scene);

```

Pero la utilidad del patrón no se limita a implementar visitantes. Es un mecanismo genérico para implementar *mixins*. En programación orientada a objetos un *mixín* es una clase que proporciona funcionalidad para ser reusada directamente por sus subclases. Es decir, las subclases no especializan al *mixín* sino que simplemente incorporan funcionalidad derivando de él.

Un ejemplo clásico es la implementación automática de operadores a partir de otros. Es muy utilizado en aritmética, pero también utilizable en otros tipos, como el siguiente ejemplo de Eli Bendersky¹⁷:

Listado 2.92: Ejemplo de CRT como *mixín*.

```

1     template <typename Derived>
2     struct Comparisons { };
3
4     template <typename Derived>
5     bool operator==(const Comparisons<Derived>& o1, const Comparisons<
        Derived>& o2)
6     {
7         const Derived& d1 = static_cast<const Derived&>(o1);
8         const Derived& d2 = static_cast<const Derived&>(o2);

```

¹⁷<http://eli.thegreenplace.net/2011/05/17/the-curiously-recurring-template-pattern-in-c/>

```

9
10     return !(d1 < d2) && !(d2 < d1);
11 }
12
13 template <typename Derived>
14 bool operator!=(const Comparisons<Derived>& o1, const Comparisons<
    Derived>& o2)
15 {
16     return !(o1 == o2);
17 }

```

Y con ello podemos definir todos los operadores de golpe sin más que definir *operator <*.

Listado 2.93: Ejemplo de *mixin* con CRT para implementación automática de operadores.

```

1 class Person : public Comparisons<Person> {
2 public:
3     Person(string name_, unsigned age_)
4         : name(name_), age(age_) {}
5
6     friend bool operator<(const Person& p1, const Person& p2);
7 private:
8     string name;
9     unsigned age;
10 };
11
12 bool operator<(const Person& p1, const Person& p2) {
13     return p1.age < p2.age;
14 }

```

Consideraciones

La técnica que explota el patrón CRT es denominada a veces como *polimorfismo estático*, por contraposición al dinámico de las funciones virtuales. La clase base utiliza la implementación correcta de los métodos redefinidos en las clases derivadas porque se le pasa como parámetro de plantilla. Esto es una ventaja y un inconveniente a la vez.

Por un lado la utilización de funciones no virtuales elimina las direcciones y permite que sea lo más eficiente posible. Pero por otro lado no puede inferir el tipo de un objeto a través de un puntero a la clase base. Por ejemplo, si en el caso del visitante hubiera varios tipos derivados de `ObjectScene` y la clase `Scene` almacenara punteros a `ObjectScene`, el método `traverseObject()` no podría determinar qué función de visita debe invocar. La solución estándar en este caso sería emplear RTTI (*run-time type information*) para determinar el tipo de objeto en tiempo de ejecución, pero eso es mucho menos eficiente que las funciones virtuales.

Listado 2.94: Uso de RTTI para especializar la visita de objetos.

```
1 void traverseObject (ObjectScene* o) {
2     Character* c = dynamic_cast<Character*>(o);
3     if (c) {
4         getDerived().visitCharacter(c);
5         return;
6     }
7     Weapon* w = dynamic_cast<Weapon*>(o);
8     if (w) {
9         getDerived().visitCharacter(w);
10        return;
11    }
12 }
```

2.4.6. Acceptor/Connector

Acceptor-Connector es un patrón de diseño propuesto por Douglas C. Schmidt [33] y utilizado extensivamente en ACE (*Adaptive Communications Environment*), su biblioteca de comunicaciones.

La mayoría de los videojuegos actuales necesitan comunicar datos entre jugadores de distintos lugares físicos. En toda comunicación en red intervienen dos ordenadores con roles bien diferenciados. Uno de los ordenadores toma el rol activo en la comunicación y solicita una conexión con el otro. El otro asume un rol pasivo esperando solicitudes de conexión. Una vez establecida la comunicación cualquiera de los ordenadores puede a su vez tomar el rol activo enviando datos o el pasivo, esperando la llegada de datos. Es decir, en toda comunicación aparece una fase de conexión e inicialización del servicio y un intercambio de datos según un patrón de intercambio de mensajes pre-establecido.

El patrón *acceptor-connector* se ocupa de la primera parte de la comunicación. Desacopla el establecimiento de conexión y la inicialización del servicio del procesamiento que se realiza una vez que el servicio está inicializado. Para ello intervienen tres componentes: *acceptors*, *connectors* y manejadores de servicio (*service handlers*). Un *connector* representa el rol activo, y solicita una conexión a un *acceptor*, que representa el rol pasivo. Cuando la conexión se establece ambos crean un manejador de servicio que procesa los datos intercambiados en la conexión.

Problema

El procesamiento de los datos que viajan por la red es en la mayoría de los casos independiente de qué protocolos, interfaces de programación de comunicaciones, o tecnologías específicas se utilicen para transportarlos. El establecimiento de la comunicación es un proceso inherentemente asimétrico (uno inicia la conexión mientras otro espera conexiones) pero una vez establecida la comunicación el transporte de datos es completamente ortogonal.

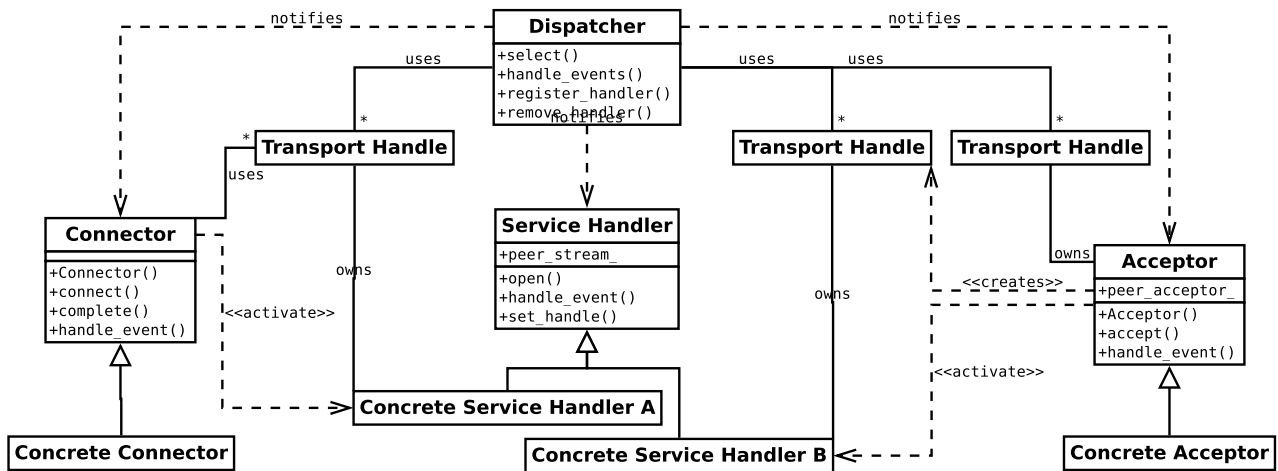


Figura 2.17: Estructura del patrón *acceptor-connector*.

Desde el punto de vista práctico resuelve los siguientes problemas:

- Facilita el cambio de los roles de conexión sin afectar a los roles en el intercambio de datos.
- Facilita la adición de nuevos servicios y protocolos sin afectar al resto de la arquitectura de comunicación.
- En los juegos de red a gran escala (*Massive Multiplayer Online Playing Games*, MMORPG) facilita la reducción de la latencia en el establecimiento de conexión usando mecanismos avanzados del sistema operativo, como conexiones asincrónicas.

Solución

El funcionamiento es como sigue:

- Un *Acceptor* es una factoría que implementa el rol pasivo para establecer conexiones. Ante una conexión crea e inicializa un *Transport Handle* y un *Service Handler* asociados. En su inicialización, un *Acceptor* se asocia a una dirección de transporte (e.g. dirección IP y puerto TCP), y se configura para aceptar conexiones en modo pasivo. Cuando llega una solicitud de conexión realiza tres pasos:
 1. Acepta la conexión creando un *Transport Handle* que encapsula un extremo conectado.
 2. Crea un *Service Handler* que se comunicará directamente con el del otro extremo a través del *Transport Handle* asociado.
 3. Activa el *Service Handler* para terminar la inicialización.

- Un `Connector` es una factoría que implementa el rol activo de la conexión. En la inicialización de la conexión `connect()()` crea un `Transport Handle` que encapsula un extremo conectado con un `Acceptor` remoto, y lo asocia a un `Service Handler` preexistente.

Tanto `Acceptor` como `Connector` pueden tener separadas las funciones de inicialización de la conexión de la función de completado de la conexión (cuando ya se tiene garantías de que el otro extremo ha establecido la conexión). De esta forma es fácil soportar conexiones asíncronas y síncronas de forma completamente transparente.

El `Dispatcher` es responsable de demultiplexar eventos del canal, tales como peticiones de conexión o peticiones de datos. Para el `Acceptor` demultiplexa indicaciones de conexión a través de los `Transport Handles` que encapsulan direcciones de nivel de transporte. Para el `Connector` demultiplexa eventos de establecimiento de conexión que llegan cuando la solicitud de conexión es asíncrona.

El patrón *acceptor-connector* coopera perfectamente con el patrón *reactor*. Tanto el `Transport Handle` asociado al `Acceptor`, como el asociado al `Connector`, e incluso los asociados a los manejadores de servicio pueden ser un manejadores de eventos registrados en el *reactor* del sistema. De esta forma el `Dispatcher` pasa a ser un *reactor* que demultiplexa no solo eventos de red, sino de interfaz de usuario, o eventos del propio juego.

World of Warcraft

WoW es el mayor MMORPG de la actualidad con más de 11.5 millones de suscriptores mensuales.

Implementación

Desde el punto de vista de la implementación, si nos restringimos a TCP y la API sockets el `Acceptor` no es más que una envoltura de la llamada al sistema `accept()`, el `Connector` una envoltura de la llamada al sistema `connect()`, y el `Dispatcher` o `Reactor` una envoltura de la llamada al sistema `select()` o `poll()`.

Una de las más flexibles implementaciones que existen de este patrón es la que ofrece ACE (*Adaptive Communications Environment*¹⁸), biblioteca creada por el inventor del patrón y utilizada en multitud de sistemas de comunicaciones a escala global. Sin embargo, desde el punto de vista didáctico no es muy conveniente utilizarla como ejemplo, porque requiere de un respetable conjunto de conceptos previos.

Otra implementación muy escalable y extremadamente elegante del patrón *acceptor-connector* es la incluida en la biblioteca ZeroC Ice, que ya conocemos. Sin embargo, el objeto de Ice es implementar un *middleware* de comunicaciones basado en el modelo de objetos distribuidos. Por tanto la implementación del patrón es privada, y no se expone a los usuarios. Ya examinaremos este modelo más adelante.

En ACE un servidor TCP mínimo atendiendo conexiones en el puerto 9999 tendría el siguiente aspecto:

¹⁸<http://www.cs.wustl.edu/~schmidt/ACE.html>

Listado 2.95: Ejemplo de uso de patrón *acceptor-connector* (servidor).

```

1 #include <ace/SOCK_Acceptor.h>
2 #include <ace/Acceptor.h>
3 #include <ace/Svc_Handler.h>
4
5 class MySvcHandler : public ACE_Svc_Handler<ACE SOCK_STREAM,
    ACE_MT_SYNCH> {
6     virtual int handle_input (ACE_HANDLE) {
7         char buf[256];
8         int n = peer().recv(buf, sizeof buf);
9         if (n <= 0) return -1;
10        // procesar buf ...
11        return 0;
12    }
13 };
14
15 typedef ACE_Acceptor <MySvcHandler, ACE SOCK_ACCEPTOR> MyAcceptor;
16
17 int main (int argc, const char *argv[]) {
18     ACE_Reactor reactor;
19     MyAcceptor acceptor;
20
21     acceptor.open(ACE_INET_Addr(9999), &reactor);
22     for(;;)
23         reactor.handle_events();
24
25     return 0;
26 }

```

Especializamos la plantilla del `Acceptor` con un `Svc_Handler` que tiene la lógica de intercambio de mensajes. Al instanciar el `Acceptor` le pasamos un `Reactor` para que automáticamente registre los nuevos `Svc_Handler` que crea en las nuevas conexiones.

El lado del cliente es muy similar, salvo que en este caso utilizamos un `Connector`.

Listado 2.96: Ejemplo de uso de patrón *acceptor-connector* (cliente).

```

1 #include <ace/SOCK_Connector.h>
2 #include <ace/Connector.h>
3 #include <ace/Svc_Handler.h>
4
5 class MySvcHandler : public ACE_Svc_Handler<ACE SOCK_STREAM,
    ACE_MT_SYNCH> {
6     virtual int handle_output (ACE_HANDLE) {
7         char buf[]="Hello, World!\n";
8         int n = peer().send(buf, sizeof buf);
9         if (n <= 0) return -1;
10        return 0;
11    }
12 };
13
14 typedef ACE_Connector <MySvcHandler, ACE SOCK_CONNECTOR>
    MyConnector;
15
16 int main (int argc, const char *argv[]) {
17     ACE_Reactor reactor;
18     MyConnector connector;
19     MySvcHandler* psvc = 0;
20

```

```
21     int n = connector.connect(psvc, ACE_INET_Addr(9999, "127.0.0.1")
22         );
23     if (n < 0) return 1;
24     reactor.register_handler(psvc, ACE_Event_Handler::WRITE_MASK);
25     for(;;)
26         reactor.handle_events();
27
28     return 0;
29 }
```

Como puede verse el `Connector` construye un `Svc_Handler` para procesar eventos. Nosotros registramos ese manejador en el reactor para generar mensajes hacia el servidor.

Téngase en cuenta que estos ejemplos son simples en exceso, con el propósito de ilustrar el uso del patrón. En un videojuego habría que tratar los errores adecuadamente y ACE permite también configurar el esquema de concurrencia deseado.

Consideraciones

Este patrón permite manejar de forma uniforme las comunicaciones multi-protocolo en juegos *online*. Además, coopera con el *reactor* de manera que podemos tener una única fuente de eventos en el sistema. Esto es muy interesante desde todos los puntos de vista, porque facilita enormemente la depuración, la síntesis de eventos en el sistema, la grabación de secuencias completas de eventos para su posterior reproducción, etc.

2.5. C++11: Novedades del nuevo estándar

El 12 de Agosto de 2011 la Organización Internacional de Estándares (ISO) aprobó el nuevo estándar de C++, anteriormente conocido como C++0x. Además de añadir funcionalidades nuevas al lenguaje, C++11 también amplía la STL, incluyendo en la misma casi todas las plantillas y clases ya presentes en el TR1.

C++11 es compatible con C++98 (también con la corrección de 2003) y con C. Aparte de esta, las cualidades que se han pretendido conseguir con el nuevo estándar incluyen la mejora de rendimiento, una programación más evolucionada y su accesibilidad para los programadores no-expertos sin privar al lenguaje de su potencia habitual.

En los siguientes apartados se introducirán algunas de las nuevas características que añade el estándar.

2.5.1. Compilando con g++

GCC junto con Clang son los dos compiladores que dan soporte a mayor número de características del nuevo estándar. Concretamente, la versión 4.7 del primero da soporte a casi todas las novedades que se

van a presentar aquí. En el momento de escribir esta documentación, la última versión estable de GCC es la 4.6, pero es posible compilar un versión en desarrollo y disfrutar desde ahora de casi todas las ventajas que brinda el nuevo estándar.

Por el momento, para compilar un programa de C++ usando el nuevo estándar hay que utilizar la opción `-std=c++0x` al compilar. Por ejemplo:

```
g++ -o main main.cc -std=c++0x
```

Si no se utiliza esta opción, GCC compilará usando el estándar C++03.

2.5.2. Cambios en el núcleo del lenguaje

Expresiones constantes

Un compilador de C++ (en concreto g++) es capaz de optimizar ciertas expresiones que serán siempre constantes, por ejemplo:

```
1  int a = 1 + 2;
2  cout << 3.2 - 4.5 << endl;
3
4  int miArray[4 * 2];
```

En este código, el compilador sustituirá las expresiones anteriores por su valor en tiempo de compilación. De este modo, en cualquier buen compilador, el código anterior no generará ninguna suma, resta o producto. Sin embargo, C++03 no permite utilizar funciones que devuelvan constantes (por ejemplo `return 5;`).

C++11 introduce la palabra reservada `constexpr` para brindar la posibilidad de utilizar funciones como expresiones constantes. Anteriormente no era posible puesto que el compilador no tenía ninguna forma de saber que podía aplicar esta optimización. De este modo, es posible escribir algo como lo siguiente:

```
1  constexpr int siete(){ return 7; }
2
3  void miFunc(){
4      char cadena[siete() + 3];
5      cadena[0]='\0';
6      // ...
7  }
```

Una función se podrá declarar como `constexpr` siempre que no devuelva `void` y que termine del modo `return <expresión>`. Dicha expresión tendrá que ser constante una vez que se sustituyan todas las variables y si llama a otras funciones tendrán que estar definidas como `constexpr`.

Es posible declarar variables utilizando `constexpr` que equivale al uso de `const`.

```

1  constexpr int saludJefeNivel = 1337;
2
3  const int saludJefeFinal    = 31337;
4  const int saludJefeEspecial = 3110 + siete();

```

La introducción de esta característica es muy útil con las plantillas. El siguiente código se evaluará en tiempo de compilación y no en tiempo de ejecución, sustituyendo la llamada por el valor devuelto.

```

1  template<typename T> constexpr T max(T a, T b)
2  {
3      return a < b ? b : a;
4  }

```

Inicializador de listas

Antes de la entrada del nuevo estándar, la inicialización de los contenedores de la STL era posible utilizando una zona de memoria con una secuencia de elementos del tipo instanciado. Normalmente se utiliza un array para llevar esto a cabo.

Definiendo una estructura del siguiente modo

```

1  struct miStruct {
2      int    a;
3      float b;
4  };

```

se podría inicializar un vector como sigue (también se incluyen ejemplos con enteros).

```

1  miStruct mS[] = { {0, 0.0}, {0, 0.0}, {0, 0.0} };
2  vector<miStruct> mVS(mS, mS + sizeof(mS)/sizeof(miStruct));
3
4  int mA[] = {1, 1, 1, 2, 3, 4, 1};
5  vector<int> mVI(mA, mA + sizeof(mA)/sizeof(int));
6
7  int mB[] = {1, 2, 3, 4};
8  set<int> mC(mB, mB + sizeof(mB)/sizeof(int));

```

A partir de ahora, es posible utilizar lo que se conoce como el inicializador de listas, que permite realizar inicializaciones de manera mucho más sencilla.

```

1  vector<miStruct> miVS {{0, 0.0}, {0, 0.0}, {0, 0.0}};
2  vector<int>      miVI {1, 1, 1, 2, 3, 4, 1};
3  set<int>        miC  {0, 4, 5, 9};

```

Esto es posible gracias al uso de una nueva sintaxis y del contenedor `std::initializer_list`.

Si se utiliza como parámetro en el constructor de una clase

Listado 2.97: Clase que utiliza un inicializador de listas

```

1 class LODDistancias {
2 public:
3   LODDistancias(initializer_list<int> entrada) :
4     distancias(entrada) {}
5 private:
6   vector<int> distancias;
7 };

```

es posible hacer uso de las llaves para inicializarla:

```

1 LODDistancias lodD {90, 21, 32, 32, 35, 45};

```

Hay que tener en cuenta que este tipo de contenedores se utilizan en tiempo de compilación, que sólo pueden ser construidos estáticamente por el compilador y que no podrán ser modificados en tiempo de ejecución. Aun así, como son un tipo, pueden ser utilizado en cualquier tipo de funciones.

También se pueden utilizar las llaves junto con el operador =, para inicializar o para asignar nuevos valores.

```

1 vector<miStruct> miVS2 = {{0, 0.0}, {0, 0.0}, {0, 0.0}};
2 vector<int>      miVI2 = {1, 1, 1, 2, 3, 4, 1};
3 set<int>        miC2 = {0, 4, 5, 9};
4
5 miVS2 = {{9, 1.2}};

```

Inicialización uniforme

En C++03 no existe una forma uniforme de inicializar los objetos. En el apartado anterior, en la parte compatible con el antiguo estándar, se ha utilizado la inicialización de un *array* utilizando `.` Esto es posible ya que esa estructura es un agregado¹⁹, ya que sólo este tipo de objetos y los *arrays* pueden ser inicializados de esta manera.

Con la aparición de C++11, es posible utilizar las llaves para inicializar cualquier clase o estructura. Por ejemplo, supongamos una clase para representar un vector de tres dimensiones.

```

1 class Vector3D {
2 public:
3   Vector3D(float x, float y, float z):
4     _x(x), _y(y), _z(z) {}
5
6 private:
7   float _x;
8   float _y;

```

¹⁹Un agregado (*aggregate*) es una clase o estructura que no tiene destructor definido por el usuario ni operador de asignación. Tampoco tendrán miembros privados o protegidos que no sean estáticos, ni una clase base o funciones virtuales.

```
9  float _z;  
10  
11  friend Vector3D normalize(const Vector3D& v);  
12  };
```

Es posible iniciar un objeto de tipo `Vector3D` de las dos formas siguientes.

```
1  Vector3D p{0.0, 1.1, -3.4};  
2  
3  Vector3D p1(1.8, 1.4, 2.3);
```

La primera utiliza la nueva inicialización uniforme, la segunda la clásica, invocando el constructor de forma explícita.

En C++11 también es posible utilizar esta inicialización para construir de manera implícita objetos que son devueltos por una función. El compilador utilizará el valor de retorno del prototipo de la función y lo usará junto con los valores proporcionados para construir y devolver un objeto de dicho tipo.

```
1  Vector3D normalize(const Vector3D& v) {  
2      float len = sqrt(v._x*v._x + v._y*v._y + v._z*v._z);  
3  
4      return {v._x/len, v._y/len, v._z/len};  
5  }
```

Esta notación no sustituye a la anterior. Cabe destacar que cuando se utiliza esta sintaxis para inicializar un objeto, el constructor que acepta una lista de inicialización como las presentadas anteriormente tendrá prioridad sobre otros. Debido a esto, algunas veces será necesario invocar directamente al constructor adecuado con la notación antigua.

Esta forma de devolver objetos es compatible con RVO (*return value optimization*), que se verá en optimizaciones. Con lo cual una llamada como la siguiente generará código óptimo y seguramente si ninguna copia.

```
1  Vector3D p2 = normalize(p);
```

Inferencia de tipos

Hasta ahora cada vez que se declaraba una variable en C++ había que especificar de qué tipo era de manera explícita. En C++11 existe la inferencia de tipos. Usando la palabra reservada `auto` en una inicialización en vez del tipo, el compilador deducirá el mismo de manera automática.

En el ejemplo siguiente se ve cómo funciona esta característica, tanto para tipos básicos como para la clase definida en el apartado anterior.

```

1  auto vidaJefe = 500;
2  auto precision = 1.00001;
3
4  Vector3D v(3.0, 2.1, 4.0);
5  auto v2 = normalize(v);

```

Esta nueva característica es especialmente adecuada para simplificar algunas declaraciones complejas. A continuación de muestra la diferencia en la declaración del iterador al recorrer un contenedor.

```

1  for (vector<double>::iterator it = dist.begin();
2      it != dist.end(); ++it)
3      cout << *it << endl ;
4
5  for (auto it = dist.begin(); it != dist.end(); ++it)
6      cout << *it << endl ;

```

Existe otra palabra reservada que se usa de forma similar a `sizeof()`, pero que devuelve el tipo de una variable. Esta palabra es `decltype` y se puede usar para extraer el tipo de una variable y usarlo para la declaración de otra.

```

1  decltype(v2) otro_vector3d = {4.1, 3.0, 1.1};

```

Bucle for basado en rangos

En C++11 se introduce una característica muy útil para recorrer listas de elementos, ya sean *arrays*, lista de inicialización o contenedores con las operaciones `begin()` y `end()`.

```

1  int records[4] = {900, 899, 39, 3};
2  for (int& i: records)
3      cout << i << endl;
4
5  list<float> punteria = {20.0, 10.9};
6  for (float& f: punteria)
7      cout << f << endl;

```

En el ejemplo anterior se utiliza una referencia para evitar la copia y la penalización de rendimiento.

Funciones Lambda

Las funciones lambda son simplemente funciones anónimas. La sintaxis para declarar este tipo de funciones es especial y es posible no declarar el tipo devuelto de manera explícita sino que está definido de forma implícita por `decltype(<expresión_devuelta>)`. Las dos formas posible de declarar y definir estas funciones son las siguientes.

```
[captura] (parámetros) ->tipo_de_retorno{cuerpo}
```

```
[captura] (parámetros) {cuerpo}
```

La primera hace explícito el tipo que se devuelve. De esto modo, las funciones que se muestran a continuación son equivalentes.

```
1 [](int p1, int p2)->int{ return p1+p2; };
2 [](int p1, int p2){ return p1+p2; };
```

Las variables que se utilizan dentro de estas funciones pueden ser capturadas para utilizarse en el exterior. Se pueden capturar por valor o por referencia, dependiendo de la sintaxis dentro de []. Si se utiliza por ejemplo [p1, &p2], p1 será capturada por valor y p2 por referencia. Si se usa [=, &p1], todas las variables serán capturadas por valor (al usar =) excepto p1 que será capturada por referencia. Si se utiliza [&, p2], todas se capturarán por referencia (usando &), excepto p2.

En el siguiente ejemplo, se utiliza una función lambda para sumar la puntuaciones de todos los jugadores, que han sido previamente almacenadas en una lista. Se muestran tres formas de hacerlo.

```
1 list<int> puntos = {330, 300, 200, 3892, 1222};
2 int suma = 0;
3
4 // 1)
5 auto f = [&suma](int& i){suma+=i;};
6 for (int& i: puntos)
7     f(i);
8 // 2)
9 for_each(puntos.begin(), puntos.end(),
10         [&suma](int& i){suma+=i;});
11 // 3)
12 for_each(puntos.begin(), puntos.end(), f);
```

Declaración alternativa de funciones

C++11 introduce una nueva forma de declarar funciones. Su utilidad es permitir declarar los tipos de retorno de funciones *templatzadas* donde este no se puede averiguar a priori.

En el ejemplo siguiente se define una clase y se declaran dos funciones *templatzadas*.

```
1 class K {
2 public:
3     int operator*(const K& k) const {return 2;};
4 };
5
6 template <typename T>
7 T pow2Bad(const T& t){return t*t;};
8
9 template <typename T>
10 auto pow2(const T& t)->decltype(t*t){return t*t;};
```

La primera función no compilará si el tipo que se devuelve al ejecutar la operación es diferente al tipo para el que se invoca. La segunda sí lo hará.

```

1  K kObj;
2  cout << pow2Bad(kObj) << endl; // <- no compila
3  cout << pow2(kObj) << endl;

```

También se puede usar esta nueva sintaxis para funciones no *templizadas*.

```

1  auto getHours()->int{ return _hours;}

```

Mejora en la construcción de objetos: delegación

En C++03 es imposible invocar a un constructor desde otro constructor del mismo objeto. En C++11 sí es posible.

```

1  class playerInfo {
2  public:
3      playerInfo(const string& name) :
4          _name(name) {}
5
6      playerInfo() : playerInfo("default") {}
7
8  private:
9      string _name;
10 };

```

Sobrescritura explícita y declaración final

En C++11 es posible utilizar dos palabras reservadas para añadir funcionalidad e información para el compilador a la declaración de los métodos de una clase.

La palabra reservada `override` proporciona una forma de expresar que el método que se está declarando sobrescribe a otro de una clase base. Esto es útil para expresar explícitamente las intenciones y facilitar la detección de fallos en tiempos de compilación. Así, si se declara un método como usando `override` y no existe uno con el mismo prototipo que este en una base clase, el compilador mostrará un error.

Listado 2.98: Uso de `final` y `override`

```

1  class Base {
2  public:
3      virtual int  getX(){return _x;}
4      virtual bool isValid() final {
5          return true;
6      }
7  private:
8      int _x;
9  };
10
11 class Padre : public Base {
12 public:
13     //Ok, compila.
14     virtual int  getX() override {

```

```

15     return _anotherX;
16 }
17 //Fallo al compilar
18 virtual int getX(int a) override {
19     return _anotherX;
20 };
21 // Fallo ...
22 virtual bool isValid() { return false; }
23 private:
24     int _anotherX;
25 };

```

En el ejemplo anterior también se muestra (líneas (4-6) y (22)) el uso de `final`. Cuando se utiliza en la declaración de un método, indica que ninguna clase que herede de esta podrá sobrescribirlo.

Puntero *null*

Se introduce también un nuevo valor sólo asignable a punteros: `nullptr`. Este valor no se puede asignar a ningún otro tipo. En C++03, se usaba el valor 0 para los punteros *null*, de este modo, se podía asignar el valor de estos punteros a un entero o a un booleano. Con `nullptr` esto ya no es posible, ayudando a prevenir errores y a sobrescribir funciones.

```

1 int* c = nullptr;

```

Cabe destacar que es un tipo compatible con los booleanos, pero que no es compatible con los enteros.

```

1 bool isNull = nullptr;
2 int zero = nullptr; // <- Error

```

Enumeraciones fuertemente tipadas

En las enumeraciones de C++03 no se podía distinguir el tipo de entero utilizado para las mismas. En C++11 sí, y además se brinda la posibilidad de usar una visibilidad más restrictiva, para agrupar la enumeraciones sin tener que anidarlas dentro de clases.

```

1 enum TipoPortal : unsigned char {
2     NORMAL,
3     MIRROR
4 };
5
6 enum class TipoArma : unsigned short {
7     BLANCA,
8     EXPLOSIVA
9 };

```

Para utilizarlo, se hará igual que en C++03, excepto en el segundo caso.


```
1 TipoPortal ptype = NORMAL;
2 TipoArma atype = TipoArma::BLANCA;
```

Además de esto, ahora se permite la declaración anticipada (*forward declaration*) de enumeraciones.

Alias de plantillas

Ya que `typedef` no se puede utilizar con plantillas, C++11 incluye una forma de crear alias para las mismas. Se basa en utilizar `using`.

```
1 template<typename T, typename M>
2 class miTipo;
3
4 template<typename N>
5 using miTipo2 = miTipo<N,N>;
```

También se puede utilizar la nueva sintaxis para realizar las definiciones de tipo que se hacían con `typedef`.

```
1 typedef unsigned int uint;
2 using uint = unsigned int;
```

Uniones sin restricciones

Ahora se permite la creación de uniones con la participación de objetos no no-triviales en las mismas. El siguiente fragmento de código sólo compilará usando el estándar C++11.

```
1 class Vector3D {
2 public:
3   Vector3D(float x, float y, float z) {}
4 };
5
6 union miUnion {
7   int a;
8   float b;
9   Vector3D v;
10};
```

Nuevos literales de cadenas

C++03 no soportaba ningún tipo de codificación Unicode. Sólo se podían utilizar dos tipos de literales: los que estaban entrecomillados ("hola", que se convertían en `arrays` de `const char`, y los entrecomillados con una L delante (L"hola"), que se transformarán en `arrays` de `const wchar_t`.

Se introduce tres nuevos tipos de literales, para UTF-8, UTF-16 y UTF-32, que serán `arrays` de `const char`, `const char16_t` y `const char32_t` respectivamente.

```

1  const char    cad1[] = u8"Cadena UTF-8";
2  const char16_t cad2[] = u"Cadena UTF-16";
3  const char32_t cad3[] = U"Cadena UTF-32";

```

También se permite la construcción de cadenas *raw*, que no interpretarán los caracteres de escape (`\`), ni las propias comillas (`"`). Para definir este tipo de cadenas se usa `R"(literal) `"`. También es posible usar cadenas *raw* con la modificación Unicode.

```

1  string raw(R"(Cadena "RAW" \n%d)");
2
3  const char16_t rcad2[] = uR"(Cadena UTF-16 RAW\n)";
4  const char32_t rcad3[] = UR"(Cadena UTF-32 RAW%d)";

```

Literales creados a medida

C++11 brinda al programador con la capacidad de crear nuevos tipos de literales. Anteriormente los literales estaban preestablecidos, por ejemplo `9` es un literal entero, `9.0` uno *double*, y `9.0f` uno de tipo *float*.

A partir de ahora se pueden crear nuevos literales usando sufijos. Los sufijos podrán ir detrás de números (los que puedan ser representados por `unsigned long long` o `long double`) o detrás de literales de cadena. Estos sufijos corresponden a funciones con el prototipo `retval operator"" _sufijo (unsigned long long)`.

```

1  double operator"" _d (unsigned long long i) {
2      return (double) i;
3  }

```

La función anterior define el sufijo `_d`, que podrá ser usado para crear un *double* usando un número natural como literal.

```

1  auto d = 30_d;

```

Un ejemplo un poco más complejo del uso de este operador se expone a continuación. Sea la siguiente una clase que podría representar un vector de tres dimensiones, incluyendo la operación de suma.

```

1  class Vector3D {
2  public:
3      Vector3D() :
4          x_(0), y_(0), z_(0) {} ;
5
6      Vector3D(float x, float y, float z) :
7          x_(x), y_(y), z_(z) {} ;
8
9      Vector3D operator+(const Vector3D& v) {
10         return Vector3D(x_ + v.x_,
11                         y_ + v.y_,
12                         z_ + v.z_ );
13     }
14 }

```

```

15 private:
16     float x_;
17     float y_;
18     float z_;
19
20     friend Vector3D operator"" _vx(long double x);
21     friend Vector3D operator"" _vy(long double y);
22     friend Vector3D operator"" _vz(long double z);
23 };

```

Se podrían definir los siguientes literales de usuario, por ejemplo para construir vectores ortogonales.

```

1 Vector3D operator"" _vx(long double x) {
2     return Vector3D(x, 0, 0);
3 }
4
5 Vector3D operator"" _vy(long double y) {
6     return Vector3D(0, y, 0);
7 }
8
9 Vector3D operator"" _vz(long double z) {
10    return Vector3D(0, 0, z);
11 }

```

Como se definió la suma, se podría crear un vector con la misma.

```

1     auto v = 1.0_vx + 3.0_vy + 8.1_vz;

```

Para utilizar los sufijos con los literales de cadenas, se muestra el siguiente ejemplo, que representa un jugador, con un nombre.

```

1 class Player {
2 public:
3     Player(string name):
4         name_(name) {}
5
6 private:
7     string name_;
8 };
9
10 Player operator"" _player(const char* name, size_t nChars) {
11     return Player(name);
12 };

```

Se podrá entonces crear un jugador como sigue.

```

1     auto p = "bRue"_player;

```

Aserciones estáticas

Algo muy útil que ya incluía Boost es una aserción estática. Este tipo de aserciones se comprobarán en tiempo de compilación, y será el mismo compilador el que avise de la situación no deseada.

En C++11 se puede usar `static_assert` (expresión-constante, "Mensaje de error") para utilizar estas aserciones en cualquier punto del código.

```

1  template <typename T>
2  bool equal(T a, T b, T epsilon) {
3
4      static_assert( sizeof(T) >= 8, "4 bytes como poco" );
5
6      return (a > b - epsilon || a < b + epsilon);
7  }
8
9  int main(int argc, char *argv[])
10 {
11     equal(8.0, 8.0000001, 0.00001); // OK (double 8 bytes)
12     equal(8.0f, 8.0000001f, 0.00001f); // Error!!
13
14     return 0;
15 }

```

La salida de la compilación será la siguiente:

```

$ g++ -o statica statica.cc -std=c++0x

statica.cc: In instantiation of 'bool equal(T, T, T)
[with T = float]':
statica.cc:17:35:   required from here
statica.cc:9:3: error: static assertion failed:
4 bytes como poco

```

Eliminación y selección por defecto explícita de funciones

En C++11 es posible prohibir el uso de las funciones de una clase, incluyendo los constructores, haciendo uso de la palabra reservada `delete`. Esto es muy útil para evitar que alguien use un constructor no deseado (en C++03 se declaraba como privado para obtener el mismo resultado).

```

1  class NoCopiable {
2  public:
3      NoCopiable(){}
4      NoCopiable(const NoCopiable&) = delete;
5      NoCopiable& operator=(const NoCopiable&) = delete;
6  };

```

También es útil para evitar llamadas implícitas a funciones. En C++03 esto es posible para los constructores, utilizando la palabra reservada `explicit`. En C++11, se pueden evitar la invocación no deseada de funciones usando `delete`.

En el ejemplo siguiente, si no se declara la función que acepta un entero, si se realizase una llamada de tipo `setA(3)`, se realizaría una conversión implícita desde un entero a un *double*. Este tipo de comportamientos no siempre es deseable y puede provocar sorpresas, sobre todo con tipos no-básicos.

```

1 class Ex {
2 public:
3     explicit Ex(double a) :
4         a_(a) {}
5
6     Ex() = default;
7
8     void setA(double a) {
9         a_ = a;
10    }

```

En el mismo ejemplo se usa `default` con uno de los constructores, lo que pide de forma explícita al compilador que él cree uno por defecto.

Constructores de movimiento

Se introduce el concepto de constructor de movimiento, en contraste con el aun necesario constructor de copia. Mientras que es este último se usa para determinar la forma en la que se copian los objetos, el de movimiento determina qué significa mover las propiedades de un objeto a otro (no son dos objetos independientes).

Aunque sea de forma transparente al programador, el compilador genera variables temporales para realizar determinadas operaciones. El constructor de movimiento es una forma de evitar este tipo de variables intermedias (de copia) y así poder optimizar determinadas operaciones (asignaciones normalmente).

Se introduce también el concepto de referencias-*rvalue* (&&). Ya que en esta sección se introducen muchas características, esta en concreto sólo se va a mencionar por encima, puesto que profundizar en ella podría llevar tanto como para el resto juntas.

El constructor de movimiento se declara como el de copia, pero el parámetro de entrada usa &&.

Listado 2.99: Ejemplo de constructor de movimiento

```

1 #include <iostream>
2 #include <string.h>
3
4 using namespace std;
5
6 class Movable {
7 public:
8     Movable() :s("cadena") {
9         cout << "Constructor" << endl;
10    }
11
12    Movable(const Movable& m){
13        // Copia
14        cout << "Copiando" << endl;
15        if (this != &m) {
16            s = m.s;
17        }
18    }
19
20    Movable(const Movable&& m) {

```

```

21     s = std::move(m.s);
22     cout << "Moviendo" << endl;
23 };
24
25 private:
26     string s;
27 };
28
29 Movable getM() {
30     Movable m;
31     return m;
32 }
33
34 int main(int argc, char *argv[])
35 {
36
37     Movable m;           // Constructor
38     Movable n = getM(); // RVO - (Constructor)
39     Movable o (m);      // Copy-Constr.
40
41     Movable p = std::move(m); // Move-constr.
42
43     return 0;
44 }

```

La salida del programa anterior es la siguiente:

```
$ ./move
```

```

Constructor
Constructor
Copiando
Moviendo

```

2.5.3. Cambios en la biblioteca de C++

Una de las adiciones más importantes a la STL es la inclusión de la mayoría del TR1. Así, plantillas como `auto_ptr` (ahora `unique_ptr`), `shared_ptr` y `weak_ptr` forman parte del estándar.

Generación de número aleatorios

C++11 introduce una nueva forma de generar números pseudo-aleatorios. La novedad que se introduce es que el generador se divide en dos partes, el motor y la distribución que se usa.

Los posibles motores a utilizar son: `std::linear_congruential` (generador lineal congruencial), `std::subtract_with_carry` (resta con acarreo) y `std::mersenne_twister`, que se representan con plantillas. Existen definiciones de tipo, para poder usarlas sin configurar cada parámetro de las mismas: `minstd_rand0` y `minstd_rand` (lineales), `mt19937` y `mt19937_64` (mersenne twister), y `ranlux24_base`, `ranlux48_base` y `ranlux24` (resta con acarreo).

Las distribuciones: `uniform_int_distribution`, `bernoulli_distribution`, `geometric_distribution`,

poisson_distribution, binomial_distribution, uniform_real_distribution, exponential_distribution, normal_distribution y gamma_distribution.

En el siguiente ejemplo se muestra un posible uso, sacando la semilla del reloj del sistema en este caso.

```
1 #include <iostream>
2 #include <functional>
3 #include <random>
4 #include <sys/time.h>
5
6 using namespace std;
7
8 int main(int argc, char *argv[])
9 {
10     struct timeval now;
11     gettimeofday(&now, 0);
12
13     minstd_rand motor;
14     motor.seed(now.tv_usec);
15
16     uniform_int_distribution<int> dist(1,6);
17     uniform_int_distribution<int> dist_2(1,50);
18
19     int loto = dist(motor);           // Uso directo
20
21     auto generador = bind(dist, motor); // Bind
22     int valor_dado = generador();     // Uso "bindeado"
23
24     cout << loto << " : " << valor_dado << endl;
25
26     return 0;
27 }
```

Tablas Hash

Se introducen 4 tipos de tablas hash (en GCC sólo se soportan dos a día de hoy). La que se corresponde con el concepto tradicional en la que está representada por `std::unordered_map`. Ya que su uso es similar a `std::map`, simplemente se muestra un ejemplo a continuación. Hay que incluir también la cabecera correspondiente (línea [2]).

```
1 #include <iostream>
2 #include <unordered_map>
3
4 int main(int argc, char *argv[])
5 {
6     std::unordered_map<int, float> miHash;
7     miHash[13] = 1.1;
8     std::cout << miHash[13] << std::endl;
9     return 0;
10 }
```

Expresiones regulares

Una de las características nuevas más interesantes que se han añadido al estándar son las expresiones regulares. Para ello se utiliza un objeto `std::regex` para construir la expresión.

```
1  regex eRE(".*Curso.*");
```

Para almacenar las coincidencias será necesario utilizar un objeto `std::cmatch`.

```
1  cmatch match;
```

Es posible buscar todas las coincidencias dentro de una cadena como se muestra a continuación.

```
1  const char* entrada = "<h1>Curso de experto en videojuegos</h1>";
2
3  // Si hay coincidencias ...
4  if(std::regex_search( entrada, match, eRE)) {
5      size_t n = match.size();
6      // Crear una cadena con cada una de ellas
7      // e imprimirla.
8      for( size_t a = 0; a < n; a++ ) {
9          std::string str(match[a].first, match[a].second);
10         std::cout << str << "\n";
11     }
12 }
```

Y saber si una cadena cumple la expresión regular así:

```
1  if(regex_match(entrada, eRE))
2      cout << entrada << " cumple la regex" << endl;
```

Tuplas

C++11 da soporte a la creación de tuplas que contengan diferentes tipos. Para ello se utiliza la plantilla `std::tuple`. Como se ve en el ejemplo siguiente, para obtener los valores se utiliza la función *templatazada* `std::get()`.

```
1  #include <iostream>
2  #include <tuple>
3
4  using namespace std;
5
6  typedef tuple <string, int, float> tuplaPuntos;
7
8  int main(int argc, char *argv[])
9  {
10     tuplaPuntos p1("Bilbo", 20, 35.0);
11
12     cout << "El jugador " << get<0>(p1)
13         << " ha conseguido " << get<2>(p1)
```

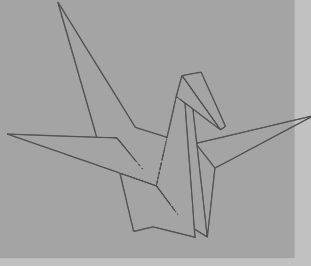
Soporte de **regex**

En el momento de escribir esta documentación, aunque el ejemplo mostrado compila en GCC 4.7, aun no funciona completamente. En la última versión de CLang sí.


```
14         << " puntos en "      << get<1>(p1)
15         << " jugadas"        << endl;
16
17     return 0;
18 }
```

Otras características

Aparte de las características mencionadas en las secciones anteriores, C++11 incluye una biblioteca para el uso de tipos *traits* para la metaprogramación, también envoltorios para poder utilizar referencias en plantillas, métodos uniformes para calcular el tipo devuelto en objetos funciones, soporte para hilos y alias de plantillas.



Capítulo 3

Técnicas específicas

Francisco Moya Fernández
Félix J. Villanueva Molina
Sergio Pérez Camacho

En este capítulo se cubren técnicas específicas vinculadas al desarrollo avanzado de software utilizando el lenguaje de programación C++. En concreto, este capítulo discute el concepto de *plugin*, la serialización de objetos y la integración de *scripts* en C++.

3.1. Plugins

En términos generales se denomina *plug-in* (o *add-on* a cualquier componente que añade (o modifica) la funcionalidad de una aplicación principal integrándose con ella mediante un API proporcionando *ex profeso*. Los *plugins* son un mecanismo que se emplea habitualmente cuando se desea que programadores ajenos al desarrollo del proyecto matriz puedan integrarse con la aplicación. Ofrece algunas ventajas interesantes respecto a una aplicación monolítica:

- Reduce la complejidad de la aplicación principal.
- Permite experimentar con nuevas características, que si resultan de interés, más tarde se pueden integrar en la línea de desarrollo principal.
- Ahorra mucho tiempo a los desarrolladores de las extensiones puesto que no necesitan compilar el proyecto completo.
- Permite a empresas o colectivos concretos implementar funcionalidades a la medida de sus necesidades, que normalmente no serían admitidas en la aplicación principal.

- En entornos de código privativo, permite a los fabricantes distribuir parte del programa en formato binario, ya sea la aplicación central o alguno de los plugins. También ocurre cuando partes distintas tienen licencias diferentes.

Asumiendo que la aplicación principal esté escrita en un lenguaje compilado (como C++) se pueden distinguir tres mecanismos básicos que puede utilizar una aplicación para ofrecer soporte de plugins:

- Empotrar un interprete para un lenguaje dinámico, tal como Lua, Python o Scheme. Esta opción se desarrolla en la sección C++ y scripting. Si la aplicación matriz está escrita en un lenguaje dinámico no se requiere normalmente ningún mecanismo especial más allá de localizar y cargar los plugins desde sus ficheros.
- Proporcionar un protocolo basado en mensajes para que la aplicación principal se pueda comunicar con los plugins. Este es el caso de OSGi y queda fuera el ámbito de este curso. Este tipo de arquitectura es muy versátil (los plugins pueden incluso estar escritos en distintos lenguajes) aunque resulta bastante ineficiente.
- Proporcionar un API binaria y cargar los plugins como bibliotecas dinámicas. Es la opción más eficiente ya que la única penalización ocurre en el momento de la carga. Esta sección se ocupa de describir este mecanismo.

3.1.1. Entendiendo las bibliotecas dinámicas

En el módulo 1 ya se mostró el proceso necesario para generar una biblioteca dinámica. Hasta ahora hemos utilizado las bibliotecas como repositorios de funcionalidad común que puede ser utilizada por los ejecutables sin más que indicárselo al montador (*linker*). Sin embargo las bibliotecas dinámicas en los sistemas operativos con formato de ejecutables ELF (*executable and linkable format*) pueden servir para mucho más.

Una característica interesante de los ejecutables y bibliotecas ELF es que pueden tener símbolos no definidos, que son resueltos en tiempo de ejecución. Con las bibliotecas esta característica va más allá, hasta el punto de que no es necesario resolver todos los símbolos en tiempo de compilación. Veamos todo esto con ejemplos.

Hagamos un pequeño programa que utiliza una biblioteca.

Listado 3.1: El programa principal simplemente usa una biblioteca

```
1 void mylib_func(const char* str, int val);
2
3 int main() {
4     mylib_func("test", 12345);
5     return 0;
6 }
```

E implementemos una biblioteca trivial.

Listado 3.2: La biblioteca simplemente traza las llamadas

```
1 #include <stdio.h>
2
3 void mylib_func(const char* str, int val) {
4     printf("mylib_func %s %d\n", str, val);
5 }
```

Compilando el ejemplo como se indicó en el módulo 1 obtenemos el ejecutable y la biblioteca. Recordaremos que toda la biblioteca debe ser compilada con la opción `-fPIC` para generar código independiente de posición.

```
$ gcc -shared -fPIC -o libmylib.so mylib.c
$ gcc -o main main.c -L. -lmylib
```

Para ejecutarlo hay que indicarle al sistema operativo que también tiene que buscar bibliotecas en el directorio actual. Para eso basta definir la variable de entorno `LD_LIBRARY_PATH`.

```
$ LD_LIBRARY_PATH=. ./main
mylib_func test 12345
```

Sin tocar para nada todo lo hecho hasta ahora vamos a generar otra biblioteca dinámica con la misma función definida de otra forma.

Listado 3.3: Otra implementación de la biblioteca mínima

```
1 #include <stdio.h>
2
3 void mylib_func(const char* str, int val) {
4     printf("cambiada mylib_func %d %s\n", val, str);
5 }
```

Hemos cambiado ligeramente el mensaje pero podría haberse implementado de forma completamente diferente. Ahora compilamos como una biblioteca dinámica, pero ni siquiera tenemos que seguir el convenio de nombres tradicional.

```
$ gcc -shared -fPIC -o ml2.so mylib2.c
```

Y volvemos a ejecutar el programa de una forma muy peculiar:

```
$ LD_PRELOAD=ml2.so LD_LIBRARY_PATH=. ./main
cambiada mylib_func 12345 test
```

¿Sorprendido? No hemos recompilado el programa, no hemos cambiado la biblioteca original, pero hemos alterado el funcionamiento. Esta técnica puede utilizarse para multitud de fines, desde la depuración (e.g. *ElectricFence*) hasta la alteración de los ejecutables para corregir errores cuando no se dispone del código fuente.

Lo que pasa tras el telón podemos analizarlo con herramientas estándar:

```
$ ldd main
linux-vdso.so.1 => (0x00007fff701ff000)
libmylib.so => not found
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f13043dd000)
/lib64/ld-linux-x86-64.so.2 (0x00007f130477c000)
```

Todos los ejecutables dinámicos están montados con la biblioteca `ld.so` o `ld-linux.so`. Se trata del montador dinámico. Obsérvese cómo se incluye en el ejecutable la ruta completa (última línea). Esta biblioteca se encarga de precargar las bibliotecas especificadas en `LD_PRELOAD`, buscar el resto en las rutas del sistema o de `LD_LIBRARY_PATH`, y de cargarlas. El proceso de carga en el ejecutable incluye resolver todos los símbolos que no estuvieran ya definidos.

Cuando desde la biblioteca dinámica es preciso invocar funciones (o simplemente utilizar símbolos) definidas en el ejecutable, éste debe ser compilado con la opción `-rdynamic`. Por ejemplo:

Listado 3.4: El programa principal define símbolos públicos

```
1 #include <stdio.h>
2
3 void mylib_func(const char* str, int val);
4 int main_i = 54321;
5
6 void main_func(int v) {
7     printf("main_func %d\n", v);
8 }
9
10 int main() {
11     mylib_func("test", 12345);
12     return 0;
13 }
```

Y la biblioteca llama a las funciones definidas en el ejecutable:

Listado 3.5: La biblioteca llama a una función definida en el programa

```
1 #include <stdio.h>
2
3 void main_func(int v);
4 extern int main_i;
5
6 void mylib_func(const char* str, int val) {
7     printf("mylib_func %s\n", str);
8     main_func(main_i);
9 }
```

Compilar este ejemplo solo cambia en la opción `-rdynamic`.

```
$ gcc -shared -fPIC -o libmylib3.so mylib3.c
$ gcc -rdynamic -o main2 main2.c -L. -lmylib3
```

Y al ejecutarlo como antes:

```
$ LD_LIBRARY_PATH=. ./main2
mylib_func test
main_func 12345
```

Si todas estas actividades son realizadas por una biblioteca (`ld.so`) no debería extrañar que esta funcionalidad esté también disponible mediante una API, para la carga explícita de bibliotecas desde nuestro programa.

3.1.2. Plugins con `libdl`

El modo más sencillo (aunque rudimentario) para implementar plugins es utilizar la biblioteca `libdl` cuyo nombre significa exactamente eso: *dynamic loading*. El API de esta biblioteca se encuentra en el fichero de cabecera `dlfcn.h` es bastante simple:

```
1 void* dlopen(const char* filename, int flag);
2 void* dlsym(void* handle, const char* symbol);
3 int dlclose(void* handle);
4 char* dlerror(void);
```

La utilidad de las funciones es sencilla:

`dlopen()` abre una biblioteca dinámica (un fichero `.so`) y devuelve un manejador.

`dlsym()` carga y devuelve la dirección de símbolo cuyo nombre se especifique como `symbol`.

`dlclose()` le indica al sistema que ya no se va a utilizar la biblioteca y puede ser descargada de memoria.

`dlerror()` devuelve una cadena de texto que describe el último error por cualquier de las otras funciones de la biblioteca.

Vamos a seguir un ejemplo muy sencillo en las próximas secciones. El ejemplo está formado por un programa principal que tiene la lógica de registro de los plugins (`main.c`), una biblioteca estática (`liba`) y una dinámica (`libb`) que se cargará dinámicamente. Ambas bibliotecas tienen un fichero de cabecera (`a.h` y `b.h`) y dos ficheros de implementación cada una (`a.c`, `a2.c`, `b.c` y `b2.c`). La funcionalidad es absolutamente trivial y sirve únicamente para ilustrar la ejecución de las funciones correspondientes.

Listado 3.6: Biblioteca estática liba: a.h

```
1 #ifndef A_H
2 #define A_H
3
4 void a(int i);
5 int a2(int i);
6
7 #endif
```

Listado 3.7: Biblioteca estática liba: a.c

```
1 #include <stdio.h>
2 #include "a.h"
3
4 void a(int i) {
5     printf("a(%d) returns '%d'\n", i, a2(i));
6 }
```

Listado 3.8: Biblioteca estática liba: a2.c

```
1 #include "a.h"
2
3 int a2(int i) {
4     return i + 1;
5 }
```

Listado 3.9: Biblioteca dinámica libb: b.h

```
1 #ifndef B_H
2 #define B_H
3
4 void b(int i);
5 int b2(int i);
6
7 #endif
```

Listado 3.10: Biblioteca estática libb: b.c

```
1 #include <stdio.h>
2 #include "b.h"
3
4 void b(int i) {
5     printf("b(%d) returns '%d'\n", i, b2(i));
6 }
```

Listado 3.11: Biblioteca estática libb: b2.c

```
1 #include "b.h"
2
3 int b2(int i) {
4     return i * i;
5 }
```

Estas bibliotecas se construyen exactamente del mismo modo que ya se explicó en el capítulo «Herramientas de Desarrollo». Veamos como ejemplo el Makefile para libb:

Listado 3.12: Makefile para la compilación de libb

```
1 CC = gcc
2 CFLAGS = -Wall -ggdb -fPIC
3 LDFLAGS = -fPIC -shared
4
5 TARGET = libb.so.1.0.0
6
7 all: $(TARGET)
8
9 $(TARGET): b.o b2.o
10     $(CC) -Wl,-soname,libb.so.1.0.0 $(LDFLAGS) -o $@ $^
11
12 clean:
13     $(RM) *.o *~ *.a $(TARGET)
```

Carga explícita

En primer lugar veamos cómo cargar y ejecutar un símbolo (la función `b()`) de forma explícita, es decir, el programador utiliza `libdl` para buscar la biblioteca y cargar el símbolo concreto que desea:

Listado 3.13: Carga explícita de símbolos con `libdl`: `main.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <dlfcn.h>
4 #include "a.h"
5 #define LIBB_PATH "./dirb/libb.so.1.0.0"
6
7 void error() {
8     fprintf(stderr, dlerror()); exit(1);
9 }
10
11 int main() {
12     int i = 3;
13     void *plugin;
14     void (*function_b)(int);
15
16     if ((plugin = dlopen(LIBB_PATH, RTLD_LAZY)) == NULL)
17         error();
18
19     if ((function_b = dlsym(plugin, "b")) == NULL)
20         error();
21
22     printf("Results for ' %d':\n", i);
23     a(i);
24     function_b(i);
25
26     dlclose(plugin);
27     return 0;
28 }
```

La diferencia más importante respecto al uso habitual de una biblioteca dinámica es que **no hay ninguna referencia** a `libb` en la construcción del programa `main.c` del listado anterior. Veamos el `Makefile` de la aplicación:

Listado 3.14: Carga explícita de símbolos con `libdl`: `Makefile`

```

1 CC = gcc
2 CFLAGS = -Wall -ggdb -Idira
3 LDFLAGS = -Ldira
4 LDLIBS = -la -ldl
5
6 all: libs main
7
8 main: main.o
9
10 libs:
11     $(MAKE) -C dira
12     $(MAKE) -C dirb
13
14 clean:
15     $(RM) main *.o *~
16     $(MAKE) -C dira clean
17     $(MAKE) -C dirb clean

```

Carga implícita

Veamos ahora cómo construir un sencillo mecanismo que cargue automáticamente el símbolo en el momento de solicitar su uso.

Listado 3.15: Carga implícita de símbolos con `libdl`: `plugin.c`

```

1 typedef struct plugin {
2     char* key;
3     void (*function)(int);
4     struct plugin* next;
5 } plugin_t;
6
7 static plugin_t* plugins;
8
9 void
10 plugin_register(char* key, void (*function)(int)) {
11     plugin_t* p = (plugin_t*) malloc(sizeof(plugin_t));
12     p->key = key;
13     p->function = function;
14     p->next = plugins;
15     plugins = p;
16     printf("*** Plugin '%s' successfully registered.\n", key);
17 }
18
19 void
20 plugin_unregister(char* key) {
21     plugin_t *prev = NULL, *p = plugins;
22
23     while (p) {
24         if (0 == strcmp(p->key, key))
25             break;
26
27         prev = p;
28         p = p->next;

```

```

29  }
30
31  if (!p)
32      return;
33
34  if (prev)
35      prev->next = p->next;
36  else
37      plugins = p->next;
38
39  free(p);
40 }
41
42 static plugin_t*
43 plugin_find(char* key) {
44     plugin_t* p = plugins;
45     while (p) {
46         if (0==strcmp(p->key, key))
47             break;
48
49         p = p->next;
50     }
51     return p;
52 }
53
54 void
55 call(char* key, int i) {
56     plugin_t* p;
57
58     p = plugin_find(key);
59     if (!p) {
60         char libname[PATH_MAX];
61         sprintf(libname, "./dir%s/lib%s.so", key, key);
62         printf("Trying load '%s'.\n", libname);
63         dlopen(libname, RTLD_LAZY);
64         p = plugin_find(key);
65     }
66
67     if (p)
68         p->function(i);
69     else
70         fprintf(stderr, "Error: Plugin '%s' not available.\n", key);
71 }

```

Los plugins (líneas 1–5) se almacenan en una lista enlazada (línea 7). Las funciones `plugin_register()` y `plugin_unregister()` se utilizan para añadir y eliminar plugins a la lista. La función `call()` (líneas 54–71) ejecuta la función específica (contenida en el plugin) y le pasa el parámetro, es decir, invoca una función a partir de su nombre¹. Esa invocación se puede ver en la línea 10 del programa principal:

¹Este proceso se denomina enlace tardío (*late binding*) o *name binding*.

Listado 3.16: Carga implícita de símbolos con libdl: main.c

```

1 #include <stdio.h>
2 #include "plugin.h"
3 #include "a.h"
4
5 int main() {
6     int i = 3;
7
8     printf("Results for ' %d':\n", i);
9     a(i);
10    call("b", i);
11
12    return 0;
13 }
```

Para que los plugins (en este caso `libb`) se registre automáticamente al ser cargado se requiere un pequeño truco: el «atributo» `constructor` (línea 9) que provoca que la función que lo tiene se ejecute en el momento de cargar el objeto:

Listado 3.17: Carga implícita de símbolos con libdl: b.c

```

1 #include <stdio.h>
2 #include "../plugin.h"
3 #include "b.h"
4
5 void b(int i) {
6     printf("b(%d) returns ' %d'\n", i, b2(i));
7 }
8
9 static void init() __attribute__((constructor));
10
11 static void init() {
12     plugin_register("b", &b);
13 }
```

Aunque este sistema es muy simple (intencionadamente) ilustra el concepto de la carga de símbolos bajo demanda desconocidos en tiempo de compilación. A partir de él es más fácil entender mecanismos más complejos puesto que se bajan en la misma idea básica.



El atributo *constructor* indica que el símbolo al que va asociado debe almacenarse en una sección de la biblioteca reservada para el código de los constructores de variables estáticas. Estos constructores deben ejecutarse tan pronto como la biblioteca se carga en memoria. Análogamente, la sección *destructor* aglutina los destructores de las variables estáticas, que se invocan tan pronto como la biblioteca es cerrada.

(constructor)

En C++ no es necesario indicar manualmente estos atributos, basta definir un constructor para una variable estática.

3.1.3. Plugins con Glib gmodule

La biblioteca `glib` es un conjunto de utilidades, tipos abstractos de datos y otras herramientas de uso general y absolutamente portables. Es una biblioteca muy utilizada en los desarrollos del proyecto GNU. Un buen ejemplo de su uso es la biblioteca GTK y el entorno de escritorio GNOME. Una de esas utilidades es `GModule`, un sistema para realizar carga dinámica de símbolos compatible con múltiples sistemas operativos, incluyendo Sun, GNU/Linux, Windows, etc.

`GModule` ofrece un API muy similar a `libdl` con funciones prácticamente equivalentes:

```
1 GModule* g_module_open(const gchar* file_name, GModuleFlags flags);
2 gboolean g_module_symbol(GModule* module, const gchar* symbol_name,
3                           gpointer* symbol);
4 gboolean g_module_close(GModule* module);
5 const gchar * g_module_error(void);
```

Carga explícita

El siguiente listado muestra cómo hacer la carga y uso de la función `b()`, equivalente al listado 3.13:

Listado 3.18: Carga explícita de símbolos con `GModule`: `main.c`

```
1 #include <stdio.h>
2 #include <glib.h>
3 #include <gmodule.h>
4 #include "a.h"
5
6 #define LIBB_PATH "./dirb/libb.so.1.0.0"
7
8 void error() {
9     g_error(g_module_error());
10 }
11
12 int main(){
13     int i = 3;
14     GModule* plugin;
15     void (*function_b)(int);
16
17     if ((plugin = g_module_open(LIBB_PATH, G_MODULE_BIND_LAZY)) ==
18         NULL)
19         error();
20     if (!g_module_symbol(plugin, "b", (gpointer*)&function_b))
21         error();
22
23     printf("Results for ' %d'.\n", i);
24     a(i);
25     function_b(i);
26
27     g_module_close(plugin);
28
29     return 0;
30 }
```

Carga implícita

Por último, este módulo implementa el sistema de registro y carga automática usando una tabla hash de glib para almacenar los plugins:

Listado 3.19: Carga explícita de símbolos con GModule: plugin.c

```

1 #include <stdio.h>
2 #include <gmodule.h>
3 #include <glib/ghash.h>
4 #include "a.h"
5
6 #ifndef PATH_MAX
7 #define PATH_MAX 1024
8 #endif
9
10 static GHashTable* plugins = NULL;
11
12 void
13 plugin_register (char* key, void (*f)(int)) {
14     if (plugins == NULL)
15         plugins = g_hash_table_new_full(g_str_hash, g_str_equal, g_free
16             , g_free);
17     g_hash_table_insert(plugins, key, f);
18     g_message("Plugin '%s' succesfully registered.", key);
19 }
20
21 void
22 plugin_unregister(char* key) {
23     if (plugins != NULL)
24         g_hash_table_remove(plugins, key);
25 }
26
27 void
28 call(char* key, int i) {
29     void (*p)(int) = NULL;
30
31     if (plugins != NULL)
32         p = g_hash_table_lookup(plugins, key);
33
34     if (!p) {
35         char libname[PATH_MAX];
36
37         sprintf(libname, "./dir%s/lib%s.so", key, key);
38         g_message("Trying load '%s'.", libname);
39         if (g_module_open(libname, G_MODULE_BIND_LAZY) == NULL)
40             g_error("Plugin '%s' not available", libname);
41
42         if (plugins != NULL)
43             p = g_hash_table_lookup(plugins, key);
44     }
45
46     if (!p)
47         g_error("Plugin '%s' not availableeeee", key);
48
49     p(i);
50 }

```

3.1.4. Carga dinámica desde Python

El módulo `ctypes`, de la librería estándar de Python, permite mapear los tipos de datos de C a Python para conseguir una correspondencia binaria. Eso hace posible cargar funciones definidas en librerías dinámicas creadas con C/C++ y utilizarlas directamente desde Python.

El siguiente listado muestra como cargar y utilizar la misma función `b()` de la librería dinámica de las secciones anteriores:

Listado 3.20: Carga de símbolos desde Python con `ctypes`

```
1 LIBB_PATH = "./dirb/libb.so.1.0.0"
2
3 import ctypes
4
5 plugin = ctypes.cdll.LoadLibrary(LIBB_PATH)
6 plugin.b(3)
```

3.1.5. Plugins como objetos mediante el patrón *Factory Method*

Los plugins implican la adición y eliminación de código en tiempo de ejecución. Los problemas asociados tienen mucho que ver con los problemas que resuelven muchos de los patrones que ya conocemos. En esta sección veremos una pequeña selección.

Recordemos el patrón *factory method* ya descrito en el módulo 1. Se basa en la definición de una interfaz para crear instancias de objetos, permitiendo que las subclases redefinan este método. Este patrón se utiliza frecuentemente acoplado con la propia jerarquía de objetos, de forma parecida al patrón *prototype*, dando lugar a lo que se conoce como *constructor virtual*. Veamos un ejemplo similar al que poníamos para ilustrar el patrón prototipo pero ahora empleando el patrón *factory method*.

Listado 3.21: Ejemplo de patrón *factory method*.

```
1 class weapon {
2 public:
3     typedef shared_ptr<weapon> weapon_ptr;
4     virtual weapon_ptr make() = 0;
5     virtual void shoot() = 0;
6     virtual ~weapon() {}
7 };
8
9 class rifle: public weapon {
10 public:
11     weapon_ptr make() { return weapon_ptr(new rifle); }
12     void shoot() { cout << "shoot rifle." << endl; }
13 };
```

Empleamos `shared_ptr` para simplificar la gestión de la memoria y definimos un destructor virtual por si acaso alguna de las subclases necesitan liberar memoria dinámica. Ahora cualquier instancia de `rifle` podría ser usada como factoría, pero para simplificar aún más su uso vamos a definir una factoría que actúe de fachada frente a todos los *factory method* concretos. De esta forma disponemos de una factoría extensible.

Listado 3.22: Ejemplo de factoría extensible de armamento.

```

1 class weapon_factory {
2 public:
3     typedef shared_ptr<weapon> weapon_ptr;
4
5     weapon_ptr make(const string& key) {
6         weapon* aux = factories_[key];
7         if (aux) return aux->make();
8         return 0;
9     }
10
11    void reg(const string& key, weapon* proto) {
12        factories_[key] = proto;
13    }
14
15    void unreg(const string& key) {
16        factories_.erase(key);
17    }
18
19 protected:
20     map<string, weapon*> factories_;
21 };

```

Para añadir o eliminar nuevas subclases de `weapon` tenemos que llamar a `reg()` o `unreg()` respectivamente. Esto es adecuado para la técnica RAII (*Resource Acquisition Is Initialization*) en la que la creación y destrucción de un objeto se utiliza para el uso y liberación de un recurso:

Listado 3.23: Ejemplo de RAII para registro de nuevas armas.

```

1 template <class weapon_type>
2 class weapon_reg {
3     weapon_factory& factory_;
4     const char* key_;
5 public:
6     weapon_reg(weapon_factory& factory, const char* key)
7         : factory_(factory), key_(key) {
8         factory_.reg(key_, new weapon_type());
9     }
10    ~weapon_reg() {
11        factory_.unreg(key_);
12    }
13 };

```

Tanto la factoría como los objetos de registro podrían ser también modelados con el patrón *singleton*, pero para simplificar el ejemplo nos limitaremos a instanciarlos sin más:

Listado 3.24: Instancias de la factoría extensible y una factoría concreta.

```

1 weapon_factory factory;
2 weapon_reg<rifle> rifle_factory(factory, "rifle");

```

Veamos cómo ha quedado el ejemplo. Tenemos subclases derivadas de `weapon` que saben cómo construir nuevos elementos. Tenemos una factoría extensible que se puede poblar con nuevas subclases de `weapon`. Y finalmente tenemos una clase auxiliar para facilitar la extensión de la factoría con cualquier subclase de `weapon`. Es una estructura ideal para los plugins. Un plugin simplemente tiene que proporcionar nuevas subclases de `weapon` e instanciar un `weapon_reg` por cada una de ellas. Para ello tan solo habría que cambiar el método `make()` de la factoría:

Listado 3.25: Ejemplo de factoría extensible con plugins.

```

1 class dynamic_weapon_factory : public weapon_factory {
2 public:
3     weapon_ptr make(const string& key) {
4         weapon_ptr ret = weapon_factory::make(key);
5         if (ret) return ret;
6         load_plugin(key);
7         return weapon_factory::make(key);
8     }
9
10 private:
11     void load_plugin(const string& key);
12 };

```

El código de un plugin es completamente análogo al de las otras factorías concretas, como `rifle`.

Listado 3.26: Ejemplo de plugin para la factoría extensible.

```

1 #include "fmethod.hh"
2
3 class bow: public weapon {
4 public:
5     weapon_ptr make() { return weapon_ptr(new bow); }
6     void shoot() { cout << "shoot arrow." << endl; }
7 };
8
9 extern dynamic_weapon_factory dfactory;
10 weapon_reg<bow> bow_factory(dfactory, "bow");

```

La variable `dfactory` es la instancia de la factoría dinámica extensible. Está declarada en el programa principal, así que para poder ser utilizada desde una biblioteca es preciso que el *linker* monte el programa principal con la opción `-rdynamic`.

Por último pondremos un ejemplo de uso de la factoría:

Listado 3.27: Ejemplo de uso de la factoría.

```

1 int main(int argc, char* argv[])
2 {
3     while (argc > 1) {
4         shared_ptr<weapon> w = dfactory.make(argv[1]);
5         if (w) w->shoot();
6         else cout << "Missing weapon " << argv[1] << endl;
7         argc--; ++argv;
8     }
9 }

```

La descarga de la biblioteca dinámica (por ejemplo, utilizando la función `dlclose()`) provocaría que se llamara al destructor de la clase `bow_factory` y con ello que se des-registrara la factoría concreta.

Nótese que en este ejemplo empleamos la infraestructura de plugins para mantener extensible nuestra aplicación, pero no manejamos explícitamente los plugins. Así, por ejemplo, no hemos proporcionado ninguna función de descarga de plugins. Incidiremos en este aspecto en el siguiente ejemplo.

3.1.6. Plugins multi-plataforma

La biblioteca `GModule` que hemos visto en la sección es compatible con múltiples sistemas operativos. Sin embargo no está todo resuelto automáticamente. Es preciso conocer algunos detalles de las plataformas más comunes para poder implantar con éxito una arquitectura de plugins. Para ello veremos una adaptación del ejemplo anterior para ejecutables PE (ReactOS, Microsoft Windows).

En el caso de los ejecutables PE no es posible compilar bibliotecas (DLL) sin determinar las referencias a todos los símbolos. Por tanto no es posible referirnos a un símbolo definido en el programa principal (EXE). La solución más sencilla es extraer la parte común del ejecutable en una biblioteca dinámica que se monta tanto con el ejecutable como con las otras bibliotecas.

El programa principal queda reducido a:

Listado 3.28: Programa principal para Windows.

```

1 #include "fmethod-win.hh"
2
3 extern dynamic_weapon_factory dfactory;
4
5 int main(int argc, char* argv[])
6 {
7     while (argc > 1) {
8         shared_ptr<weapon> w = dfactory.make(argv[1]);
9         if (w) w->shoot();
10        else cout << "Missing weapon " << argv[1] << endl;
11        argc--; ++argv;
12    }
13 }

```

Y la parte común se extraería en:

Listado 3.29: Biblioteca común con la factoría para Windows.

```

1 #include "fmethod-win.hh"
2 #include <windows.h>
3
4 void
5 dynamic_weapon_factory::load_plugin(const string& key)
6 {
7     string libname = "./fmethod-" + key + "-win.dll";
8     LoadLibrary(libname.c_str());
9 }
10
11 dynamic_weapon_factory dfactory;
12 weapon_reg<rifle> rifle_factory(dfactory, "rifle");

```

Nótese cómo se cargan las bibliotecas con `LoadLibrary()`.

El plugin es muy similar a la versión ELF:

Listado 3.30: Plugin para Windows.

```

1 #include "fmethod-win.hh"
2
3 class bow: public weapon {
4 public:
5     weapon_ptr make() { return weapon_ptr(new bow); }
6     void shoot() { cout << "shoot arrow." << endl; }
7 };
8
9 extern dynamic_weapon_factory dfactory;
10 weapon_reg<bow> bow_factory(dfactory, "bow");

```

Para compilar y probar todo no es necesario utilizar ReactOS o Microsoft Windows. Podemos usar el compilador cruzado GCC para MINGW32 y el emulador wine.

```

$ i586-mingw32msvc-g++ -std=c++0x -shared \
-Wl,--enable-runtime-pseudo-reloc \
-o fmethod-fac-win.dll fmethod-fac-win.cc
$ i586-mingw32msvc-g++ -std=c++0x \
-Wl,--enable-runtime-pseudo-reloc \
-Wl,--enable-auto-import -o fmethod-win.exe \
fmethod-win.cc fmethod-fac-win.dll
$ i586-mingw32msvc-g++ -std=c++0x -shared
-Wl,--enable-runtime-pseudo-reloc \
-o fmethod-bow-win.dll fmethod-bow-win.cc \
fmethod-fac-win.dll
$ wine fmethod-win.exe rifle bow 2>/dev/null

```

La opción del montador `-enable-runtime-pseudo-reloc` permite utilizar la semántica tradicional de visibilidad de símbolos de Unix. Todos los símbolos externos son automáticamente exportados. La opción `-enable-auto-import` permite que todos los símbolos usados en el ejecutable que no están definidos en el propio ejecutable sean automáticamente importados.



Se propone como ejercicio la generalización de este código para que el mismo programa compile correctamente con ejecutables ELF o con ejecutables PE.

3.2. Serialización de objetos

La serialización de objetos tiene que ver en parte con la persistencia del estado de un videojuego. Serializar un objeto consiste en convertirlo en algo almacenable y recuperable. De este modo, el estado completo del objeto podrá ser escrito en disco o ser enviado a través de la red, y su estado podrá ser recuperado en otro instante de tiempo o en otra máquina.

Puede parecer una operación sencilla, después de todo, bastaría con almacenar el pedazo de memoria que representa al objeto y volver a ponerlo en el mismo sitio después. Lamentablemente esto no es posible, puesto que la configuración de la memoria varía de ejecución en ejecución y de máquina en máquina. Además, cada objeto tiene sus particularidades. Por ejemplo, si lo que se desea serializar es una `std::string` seguramente sea suficiente con almacenar los caracteres que la componen.

Uno de los problemas a la hora de serializar objetos es que estos pueden contener referencias o punteros a otros objetos, y este estado ha de conservarse de forma fidedigna. El problema de los punteros, es que la direcciones de memoria que almacenan serán diferentes en cada ejecución.

Antes de hablar de la serialización propiamente dicha, se presentarán los *streams* de C++.

3.2.1. Streams

Un *stream* es como una tubería por donde fluyen datos. Existen *streams* de entrada o de salida, y de ambas, de modo que un programa puede leer (entrada) de una abstrayéndose por completo de qué es lo que está llenando la misma. Esto hace que los *streams* sean una forma de desacoplar las entradas de la forma de acceder a las mismas, al igual que las salidas. No importa si quien rellena un *stream* es la entrada del teclado o un archivo, la forma de utilizarla es la misma para ambos casos. De este modo, controlar la entrada supondría conectar un *stream* a un fichero (o al teclado) y su salida al programa. Justo al revés (donde el teclado sería ahora la pantalla) se controlaría la salida.

Normalmente los *streams* tienen un *buffer* asociado puesto que escribir o leer en bloques suele ser mucho más eficiente en los dispositivos de entrada y salida. El *stream* se encargará (usando un

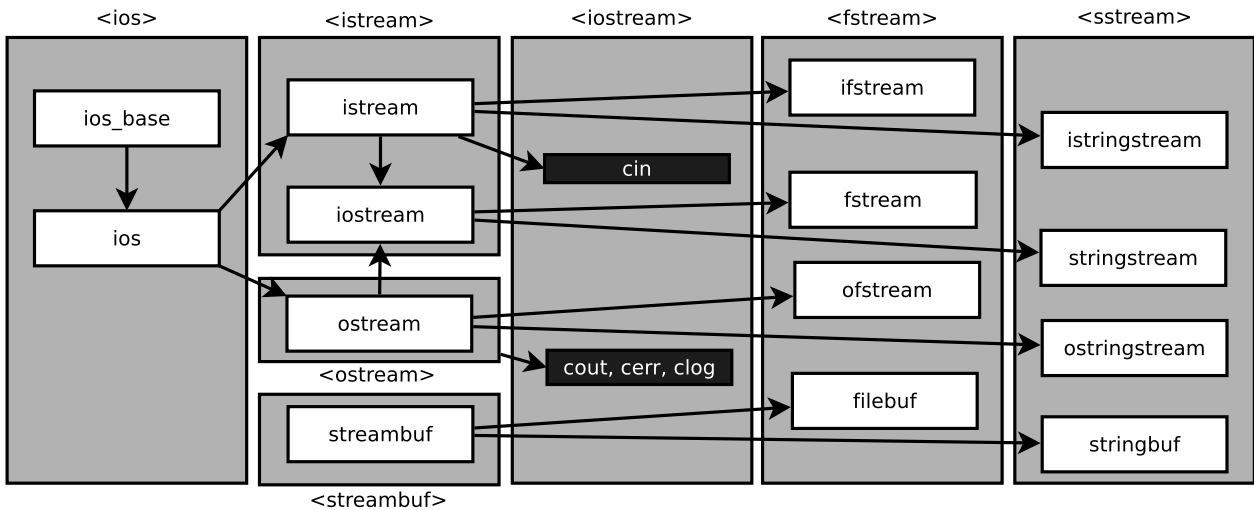


Figura 3.1: Jerarquía de *streams*

`streambuf`²) de proporcionar o recoger el número de bytes que se requiera leer o escribir en el mismo

En la figura 3.1 se muestra la jerarquía de *streams* en la biblioteca estándar de C++. La clase `ios_base` representa las propiedades generales de un *stream*, como por ejemplo si este es de entrada o de salida o si es de texto o binaria. La clase `ios`, que hereda de la anterior, contiene un `streambuf`. Las clases `ostream` y `istream`, derivan de `ios` y proporcionan métodos de salida y de entrada respectivamente.

istream

La clase `istream` implementa métodos que se utilizan para leer del *buffer* interno de manera transparente. Existen dos formas de recoger la entrada: formateada y sin formatear. La primera usa el operador `>>` y la segunda utiliza los siguientes miembros de la clase:

gcount	Devuelve el número de caracteres que retornó la última lectura no formateada
get	Obtiene datos sin formatear del <i>stream</i>
getline	Obtiene una línea completa del <i>stream</i>
ignore	Saca caracteres del <i>stream</i> y los descarta
peek	Lee el siguiente carácter sin extraerlo del <i>stream</i>
read	Lee en bloque el número de caracteres que se le pidan
readsome	Lee todo lo disponible en el <i>buffer</i>
putback	Introduce de vuelta un carácter en el <i>buffer</i>
unget	Decrementa el puntero <code>get</code> . Se leerá de nuevo el mismo carácter.

Utilizando `tellg` se obtiene la posición (`streampos`) del puntero en

²`streambuf` es una clase que provee la memoria para dicho *buffer* incluyendo además funciones para el manejo del mismo (rellenado, *flushing*, etc. . .)

el *stream*, y es posible modificar la misma utilizando `seekg` con la posición que de desee como entrada. La función `seekg` también se puede utilizar con un *offset* como primer parámetro y con una posición base como segundo. Así, `ios_base::beg`, `ios_base::cur` y `ios_base::end` representan al principio del *stream*, a la posición actual y al final del mismo respectivamente. Es posible (y de hecho necesario con `end`) utilizar números negativos para posicionarse en un *stream*.

ostream

Un `ostream` representa una tubería en la que se puede escribir. Al igual que un `istream`, se soportan los datos formateados, en este caso la inserción, usando el operador `<<`.

Las operaciones para datos no formateados son las siguientes:

put	Escribe un carácter en el <i>stream</i>
write	Escribe un conjunto de caracteres desde un <i>buffer</i>

ifstream y ofstream

Estos *streams* que se utilizan para leer y escribir de archivos.

En el ejemplo siguiente se muestra cómo leer de un archivo utilizando los visto sobre *streams*.

Listado 3.31: Ejemplo de lectura de un archivo

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4
5 using namespace std;
6
7 int main(int argc, char *argv[])
8 {
9     ifstream infile("prueba.txt", ios_base::binary);
10
11     if (!infile.is_open()) {
12         cout << "Error abriendo fichero" << endl;
13         return -1;
14     }
15
16     string linea;
17     getline(infile, linea);
18     cout << linea << endl;
19
20     char buffer[300];
21     infile.getline(buffer, 300);
22     cout << buffer << endl;
23
24     infile.read(buffer, 3);
25     buffer[3] = '\0';
26     cout << "[" << buffer << "]" << endl;
27

```

```

28  streampos p = infile.tellg();
29  infile.seekg(2, ios_base::cur);
30  infile.seekg(-4, ios_base::end);
31  infile.seekg(p);
32
33  int i;
34  while ((i = infile.get()) != -1)
35      cout << "\' " << (char) i << "\'=int(" << i << ")" << endl;
36
37  return 0;
38 }

```

En la línea [9](#) se crea el *stream* del fichero, y se intenta abrir para lectura como un fichero binario. En [11-14](#) se comprueba que el archivo se abrió y se termina el programa si no es así. En [16-18](#) se usa una función global de `string` para rellenar una de estas con una línea desde el fichero. Se hace lo mismo con un *buffer* limitado a 300 caracteres en la líneas [20-22](#). Después se leen 3 caracteres sueltos (sin tener en cuenta el final de línea) ([24-26](#)). En [28-31](#) se juega con la posición del puntero de lectura, y en el resto, se lee carácter a carácter hasta el final del archivo.

Los modos de apertura son los siguientes:

in	Permitir sacar datos del <i>stream</i>
out	Permitir introducir datos en el <i>stream</i>
ate	Al abrir el <i>stream</i> , situar el puntero al final del archivo.
app	Poner el puntero al final en cada operación de salida
trunc	Trunca el archivo al abrirlo
binary	El <i>stream</i> será binario y no de texto

En el ejemplo siguiente se muestra cómo copiar un archivo.

Listado 3.32: Ejemplo de copia desde un archivo a otro

```

1  #include <fstream>
2  using namespace std;
3
4  int main()
5  {
6      fstream in ("prueba.txt", ios_base::in | ios_base::binary);
7      fstream out("copiaP.txt", ios_base::out | ios_base::binary
8                  | ios_base::trunc );
9      if (!in.is_open() || !out.is_open())
10         return -1;
11
12     in.seekg(0, ios_base::end);
13     size_t size = in.tellg();
14     in.seekg(0, ios_base::beg);
15
16     char* buffer = new char[size];
17
18     in.read (buffer, size);
19     out.write(buffer, size);
20
21     delete [] buffer;
22     return 0;
23 }

```

Operadores de inserción y extracción

Es posible definir (sobrecargar) los operadores de inserción o de extracción para cualquier clase que nos interese, y así poder utilizarla para rellenar un *stream* o para modificarla extrayendo datos de un *stream*. Estos operadores se usan para una entrada/salida formateada.

Listado 3.33: Operadores de inserción y extracción de Vector3D

```

1 #include <iostream>
2
3 using namespace std;
4
5 class Vector3D {
6     friend ostream& operator<<(ostream& o, const Vector3D& v);
7     friend istream& operator>>(istream& i, Vector3D& v);
8 public:
9     Vector3D(float x, float y, float z) :
10         x_(x), y_(y), z_(z) {}
11 private:
12     float x_, y_, z_;
13 };
14
15 ostream& operator<<(ostream& o, const Vector3D& v)
16 {
17     o << "(" << v.x_ << ", " << v.y_ << ", " << v.z_ << ")" ;
18     return o;
19 }
20
21 istream& operator>>(istream& i, Vector3D& v)
22 {
23     char par, coma;
24     // formato: (X, Y, Z)
25     i >> par;
26     i >> v.x_;
27     i >> coma;
28     i >> v.y_;
29     i >> coma;
30     i >> v.z_;
31     return i;
32 }

```

En la líneas 15-19 se define el operador de inserción, y en las líneas 21-32 el de extracción. Es necesario definir estos operadores como amigos de la clase (líneas 6-7) ya que

La forma de utilizarlos es la siguiente:

Listado 3.34: Operadores de inserción y extracción

```

1 int main(int argc, char *argv[])
2 {
3     Vector3D v(1.0, 2.3, 4.5);
4     cout << v << endl;
5     cin >> v ;
6     cout << v << endl;
7     return 0;
8 }

```


El programa anterior imprime el valor original del vector, y espera a la entrada de un vector con el mismo formato. Al pulsar `RETURN` el vector original se rellenará con los nuevos datos tomados de la entrada estándar. De hecho, el programa funciona también con una tubería del tipo `echo "(1.0, 2.912, 3.123)" | ./ejecutable`.

`stringstream`

La clase `stringstream` proporciona un interfaz para manipular cadenas como si fueran *streams* de entrada/salida.

Su uso puede sustituir de algún modo al de `sprintf`, ya que es posible utilizar un objeto de este tipo para transformar números en cadenas y para realizar un formateo básico.

Listado 3.35: Usando un `stringstream`

```

1 #include <iostream>
2 #include <sstream>
3
4 using namespace std;
5
6 template<typename T>
7 string toString(T in)
8 {
9     stringstream ss;
10    ss << in;
11    return ss.str();
12 }
13
14 template<typename T>
15 T toNumber(const string& s)
16 {
17     stringstream ss(s);
18     T t;
19     ss << s;
20     if (!(ss >> t))
21         throw;
22     return t;
23 }
24
25
26 int main(int argc, char *argv[])
27 {
28     stringstream s;
29
30     s << 98 << endl << "texto" << endl;
31     cout << (s.str() += "op\n") ;
32
33     string str = toString(9.001);
34     long a = toNumber<long>("245345354525");
35     cout << a << endl;
36     return 0;
37 }

```

En las líneas `[6-12]` se define una función *templatazada* que se usa en `[33]` para transformar un número en una cadena, usando `stringstream` e invocando luego su método `str()`, que devuelve la cadena asociada.

En [14-23](#) se define otra que se puede utilizar para extraer un número de una cadena. Se ve un ejemplo de uso en la línea [34](#).

3.2.2. Serialización y Dependencias entre objetos

A la hora de serializar un objeto, o un conjunto de objetos, se pueden dar diferentes escenarios. No es lo mismo tener que escribir el contenido de un objeto que no tiene ninguna dependencia con otros, que tener que escribir el contenido de un conjunto de objetos que dependen unos de otros.

Sin dependencias

El escenario más simple es la serialización de un objeto sin dependencias con el resto, es decir, un objeto que no apunta a ningún otro y que está autocontenido.

La serialización será entonces trivial, y bastará con escribir cada una de los valores que contenga, y recuperarlo en el mismo orden.

Sea la siguiente una interfaz para objetos que puedan serializarse.

Listado 3.36: Interfaz simple para objetos serializables

```
1 class ISerializable {
2 public:
3     virtual void read (std::istream& in) = 0;
4     virtual void write(std::ostream& out) = 0;
5 };
```

De este modo, todos los objetos que deriven de esta clase tendrán que implementar la forma de escribir y leer de un *stream*. Es útil delegar los detalles de serialización al objeto.

Supóngase ahora una clase muy sencilla y sin dependencias, con un `double`, un `int` y un `string` para serializar.

Listado 3.37: Objeto serializable sin dependencias

```
1 class ObjetoSimple : public ISerializable
2 {
3 public:
4     ObjetoSimple(double a, int b, std::string cad);
5
6     ObjetoSimple();
7     virtual ~ObjetoSimple();
8
9     virtual void read (std::istream& in);
10    virtual void write(std::ostream& out);
11
12 private:
13
14    double    a_;
15    int       b_;
16    std::string cad_;
17 };
```

La implementación de `read()` y `write()` sería como sigue:

Listado 3.38: Detalle de implementación de un serializable simple

```
1 void
2 ObjetoSimple::read(std::istream& in)
3 {
4     in.read((char*) &a_, sizeof(double));
5     in.read((char*) &b_, sizeof(int));
6
7     size_t len;
8     in.read((char*) &len, sizeof(size_t));
9     char* auxCad = new char[len+1];
10
11     in.read(auxCad, len);
12     auxCad[len] = '\0';
13     cad_ = auxCad;
14
15     delete [] auxCad;
16
17     std::cout << "a_: " << a_ << std::endl;
18     std::cout << "b_: " << b_ << std::endl;
19     std::cout << "cad_: " << cad_ << std::endl;
20 }
21
22 void
23 ObjetoSimple::write(std::ostream& out)
24 {
25     out.write((char*) &a_, sizeof(double));
26     out.write((char*) &b_, sizeof(int));
27
28     size_t len = cad_.length();
29     out.write((char*) &len, sizeof(size_t));
30     out.write((char*) cad_.c_str(), len);
31 }
```

En la lectura y escritura se realiza un *cast* a `char*` puesto que así lo requieren las funciones `read` y `write` de un *stream*. Lo que se está pidiendo a dichas funciones es: “desde/en esta posición de memoria, tratada como un `char*`, lee/escribe el siguiente número de caracteres”.

El número de caracteres (bytes/octetos en x86+) viene determinado por el segundo parámetro, y en este ejemplo se calcula con `sizeof`, esto es, con el tamaño del tipo que se está guardando o leyendo.

Un caso especial es la serialización de un `string`, puesto que como se aprecia, no se está guardando todo el objeto, sino los caracteres que contiene. Hay que tener en cuenta que será necesario guardar la longitud de la misma (línea [30](#)) para poder reservar la cantidad de memoria correcta al leerla de nuevo ([8-9](#)).

A continuación se muestra un ejemplo de uso de dichos objetos utilizando archivos para su serialización y carga.

Listado 3.39: Uso de un objeto serializable simple

```
1 int main(int argc, char *argv[])
2 {
3     {
4         ofstream fout("data.bin", ios_base::binary | ios_base::trunc);
5         if (!fout.is_open())
6             return -1;
7
8         ObjetoSimple o(3.1371, 1337, "CEDV");
9         o.write(fout);
10        ObjetoSimple p(9.235, 31337, "UCLM");
11        p.write(fout);
12    }
13 }
14
15 ifstream fin("data.bin", ios_base::binary);
16 ObjetoSimple q;
17 q.read(fin);
18 ObjetoSimple r;
19 r.read(fin);
20
21 return 0;
22 }
```

Se está utilizando un archivo para escribir el valor de un par de objetos, y tras cerrarse, se vuelve a abrir para leer los datos almacenados y rellenar un par nuevo.

Con dependencias

Habrà dependencia entre objetos, cuando la existencia de uno esté ligada a la de otro. Normalmente esto viene determinado porque uno de los miembros de una clase es un puntero a la misma o a otra clase.

Cuando existen objetos con dependencias hay dos aproximaciones posibles para su serialización. La primera consiste en diseñar la arquitectura para que no se utilicen punteros. En vez de esto se utilizarán UUIDs (IDs únicas universales). Un objeto, en vez de almacenar un puntero al resto de objetos, almacenará su UUID y hará uso de factorías para recuperar el objeto en tiempo de carga o de ejecución. Las ventajas son claras, y las desventajas son el tiempo necesario para mantener las referencias actualizadas, y que la arquitectura dependerá de esta decisión de diseño completamente.

Otra forma de serializar clases con punteros es escribir sin preocupación y reparar el estado no-válido de ese objeto teniendo en cuenta las propiedades de los mismos. Un puntero referencia una dirección de memoria única, es decir, dos objetos diferentes no podrán compartir la misma dirección de memoria. Visto de otro modo, dos punteros iguales apuntan al mismo objeto. Teniendo esto en cuenta, el propio puntero podría valer como un UUID interno para la serialización.

De este modo, la serialización y deserialización lectura de objetos con punteros podría ser del siguiente modo:

- Almacenar todos los objetos, teniendo en cuenta que lo primero que se almacenará será la dirección de memoria que ocupa el

objeto actual. Los punteros del mismo se almacenarán como el resto de datos.

- Al leer los objetos, poner en una tabla el puntero antiguo leído, asociado a la nueva dirección de memoria.
- Hacer una pasada corrigiendo el valor de los punteros, buscando la correspondencia en la tabla.

Para ello necesitamos una interfaz nueva, que soporte la nueva función `fixPtrs()` y otras dos para leer y recuperar la posición de memoria del propio objeto.

Listado 3.40: Nueva interfaz de objeto serializable

```

1 class ISerializable {
2 public:
3     virtual void read (std::istream& in) = 0;
4     virtual void write (std::ostream& out) = 0;
5
6     virtual void fixPtrs () = 0;
7
8 protected:
9     virtual void readMemDir (std::istream& in) = 0;
10    virtual void writeMemDir (std::ostream& out) = 0;
11 };

```

Esta vez se implementará dicha interfaz con la clase `Serializable`:

Listado 3.41: Implementación de la interfaz `ISerializable`

```

1 class Serializable : public ISerializable {
2 public:
3     Serializable();
4     ~Serializable();
5
6     virtual void read (std::istream& in) = 0;
7     virtual void write (std::ostream& out) = 0;
8
9     virtual void fixPtrs () = 0;
10
11 protected:
12     virtual void readMemDir (std::istream& in);
13     virtual void writeMemDir (std::ostream& out);
14
15     Serializable* sPtr;
16 };

```

En la línea 15 se añade un puntero que almacenará la dirección de memoria de la propia clase.

La implementación de las funciones de lectura y escritura se muestra a continuación.

Listado 3.42: Implementación de la interfaz ISerializable (II)

```

1 void Serializable::readMemDir(std::istream& in)
2 {
3     in.read((char*) &sPtr, sizeof(Serializable*));
4     LookUpTable::getMap()[sPtr] = this;
5 }
6
7 void Serializable::writeMemDir(std::ostream& out)
8 {
9     sPtr = this;
10    out.write((char*) &sPtr, sizeof(Serializable*));
11 }

```

Cuando se lee la antigua dirección de memoria en `readMemDir`, esta se almacena en una tabla junto con la nueva dirección (línea 4). La implementación de la tabla se podría dar a través de una especie de *Singleton*, que envolvería un `map` y lo mostraría como una variable global.

Listado 3.43: Tabla de búsqueda de punteros

```

1 class Serializable; // Forward Dec.
2
3 class LookUpTable
4 {
5     friend class std::auto_ptr<LookUpTable*>;
6 public:
7     static std::map<Serializable*, Serializable*>& getMap();
8
9     typedef std::map<Serializable*, Serializable*>::iterator itMapS;
10
11 private:
12     LookUpTable(){}
13
14     std::map<Serializable*, Serializable*> sMap_;
15
16 };

```

Listado 3.44: Tabla de búsqueda de punteros (II)

```

1 std::map<Serializable*, Serializable*>&
2 LookUpTable::getMap()
3 {
4     static std::auto_ptr<LookUpTable> instance_(new LookUpTable);
5     return instance_->sMap_;
6 }

```

El nuevo tipo de objeto compuesto tendrá que derivar de la clase `Serializable` y no de `ISerializable` como antes.

Listado 3.45: Declaración de ObjetoCompuesto

```

1 class ObjetoCompuesto : public Serializable
2 {
3 public:
4     ObjetoCompuesto(double a, int b, std::string cad,
5                     ObjetoCompuesto* other);
6 }

```

```
7 ObjetoCompuesto();
8 virtual ~ObjetoCompuesto();
9
10 virtual void read (std::istream& in);
11 virtual void write(std::ostream& out);
12
13 virtual void fixPtrs();
14
15 void printCad();
16 void printOther();
17
18 private:
19
20 double    a_;
21 int       b_;
22 std::string cad_;
23 ObjetoCompuesto* obj_;
24 };
```

Uno de los constructores ahora acepta un puntero a un objeto del mismo tipo. En [23] se declara un puntero a un objeto del mismo tipo, y tendrá que ser serializado, recuperado y arreglado. Con motivo de probar si la lectura ha sido correcta, se han añadido un par de funciones, `printCad`, que imprime la cadena serializada del propio objeto y `printOther`, que imprime la cadena del objeto apuntado a través del primer método.

De esto modo, la implementación de la clase anterior sería la siguiente. Primero para las funciones de impresión, que son las más sencillas:

Listado 3.46: Definición de ObjetoCompuesto

```
1 void ObjetoCompuesto::printCad()
2 {
3     std::cout << cad_ << std::endl;
4 }
5
6 void ObjetoCompuesto::printOther()
7 {
8     if (obj_) obj_>printCad();
9 }
```

Y a continuación las de serialización y deserialización, con el añadido de que justo antes de leer el resto del objeto, se lee la dirección de memoria que se almacenó (línea [4]), que será la encargada de rellenar la tabla de punteros como se ha visto anteriormente. En la línea [19] se lee el puntero, como se haría de forma normal. En este momento, el puntero contendría la dirección antigua fruto de la serialización. Para la escritura pasa exactamente lo mismo, simplemente se guardan los punteros que corresponden a las direcciones de memoria en el momento de la escritura.

Listado 3.47: Definición de ObjetoCompuesto (II)

```

1 void
2 ObjetoCompuesto::read(std::istream& in)
3 {
4     readMemDir(in);
5
6     in.read((char*) &a_, sizeof(double));
7     in.read((char*) &b_, sizeof(int));
8
9     size_t len;
10    in.read((char*) &len, sizeof(size_t));
11    char* auxCad = new char[len+1];
12
13    in.read(auxCad, len);
14    auxCad[len] = '\0';
15    cad_ = auxCad;
16
17    delete [] auxCad;
18
19    in.read((char*) &obj_, sizeof(ObjetoCompuesto* ));
20
21    std::cout << "a_: " << a_ << std::endl;
22    std::cout << "b_: " << b_ << std::endl;
23    std::cout << "cad_: " << cad_ << std::endl;
24    std::cout << "obj_: " << obj_ << std::endl;
25    std::cout << "this: " << this << std::endl;
26    std::cout << "-----" << std::endl;
27 }
28
29 void
30 ObjetoCompuesto::write(std::ostream& out)
31 {
32     writeMemDir(out);
33
34     out.write((char*) &a_, sizeof(double));
35     out.write((char*) &b_, sizeof(int));
36
37     size_t len = cad_.length();
38     out.write((char*) &len, sizeof(size_t));
39     out.write((char*) cad_.c_str(), len);
40
41     out.write((char*) &obj_, sizeof(ObjetoCompuesto* ));
42     std::cout << "* obj_: " << obj_ << std::endl;
43 }

```

La función que se encarga de arreglar los punteros es la siguiente:

Listado 3.48: Definición de ObjetoCompuesto (III)

```

1 void ObjetoCompuesto::fixPtrs() {
2     if (obj_ == NULL)
3         return;
4
5     LookUpTable::itMapS it;
6     it = LookUpTable::getMap().find(obj_);
7     if (it == LookUpTable::getMap().end()) {
8         std::cout << "Puntero no encontrado" << std::endl;
9         throw;
10    }
11    obj_ = (ObjetoCompuesto*) it->second;
12    std::cout << "obj_ FIXED: " << obj_ << std::endl;
13 }

```


Si el puntero almacenado es nulo, no cambiará nada. Si el puntero no es nulo, se sustituirá por el que esté almacenado en la tabla, que será precisamente la nueva posición del objeto apuntado en memoria. Hay que tener en cuenta que para que esta función no falle, primero tendrá que estar cargado en memoria en objeto al que se debería estar apuntando.

Así, una forma de utilizar todas estas clases sería esta:

Listado 3.49: Serialización con dependencias

```

1  int main() {
2  cout << "Serializando" << endl; cout << "-----" << endl;
3  {
4      ofstream fout("data.bin", ios_base::binary | ios_base::trunc);
5      if (!fout.is_open())
6          return -1;
7
8      ObjetoCompuesto o(3.1371, 1337, "CEDV", NULL);
9      o.write(fout);
10     ObjetoCompuesto p(9.235, 31337, "UCLM", &o);
11     p.write(fout);
12     ObjetoCompuesto q(9.235, 6233, "ESI", &p);
13     q.write(fout);
14
15     ObjetoCompuesto* k = new ObjetoCompuesto(300.2, 1000, "BRUE",
16         &p);
17     k->write(fout);
18     delete k;
19
20     ObjetoCompuesto r(10.2, 3243, "2012", k);
21     r.write(fout);
22 }
23 cout << "\nRecuperando" << endl;
24 cout << "-----" << endl;
25
26 ifstream fin("data.bin", ios_base::binary);
27
28 std::vector<Serializable*> objetosLeidos;
29
30 for (int i = 0; i < 5; ++i) {
31     ObjetoCompuesto* o = new ObjetoCompuesto();
32     o->read(fin);
33     objetosLeidos.push_back(o);
34 }
35
36 cout << "\nFix punteros" << endl;
37 cout << "-----" << endl;
38
39 for_each(objetosLeidos.begin(), objetosLeidos.end(),
40     mem_fun(&Serializable::fixPtrs));
41
42 cout << "\nProbando" << endl;
43 cout << "-----" << endl;
44
45 std::vector<Serializable*>::iterator it;
46 for (it = objetosLeidos.begin();
47     it != objetosLeidos.end();
48     ++it)
49     static_cast<ObjetoCompuesto*>((*it)->printOther());
50
51 return 0;
52 }
```

En las líneas (5-23) se crea el archivo que se usará como un *stream* y algunos objetos que se van serializando. Algunos de ellos se crean en el *stack* y otro en el *heap*. El archivo se cerrará puesto que la variable `fout` sale de contexto al terminar el bloque.

En la línea (27) se abre el mismo archivo para proceder a su lectura. En (29-35) se leen los datos del archivo y se van metiendo en un vector. En (40-41) se procede a ejecutar la función `fixPtrs` de cada uno de los objetos almacenados dentro del vector. Justo después se ejecutan las funciones que imprimen las cadenas de los objetos apuntados, para comprobar que se han restaurado correctamente las dependencias.

La salida al ejecutar el programa anterior se muestra a continuación:

```
Serializando
-----
* obj_: 0
* obj_: 0x7fff3f6dad80
* obj_: 0x7fff3f6dadb0
* obj_: 0x7fff3f6dadb0
* obj_: 0x11b3320

Recuperando
-----
a_: 3.1371
b_: 1337
cad_: CEDV
obj_: 0
this: 0x11b3260
-----
a_: 9.235
b_: 31337
cad_: UCLM
obj_: 0x7fff3f6dad80
this: 0x11b3370
-----
a_: 9.235
b_: 6233
cad_: ESI
obj_: 0x7fff3f6dadb0
this: 0x11b3440
-----
a_: 300.2
b_: 1000
cad_: BRUE
obj_: 0x7fff3f6dadb0
this: 0x11b3520
-----
a_: 10.2
b_: 3243
cad_: 2012
obj_: 0x11b3320
this: 0x11b35d0
-----

Fix punteros
```

```

-----
obj_ FIXED: 0x11b3260
obj_ FIXED: 0x11b3370
obj_ FIXED: 0x11b3370
obj_ FIXED: 0x11b3520

```

Probando

```

-----
CEDV
UCLM
UCLM
BRUE

```

Cabe destacar que la dirección de memoria obtenida de los objetos en el *stack* se diferencia notablemente de la obtenida del *heap*. Como se puede ver, la serialización y la posterior lectura es correcta cuando se arreglan los punteros con la técnica presentada.

3.2.3. Serialización con Boost

Boost provee al programador de C++ con muchas utilidades, entre ellas la capacidad para serializar objetos de forma muy sencilla y metódica, convirtiendo una tarea tediosa en un mero trámite.

Objetos sin dependencias

Para serializar la clase simple expuesta en la sección anterior, primero habría del siguiente modo:

Listado 3.50: Serializando un objeto simple con Boost

```

1 #include <fstream>
2 #include <boost/archive/text_oarchive.hpp>
3 #include <boost/archive/text_iarchive.hpp>
4
5 class ObjetoSimple {
6     friend class boost::serialization::access;
7     public:
8         ObjetoSimple(double a, int b, std::string cad);
9         ObjetoSimple();
10        virtual ~ObjetoSimple();
11
12        void print();
13
14        template<class Archive>
15        void serialize(Archive & ar, const unsigned int version) {
16            ar & a_;
17            ar & b_;
18            ar & cad_;
19        }
20
21    private:
22        double    a_;
23        int       b_;
24        std::string cad_;
25 };

```

En la línea [8](#) se permite el acceso a esta clase desde la función `access` de Boost, que se usará para la invocación de `serialize` [18-22](#). El símbolo `&` utilizado dentro de dicha función *templaticada* representa a `<<` o `>>` según sea el tipo de `Archive`, que será el envoltorio de `fstreams` de Boost usado para la serialización. Es precisamente en esa función donde se lleva a cabo la serialización, puesto que para cada variable de la clase, se procede a su lectura o escritura.

A continuación se muestra cómo utilizar esta clase en un programa:

Listado 3.51: Uso de un objeto simple serializable con Boost

```

1  {
2    ofstream fout ("dataSimple", ios_base::trunc);
3    ObjetoSimple oSimple(1.0, 2, "BOOST");
4    boost::archive::text_oarchive outA(fout);
5    outA << oSimple;
6  }
7
8  {
9    ObjetoSimple otherSimple;
10   ifstream fin("dataSimple", ios_base::binary );
11   boost::archive::text_iarchive inA(fin);
12   inA >> otherSimple;
13   otherSimple.print();
14  }

```

En el primer bloque se crea un archivo de salida, y se crean y escriben dos objetos. En el segundo se leen y se imprimen. Como se muestra en la líneas [5](#) y [12](#), se usan los operadores de inserción y extracción de las clases de Boost utilizadas.

Objetos con dependencias

Sea la siguiente clase una similar a la compuesta que se planteó en la sección anterior, añadiendo además un objeto de tipo `ObjetoSimple` como miembro.

Listado 3.52: Declarando un objeto compuesto serializable con Boost

```

1  class ObjetoCompuesto
2  {
3  friend class boost::serialization::access;
4  public:
5    ObjetoCompuesto(double a, int b, std::string cad,
6                    ObjetoCompuesto* other);
7
8    ObjetoCompuesto();
9    virtual ~ObjetoCompuesto();
10
11   void print();
12   void printOther();
13
14   template<class Archive>
15   void serialize(Archive & ar, const unsigned int version) {
16     ar & a_;
17     ar & b_;
18     ar & cad_;

```

```

19     ar & simple_;
20     ar & obj_;
21 }
22
23 private:
24     double    a_;
25     int       b_;
26     std::string cad_;
27
28     ObjetoSimple    simple_;
29     ObjetoCompuesto* obj_;
30 };

```

Como se puede apreciar, la serialización se lleva a cabo de la misma manera si se utiliza Boost.

De hecho la forma de utilizarlos es similar, excepto a la hora de crear los objetos:

Listado 3.53: Uso de un objeto compuesto serializable

```

1  {
2  ofstream fout ("dataCompuesto", ios_base::trunc );
3  ObjetoCompuesto oComp (4.534, 90, "BOOST COMPO", NULL);
4  ObjetoCompuesto oComp2(43.234, 280, "OTRO BOOST COMPO", &oComp)
5  ;
6  boost::archive::text_oarchive outA(fout);
7  outA << oComp;
8  outA << oComp2;
9  }
10 {
11     ObjetoCompuesto otherComp;
12     ObjetoCompuesto otherComp2;
13
14     ifstream fin("dataCompuesto", ios_base::binary );
15     boost::archive::text_iarchive inA(fin);
16
17     inA >> otherComp;
18     inA >> otherComp2;
19
20     otherComp.print();
21     cout << "\n\n";
22     otherComp2.print();
23 }

```

De hecho, dos de los pocos casos donde esta forma difiere se muestran en el siguiente apartado.

Objetos derivados y con contenedores

En el código siguiente se muestra una clase Base y una clase ObjetoDerivadoCont que hereda de ella. Además, incluye un contenedor vector que se serializará con la misma.

Listado 3.54: Declarando un objeto base serializable con Boost

```

1 class Base {
2 friend class boost::serialization::access;
3 public:
4     Base(const std::string& bName) :
5         baseName_(bName) {}
6
7     virtual void print() {
8         std::cout << "Base::print(): " << baseName_;
9     };
10
11     virtual ~Base() {}
12
13     template<class Archive>
14     void serialize(Archive & ar, const unsigned int version) {
15         ar & baseName_;
16     }
17
18 protected:
19     std::string baseName_;
20 };

```

Listado 3.55: Declarando un objeto derivado y con contenedores serializable con Boost

```

1 class ObjetoDerivadoCont : public Base
2 {
3 friend class boost::serialization::access;
4 public:
5     ObjetoDerivadoCont(std::string s) :
6         Base(s) {}
7
8     ObjetoDerivadoCont() : Base("default") {}
9
10    virtual ~ObjetoDerivadoCont() {}
11
12    virtual void print();
13
14    void push_int(int i) {
15        v_.push_back(i);
16    };
17
18    template<class Archive>
19    void serialize(Archive & ar, const unsigned int version) {
20        ar & boost::serialization::base_object<Base>(*this);
21        ar & v_;
22    }
23
24 private:
25     std::vector<int> v_;
26 };

```

La única cosa que hay que tener en cuenta a la hora de serializar este tipo de clases es que hay que ser explícito a la hora de serializar la parte relativa a la clase base. Esto se lleva a cabo como se muestra en la línea [20](#) del código anterior.

Para que se puedan serializar contenedores, simplemente habrá que incluir la cabecera de Boost correspondiente:

Listado 3.56: Cabecera de Boost para serializar vector

```
1 #include <boost/serialization/vector.hpp>
```

Si se quisiera serializar una `list`, se usaría `list.hpp`.

A continuación, se muestra un ejemplo de uso donde se ve cómo se rellenan los `vectors`, para luego serializar dos los objeto y proceder a recuperarlos en el segundo bloque.

Listado 3.57: Uso de un objeto derivado y con contenedores

```
1  {
2  ofstream fout ("dataDerivadoCont", ios_base::trunc);
3  boost::archive::text_oarchive outA(fout);
4
5  ObjetoDerivadoCont oDeriv ("DERIVADO1");
6  oDeriv.push_int(38); oDeriv.push_int(485);
7  oDeriv.push_int(973); oDeriv.push_int(545);
8
9  ObjetoDerivadoCont oDeriv2("DERIVADO2");
10 oDeriv2.push_int(41356); oDeriv2.push_int(765);
11
12 outA << oDeriv;
13 outA << oDeriv2;
14 }
15
16 {
17 ifstream fin("dataDerivadoCont", ios_base::binary );
18 boost::archive::text_iarchive inA(fin);
19
20 ObjetoDerivadoCont oD;
21 ObjetoDerivadoCont oD2;
22
23 inA >> oD;
24 inA >> oD2;
25
26 oD.print();
27 cout << "\n\n\n";
28 oD2.print();
29 cout << "\n\n\n";
30 }
```

Con todos los ejemplos anteriores se puede afrontar casi cualquier tipo de serialización. Queda claro que el uso de Boost acelera el proceso, pero aun existen plataformas donde Boost no está portada (aquellas con compiladores que no soportan todas las características de C++, por ejemplo) y donde la STL aun lucha por parecerse al estándar. Es en éstas donde habrá que realizar una serialización más artesana y usar algún tipo de técnica parecida a la vista en las primeras secciones.

3.3. C++ y scripting

A pesar de que el uso de un lenguaje de propósito general como C++ nos permite abordar cualquier tipo de problema, existen lenguajes mas o menos apropiados para tareas específicas. En el diseño de un

lenguaje se tiene en mente aspectos como la eficiencia, portabilidad, simpleza, etc. y difícilmente se pueden alcanzar la excelencia en todas las facetas.

No obstante, sería deseable que pudiéramos realizar cada tarea en aquel lenguaje mas apropiado para la tarea a realizar. Por ejemplo, mientras que C/C++ se caracterizan, entre otras cosas, por su eficiencia, lenguajes como Python nos proporcionan un entorno de programación simple y muy productivo de cara a prototipado rápido así como una gran portabilidad.

Existen muchos proyectos que utilizan varios lenguajes de programación, utilizando el mas apropiado para cada tarea. En esta sección vamos a ver un ejemplo de esta interacción entre diversos lenguajes de programación. En concreto vamos a coger C++, como ya hemos comentado, un lenguaje orientado a objetos muy eficiente en su ejecución y Python, un lenguaje interpretado (como java, php, Lua etc.) muy apropiado por su simpleza y portabilidad que nos permite desarrollar prototipos de forma rápida y sencilla.

3.3.1. Consideraciones de diseño

En el caso de juegos, el planteamiento inicial es qué partes implementar en C++ y qué partes dejar al lenguaje de *scripting*.

En el caso del desarrollo de juegos cuyo lenguaje principal sea de *scripting* (por ejemplo, Python), una aproximación genérica sería, desarrollar el juego por completo, y después, mediante técnicas de *profiling* se identifican aquellas partes críticas para mejorar las prestaciones, que son las que se implementan en C/C++. Obviamente aquellas partes que, a priori, ya sabemos que sus prestaciones son críticas podemos anticiparnos y escribirlas directamente en C/C++.

En el caso de que la aplicación se implemente en C/C++, utilizamos un lenguaje de *scripting* para el uso de alguna librería concreta o para poder modificar/adaptar/extender/corregir el comportamiento sin tener que recompilar. En general, cuando hablamos de C++ y *scripting* hablamos de utilizar las características de un lenguaje de prototipado rápido desde C++, lo cual incluye, a grandes rasgos:

- Crear y borrar objetos en el lenguaje de *scripting* e interactuar con ellos invocando métodos .
- pasar datos y obtener resultados en invocaciones a funciones
- Gestionar posibles errores que pudieran suceder en el proceso de interacción, incluyendo excepciones.

Otro ejemplo de las posibilidades de los lenguajes de *scripting* son utilizar lenguajes específicos ampliamente usados en otros entornos como la inteligencia artificial, para implementar las partes relacionadas del juego. Ejemplos de este tipo de lenguajes serían LISP y Prolog ampliamente usados en inteligencia artificial, y por lo tanto, muy apropiados para modelar este tipo de problemas.

Lua vs Python

Mientras que Lua está pensado para extender aplicaciones y como lenguaje de configuración, Python es mas completo y puede ser utilizado para funciones mas complejas.

En la actualidad, las decisiones de diseño en cuanto a qué lenguaje de *scripting* usar viene determinado por las características de dichos lenguajes. Sin tener en cuenta lenguajes muy orientados a problemas concretos como los mencionados LISP y Prolog, y considerando sólo aquellos lenguajes de *scripting* de propósito general, las opciones actuales pasan por Lua y Python principalmente.

Atendiendo a sus características, Python:

- Tiene una gran librería y, generalmente, bien documentada.
- Facilita la gestión de cadenas y tiene operadores binarios.
- A partir de la versión 2.4, Python tiene los denominados *ctypes* que permiten acceder a tipos de librerías compartidas sin tener que hacer un *wrapper C*.
- Tiene buenas prestaciones en computación numérica (lo cual es muy deseable en simuladores de eventos físicos)

En contraste Lua es un lenguaje mas simple, originalmente pensado para labores de configuración y que ha sido orientado específicamente a la extensión de aplicaciones, algunas de sus características son:

- En general, usa menos memoria y el intérprete es mas rápido que el de Python.
- Tiene una sintaxis simple y fácil de aprender si bien es cierto que no tiene la documentación, ejemplos y tutoriales que Python.

Es cierto que tanto Lua como Python pueden ser utilizados para extender aplicaciones desarrolladas en C/C++, la decisión de qué lenguaje usar depende de qué características queremos implementar en el lenguaje de *scripting*. Al ser Python un lenguaje mas genérico, y por tanto versátil, que Lua será el que estudiaremos mas en profundidad.

3.3.2. Invocando Python desde C++ de forma nativa

En nomenclatura Python, hablamos de extender Python cuando usamos funciones y objetos escritos en un lenguaje (por ejemplo C++) desde programas en Python. Por el contrario, se habla de Python embebido cuando es Python el que se invoca desde una aplicación desarrollada en otro lenguaje. Desde la nomenclatura C/C++ se habla de *scripting* cuando accedemos a un lenguaje de script desde C++.

El interprete Python ya incluye extensiones para empotrar Python en C/C++. Es requisito imprescindible tener instalado en la máquina a ejecutar los ejemplos de esta sección, el intérprete de Python (usaremos la versión 2.7) aunque dichas extensiones están desde la versión 2.2.

En el primer ejemplo, vamos a ver la versión Python del intérprete y que nos sirve para ver cómo ejecutar una cadena en dicho intérprete desde un programa en C++.

Lenguajes compilados

Aquellas partes de cálculo intensivo deben ir implementadas en los lenguajes eficientes (compilados)

Listado 3.58: Imprimiendo la versión de Python desde C++

```
1 #include <python2.7/Python.h>
2
3 int main(int argc, char *argv[])
4 {
5     Py_Initialize();
6     PyRun_SimpleString("import sys; print '%d.%d' % sys.version_info
7                        [:2]\n");
8     Py_Finalize();
9     return 0;
}
```

La función `Py_Initialize()` inicializa el intérprete creando la lista de módulos cargados (`sys.modules`), crea los módulos básicos (`__main__`, `__builtin__` y `sys`) y crea la lista para la búsqueda de módulos `sys.path`. En definitiva lo prepara para recibir órdenes. `PyRun_SimpleString()` ejecuta un comando en el intérprete, podemos ver que en este caso, importamos el módulo `sys` y a continuación imprimimos la versión del intérprete que estamos ejecutando. Por último, finalizamos la instancia del intérprete liberando la memoria utilizada y destruyendo los objetos creados en la sesión.

Todas estas funciones se definen en el archivo `Python.h` que proporciona la instalación de Python y que proporciona un API para acceder al entorno de ejecución de este lenguaje. El propio intérprete de Python utiliza esta librería.

Estas extensiones permiten invocar todo tipo de sentencias e interactuar con el intérprete de Python, eso si, de forma no muy orientada a objetos. Utilizando el tipo *PyObject* (concretamente punteros a este tipo) podemos obtener referencias a cualquier módulo e invocar funciones en ellas. En la tabla 3.1 podemos ver, de forma muy resumida, algunas funciones que nos pueden ser muy útiles. Por supuesto no están todas pero nos pueden dar una referencia para los pasos principales que necesitaríamos de cara a la interacción C++ y Python.

La gestión de errores (del módulo `sys`) en la actualidad está delegada en la función `exc_info()` () que devuelve una terna que representan el tipo de excepción que se ha producido, su valor y la traza (lo que hasta la versión 1.5 representaban las variables `sys.exc_type`, `sys.exc_value` y `sys.exc_traceback`).

Con el ejemplo visto en esta subsección no existe un intercambio entre nuestro programa C++ y el entorno Python. Por supuesto, el soporte nativo de Python nos permite realizar cualquier forma de interacción que necesitemos. No obstante, podemos beneficiarnos de librerías que nos hacen esta interacción mas natural y orientada a objetos. Vamos a estudiar la interacción entre ambos entornos mediante la librería `boost`.

3.3.3. Librería `boost`

La librería `boost` [1] nos ayuda en la interacción de C++ y Python. Es necesario resaltar que está mas evolucionada en el uso de C++ desde Python que al revés. Esto es debido a que generalmente, es un

Función	Cometido
Py_Initialize()	Inicializa el intérprete
PyString_FromString("cadena")	Retorna un puntero a PyObject con una cadena (E.j. nombre del módulo a cargar).
PyImport_Import(PyObject* name)	Carga un módulo, retorna un puntero a PyObject.
PyModule_GetDict(PyObject* modulo)	Obtiene el diccionario con atributos y métodos del módulo. Retorna un puntero a PyObject.
PyDict_GetItemString(PyObject *Diccionario, "función")	Obtiene una referencia a una función. Retorna un puntero a PyObject
PyObject_CallObject(PyObject *función, argumentos)	Llama a la función con los argumentos proporcionados.
PyCallable_Check(PyObject *funcion)	Comprueba que es un objeto invocable.
PyRun_File	Interpreta un archivo
PyTuple_New(items)	Crea una tupla
PyTuple_SetItem(tupla, posición, item)	Almacena un Item en una tupla
PyErr_Print()	Imprime error.
PyList_Check(PyObject*)	Comprueba si PyObject es una lista

Tabla 3.1: Funciones útiles de invocación de Python desde C++

caso de uso mas frecuente el usar C++ desde Python por dos motivos principalmente:

- Aumentar la eficiencia del programa implementando partes críticas en C++.
- Usar alguna librería C++ para la cual no existen *bindings* en Python.

No obstante, como ya hemos indicado anteriormente, el uso de Python desde C++ también cuenta con ventajas y para introducir la librería boost, vamos a continuar con nuestro ejemplo de obtener la versión del interprete desde nuestro programa en C++.

Usando Python desde nuestro programa en C++

Nuestra primera modificación va a ser imprimir la versión del intérprete desde C++, por lo que debemos realizar un intercambio de datos desde el intérprete de Python al código en C++.

Listado 3.59: Obteniendo información de Python desde C++

```

1 #include <boost/python.hpp>
2 #include <boost/python/import.hpp>
3 #include <iostream>
4
5 using namespace boost::python;
6 using namespace std;
7
8 int main(int argc, char *argv[])
9 {
10
11     Py_Initialize();
12     PyRun_SimpleString("import sys; major, minor = sys.version_info
13         [:2]");
14     object mainobj = import("__main__");
15     object dictionary = mainobj.attr("__dict__");
16     object major = dictionary["major"];
17     int major_version = extract<int>(major);
18     object minor = dictionary["minor"];
19     int minor_version = extract<int>(minor);
20     cout<<major_version<<". "<<minor_version<<endl;
21     Py_Finalize();
22     return 0;
23 }

```

Debemos observar varios puntos en este nuevo listado:

- Seguimos usando *Py_Initialize* y *Py_Finalize*. Estas funciones se utilizan siempre y son obligatorias, en principio, no tienen equivalente en boost.
- Se usa *Run_SimpleString* para seguir con el ejemplo anterior, luego veremos como substituir esta sentencia por usos de la librería boost.
- Para acceder al interprete de Python, necesitamos acceder al módulo principal y a su diccionario (donde se definen todos los atributos y funciones de dicho módulo). Este paso se realiza en las líneas 13 y 14.
- Una vez obtenemos el diccionario, podemos acceder a sus variables obteniéndolas como referencias a *object()*, línea 15.

La plantilla *extract()* nos permite extraer de una instancia de *object*, en principio, cualquier tipo de C++. En nuestro ejemplo extraemos un entero correspondiente a las versiones del intérprete de Python (versión mayor y menor). De forma genérica, si no existe una conversión disponible para el tipo que le pasamos a *extract()*, una excepción Python (*TypeError*) es lanzada.

Como vemos en este ejemplo, la flexibilidad de Python puede simplificar nos la interacción con la parte de C++. La sentencia (línea 12) *sys.version_info* nos devuelve un tupla en Python, no obstante, hemos guardado esa tupla como dos enteros (*major* y *minor*) al cual accedemos de forma individual (líneas 16 y 19 mediante *extract*). Como ya hemos comentado, esta plantilla es clave de cara a obtener referencias a los tipos básicos desde C++ y puede ser empleado para aquellos tipos

básicos definidos como pueden ser `std::string`, `double`, `float`, `int`, etc. Para estructuras mas complejas (por ejemplo, tuplas), esta extracción de elementos se puede realizar mediante el anidamiento de llamadas a la plantilla *extract*.

Modificando brevemente el ejemplo anterior podemos mostrar el caso más básico de una tupla. tal y como podemos ver en este listado:

Listado 3.60: Extracción de tipos compleja

```

1  PyRun_SimpleString("import sys; result = sys.version_info[:2]");
2  object mainobj = import("__main__");
3  object dictionary = mainobj.attr("__dict__");
4  object result = dictionary["result"];
5  tuple tup = extract<tuple>(result);
6  if (!extract<int>(tup[0]).check() || !extract<int>(tup[1]).check
    ())
7      return 0;
8  int major =extract<int>(tup[0]);
9  int minor =extract<int>(tup[1]);
10 cout<<major<<". "<<minor<<endl;
11 Py_Finalize();
12 return 0;
13 }
```

ahora vemos como guardamos en *result* la tupla que, posteriormente, es guardada en la variable *tup* mediante el `extract()` correspondiente (línea 5).

A partir de este punto podemos obtener los elementos de la tupla (obviamente conociendo de antemano los campos de dicha tupla y su disposición en la misma) como podemos ver en las líneas 8 y 9. Obviamente, es recomendable realizar la comprobación de que la conversión de un entorno a otro se ha realizado correctamente mediante el uso de la función `check()` (línea 6).

Para el siguiente ejemplo vamos a dar un paso mas allá en nuestra forma de pasar datos de un entorno a otro.

Listado 3.61: Clase hero

```

1  class hero{
2      string _name;
3      string _weapon;
4      int amunition;
5  public:
6      hero(){}
7      hero(string name){
8          _name=name;
9      }
10
11     void configure()
12     {
13         cout<<"Getting configuration:"<<_name<<": "<<_weapon<<endl;
14     }
15     void weapon(string weapon){
16         _weapon=weapon;
17     }
18 };
```

Particularizando en la programación de videojuegos, vamos a suponer que tenemos una clase hero la cual, va a representar un héroe. Cada instancia coge su nombre del héroe que representa y a continuación se le asigna un arma.

Además se tiene un método `configure()`, que nos permite obtener la configuración del héroe en concreto, en este caso, simplemente la imprime. Bien asumimos como decisión de diseño, que, salvo el nombre, el arma asignada a cada héroe será variable y podremos ir obteniendo diversas armas conforme avancemos en el juego. Esta última parte la decidimos implementar en Python. Por lo tanto, habrá un método en Python, al cual le pasaremos un objeto de la clase hero y ese método lo configurará de forma apropiada (en nuestro caso sólo con el tipo de arma). En el siguiente listado podemos ver esta función. En este ejemplo simplemente le pasa el arma (Kalasnikov) invocando el método correspondiente.

Listado 3.62: Configurar una instancia de la clase hero desde Python

```
1 def ConfHero(hero):
2     hero.weapon("Kalasnikov")
3     hero.configure()
```

Para conseguir este ejemplo, necesitamos exponer la clase hero al intérprete de Python.

En boost, se usa la macro `BOOST_PYTHON_MODULE` que básicamente crea un módulo (*ConfActors*), que podremos usar en Python, definiendo las clases y métodos que le proporcionemos (en nuestro caso el constructor que acepta una cadena y los métodos `configure()` y `weapon()`)

Listado 3.63: Exponer clases C++ a entornos Python

```
1 // Exposing class hero to python
2 BOOST_PYTHON_MODULE( ConfActors )
3 {
4     class_<hero>("hero")
5         .def(init<std::string>() )
6         .def("configure", &hero::configure)
7         .def("weapon", &hero::weapon)
8     ;
9 }
```

Con esta infraestructura vamos a invocar la función en Python `ConfHero()` para que le asigne el arma y, a continuación vamos a comprobar que esa asignación se realiza de forma satisfactoria.

En el listado siguiente, en la línea 7 cargamos el contenido del archivo Python en el diccionario, con esta sentencia ponemos en el diccionario toda la información relativa a atributos y a funciones definidas en dicho archivo. A continuación ya podemos obtener un objeto que representa a la función Python que vamos a invocar (línea 8). Si este objeto es válido (línea 9), obtenemos un puntero compartido al objeto que vamos a compartir entre el intérprete Python y el espacio C++. En este caso, creamos un objeto de la clase *hero* (línea 11).

Listado 3.64: Pasando objetos C++ como argumentos de funciones en Python

```

1 int main(int argc, char *argv[])
2 {
3     Py_Initialize();
4     initConfActors(); //initialize the module
5     object mainobj = import("__main__");
6     object dictionary(mainobj.attr("__dict__"));
7     object result = exec_file("configureActors.py", dictionary,
8         dictionary);
9     object ConfHero_function = dictionary["ConfHero"];
10    if(!ConfHero_function.is_none())
11    {
12        boost::shared_ptr<hero> Carpanta(new hero("Carpanta"));
13        ConfHero_function(ptr(Carpanta.get()));
14        hero *obj = ptr(Carpanta.get());
15        obj->configure();
16    }
17    Py_Finalize();
18    return 0;
19 }

```

Ya estamos listos para invocar la función proporcionándole la instancia que acabamos de crear. Para ello, utilizamos la instancia del puntero compartido y obtenemos con `get()` la instancia en C++, con el cual podemos llamar a la función (línea 13) y por supuesto comprobar que nuestro héroe se ha configurado correctamente (línea 14).

Invocando C++ desde el intérprete Python

Veamos ahora el caso contrario. Vamos a tener una clase en C++ y vamos a acceder a ella como si de un módulo en Python se tratara. De hecho el trabajo duro ya lo hemos realizado, en el ejemplo anterior, ya usábamos un objeto definido en C++ desde el intérprete en Python.

Aprovechemos ese trabajo, si tenemos en un archivo el código relativo a la clase *hero* (listado 3.61) y la exposición realizada de la misma (listado 3.63) lo que nos falta es construir un módulo dinámico que el intérprete Python pueda cargar. En este punto nos puede ayudar el sistema de construcción del propio intérprete. Efectivamente podemos realizar un archivo `setup.py` tal y como aparece en el listado 3.65

Listado 3.65: Configuración para generar el paquete Python a partir de los fuentes C++

```

1 from distutils.core import setup, Extension
2
3 module1 = Extension('ConfActors', sources = ['hero.cc'], libraries
4     = ['boost_python-py27'])
5
6 setup (name = 'PackageName',
7     version = '1.0',
8     description = 'A C++ Package for python',
9     ext_modules = [module1])

```

De esta forma, podemos decirle a las herramientas de construcción y distribución de paquetes Python toda la información necesaria para que haga nuestro nuevo paquete a partir de nuestros fuentes en C++. En él, se le indica los fuentes.

Para compilar y generar la librería que, con posterioridad, nos permitirá importarla desde el intérprete de comandos, debemos invocar el archivo `setup.py` con el intérprete indicándole que construya el paquete:

```
python setup.py build
```

Esto nos generará la librería específica para la máquina donde estamos y lo alojará en el directorio `build` que creará en el mismo directorio donde esté el `setup.py` (`build/lib.linux-i686-2.7/` en nuestro caso) y con el nombre del módulo (`ConfActors.so`) que le hemos indicado. A partir de este punto, previa importación del módulo `ConfActors`, podemos acceder a todas sus clases y métodos directamente desde el intérprete de python como si fuera un módulo mas escrito de forma nativa en este lenguaje.

3.3.4. Herramienta SWIG

No se puede terminar esta sección sin una mención explícita a la herramienta SWIG [3], una herramienta de desarrollo que permite conectar programas escritos en C/C++ con una amplia variedad de lenguajes de programación de scripting incluidos Python, PHP, Lua, C#, Java, R, etc.

Para C++ nos automatiza la construcción de *wrappers* para nuestro código mediante una definición de las partes a utilizar en el lenguaje destino.

A modo de ejemplo básico, vamos a usar una nueva clase en C++ desde el intérprete Python, en este caso una clase `player` al cual le vamos a proporcionar parámetros de configuración.

Listado 3.66: Definición de la clase `Player`

```
1
2 #include <string>
3 #include <iostream>
4
5 class Player
6 {
7     std::string _name;
8     std::string _position;
9 public:
10     Player(std::string name);
11     void position(std::string pos);
12     void printConf();
13 };
```


Y su implementación:

Listado 3.67: Implementación de la clase en C++

```
1
2 using namespace std;
3 Player::Player(string name){
4     _name=name;
5 }
6
7 void Player::position(string pos){
8     _position=pos;
9 }
10
11 void Player::printConf(){
12     cout<<_name<<" "<<_position<<endl;
13 }
```

Sin modificación de estos archivos construimos un archivo de configuración para swig:

Listado 3.68: Archivo de configuración de SWIG

```
1 #define SWIG_FILE_WITH_INIT
2 #include "player.h"
3 %)
4
5 %include "std_string.i"
6 %include "player.h"
```

Con este archivo de configuración generamos `player_wrap.cc` y `player.py`:

```
swig -shadow -c++ -python player.i
```

El *wrapper* se debe compilar y enlazar con la implementación de la clase en una librería dinámica que se puede importar directamente desde el intérprete.

Listado 3.69: Testeando nuestro nuevo módulo Python

```
1 import player
2 p = player.Player('Carpanta')
3 dir(player)
4 p.printConf()
5 p.position("Goalkeeper")
6 p.printConf()
```

3.3.5. Conclusiones

Realizar un tutorial completo y guiado de la interacción entre C++ y los lenguajes de *scripting* queda fuera del ámbito de este libro. Hemos proporcionado, no obstante, algunos ejemplos sencillos que permiten al lector hacerse una idea de los pasos básicos para una interacción

básica entre C++ y un lenguaje de *scripting* de propósito general como es Python.

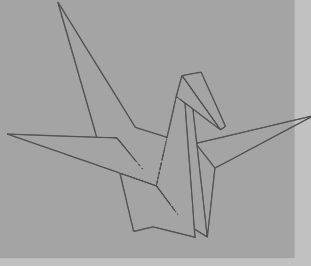
Se inició esta sección proporcionando los motivos por los que la integración de varios lenguajes de programación en una misma aplicación es una técnica muy útil y ampliamente utilizada en el mundo de los videojuegos. El objetivo final es utilizar el lenguaje más apropiado para la tarea que estamos desarrollando, lo cual da como resultado una mayor productividad y juegos más flexibles y extensibles.

A continuación hemos proporcionado algunas directivas básicas de cómo decidir entre lenguajes de *scripting* y compilados y qué partes son apropiadas para unos lenguajes u otros.

La mayor parte de esta sección se ha dedicado a mostrar cómo podemos integrar C++ y Python de tres maneras posibles:

- El soporte nativo del intérprete de Python es lo más básico y de más bajo nivel que hay para integrar Python en C++ o viceversa. La documentación del intérprete puede ayudar al lector a profundizar en este API.
- La librería boost nos aporta una visión orientada a objetos y de más alto nivel para la interacción entre estos lenguajes. Esta librería, o mejor dicho conjunto de librerías, de propósito general nos ayuda en este aspecto particular y nos proporciona otras potentes herramientas de programación en otros ámbitos como hemos visto a lo largo de este curso.
- Por último, hemos introducido la herramienta SWIG que nos puede simplificar de manera extraordinaria la generación de *wrappers* para nuestro código C++ de una forma automática y sin tener que introducir código adicional en nuestro código para interactuar con Python.

Herramientas y librerías similares a estas están disponibles para otros lenguajes de programación como Lua, prolog, etc.



4

Capítulo

Optimización

Francisco Moya Fernández

Antes de entrar en materia vamos a matizar algunos conceptos. Optimizar hace referencia a obtener el mejor resultado posible. Pero la bondad o maldad del resultado depende fuertemente de los criterios que se pretenden evaluar. Por ejemplo, si queremos hacer un programa lo más pequeño posible el resultado será bastante diferente a si lo que queremos es el programa más rápido posible. Por tanto cuando hablamos de optimización debemos acompañar la frase con el objetivo, con la magnitud que se pretende mejorar hasta el límite de lo posible. Así se habla frecuentemente de optimizar en velocidad u optimizar en tamaño.

La optimización normalmente es un proceso iterativo e incremental. Cada etapa produce un resultado mejor (o por lo menos más fácil de mejorar). A cada una de estas etapas del proceso se les suele denominar también optimizaciones, aunque sería más correcto hablar de etapas del proceso de optimización. Pero además el objetivo de optimización se enmarca en un contexto:

- Las mismas optimizaciones que en una arquitectura concreta generan mejores resultados pueden afectar negativamente al resultado en otras arquitecturas. Por ejemplo, la asignación de variables (o parámetros) a registros en un PowerPC aprovecha el hecho de disponer de un buen número de registros de propósito general. Si se usara el mismo algoritmo para asignar registros en un x86, en el que la mayoría de los registros son de propósito específico, obligaría a introducir multitud de instrucciones adicionales para almacenar temporalmente en la pila.

- Las mismas optimizaciones que permiten mejorar el rendimiento en un procesador pueden perjudicar al rendimiento cuando usamos multiprocesadores o procesadores multi-core. Por ejemplo, el paso por referencia, que permite ahorrar copias innecesarias, también exige utilizar primitivas de sincronización cuando los datos se acceden desde diferentes procesos. Estas primitivas afectan al paralelismo global y los bloqueos pueden superar con mucho el tiempo de copia del objeto.
- Incluso dentro de una misma arquitectura hay optimizaciones que penalizan a determinados procesadores de la misma familia. Por ejemplo en la familia Intel Pentium la forma más eficiente para transferir bloques de memoria era mediante el uso de instrucciones del coprocesador matemático debido al mayor tamaño de dichos registros frente a los de propósito general [29]. Eso ya no aplica para ninguna de las variantes modernas de la familia x86.

En cualquier caso es muy importante tener presente el objetivo global desde el principio, porque las oportunidades de mejora más destacables no están en mano del compilador, sino del programador. Los algoritmos y las estructuras de datos empleados son los que verdaderamente marcan la diferencia, varios órdenes de magnitud mejor que otras alternativas.

El programador de videojuegos siempre tiene que mantener un equilibrio entre dos frases célebres de Donald Knuth¹:

1. *In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering.* En las disciplinas de ingeniería tradicionales una mejora de un 12%, fácil de obtener, nunca se considera marginal y pienso que el mismo punto de vista debe prevalecer en la ingeniería de software.
2. *Premature optimization is the root of all evil.* La optimización prematura es la raíz de toda maldad.

Es decir, cuando se está desarrollando un videojuego la optimización no es una prioridad. No debemos ocuparnos de mejorar cuando todavía no sabemos qué debemos mejorar. Está ampliamente documentado que el ser humano es extremadamente malo prediciendo cuellos de botella.

Pero eso no puede justificar la programación descuidada. No es justificable incluir fragmentos de código o algoritmos claramente ineficientes cuando se puede hacer bien desde el principio a un mínimo coste, o incluso a un coste menor.

¹Ambas frases aparecen prácticamente juntas en la página 268 de [23].

4.1. Perfilado de programas

Una vez que se dispone de un prototipo o un fragmento funcional del programa podemos determinar los cuellos de botella del programa para intentar mejorarlo. Para ello se suele emplear una técnica conocida como perfilado de software (*software profiling*). El perfilado permite contestar preguntas como:

- ¿Dónde se gasta la mayor parte del tiempo de ejecución? De cara a concentrar los esfuerzos de optimización donde más se notará.
- ¿Cuál es el camino crítico? Para incrementar las prestaciones globales. Por ejemplo, el número de *frames* por segundo.
- ¿Cuál es la tasa de fallos de la memoria caché? Con el objetivo de mejorar la localidad de la memoria.

Normalmente recabar este tipo de información implica instrumentar el código añadiendo algunas instrucciones que permiten acumularla en un archivo (o varios) para cada ejecución del programa. La información de perfilado es posteriormente analizada con un programa, el perfilador o *profiler*.

Cada *profiler* implementa el registro de la información de forma diferente. Básicamente se utilizan cuatro técnicas: trazas, muestreo estadístico, puntos de ruptura hardware y contadores hardware. Veamos cada una de ellas en más detalle:

- Cuando el evento de interés corresponde a una operación que requiere un tiempo considerable es posible trazar cada ejecución de la operación sin un impacto significativo en las prestaciones del programa. Ésta es la técnica empleada por el perfilador de Linux *perf* (descrito más adelante) para trazar las operaciones sobre el sistema de archivos, las operaciones de *writeback*, las operaciones de gestión de energía, la recepción y el manejo de interrupciones, las operaciones de planificación de procesos, etc. También es la técnica empleada por utilidades como *strace*, que traza las llamadas al sistema de un proceso.
- Sin embargo, en un programa de tamaño considerable no es posible ejecutar código adicional en todos los eventos de interés (por ejemplo, en todas las llamadas a función). En ese caso se realiza un análisis estadístico. Periódicamente se realiza un muestreo del contador de programa y se analiza en qué función se encuentra. Es más, en lugar de solo observar el valor del contador de programa puede analizar el contenido de la pila para determinar todos marcos de pila activos, es decir, la *call trace*. Con esto es posible determinar el grafo de llamadas y el tiempo estimado destinado a cada función.
- En lugar de instrumentar el código o muestrear de forma estadística, es posible utilizar los mecanismos previstos en los procesadores actuales para facilitar el perfilado. Por ejemplo, una posibilidad es el empleo de puntos de ruptura hardware para detectar

cuándo se escribe una posición de memoria, cuándo se escribe, o cuándo se ejecuta la instrucción que contiene. Esta técnica se puede emplear para trazar solo un conjunto limitado de funciones, o para estudiar el patrón de accesos a un objeto. También se emplea en la utilidad *ltrace*, que traza las llamadas a procedimientos de bibliotecas dinámicas desde un proceso determinado.

- Por último los procesadores modernos proporcionan otra funcionalidad especialmente interesante para el perfilado. Disponen de una *Performance Monitoring Unit* que controla un conjunto de registros especiales denominados *performance counters*. Estos registros son capaces de contar determinados eventos, tales como ciclos de la CPU, ciclos de bus, instrucciones, referencias a la cache, fallos de la memoria caché, saltos o fallos en la predicción de saltos. Estos registros pueden ser utilizados en *profilers* tales como *perf* para realizar mediciones muy precisas.

Es importante conocer cuándo se emplea cada una de estas técnicas para poder interpretar con precisión los datos del perfilado. Así, por ejemplo, las técnicas basadas en muestreo de la traza de llamadas debe entenderse en un contexto estadístico. Valores bajos en los contadores de llamadas no tienen significado absoluto, sino en relación a otros contadores. Es muy posible que tengamos que ejecutar el mismo fragmento de código múltiples veces para eliminar cualquier sesgo estadístico.



Para cualquier análisis que requiera examinar la pila (perfilado de la traza de llamadas, o del grafo de llamadas, o simplemente la depuración interactiva), se asume el convenio de que un registro contiene la dirección del marco de pila actual (*frame pointer*) y al principio del marco de pila actual se almacena una copia del *frame pointer* anterior a la llamada actual. Sin embargo los compiladores actuales pueden generar código perfectamente funcional sin necesidad de *frame pointer*. Es importante compilar los programas evitando la opción `-fomit-frame-pointer` o incluso explícitamente indicando `-fno-omit-frame-pointer` durante el desarrollo para que estos análisis funcionen correctamente.

4.1.1. El perfilador de Linux *perf*

El subsistema *Linux Performance Counters* proporciona una abstracción de los *performance counters* disponibles en los procesadores modernos. Independientemente del hardware subyacente Linux ofrece una serie de contadores de 64 bits virtualizados por CPU o por tarea y combinado con un sistema de traza de eventos de otro tipo (eventos software, trazas). Es más sencillo de lo que parece, veamos algún ejemplo.

La herramienta *perf* está incluida en el paquete `linux-tools-X.Y` donde X.Y hace referencia a la versión del kernel empleada. Por ejemplo, la más actual en el momento de escribir este texto es `linux-tools-3.2`. Por tanto para instalar la herramienta deberemos ejecutar:

```
$ sudo apt-get install linux-tools-3.2
```

A continuación conviene configurar el kernel para que permita a los usuarios normales recabar estadísticas de todo tipo. Esto no debe hacerse con carácter general, sino solo en las computadoras empleadas en el desarrollo, puesto que también facilita la obtención de información para realizar un ataque.

```
$ sudo sh -c "echo -1 > /proc/sys/kernel/perf_event_paranoid"
```

Ahora ya como usuarios normales podemos perfilar cualquier ejecutable, e incluso procesos en ejecución. Tal vez la primera tarea que se debe realizar para perfilar con *perf* es obtener la lista de eventos que puede contabilizar. Esta lista es dependiente de la arquitectura del procesador y de las opciones de compilación del kernel.

```
$ perf list
```

```
List of pre-defined events (to be used in -e):
```

<code>cpu-cycles</code> OR <code>cycles</code>	[Hardware event]
<code>stalled-cycles-frontend</code> OR <code>idle-cycles-frontend</code>	[Hardware event]
<code>stalled-cycles-backend</code> OR <code>idle-cycles-backend</code>	[Hardware event]
<code>instructions</code>	[Hardware event]
<code>cache-references</code>	[Hardware event]
<code>cache-misses</code>	[Hardware event]
<code>branch-instructions</code> OR <code>branches</code>	[Hardware event]
<code>branch-misses</code>	[Hardware event]
<code>bus-cycles</code>	[Hardware event]
<code>cpu-clock</code>	[Software event]
<code>task-clock</code>	[Software event]
<code>page-faults</code> OR <code>faults</code>	[Software event]
<code>minor-faults</code>	[Software event]
<code>major-faults</code>	[Software event]
<code>context-switches</code> OR <code>cs</code>	[Software event]
<code>cpu-migrations</code> OR <code>migrations</code>	[Software event]
<code>alignment-faults</code>	[Software event]
<code>emulation-faults</code>	[Software event]
<code>L1-dcache-loads</code>	[Hardware cache event]
<code>L1-dcache-load-misses</code>	[Hardware cache event]
<code>L1-dcache-stores</code>	[Hardware cache event]
<code>L1-dcache-store-misses</code>	[Hardware cache event]
<code>L1-dcache-prefetches</code>	[Hardware cache event]

```

L1-dcache-prefetch-misses [Hardware cache event]
L1-icache-loads [Hardware cache event]
L1-icache-load-misses [Hardware cache event]
L1-icache-prefetches [Hardware cache event]
L1-icache-prefetch-misses [Hardware cache event]
LLC-loads [Hardware cache event]
LLC-load-misses [Hardware cache event]
LLC-stores [Hardware cache event]
LLC-store-misses [Hardware cache event]
LLC-prefetches [Hardware cache event]
LLC-prefetch-misses [Hardware cache event]
dTLB-loads [Hardware cache event]
dTLB-load-misses [Hardware cache event]
dTLB-stores [Hardware cache event]
dTLB-store-misses [Hardware cache event]
dTLB-prefetches [Hardware cache event]
dTLB-prefetch-misses [Hardware cache event]
iTLB-loads [Hardware cache event]
iTLB-load-misses [Hardware cache event]
branch-loads [Hardware cache event]
branch-load-misses [Hardware cache event]
node-loads [Hardware cache event]
node-load-misses [Hardware cache event]
node-stores [Hardware cache event]
node-store-misses [Hardware cache event]
node-prefetches [Hardware cache event]
node-prefetch-misses [Hardware cache event]

rNNN (...) [Raw hardware event descriptor]

mem:<addr>[:access] [Hardware breakpoint]

i915:i915_gem_object_create [Tracepoint event]
i915:i915_gem_object_bind [Tracepoint event]
i915:i915_gem_object_unbind [Tracepoint event]
...
sched:sched_wakeup [Tracepoint event]
sched:sched_wakeup_new [Tracepoint event]
sched:sched_switch [Tracepoint event]
...

```

En la lista de eventos podemos apreciar seis tipos diferentes.

- *Software event*. Son simples contadores del kernel. Entre otros permite contar cambios de contexto o fallos de página.
- *Hardware event*. Se refiere a los contadores incluidos en las PMU (Performance Monitoring Units) de los procesadores modernos. Permite contar ciclos, instrucciones ejecutadas, fallos de caché. Algunos de estos contadores se ofrecen de forma unificada como contadores de 64 bits, de tal forma que oculta los detalles de la PMU subyacente. Pero en general su número y tipo dependerá del modelo de procesador donde se ejecuta.
- *Hardware cache event*. Dentro del subsistema de memoria las PMU modernas² permiten extraer estadísticas detalladas de las

²Los eventos de las PMU se documentan en los manuales de los fabricantes. Por ejemplo, los contadores de la arquitectura Intel 64 e IA32 se documentan en el apéndice A de [17] disponible en <http://www.intel.com/Assets/PDF/manual/253669.pdf> y los de los procesadores AMD64 en [5] disponible en http://support.amd.com/us/Processor_TechDocs/31116.pdf

memorias caché de primer nivel de último nivel o del TLB. Nuevamente se trata de contadores que dependen fuertemente del modelo de procesador sobre el que se ejecuta.

- *Hardware breakpoint.* Los puntos de ruptura hardware permiten detener la ejecución del programa cuando el procesador intenta leer, escribir o ejecutar el contenido de una determinada posición de memoria. Esto nos permite monitorizar detalladamente objetos de interés, o trazar la ejecución de instrucciones concretas.
- *Tracepoint event.* En este caso se trata de trazas registradas con la infraestructura *ftrace* de Linux. Se trata de una infraestructura extremadamente flexible para trazar todo tipo de eventos en el kernel o en cualquier módulo del kernel. Esto incluye eventos de la GPU, de los sistemas de archivos o del propio *scheduler*.
- *Raw hardware event.* En el caso de que *perf* no incluya todavía un nombre simbólico para un contador concreto de una PMU actual se puede emplear el código hexadecimal correspondiente, de acuerdo al manual del fabricante.

4.1.2. Obteniendo ayuda

La primera suborden de *perf* que debe dominarse es *help*, que se emplea para obtener ayuda. La ejecución de *perf help* sin más nos muestra todas las órdenes disponibles. Las más utilizadas son *perf stat*, *perf record*, *perf report* y *perf annotate*.

Cada una de estas órdenes tienen ayuda específica que puede obtenerse con *perf help suborden*.

4.1.3. Estadísticas y registro de eventos

La operación más sencilla que se puede hacer con *perf* es contar eventos. Eso puede realizarse con la suborden *perf stat*:

```
$ perf stat glxgears
Performance counter stats for 'glxgears':

80,416861 task-clock                # 0,069 CPUs utilized
      171 context-switches         # 0,002 M/sec
       71 CPU-migrations           # 0,001 M/sec
    10732 page-faults              # 0,133 M/sec
109061681 cycles                   # 1,356 GHz
 [86,41%]
75057377 stalled-cycles-frontend  # 68,82% frontend cycles idle
 [85,21%]
58498153 stalled-cycles-backend   # 53,64% backend  cycles idle
 [62,34%]
68356682 instructions              # 0,63 insns per cycle
                                     # 1,10 stalled cycles per insn
                                     [80,66%]
14463080 branches                  # 179,851 M/sec
 [86,78%]
 391522 branch-misses              # 2,71% of all branches
 [80,19%]
```

```
1,158777481 seconds time elapsed
```

Basta indicar el ejecutable a continuación de `perf stat`. Por defecto muestra un conjunto de métricas comunes, que incluye eventos hardware (como los ciclos o las instrucciones), eventos software (como los cambios de contexto), y métricas derivadas a la derecha (como el número de instrucciones por ciclo).

Puede utilizarse `perf` para medir un tipo de eventos concreto empleando la opción `-e`:

```
$ perf stat -e cycles,instructions precompute_landscape

Performance counter stats for 'precompute_landscape':

   4473759 cycles          #    0,000 GHz
   3847463 instructions    #    0,86  insns per cycle

0,004595748 seconds time elapsed
```

Y podemos dividir entre los eventos que ocurren en espacio de usuario y los que ocurren en espacio del kernel.

```
$ perf stat -e cycles:u,cycles:k precompute_landscape

Performance counter stats for 'precompute_landscape':

   1827737 cycles:u        #    0,000 GHz
   2612202 cycles:k        #    0,000 GHz

0,005022949 seconds time elapsed
```

Todos los eventos hardware aceptan los modificadores `u` para filtrar solo los que ocurren en espacio de usuario, `k` para filtrar los que ocurren en espacio del kernel y `uk` para contabilizar ambos de forma explícita. Hay otros modificadores disponibles, incluso alguno dependiente del procesador en el que se ejecuta.

4.1.4. Multiplexación y escalado

Las PMU tienen dos tipos de contadores: los contadores fijos, que cuentan un único tipo de evento, y los contadores genéricos, que pueden configurarse para contar cualquier evento hardware. Cuando el usuario solicita más eventos de los que físicamente se pueden contar con los contadores implementados el sistema de perfilado multiplexa los contadores disponibles. Esto hace que parte del tiempo se estén contando unos eventos y parte del tiempo se están contando otros eventos distintos.

Posteriormente el propio sistema escala los valores calculados en proporción al tiempo que se ha contado el evento respecto al tiempo total. Es muy fácil de ver el efecto con un ejemplo. El computador sobre el que se escriben estas líneas dispone de un procesador Intel Core i5. Estos procesadores tienen 4 contadores genéricos³.

³Lo más normal es disponer de dos o cuatro contadores genéricos y otros tantos es-

Vamos a ver qué pasa cuando se piden 4 eventos idénticos:

```
$ perf stat -e cycles,cycles,cycles,cycles render_frame

Performance counter stats for 'render_frame':

   803261796 cycles      #    0,000 GHz
   803261796 cycles      #    0,000 GHz
   803261796 cycles      #    0,000 GHz
   803261799 cycles      #    0,000 GHz

0,306640126 seconds time elapsed
```

Puede verse que la precisión es absoluta, los cuatro contadores han contado prácticamente la misma cantidad de ciclos. En cambio, veamos qué pasa cuando se solicitan 5 eventos idénticos:

```
$ perf stat -e cycles,cycles,cycles,cycles,cycles render_frame

Performance counter stats for 'render_frame':

   801863997 cycles      #    0,000 GHz   [79,06%]
   801685466 cycles      #    0,000 GHz   [80,14%]
   792515645 cycles      #    0,000 GHz   [80,37%]
   792876560 cycles      #    0,000 GHz   [80,37%]
   793921257 cycles      #    0,000 GHz   [80,08%]

0,306024538 seconds time elapsed
```

Los valores son significativamente diferentes, pero los porcentajes entre corchetes nos previenen de que se ha realizado un escalado. Por ejemplo, el primer contador ha estado contabilizando ciclos durante el 79,06% del tiempo. El valor obtenido en el contador se ha escalado dividiendo por 0,7906 para obtener el valor mostrado.

En este caso los contadores nos dan una aproximación, no un valor completamente fiable. Nos vale para evaluar mejoras en porcentajes significativos, pero no mejoras de un 1%, porque como vemos el escalado ya introduce un error de esa magnitud. Además en algunas mediciones el resultado dependerá del momento concreto en que se evalúen o de la carga del sistema en el momento de la medida. Para suavizar todos estos efectos estadísticos se puede ejecutar varias veces empleando la opción `-r`.

```
$ perf stat -r 10 -e cycles,cycles,cycles,cycles,cycles render_frame

Performance counter stats for 'render_frame' (10 runs):

   803926738 cycles      #    0,000 GHz   ( +- 0,15% ) [79,42%]
   804290331 cycles      #    0,000 GHz   ( +- 0,14% ) [79,66%]
   802303057 cycles      #    0,000 GHz   ( +- 0,17% ) [80,21%]
   797518018 cycles      #    0,000 GHz   ( +- 0,11% ) [80,59%]
   799832288 cycles      #    0,000 GHz   ( +- 0,19% ) [80,15%]

0,310143008 seconds time elapsed ( +- 0,39% )
```

Entre paréntesis se muestra la variación entre ejecuciones.

pecíficos. Realiza la misma prueba en tu ordenador para comprobar cuántos contadores genéricos tiene.

4.1.5. Métricas por hilo, por proceso o por CPU

Es posible contabilizar los eventos solo en un hilo, o en todos los hilos de un proceso, o en todos los procesos de una CPU, o de un conjunto de ellas. Por defecto `perf` contabiliza eventos del hilo principal incluyendo todos los subprocessos, creados con `fork()`, o hilos, creados con `pthread_create()`, lanzados durante la ejecución. Este comportamiento se implementa con un mecanismo de herencia de contadores que puede desactivarse con la opción `-i` de `perf stat`.

Alternativamente se puede recolectar datos de un conjunto de procesadores en lugar de un proceso concreto. Este modo se activa con la opción `-a` y opcionalmente complementado con la opción `-C`. Al utilizar la opción `-a` se activa la recolección de datos por CPU, pero por defecto se agregan todos los contadores de todas las CPU (recolección de datos a nivel de sistema). Con la opción `-C` podemos seleccionar la CPU o conjunto de CPUs de los que se recaban estadísticas. Por ejemplo, para recolectar el número de fallos de página en espacio de usuario de las CPUs 0 y 2 durante 5 segundos:

```
$ perf stat -a -e faults -C 0,2 sleep 5
Performance counter stats for 'sleep 5':

    233 faults

    5,001227158 seconds time elapsed
```

Nótese que utilizamos la orden `sleep` para no consumir ciclos y de esta forma no influir en la medida.

4.1.6. Muestreo de eventos

Además de contar eventos, `perf` puede realizar un muestreo similar a otros *profilers*. En este caso `perf record` recolecta datos en un archivo llamado `perf.data` que posteriormente puede analizarse con `perf report` o `perf annotate`.

El periodo de muestreo se especifica en número de eventos. Si el evento que se utiliza para el muestreo es `cycles` (es así por defecto) entonces el periodo tiene relación directa con el tiempo, pero en el caso general no tiene por qué. Incluso en el caso por defecto la relación con el tiempo no es lineal, en caso de que el procesador tenga activos modos de escalado de frecuencia.

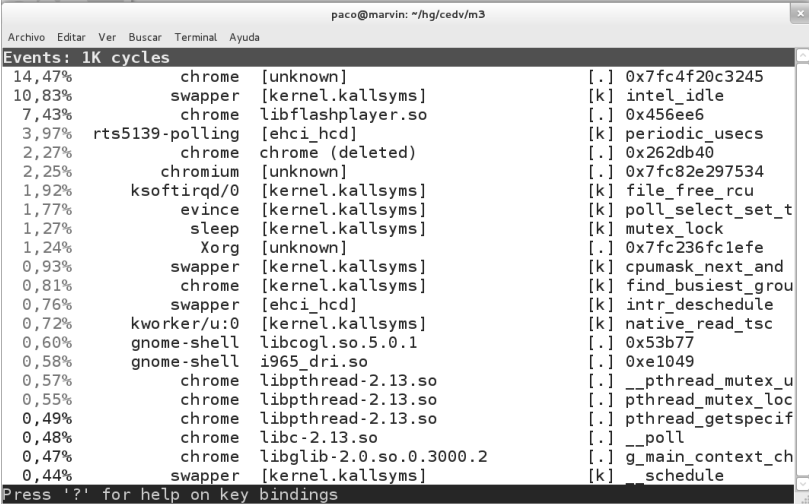
Por defecto `perf record` registra 1000 muestras por segundo y ajusta dinámicamente el periodo para que mantenga esta tasa. El usuario puede establecer una frecuencia de muestreo utilizando la opción `-F` o puede establecer un periodo fijo con la opción `-c`.

A diferencia de otros *profilers*, `perf record` puede recoger estadísticas a nivel del sistema completo o de un conjunto de CPUs concreto empleando las opciones `-a` y `-C` que ya hemos visto al explicar `perf stat`.

Es especialmente interesante el muestreo de la traza de llamada empleando la opción `-g`, aunque para que esta característica muestre resultados fiables es necesario mantener el convenio de *marcos de llamada* y no compilar con la opción `-fomit-frame-pointer`.

Para mostrar los resultados almacenados en `perf.data` puede emplearse `perf report`. Por ejemplo, a continuación recolectaremos datos de la actividad del sistema durante 5 segundos y mostraremos el resultado del perfilado.

```
$ perf record -a sleep 10
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.426 MB perf.data (~18612 samples)
]
$ perf report
```



The screenshot shows a terminal window titled 'paco@marvin: ~/hg/cedv/m3'. The output of the 'perf report' command is displayed, showing a list of events and their percentages. The top of the output indicates 'Events: 1K cycles'. The list includes various processes and kernel functions, such as 'chrome', 'swapper', 'ksoftirqd/0', and 'Xorg', with their respective percentages and kernel symbols.

Percentage	Process	Kernel Symbol	Address
14,47%	chrome	[unknown]	[.] 0x7fc4f20c3245
10,83%	swapper	[kernel.kallsyms]	[k] intel_idle
7,43%	chrome	libflashplayer.so	[.] 0x456ee6
3,97%	rts5139-polling	[ehci_hcd]	[k] periodic_usecs
2,27%	chrome	chrome (deleted)	[.] 0x262db40
2,25%	chromium	[unknown]	[.] 0x7fc82e297534
1,92%	ksoftirqd/0	[kernel.kallsyms]	[k] file_free_rcu
1,77%	evince	[kernel.kallsyms]	[k] poll_select_set_t
1,27%	sleep	[kernel.kallsyms]	[k] mutex_lock
1,24%	Xorg	[unknown]	[.] 0x7fc236fc1efe
0,93%	swapper	[kernel.kallsyms]	[k] cpumask_next_and
0,81%	chrome	[kernel.kallsyms]	[k] find_busiest_grou
0,76%	swapper	[ehci_hcd]	[k] intr_deschedule
0,72%	kworker/u:0	[kernel.kallsyms]	[k] native_read_tsc
0,60%	gnome-shell	libcogl.so.5.0.1	[.] 0x53b77
0,58%	gnome-shell	i965_dri.so	[.] 0xe1049
0,57%	chrome	libpthread-2.13.so	[.] __pthread_mutex_u
0,55%	chrome	libpthread-2.13.so	[.] pthread_mutex_loc
0,49%	chrome	libpthread-2.13.so	[.] pthread_getspecif
0,48%	chrome	libc-2.13.so	[.] __poll
0,47%	chrome	libgl-2.0.so.0.3000.2	[.] g_main_context_ch
0,44%	swapper	[kernel.kallsyms]	[k] __schedule

Press '?' for help on key bindings

Figura 4.1: Interfaz textual de `perf report`.

El muestreo permite analizar qué funciones se llevan la mayor parte del tiempo de ejecución y, en caso de muestrear también la traza de llamada permite identificar de forma rápida los cuellos de botella. No se trata de encontrar la función que más tiempo se lleva, sino de identificar funciones que merezca la pena optimizar. No tiene sentido optimizar una función que se lleva el 0,01 % del tiempo de ejecución, porque simplemente no se notaría.

Una vez identificada la función o las funciones susceptibles de mejora podemos analizarlas en mayor detalle, incluso a nivel del código ensamblador empleando `perf annotate símbolo`. También desde la interfaz de texto es posible examinar el código anotado seleccionando el símbolo y pulsando la tecla `a`.



Para poder utilizar las características de anotación del código de los perfiladores es necesario compilar el programa con información de depuración.

Para ilustrar la mecánica veremos un caso real. Ingo Molnar, uno de los principales desarrolladores de la infraestructura de perfilado de Linux, tiene multitud de mensajes en diversos foros sobre optimizaciones concretas que fueron primero identificadas mediante el uso de `perf`. Uno de ellos⁴ describe una optimización significativa de `git`, el sistema de control de versiones.

En primer lugar realiza una fase de análisis de una operación concreta que revela un dato intranquilizador. Al utilizar la operación de compactación `git gc` descubre un número elevado de ciclos de estancamiento (*stalled cycles*⁵):

```
$ perf record -e stalled-cycles -F 10000 ./git gc
$ perf report --stdio

# Events: 26K stalled-cycles
#
# Overhead  Command      Shared Object      Symbol
# .....
#
# 26.07%    git          git                [.] lookup_object
# 10.22%    git          libz.so.1.2.5      [.] 0xc43a
#  7.08%    git          libz.so.1.2.5      [.] inflate
#  6.63%    git          git                [.] find_pack_entry_one
#  5.37%    git          [kernel.kallsyms] [k] do_raw_spin_lock
#  4.03%    git          git                [.] lookup_blob
#  3.09%    git          libc-2.13.90.so    [.] __strlen_sse42
#  2.81%    git          libc-2.13.90.so    [.] __memcpy_sse3_back
```

Ingo descubre que la función `find_pack_entry_one()` se lleva un porcentaje significativo de los ciclos de estancamiento. Por tanto examina el contenido de esa función con `perf annotate`. Para poder extraer todo el beneficio de esta orden es interesante compilar el programa con información de depuración.

```
$ perf annotate find_pack_entry_one

Percent | Source code & Disassembly of git
-----|-----
:
```

⁴<http://thread.gmane.org/gmane.comp.version-control.git/172286>

⁵En las versiones actuales de `perf` habría que usar `stalled-cycles-frontend` en lugar de `stalled-cycles` pero mantenemos el texto del caso de uso original para no confundir al lector.

```

...
      :          int cmp = hashcmp(index + mi * stride, shal);
0.90 : 4b9264: 89 ee      mov    %ebp, %esi
0.45 : 4b9266: 41 0f af f2  imul  %r10d, %esi
2.86 : 4b926a: 4c 01 de      add  %r11, %rsi
53.34 : 4b926d: f3 a6      repz cmpsb %es:(%rdi), %ds:(%rsi)
14.37 : 4b926f: 0f 92 c0      setb %al
5.78 : 4b9272: 41 0f 97 c4  seta  %r12b
1.52 : 4b9276: 41 28 c4      sub  %al, %r12b

```

La mayoría de la sobrecarga está en la función `hashcmp()` que usa `memcmp()`, pero esta última se expande como instrucciones ensamblador por el propio compilador.

Ingo Molnar estudia el caso concreto. La función `hashcmp()` compara *hashes*, y por eso se utiliza `memcmp()`, pero si no coinciden el primer byte diferirá en el 99% de los casos. Por tanto modifica el programa para escribir la comparación manualmente, evitando entrar en la comparación para la mayor parte de los casos.

El resultado es realmente sorprendente. Antes de la optimización obtuvo estos números:

```

$ perf stat --sync --repeat 10 ./git gc
Performance counter stats for './git gc' (10 runs):

   2771.119892 task-clock          #    0.863 CPUs utilized      ( +-
   0.16% )
     1,813 context-switches      #    0.001 M/sec              ( +-
   3.06% )
       167 CPU-migrations        #    0.000 M/sec              ( +-
   2.92% )
    39,210 page-faults           #    0.014 M/sec              ( +-
   0.26% )
  8,828,405,654 cycles            #    3.186 GHz                ( +-
   0.13% )
  2,102,083,909 stalled-cycles   # 23.81% of all cycles are idle ( +-
   0.52% )
  8,821,931,740 instructions     #    1.00 insns per cycle
                                     #    0.24 stalled cycles per insn ( +-
   0.04% )
  1,750,408,175 branches         # 631.661 M/sec              ( +-
   0.04% )
    74,612,120 branch-misses     #    4.26% of all branches   ( +-
   0.07% )

  3.211098537 seconds time elapsed ( +- 1.52% )

```

La opción `-sync` hace que se ejecute una llamada `sync()` (vuelca los buffers pendientes de escritura de los sistemas de archivos) antes de cada ejecución para reducir el ruido en el tiempo transcurrido.

Después de la optimización el resultado es:

```

$ perf stat --sync --repeat 10 ./git gc
Performance counter stats for './git gc' (10 runs):

   2349.498022 task-clock          #    0.807 CPUs utilized      (
+- 0.15% )
     1,842 context-switches      #    0.001 M/sec              (
+- 2.50% )

```

```

    164 CPU-migrations # 0.000 M/sec (
      +- 3.67% )
    39,350 page-faults # 0.017 M/sec (
      +- 0.06% )
  7,484,317,230 cycles # 3.185 GHz (
      +- 0.15% )
  1,577,673,341 stalled-cycles # 21.08% of all cycles are idle (
      +- 0.67% )
11,067,826,786 instructions # 1.48 insns per cycle
                        # 0.14 stalled cycles per insn (
                        +- 0.02% )
  2,489,157,909 branches # 1059.442 M/sec (
      +- 0.02% )
  59,384,019 branch-misses # 2.39% of all branches (
      +- 0.22% )

  2.910829134 seconds time elapsed ( +- 1.39% )

```

La misma operación se aceleró en un 18%. Se han eliminado el 33% de los ciclos de estancamiento y la mayoría de ellos se han traducido en ahorro efectivo de ciclos totales y con ello en mejoras de velocidad.

Este ejemplo deja claro que las instrucciones ensamblador que emite el compilador para optimizar `memcpy()` no son óptimas para comparaciones pequeñas. La instrucción `repz cmpsb` requiere un tiempo de *setup* considerable durante el cual la CPU no hace nada más.

Otro efecto interesante que observa Ingo Molnar sobre esta optimización es que también mejora la predicción de saltos. Midiendo el evento `branch-misses` obtiene los siguientes resultados:

	branch-misses	% del total
Antes	74,612,120	4.26% (± 0.07%)
Después	59,384,019	2.39% (± 0.22%)

Tabla 4.1: Mejora en predicción de saltos.

Por alguna razón el bucle abierto es más sencillo de predecir por parte de la CPU por lo que produce menos errores de predicción.

No obstante es importante entender que estas optimizaciones corresponden a problemas en otros puntos (compilador que genera código subóptimo, y arquitectura que privilegia un estilo frente a otro). Por tanto se trata de optimizaciones con fecha de caducidad. Cuando se utilice una versión más reciente de GCC u otro compilador más agresivo en las optimizaciones esta optimización no tendrá sentido.

Un caso célebre similar fue la optimización del recorrido de listas en el kernel Linux⁶. Las listas son estructuras muy poco adecuadas para la memoria caché. Al no tener los elementos contiguos generan innumerables fallos de caché. Mientras se produce un fallo de caché el procesador está parcialmente parado puesto que necesita el dato de la memoria para operar. Por esta razón en Linux se empleó una optimización denominada *prefetching*. Antes de operar con un elemento se accede al siguiente. De esta forma mientras está operando con el

⁶<https://lwn.net/Articles/444336/>

elemento es posible ir transfiriendo los datos de la memoria a la caché.

Desgraciadamente los procesadores modernos incorporan sus propias unidades de *prefetch* que realizan un trabajo mucho mejor que el manual, puesto que no interfiere con el TLB. El propio Ingo Molnar reporta que esta optimización estaba realmente causando un impacto de 0,5%.

La lección que debemos aprender es que nunca se debe optimizar sin medir, que las optimizaciones dependen del entorno de ejecución, y que si el entorno de ejecución varía las optimizaciones deben reevaluarse.

4.1.7. Otras opciones de `perf`

Puede resultar útil también la posibilidad de contabilizar procesos o hilos que ya están en ejecución (opciones `-p` y `-t` respectivamente). A pesar de usar cualquiera de estas opciones se puede especificar un orden para limitar el tiempo de medición. En caso contrario mediría el proceso o hilo hasta su terminación.

También es posible generar gráficos de líneas temporales. Para ello es necesario utilizar la suborden `perf timechart record` para registrar los eventos de forma similar a como se hacía con `perf record` y posteriormente emplear `perf timechart` para generar el archivo `output.svg`. Este archivo puede editarse o convertirse a PDF con *inkscape*. El problema es que el tiempo de captura debe ser reducido o de lo contrario el archivo SVG se volverá inmanejable. No obstante es muy útil para detectar problemas de bloqueo excesivo. Por ejemplo, los datos de la figura 4.2 se grabaron con `perf timechart record -a sleep 1`.

Por último conviene citar la suborden `perf top` que permite monitorizar en tiempo real el sistema para analizar quién está generando más eventos.

4.1.8. Otros perfiladores

La tabla 4.2 muestra una colección de herramientas de perfilado disponibles en entornos GNU y GNU/Linux.

La mayoría de los perfiladores requieren compilar el programa de una manera especial. El más extendido y portable es GNU Profiler, incluido dentro de `binutils`, que es directamente soportado por el compilador de GNU. Si se compilan y se montan los programas con la opción `-pg` el programa quedará instrumentado para perfilado.

Paquete	Herramienta	Descripción
Valgrind Callgrind ⁷	kCacheGrind	Excelentes capacidades de representación gráfica.
Google Performance Tools ⁸	google-pprof	Permite perfilado de CPU y de memoria dinámica. Permite salida en formato <i>callgrind</i> para poder analizar con kCacheGrind.
GNU Profiler ⁹	gprof	Es una herramienta estándar pero ha ido perdiendo su utilidad conforme fueron surgiendo los perfiladores basados en PMU.
nVidia Visual Profiler	nvvp	Es específico para GPUs nVidia.
AMD APP Profiler	sprofile	Es específico para GPUs AMD/ATI Radeon.

Tabla 4.2: Herramientas de perfilado en GNU/Linux.

Todas las ejecuciones del programa generan un archivo `gmon.out` con la información recolectada, que puede examinarse con `gprof`. GNU Profiler utiliza muestreo estadístico sin ayuda de PMU. Esto lo hace muy portable pero notablemente impreciso.

Google Performance Tools aporta un conjunto de bibliotecas para perfilado de memoria dinámica o del procesador con apoyo de PMU. Por ejemplo, el perfilado de programas puede realizarse con la biblioteca `libprofiler.so`. Esta biblioteca puede ser cargada utilizando la variable de entorno `LD_PRELOAD` y activada mediante la definición de la variable de entorno `CPUPROFILE`. Por ejemplo:

```
$ LD_PRELOAD=/usr/lib/libprofiler.so.0 CPUPROFILE=prof.data \
./light-model-test
```

Esto genera el archivo `prof.data` con los datos de perfilado, que luego pueden examinarse con `google-pprof`. Entre otras capacidades permite representación gráfica del grafo de llamadas o compatibilidad con el formato de `kcachegrind`.

Una característica interesante de Google Performance Tools es la capacidad de realizar el perfilado solo para una sección concreta del código. Para ello, en lugar de definir la variable `CPUPROFILE` basta incluir en el código llamadas a las funciones `ProfilerStart()` y `ProfilerStop()`.

Para un desarrollador de videojuegos es destacable la aparición de perfiladores específicos para GPUs. Las propias GPUs tienen una PMU (*Performance Monitoring Unit*) que permite recabar información de contadores específicos. De momento en el mundo del software libre han emergido nVidia Visual Profiler, AMD APP Profiler y extensiones de Intel a `perf` para utilizar los contadores de la GPU (`perf gpu`). Probablemente en un futuro cercano veremos estas extensiones incorporadas en la distribución oficial de `linux-tools`.

GPU Profilers

De momento solo nVidia proporciona un profiler con capacidades gráficas sobre GNU/Linux. AMD APP Profiler funciona en GNU/Linux pero no con interfaz gráfica.

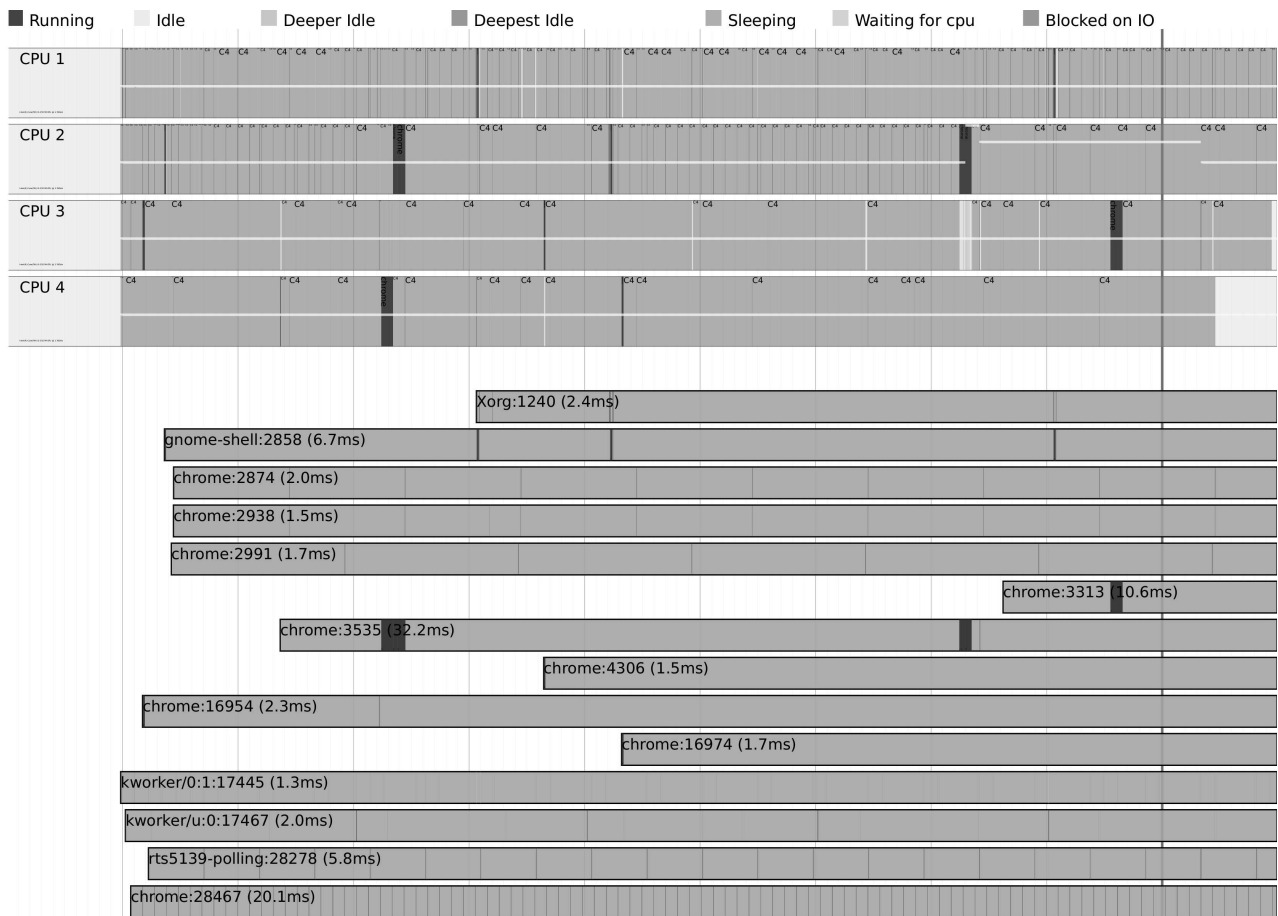


Figura 4.2: Ejemplo de `perf timechart`.

4.2. Optimizaciones del compilador

Los compiladores modernos implementan un enorme abanico de optimizaciones. Con frecuencia son tan eficientes como el código ensamblador manualmente programado. Por esta razón es cada vez más raro encontrar fragmentos de código ensamblador en programas bien optimizados.

El lenguaje C++, y su ancestro C son considerados como lenguajes de programación de sistemas. Esto se debe a que permiten acceso a características de muy bajo nivel, hasta el punto de que algunos autores lo consideran un *ensamblador portable*. Los punteros no dejan de ser una forma de expresar direccionamiento indirecto, así como el operador de indexación no deja de ser una expresión de los modos de direccionamiento relativo.

C fue diseñado con el objetivo inicial de programar un sistema operativo. Por este motivo, desde las primeras versiones incorpora carac-

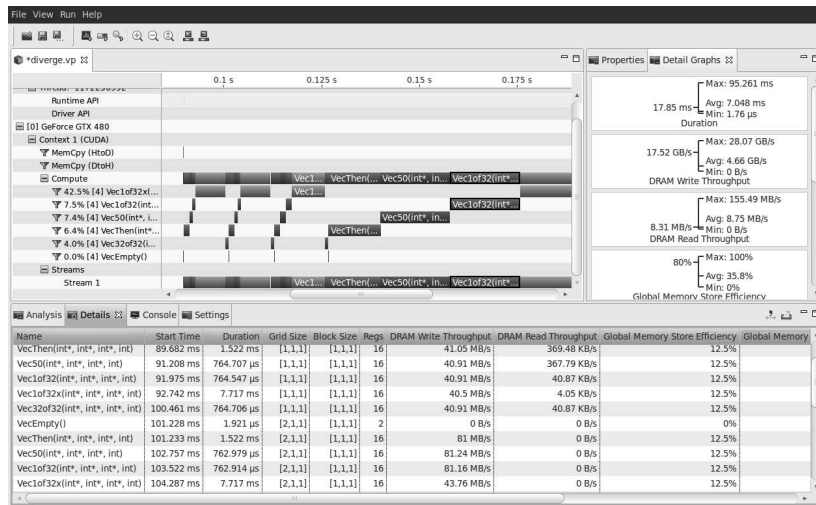


Figura 4.3: Aspecto de la interfaz de nVidia Visual Profiler.

terísticas de muy bajo nivel que permite dirigir al compilador para generar código más eficiente. Variables registro, funciones en línea, paso por referencia, o plantillas son algunas de las características que nos permiten indicar al compilador cuándo debe esforzarse en buscar la opción más rápida. Sin embargo, la mayoría de las construcciones son simplemente indicaciones o sugerencias, que el compilador puede ignorar libremente si encuentra una solución mejor. En la actualidad tenemos compiladores libres maduros con capacidades comparables a los mejores compiladores comerciales, por lo que frecuentemente las indicaciones del programador son ignoradas.

4.2.1. Variables registro

Los más viejos del lugar recordarán un modificador opcional para las variables denominado `register`. Este modificador indica al compilador que se trata de una variable especialmente crítica, por lo que sugiere almacenarla en un registro del procesador. Era frecuente ver código como éste:

Listado 4.1: Utilización arcaica de `register` para sumar los 1000 primeros números naturales.

```
1 register unsigned i, sum = 0;
2 for (i=1; i<1000; ++i)
3   sum += i;
```

Esta palabra clave está en desuso porque los algoritmos de asignación de registros actuales son mucho mejores que la intuición humana. Pero además, aunque se utilizara, sería totalmente ignorada por el compilador. La mera aparición de `register` en un programa debe ser

considerada como un *bug*, porque engaña al lector del programa haciéndole creer que dicha variable será asignada a un registro, cuando ese aspecto está fuera del control del programador.

4.2.2. Código estático y funciones *inline*

Ya se ha comentado el uso del modificador `inline` en el módulo 1. Sirve para indicar al compilador que debe replicar el código de dicha función cada vez que aparezca una llamada. Si no se hiciera generaría código independiente para la función, al que salta mediante una instrucción de llamada a subrutina. Sin embargo no siempre es posible la sustitución en línea del código y además el compilador es libre de hacer sustitución en línea de funciones aunque no estén marcadas como `inline`. Veamos un ejemplo:

Listado 4.2: Ejemplo sencillo de función apropiada para la expansión en línea.

```

1 int sum(int* a, unsigned size)
2 {
3     int ret = 0;
4     for (int i=0; i<size; ++i) ret += a[i];
5     return ret;
6 }
7
8 int main() {
9     int a[] = { 1, 2, 3, 4, 5};
10    return sum(a, sizeof(a)/sizeof(a[0]));
11 }
```

Compilemos este ejemplo con máximo nivel de optimización. No es necesario dominar el ensamblador de la arquitectura `x86_64` para entender la estructura.

```
$ gcc -S -O3 -c inl.cc
```

El resultado es el siguiente:

Listado 4.3: Resultado de la compilación del ejemplo anterior.

```

1     .file    "inl.cc"
2     .text
3     .p2align 4,,15
4     .globl  _Z3sumPij
5     .type   _Z3sumPij, @function
6 _Z3sumPij:
7     .LFB0:
8     .cfi_startproc
9     xorl   %eax, %eax
10    testl  %esi, %esi
11    pushq  %rbx
12    .cfi_def_cfa_offset 16
13    .cfi_offset 3, -16
14    je     .L2
15    movq   %rdi, %r8
16    movq   %rdi, %rcx
```

```

17     andl    $15, %r8d
18     shrq   $2, %r8
19     negq   %r8
20     andl    $3, %r8d
21     cmpl   %esi, %r8d
22     cmova  %esi, %r8d
23     xorl   %edx, %edx
24     testl  %r8d, %r8d
25     movl   %r8d, %ebx
26     je     .L11
27     .p2align 4,,10
28     .p2align 3
29 .L4:
30     addl   $1, %edx
31     addl   (%rcx), %eax
32     addq   $4, %rcx
33     cmpl   %r8d, %edx
34     jb     .L4
35     cmpl   %r8d, %esi
36     je     .L2
37 .L3:
38     movl   %esi, %r11d
39     subl   %r8d, %r11d
40     movl   %r11d, %r9d
41     shrl   $2, %r9d
42     leal   0(%r9,4), %r10d
43     testl  %r10d, %r10d
44     je     .L6
45     pxor   %xmm0, %xmm0
46     leaq   (%rdi,%rbx,4), %r8
47     xorl   %ecx, %ecx
48     .p2align 4,,10
49     .p2align 3
50 .L7:
51     addl   $1, %ecx
52     paddd  (%r8), %xmm0
53     addq   $16, %r8
54     cmpl   %r9d, %ecx
55     jb     .L7
56     movdqa %xmm0, %xmm1
57     addl   %r10d, %edx
58     psrldq $8, %xmm1
59     paddd  %xmm1, %xmm0
60     movdqa %xmm0, %xmm1
61     psrldq $4, %xmm1
62     paddd  %xmm1, %xmm0
63     movd   %xmm0, -4(%rsp)
64     addl   -4(%rsp), %eax
65     cmpl   %r10d, %r11d
66     je     .L2
67 .L6:
68     movslq %edx, %rcx
69     leaq   (%rdi,%rcx,4), %rcx
70     .p2align 4,,10
71     .p2align 3
72 .L9:
73     addl   $1, %edx
74     addl   (%rcx), %eax
75     addq   $4, %rcx
76     cmpl   %edx, %esi
77     ja     .L9
78 .L2:
79     popq   %rbx
80     .cfi_remember_state
81     .cfi_def_cfa_offset 8

```

```

82     ret
83 .L11:
84     .cfi_restore_state
85     movl  %r8d, %eax
86     jmp  .L3
87     .cfi_endproc
88 .LFE0:
89     .size  _Z3sumPij, .-_Z3sumPij
90     .section  .text.startup,"ax",@progbits
91     .p2align 4,,15
92     .globl  main
93     .type   main, @function
94 main:
95 .LFB1:
96     .cfi_startproc
97     movl  $15, %eax
98     ret
99     .cfi_endproc
100 .LFE1:
101     .size  main, .-main
102     .ident  "GCC: (Debian 4.6.3-1) 4.6.3"
103     .section  .note.GNU-stack,"",@progbits

```

El símbolo `_Z3sumPij` corresponde a la función `sum()` aplicando las reglas de *mangling*. Podemos decodificarlo usando `c++filt`.

```

$ echo _Z3sumPij | c++filt
sum(int*, unsigned int)

```

El símbolo codifica la signatura entera de la función. Sin embargo no se utiliza en ninguna parte. Observemos en detalle las instrucciones de la función `main()` eliminando las directivas no necesarias.

Listado 4.4: Código de la función `main()` del ejemplo anterior.

```

1 main:
2     movl  $15, %eax
3     ret

```

El código se limita a retornar el resultado final, un 15. El compilador ha realizado la expansión en línea y sucesivamente ha aplicado propagación de constantes y evaluación de expresiones constantes para simplificarlo a lo mínimo. Y entonces ¿por qué aparece el código de la función `sum()`?

El motivo es simple, la función puede ser necesaria desde otra unidad de compilación. Por ejemplo, supóngase que en otra unidad de compilación aparece el siguiente código.

Listado 4.5: Otra unidad de compilación puede requerir la función `sum()`.

```

1 #include <iostream>
2
3 using namespace std;
4
5 int sum(int* a, unsigned sz);
6

```

```

7 struct A {
8     A() {
9         int a[] = { 1, 1, 1, 1 };
10        cout << sum(a, 4) << endl;
11    }
12 };
13
14 A a;

```

¿Significa eso que el código no utilizado ocupa espacio en el ejecutable? Podemos responder a esa pregunta compilando el ejemplo inicial y examinando los símbolos con `nm`:

```

$ g++ -O3 -o inl inl.cc
$ nm --dynamic inl
                 w __Jv_RegisterClasses
                 w __gmon_start__
                 U __libc_start_main

```

No ha quedado ningún símbolo reconocible. El montador ha optimizado el ejecutable para que solo contenga los símbolos utilizados. ¿Y si un plugin necesita la función `sum()`? La respuesta la conocemos, aunque no conocíamos los detalles, basta montar con la opción `-rdynamic`:

```

$ g++ -O3 -rdynamic -o inl inl.cc
$ nm --dynamic inl
00000000004008d8 R __IO_stdin_used
                 w __Jv_RegisterClasses
0000000000400720 T __Z3sumPij
0000000000600c00 A __bss_start
0000000000600bf0 D __data_start
                 w __gmon_start__
00000000004007f0 T __libc_csu_fini
0000000000400800 T __libc_csu_init
                 U __libc_start_main
0000000000600c00 A __edata
0000000000600c10 A __end
00000000004008c8 T __fini
00000000004005f8 T __init
0000000000400638 T __start
0000000000600bf0 W data_start
0000000000400630 T main

```

Si el código está en una biblioteca dinámica el montador no eliminará los símbolos porque no puede determinar si se usarán en el futuro. Sin embargo algunas funciones solo serán necesarias en un archivo concreto. En ese caso pueden declararse como `static`, lo que evita que se exporte el símbolo.



La palabra `static` es seguramente la palabra clave más sobrecargada de C++. Aplicado a las funciones o las variables globales quiere decir que el símbolo no se exporta. Aplicado a un método quiere decir que se trata de un método de clase, no aplicado a una instancia concreta. Aplicado a una variable local quiere decir que se almacena en la zona de datos estáticos.

Listado 4.6: Esta biblioteca solo exporta la función `sum10()`.

```
1 static int sum(int* a, unsigned size)
2 {
3     int ret = 0;
4     for (int i=0; i<size; ++i) ret += a[i];
5     return ret;
6 }
7
8 int sum10(int* a)
9 {
10    return sum(a,10);
11 }
```

La expansión en línea de las funciones no siempre produce un código óptimo. Para ilustrar este punto vamos a utilizar un ejemplo ya conocido de la sección anterior. En dicha sección describíamos un caso de optimización de `git` de Ingo Molnar. Simplificando al máximo el caso se trataba del siguiente fragmento de código:

Listado 4.7: Funciones críticas en la ejecución de `git gc`.

```
1 #include <string.h>
2
3 static inline int hashcmp(const char *sha1, const char *sha2)
4 {
5     return memcmp(sha1, sha2, 20);
6 }
7
8 extern const char null_sha1[20] __attribute__((aligned(8)));
9 static inline int is_null_sha1(const char *sha1)
10 {
11     return !hashcmp(sha1, null_sha1);
12 }
13
14
15 int ejemplo(char* sha1, char* index, unsigned mi)
16 {
17     int cmp, i;
18     for (i=0; i<mi; ++i) {
19         cmp = hashcmp(index + i * 1024, sha1);
20         if (cmp == 0) return 0;
21     }
22     return cmp;
23 }
```

Estas funciones, que eran expandidas en línea por el compilador, exhibían un comportamiento anómalo con respecto a los ciclos de estancamiento y a la predicción de saltos. Por lo que Ingo propone la siguiente optimización:

Listado 4.8: Optimización de funciones críticas en la ejecución de git gc.

```

1 static inline int hashcmp(const char *shal, const char *sha2)
2 {
3     int i;
4
5     for (i = 0; i < 20; i++, shal++, sha2++) {
6         if (*shal != *sha2)
7             return *shal - *sha2;
8     }
9
10    return 0;
11 }
12
13 extern const char null_shal[20];
14 static inline int is_null_shal(const char *shal)
15 {
16     return !hashcmp(shal, null_shal);
17 }
18
19
20 int ejemplo(char* shal, char* index, unsigned mi)
21 {
22     int cmp, i;
23     for (i=0; i<mi; ++i) {
24         cmp = hashcmp(index + i * 1024, shal);
25         if (cmp == 0) return 0;
26     }
27     return cmp;
28 }

```

Lo interesante de este caso de estudio es que partió de un análisis con el perfilador que determinaba que la función `memcmp()` era subóptima para comparaciones cortas. La función `memcmp()` se expandía automáticamente en línea en forma de un puñado de instrucciones ensamblador. Una de ellas, `repz cmpsb`, era identificada como la culpable del problema. Actualmente ni `gcc-4.6` ni `clang` expanden automáticamente la función `memcmp()`. Por tanto el resultado es bien distinto. Empleando `perf stat -r 100 -e cycles:u` se obtienen los resultados que muestra la tabla 4.3.

Compilador	Ciclos	Ciclos Opt.	Mejora
gcc-4.6	192458	190022	1,3 %
clang-3.0	197163	198232	-0,5 %
llvm-gcc-4.6	189164	191826	-1,4 %

Tabla 4.3: Resultados de la optimización de Ingo Molnar con compiladores actuales (100 repeticiones).

El mejor resultado lo obtiene `llvm-gcc` con el caso sin optimizar. El caso de `clang` genera resultados absolutamente comparables, dentro de los márgenes de error de `perf`. En cualquiera de los casos el resultado es mucho menos significativo que los resultados que obtuvo Ingo Molnar. Una optimización muy efectiva en un contexto puede no ser tan efectiva en otro, y el contexto es siempre cambiante (nuevas versiones de los compiladores, nuevas arquitecturas, etc.).

4.2.3. Eliminación de copias

En la mayor parte del estándar de C++ se suele indicar que el compilador tiene libertad para optimizar siempre que el resultado se comporte *como si* esas optimizaciones no hubieran tenido lugar. Sin embargo el estándar permite además un rango de optimizaciones muy concreto pero con gran impacto en prestaciones, que pueden cambiar el comportamiento de un programa. En [18], sección 12.8, § 32 introduce la noción de *copy elision*. Lo que sigue es una traducción literal del estándar.

Cuando se cumplen determinados criterios una implementación puede omitir la llamada al constructor de copia o movimiento de un objeto, incluso cuando el constructor y/o destructor de dicho objeto tienen efectos de lado. En estos casos la implementación simplemente trata la fuente y el destino de la operación de copia o movimiento omitida como dos formas diferentes de referirse al mismo objeto, y la destrucción de dicho objeto ocurre cuando ambos objetos hubieran sido destruidos sin la optimización. Esta elisión de las operaciones de copia o movimiento, denominada elisión de copia, se permite en las siguientes circunstancias (que pueden ser combinadas para eliminar copias múltiples):

- *En una sentencia return de una función cuyo tipo de retorno sea una clase, cuando la expresión es el nombre de un objeto automático no volátil (que no sea un parámetro de función o un parámetro de una cláusula catch) con el mismo tipo de retorno de la función (que no puede ser const ni volatile), la operación de copia o movimiento puede ser omitida mediante la construcción directa del objeto automático en el propio valor de retorno de la función.*
- *En una expresión throw, cuando el operando es el nombre de un objeto automático no volátil (que no sea un parámetro de función o un parámetro de una cláusula catch) cuyo ámbito de declaración no se extienda más allá del final del bloque try más interior que contenga a dicha expresión (si es que existe), la operación de copia o movimiento desde el operando hasta el objeto excepción puede ser omitida mediante la construcción del objeto automático directamente sobre el objeto excepción.*
- *Cuando un objeto temporal de clase que no ha sido ligado a una referencia sería copiado o movido a un objeto con la misma calificación de const/volatile, la operación de copia o movimiento puede ser omitida construyendo el temporal directamente sobre el destino de la copia o movimiento.*
- *Cuando la declaración de excepción en una cláusula catch declara un objeto del mismo tipo (salvo por modificadores const o volatile) como el objeto excepción, la operación de copia o movimiento puede ser omitida tratando la declaración de excepción como un alias del objeto excepción siempre que el significado del programa no sea cambiado salvo por la ejecución de constructores y destructores del objeto de la declaración de excepción.*

```

1     class Thing {
2         public:
3             Thing();
4             ~Thing();
5             Thing(const Thing&);
6         };
7
8     Thing f() {
9         Thing t;
10        return t;
11    }
12
13    Thing t2 = f();

```

Aquí los criterios de elisión pueden combinarse para eliminar dos llamadas al constructor de copia de *Thing*: la copia del objeto automático local *t* en el objeto temporal para el valor de retorno de la función *f()* y la copia de ese objeto temporal al objeto *t2*. Por tanto la construcción del objeto local *t* puede verse como la inicialización directa del objeto *t2*, y la destrucción de dicho objeto tendrá lugar al terminar el programa. Añadir un constructor de movimiento a *Thing* tiene el mismo efecto, en cuyo caso es el constructor de movimiento del objeto temporal a *t2* el que se elide.

Copy elision es un concepto que incluye dos optimizaciones frecuentes en compiladores de C++: *Return Value Optimization (RVO)* (tercera circunstancia contemplada en el estándar) y *Named Return Value Optimization (NRVO)* (primera circunstancia contemplada en el estándar).

4.2.4. Volatile

Las optimizaciones del compilador pueden interferir con el funcionamiento del programa, especialmente cuando necesitamos comunicarnos con periféricos. Así por ejemplo, el compilador es libre de reordenar y optimizar las operaciones mientras mantenga una equivalencia funcional. Así, por ejemplo este caso se encuentra no pocas veces en código de videojuegos caseros para consolas.

```

1 void reset(unsigned& reg)
2 {
3     reg = 1;
4     for(int i=0; i<1000000; ++i);
5     reg = 0;
6 }

```

El programador piensa que el bucle implementa un retardo y por tanto la función permite generar un pulso en el bit menos significativo. Compilando el ejemplo con máximo nivel de optimización obtenemos lo siguiente:

```

1 _Z5resetRj:
2     movl    $0, (%rdi)
3     ret

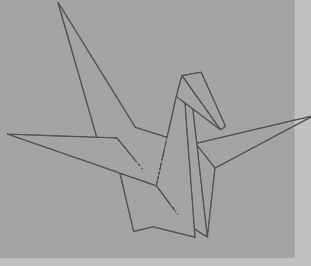
```

El compilador ha eliminado todo hasta el punto de que ni siquiera escribe el pulso. Una forma sencilla de corregir este comportamiento es declarar el contador `i` y el registro `reg` como `volatile`. Esto indica al compilador que no debe hacer optimizaciones con respecto a esas variables. Otra forma sería sustituir el bucle de espera por una llamada a función (por ejemplo `usleep(10)`).

4.3. Conclusiones

La optimización de programas es una tarea sistemática, pero a la vez creativa. Toda optimización parte de un análisis cuantitativo previo, normalmente mediante el uso de perfiladores. Existe un buen repertorio de herramientas que nos permite caracterizar las mejores oportunidades, pero no todo lo que consume tiempo está en nuestra mano cambiarlo. Las mejores oportunidades provienen de la reestructuración de los algoritmos o de las estructuras de datos.

Por otro lado el programador de videojuegos deberá optimizar para una plataforma o conjunto de plataformas que se identifican como objetivo. Algunas de las optimizaciones serán específicas para estas plataformas y deberán re-evaluarse cuando el entorno cambie.



Capítulo 5

Validación y pruebas

David Villa Alises

La programación, igual que cualquier otra disciplina técnica, debería ofrecer garantías sobre los resultados. La formalización de algoritmos ofrece garantías indiscutibles y se utiliza con éxito en el ámbito de los algoritmos numéricos y lógicos. Sin embargo, el desarrollo de software en muchos ámbitos está fuertemente ligado a requisitos que provienen directamente de necesidades de un cliente. En la mayoría de los casos, esas necesidades no se pueden formalizar dado que el cliente expresa habitualmente requisitos ambiguos o incluso contradictorios. El desarrollo de software no puede ser ajeno a esa realidad y debe integrar al cliente de forma que pueda ayudar a validar, refinar o rectificar la funcionalidad del sistema durante todo el proceso.

Un programador responsable comprueba que su software satisface los requisitos del cliente, comprueba los casos típicos y se asegura que los errores detectados (ya sea durante el desarrollo o en producción) se resuelven y no vuelven a aparecer. Es imposible escribir software perfecto (a día de hoy) pero un programador realmente profesional escribe código limpio, legible, fácil de modificar y adaptar a nuevas necesidades. En este capítulo veremos algunas técnicas que pueden ayudar a escribir código más limpio y robusto.

5.1. Programación defensiva

La expresión «programación defensiva» se refiere a las técnicas que ayudan al programador a evitar, localizar y depurar fallos, especialmente aquellos que se producen en tiempo de ejecución. En muchas

situaciones, especialmente con lenguajes como C y C++, el programa puede realizar una operación ilegal que puede terminar con la ejecución del proceso por parte del sistema operativo. El caso más conocido en este sentido se produce cuando se dereferencia un puntero que apunta a memoria fuera de los límites reservados para ese proceso: el resultado es el fatídico mensaje *segmentation fault* (abreviado como SEGFAULT). Cuando esa situación no ocurre en todos los casos sino que aparece esporádicamente, encontrar la causa del problema puede ser realmente complicado y puede llevar mucho tiempo. Para este tipo de problemas la depuración *postmortem* es una gran ayuda, pero antes de llegar a la autopsia, hay algunas medidas preventivas que podemos tomar: el control de invariantes.

En programación, una invariante es un predicado que asumimos como cierto antes, durante y después de la ejecución de un bloque de código (típicamente una función o método). Definir invariantes en nuestras funciones puede ahorrar mucho tiempo de depuración porque tenemos garantías de que el problema está limitado al uso correcto de la función que corresponda.

Muy ligado al concepto de invariante existe una metodología denominada «diseño por contrato». Se trata de un método para definir la lógica de una función, objeto u otro componente de modo que su interfaz no depende solo de los tipos de sus parámetros y valor de retorno. Se añaden además predicados que se evalúan antes (pre-condiciones) y después (post-condiciones) de la ejecución del bloque de código. Así, la interfaz de la función es mucho más rica, el valor del parámetro además de ser del tipo especificado debe tener un valor que cumpla con restricciones inherentes al problema.

Listado 5.1: Una función que define una *invariante* sobre su parámetro

```
1 double sqrt(double x) {
2     assert(x >= 0);
3     [...]
4 }
```

Normalmente el programador añade comprobaciones que validan los datos de entrada procedentes de la interfaz con el usuario. Se trata principalmente de convertir y verificar que los valores obtenidos se encuentran dentro de los rangos o tengan valores según lo esperado. Si no es así, se informa mediante la interfaz de usuario que corresponda. Sin embargo, cuando se escribe una función que va a ser invocada desde otra parte, no se realiza una validación previa de los datos de entrada ni tampoco de los producidos por la función. En condiciones normales podemos asumir que la función va a ser invocada con los valores correctos, pero ocurre que un error en la lógica del programa o un simple error-por-uno puede implicar que utilicemos incorrectamente nuestras propias funciones, provocando errores difíciles de localizar.

La herramienta más simple, a la vez que potente, para definir invariantes, pre-condiciones o post-condiciones es la función `assert()`¹,

¹En C++, la función `assert()` se encuentra en el fichero de cabecera `<cassert>`.

«Error por uno»

Se denomina así a los errores (*bugs*) debidos a comprobaciones incorrectas ('>' por '>=', '<' por '<=' o viceversa), en la indexación de vectores en torno a su tamaño, iteraciones de bucles, etc. Estos casos deben ser objeto de testing concienzudo.

que forma parte de la librería estándar de prácticamente todos los lenguajes modernos. `assert()` sirve, tal como indica su nombre, para definir aserciones, que en el caso de C++ será toda expresión que pueda ser evaluada como cierta. El siguiente listado es un ejemplo mínimo de usa aserción. Se muestra también el resultado de ejecutar el programa cuando la aserción falla:

Listado 5.2: `assert-argc.cc`: Un ejemplo sencillo de `assert()`

```
1 #include <cassert>
2
3 int main(int argc, char *argv[]) {
4     assert(argc == 2);
5     return 0;
6 }
```

```
$ ./assert-argc hello
$ ./assert-argc
assert-argc: assert-argc.cc:4: int main(int, char**): Assertion `argc
== 2' failed.
Abortado
```

Veamos algunos usos habituales de `assert()`

- Validar los parámetros de una función (pre-condiciones). Por ejemplo, comprobar que una función recibe un puntero no nulo:

```
1 void Inventory::add(Weapon* weapon) {
2     assert(weapon);
3     [...]
4 }
```

- Comprobar que el estado de un objeto es consistente con la operación que se está ejecutando, ya sea como pre-condición o como post-condición.
- Comprobar que un algoritmo produce resultados consistentes. Este tipo de post-condiciones se llaman a menudo *sanity checks*.
- Detectar condiciones de error irrecuperables.

```
1 void Server::bind(int port) {
2     assert(port > 1024);
3     assert(not port_in_use(port));
4     [...]
5 }
```

5.1.1. Sobrecarga

Las aserciones facilitan la depuración del programa porque ayudan a localizar el punto exacto donde se desencadena la inconsistencia. Por eso deberían incluirse desde el comienzo de la implementación. Sin embargo, cuando el programa es razonablemente estable, las

aserciones siempre se cumplen (o así debería ser). En una versión de producción las aserciones ya no son útiles² y suponen una sobrecarga que puede afectar a la eficiencia del programa.

Obviamente, eliminar «a mano» todas las aserciones no parece muy cómodo. La mayoría de los lenguajes incorporan algún mecanismo para desactivarlas durante la compilación. En C/C++ se utiliza el preprocesador. Si la constante simbólica `NDEBUG` está definida la implementación de `assert()` (que en realidad es una macro de preprocesador) se substituye por una sentencia vacía de modo que el programa que se compila realmente no tiene absolutamente nada referente a las aserciones.

En los casos en los que necesitamos hacer aserciones más complejas, que requieran variables auxiliares, podemos aprovechar la constante `NDEBUG` para eliminar también ese código adicional cuando no se necesite:

```
1 [...]
2 #ifndef NDEBUG
3 vector<int> values = get_values();
4 assert(values.size());
5 #endif
```

Aunque esta contante se puede definir simplemente con `#define NDEBUG`, lo más cómodo y aconsejable es utilizar el soporte que los compiladores suelen ofrecer para definir contantes en línea de comandos. En el caso de `g++` se hace así:

```
$ g++ -DNDEBUG main.cc
```

Definir la constante en el código, aparte de ser incómodo cuando se necesita activar/desactivar con frecuencia, puede ser confuso porque podría haber ficheros que se preprocesan antes de que la constante sea definida.

5.2. Desarrollo ágil

El desarrollo ágil de software trata de reducir al mínimo la burocracia típica de las metodologías de desarrollo tradicionales. Se basa en la idea de que «el software que funciona es la principal medida de progreso». El desarrollo ágil recoge la herencia de varias corrientes de finales de los años 90 como Scrum o la programación extrema y todas esas ideas se plasmaron en el llamado *manifiesto ágil*:

Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

²Por contra, algunos autores como Tony Hoare, defienden que en la versión de producción es dónde más necesarias son las aserciones.



Figura 5.1: Kent Beck, uno de los principales creadores de *eXtreme programming*, TDD y los métodos ágiles.

- *Individuos e interacciones* sobre procesos y herramientas.
- *Software funcionando* sobre documentación extensiva.
- *Colaboración con el cliente* sobre negociación contractual.
- *Respuesta ante el cambio* sobre seguir un plan.

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.

Las técnicas de desarrollo ágil pretenden entregar valor al cliente pronto y a menudo, es decir, priorizar e implementar las necesidades expresadas por el cliente para ofrecerle un producto que le pueda resultar útil desde el comienzo. También favorecen la adopción de cambios importantes en los requisitos, incluso en las últimas fases del desarrollo.

5.3. TDD

Una de las técnicas de desarrollo ágil más efectiva es el Desarrollo Dirigido por Pruebas o TDD (Test Driven Development). La idea básica consiste en empezar el proceso escribiendo pruebas que representen directamente requisitos del cliente. Algunos autores creen que el término «ejemplo» describe mejor el concepto que «prueba». Una prueba es un pequeño bloque de código que se ejecuta sin ningún tipo de interacción con el usuario (ni entrada ni salida) y que determina de forma inequívoca (la prueba pasa o falla) si el requisito correspondiente se está cumpliendo.

En el desarrollo de software tradicional las pruebas se realizan una vez terminado el desarrollo asumiendo que desarrollo y pruebas son fases estancas. Incluso en otros modelos como el iterativo, en espiral o el prototipado evolutivo las pruebas se realizan después de la etapa de diseño y desarrollo, y en muchas ocasiones por un equipo de programadores distinto al que ha escrito el código.

5.3.1. Las pruebas primero

Con TDD la prueba es el primer paso que desencadena todo el proceso de desarrollo. En este sentido, las pruebas no son una mera herramienta de *testing*. Las pruebas se utilizan como un medio para capturar y definir con detalle los requisitos del usuario, pero también como ayuda para obtener un diseño consistente evitando añadir complejidad innecesaria. Hacer un desarrollo dirigido por pruebas acota el trabajo a realizar: si todas las pruebas pasan, el programa está terminado, algo que puede no resultar trivial con otros modelos de desarrollo.

Este proceso resulta muy útil para evitar malgastar tiempo y esfuerzo añadiendo funcionalidad que en realidad no se ha solicitado.

Este concepto se conoce como YAGNI (*You Ain't Gonna Need It*) y aunque a primera vista pueda parecer una cuestión trivial, si se analiza detenidamente, puede suponer un gran impacto en cualquier proyecto. Es frecuente que los programadores entusiastas y motivados por la tarea acaben generando un diseño complejo plasmado en una gran cantidad de código difícil de mantener, mejorar y reparar.

5.3.2. rojo, verde, refactorizar

Cada uno de los requisitos identificados debe ser analizado hasta obtener una serie de escenarios que puedan ser probados de forma independiente. Cada uno de esos escenarios se convertirá en una prueba. Para cada uno de ellos:

- Escribe la prueba haciendo uso de las interfaces del sistema (¡es probable que aún no existan!) y ejecútala. La prueba debería fallar y debes comprobar que es así (**rojo**).
- A continuación escribe el código de producción mínimo necesario para que la prueba pase (**verde**). Ese código «mínimo» debe ser solo el imprescindible, lo más simple posible, hasta el extremo de escribir métodos que simplemente retornan el valor que la prueba espera³. Eso ayuda a validar la interfaz y confirma que la prueba está bien especificada. Pruebas posteriores probablemente obligarán a modificar el código de producción para que pueda considerar todas las posibles situaciones. A esto se le llama «triangulación» y es la base de TDD: Las pruebas dirigen el diseño.
- Por último **refactoriza** si es necesario. Es decir, revisa el código de producción y elimina cualquier duplicidad. También es el momento adecuado para renombrar tipos, métodos o variables si ahora se tiene más claro cuál es su objetivo real. Por encima de cualquier otra consideración el código debe expresar claramente la *intención del programador*. Es importante refactorizar tanto el código de producción como las propias pruebas.

Este sencillo método de trabajo (el algoritmo TDD) favorece que los programadores se concentren en lo que realmente importa: satisfacer los requisitos del usuario. También ayuda al personal con poca experiencia en el proyecto a decidir *cuál es el próximo paso* en lugar de divagar o tratando de «mejorar» el programa añadiendo funcionalidades no solicitadas.

5.4. Tipos de pruebas

Hay muchas formas de clasificar las pruebas, y todo lo referente al *testing* tradicional es aplicable aquí, aunque quizá de un modo diferente: pruebas de caja negra y blanca, pruebas de aceptación, integración,

³Kent Beck se refiere a esto con la expresión "Fake until make it".

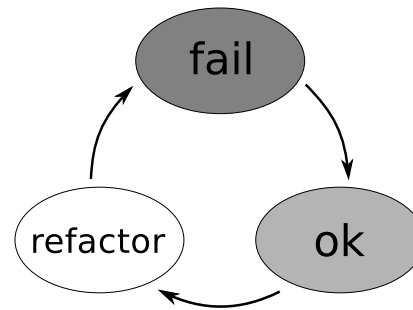


Figura 5.2: Algoritmo TDD

sistema, unitarias y largo etcétera. En el contexto de las metodologías ágiles podemos concretar los siguientes tipos de pruebas [9]:

De aceptación Idealmente debería estar especificado por el cliente o al menos por un analista con la ayuda del cliente. Se expresa en términos del dominio de la aplicación, sin detalles de implementación. Esto incluye los test no funcionales, es decir, aquellos que expresan requisitos no relacionados con los resultados obtenidos sino sobre cuestiones como tiempo de ejecución, consumo de energía, etc.

De sistema Es un test que utiliza el sistema completo, desde el interfaz de usuario hasta la base de datos, y lo hace del mismo modo que lo harían los usuarios reales. Son pruebas muy frágiles, es decir, pequeños cambios sin relación aparente pueden hacer fallar la prueba aunque funcionalmente el sistema sea correcto.

Unitarios Se utilizan para probar un único componente del sistema: un método o función, y para unas condiciones concretas (un escenario). La validación se puede hacer bien comprobando el estado final conocido el estado inicial o por la interacción entre el componente que se está probando y sus colaboradores.

Desde el punto de vista ágil hay una pauta clara: las pruebas se escriben para ejecutarse, y debería ocurrir tan a menudo como sea posible. Lo ideal sería ejecutar todas las pruebas después de cada cambio en cualquier parte de la aplicación. Obviamente eso resulta prohibitivo incluso para aplicaciones pequeñas. Hay que llegar a una solución de compromiso. Por este motivo, las pruebas unitarias son las más importantes. Si están bien escritas, las pruebas unitarias se deberían poder ejecutar en muy pocos segundos. Eso permite que, con los frameworks y herramientas adecuadas se pueda lanzar la batería de pruebas unitarias completa mientras se está editando el código (sea la prueba o el código de producción).

Para que una prueba se considere unitaria no basta con que esté escrita en un framework xUnit, debe cumplir los principios FIRST [24], que es un acrónimo para:

Fast Las pruebas unitarias deberían ser muy rápidas. Como se ha dicho, todas las pruebas unitarias de la aplicación (o al menos del módulo) deberían ejecutarse en menos de 2–3 segundos.

Independent Cada prueba debe poder ejecutarse por separado o en conjunto, y en cualquier orden, sin que eso afecte al resultado.

Repeatable La prueba debería poder ejecutarse múltiples veces dando siempre el mismo resultado. Por este motivo no es buena idea incorporar aleatoriedad a los tests. También implica que la prueba debe poder funcionar del mismo modo en entornos distintos.

Self-validating La prueba debe ofrecer un resultado concreto: pasa o falla, sin que el programador tenga que leer o interpretar un valor en pantalla o en un fichero.

Timely El test unitario debería escribirse justo cuando se necesite, es decir, justo antes de escribir el código de producción relacionado, ni antes ni después.

En cuanto al resto de las pruebas: sistema, integración y aceptación; deberían ejecutarse al menos una vez al día. Existe toda una disciplina, llamada «integración continua» que trata sobre la compilación, integración y prueba de todo el sistema de forma totalmente automática, incluyendo la instalación de dependencias e incluso el empaquetado y despliegue. Esta operación puede hacerse cada vez que un programador añade nuevo código al repositorio o bien una vez al día si la aplicación es muy grande. El objetivo es disponer de información precisa y actualizada sobre el estado de la aplicación en su conjunto y sobre los requisitos que está cumpliendo.

5.5. Pruebas unitarias con google-tests

En esta sección veremos un ejemplo de TDD intencionadamente simple para crear la función `factorial()`⁴. Para ello vamos a utilizar el framework de pruebas `google-tests` (`gtest`). El primer test prueba que el resultado de `factorial(0)` es 1:

Listado 5.3: `factorial-test.cc`: Pruebas para `factorial()`

```
1 #include "gtest/gtest.h"
2 #include "factorial.h"
3
4 TEST(FactorialTest, Zero) {
5     EXPECT_EQ(1, factorial(0));
6 }
```

Se incluye el archivo de cabecera de `gtest` (línea 1) donde están definidas las macros que vamos a utilizar para definir las pruebas. La línea 4 define una prueba llamada `Zero` mediante la macro `TEST` para

⁴Inspirado en el primer ejemplo del tutorial de GTests.

el casa de prueba (*TestCase*) `FactorialTest`. La línea 5 especifica una expectativa: el resultado de invocar `factorial(0)` debe ser igual a 1.

Además del fichero con la prueba debemos escribir el código de producción, su fichero de cabecera y un `Makefile`. Al escribir la expectativa ya hemos decidido el nombre de la función y la cantidad de parámetros (aunque no es tipo). Veamos estos ficheros:

Listado 5.4: Escribiendo `factorial()` con TDD: `Makefile`

```
1 CC=$(CXX)
2 LDLIBS=-lpthread -lgtest -lgtest_main
3
4 factorial-test: factorial-test.o factorial.o
5
6 clean:
7     $(RM) factorial-test *.o *~
```

Listado 5.5: Escribiendo `factorial()` con TDD: `factorial.h`

```
1 int factorial(int n);
```

Y el código de producción mínimo para pasar la prueba:

Listado 5.6: Escribiendo `factorial()` con TDD: `factorial.cc` (1)

```
1 #include "factorial.h"
2
3 int factorial(int n) {
4     return 1;
5 }
```

Compilamos y ejecutamos el binario obtenido:

```
$ make
g++ -c -o factorial.o factorial.cc
g++ factorial-test.o factorial.o -lpthread -lgtest -lgtest_main -o
  factorial-test
$ ./factorial-test
Running main() from gtest_main.cc
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from FactorialTest
[ RUN      ] FactorialTest.Zero
[          OK ] FactorialTest.Zero (0 ms)
[-----] 1 test from FactorialTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (0 ms total)
[ PASSED  ] 1 test.
```

Esta primera prueba no la hemos visto fallar porque sin el fichero de producción ni siquiera podríamos haberla compilado.

Añadamos ahora un segundo caso de prueba al fichero:

Listado 5.7: factorial-test.cc: Pruebas para factorial()

```

1 TEST(FactorialTest, Positive) {
2     EXPECT_EQ(1, factorial(1));
3     EXPECT_EQ(2, factorial(2));
4 }

```

Como es lógico, la expectativa de la línea 2 también pasa ya que el resultado es el mismo que para la entrada 0. Veamos el resultado:

```

$ ./factorial-test
Running main() from gtest_main.cc
[====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 2 tests from FactorialTest
[ RUN      ] FactorialTest.Zero
[ OK       ] FactorialTest.Zero (0 ms)
[ RUN      ] FactorialTest.Positive
factorial-test.cc:10: Failure
Value of: factorial(2)
  Actual: 1
 Expected: 2
[ FAILED   ] FactorialTest.Positive (0 ms)
[-----] 2 tests from FactorialTest (0 ms total)

[-----] Global test environment tear-down
[====] 2 tests from 1 test case ran. (0 ms total)
[ PASSED   ] 1 test.
[ FAILED   ] 1 test, listed below:
[ FAILED   ] FactorialTest.Positive

1 FAILED TEST

```

El resultado nos indica la prueba que ha fallado (línea 8), el valor obtenido por la llamada (línea 11) y el esperado (línea 12).

A continuación se debe modificar la función `factorial()` para que cumpla la nueva expectativa. Después escribir nuevas pruebas que nos permitan comprobar que la función cumple con su cometido para unos cuantos casos representativos.

5.6. Dobles de prueba

TDD, y el *agilismo* en general, está muy relacionada con la orientación a objetos, y en muchos sentidos se asume que estamos haciendo un diseño orientado a objetos prácticamente en todos los casos.

La mayoría de los lenguajes de programación que soportan la orientación a objetos tienen herramientas para encapsulación y ocultación. La ocultación (el hecho de no exponer los detalles de implementación de la clase) resulta crucial en un diseño orientado a objetos porque proporciona «sustituibilidad» (LSP⁵). Pero la ocultación dificulta la definición de pruebas en base a aserciones sobre de estado porque el estado del objeto está definido por el valor de sus atributos. Si tuviéramos acceso a todos los atributos del objeto (sea con *getters* o no) sería una pista de un mal diseño.

⁵Se refiere al principio SLP: «Principio de Sustitución de Liskov»

Debido a ello, suele ser más factible definir la prueba haciendo aserciones sobre la interacción que el objeto que se está probando (el SUT⁶) realiza con sus colaboradores. El problema de usar un colaborador real es que éste tendrá a su vez otros colaboradores de modo que es probable que para probar un único método necesitemos montar gran parte de la aplicación. Como lo que queremos es instanciar lo mínimo posible del sistema real podemos recurrir a los dobles⁷ de prueba.

SOLID

SOLID⁸ es una serie de 5 principios esenciales para conseguir diseños orientados a objetos de calidad. Estos son:

SRP Single Responsibility
OCP Open Closed
LSP Liskov Substitution
DIP Dependency Inversion
ISP Interface Segregation

Un doble de prueba es un objeto capaz de simular la interfaz que un determinado colaborador ofrece al SUT, pero que realmente no implementa nada de lógica. El doble (dependiendo de su tipo) tiene utilidades para comprobar qué métodos y parámetros usó el SUT cuando invocó al doble.



Una regla básica: Nunca se deben crear dobles para clases implementadas por terceros, sólo para clases de la aplicación.

Un requisito importante para poder realizar pruebas con dobles es que las clases de nuestra aplicación permitan «inyección de dependencias». Consiste en pasar (inyectar) las instancias de los colaboradores (dependencias) que el objeto necesitará en el momento de su creación. Pero no estamos hablando de un requisito impuesto por las pruebas, se trata de otro de los principios SOLID, en concreto DIP (Dependency Inversion Principle).

Aunque hay cierta confusión con la terminología, hay bastante consenso en distinguir al menos entre los siguientes tipos de dobles:

Fake Es una versión rudimentaria del objeto de producción. Funcionalmente equivalente, pero tomando atajos que no serían admisibles en el código final. Por ejemplo, una base de datos cuya persistencia es un diccionario en memoria.

Stub Devuelve valores predefinidos para los métodos que el SUT va a invocar. Se trata de un colaborador que «le dice al SUT lo que necesita oír» pero nada más.

Mock El mock se programa con una serie de expectativas (invocaciones a sus métodos) que debería cumplirse durante la ejecución de la prueba. Si alguna de esas llamadas no se produce, u ocurre en una forma diferente a lo esperado, la prueba fallará.

Spy El spy es un objeto que registra todas las invocaciones que se hacen sobre él. Después de utilizado, se pueden hacer aserciones para comprobar que ciertas llamadas a sus métodos ocurrieron. A diferencia del mock, puede haber recibido otras invocaciones

⁶SUT: Subject Under Test

⁷Son «dobles» en el mismo sentido que los actores que ruedan las escenas arriesgadas en el cine.

además de las que se comprueban y el comportamiento sigue siendo válido.

5.7. Dobles de prueba con google-mock

En esta sección veremos un ejemplo muy simple de uso con google-mock, el framework de dobles C++ que complementa a google-test. Vamos a implementar el método `notify()` de la clase `Observable` (también llamada `Subject`) del patrón *observador*.

Los primeros ficheros que se muestran son el fichero de cabecera `observable.h`:

Listado 5.8: Patrón observador con TDD: observable.h

```
1 #ifndef _OBSERVABLE_H_
2 #define _OBSERVABLE_H_
3
4 #include <vector>
5 #include "observer.h"
6
7 class Observable {
8     std::vector<Observer*> observers;
9 public:
10     void attach(Observer* observer);
11     void detach(Observer* observer);
12     void notify(void);
13 };
14
15 #endif
```

Y el fichero de implementación `observable.cc`:

Listado 5.9: Patrón observador con TDD: observable.cc

```
1 #include <algorithm>
2 #include <functional>
3
4 #include "observable.h"
5 #include "observer.h"
6
7 void
8 Observable::attach(Observer* observer) {
9     observers.push_back(observer);
10 }
11
12 void
13 Observable::detach(Observer* observer) {
14     observers.erase(find(observers.begin(), observers.end(),
15                         observer));
16 }
17
18 void
19 Observable::notify(void) {
20     observers[0]->update();
21 }
```

Para escribir un test que pruebe el método `notify()` necesitamos un *mock* para su colaborador (el *observador*). El siguiente listado muestra la interfaz que deben implementar los observadores:

Listado 5.10: Patrón observador con TDD: `observer.h`

```
1 #ifndef _OBSERVER_H_
2 #define _OBSERVER_H_
3
4 class Observer {
5 public:
6     virtual void update(void) = 0;
7     virtual ~Observer() {}
8 };
9
10 #endif
```

Con ayuda de `google-mock` escribimos el *mock* para este colaborador:

Listado 5.11: Patrón observador con TDD: `mock-observer.h`

```
1 #ifndef MOCK_OBSERVER_H
2 #define MOCK_OBSERVER_H
3
4 #include <gmock/gmock.h>
5 #include "observer.h"
6
7 class MockObserver : public Observer {
8 public:
9     MOCK_METHOD0(update, void());
10 };
11
12 #endif
```

Lo interesante aquí es la definición del método *mockeado* en la línea 9. La macro `MOCK_METHOD0` indica que es para un método sin argumentos llamado `update()` que devuelve `void`. Aunque podemos escribir este fichero *a mano* sin demasiados problemas, existe una herramienta llamada `gmock_gen` que los genera automáticamente a partir de los ficheros de declaración de las clases.

Es hora de escribir la prueba. Vamos a comprobar que si tenemos *observable* con un *observador* registrado e invocamos su método `notify()` el método `update()` del observador se ejecuta una vez (y solo una).

En la prueba creamos el doble para el *observer* (línea 8) y creamos la expectativa (línea 9). Después creamos el *observable* (línea 11) y registramos el *observador* (línea 12). Por último invocamos el método `notify()` (línea 14).

Listado 5.12: Patrón observador con TDD: observable-tests.cc

```

1 #include <gmock/gmock.h>
2 #include <gtest/gtest.h>
3
4 #include "observable.h"
5 #include "mock-observer.h"
6
7 TEST(ObserverTest, UpdateObserver) {
8     MockObserver observer;
9     EXPECT_CALL(observer, update()).Times(1);
10
11     Observable observable;
12     observable.attach(&observer);
13
14     observable.notify();
15 }

```

También necesitamos un Makefile para compilar y ejecutar la prueba:

Listado 5.13: Patrón observador con TDD: Makefile

```

1 GMOCK_SRC = /usr/src/gmock
2
3 CC          = g++
4 CXXFLAGS   = -I $(GMOCK_SRC)
5 LDLIBS     = -lpthread -lgtest
6
7 TARGET     = observable-tests
8
9 vpath %.cc $(GMOCK_SRC)/src
10
11 $(TARGET): observable-tests.o observable.o gmock_main.o gmock-all.o
12
13 test: $(TARGET)
14     ./${<
15
16 clean:
17     $(RM) $(TARGET) *.o *~

```

Ejecutemos el Makefile:

```

$ make test
g++ -I /usr/src/gmock -c -o observable-tests.o observable-tests.cc
g++ -I /usr/src/gmock -c -o observable.o observable.cc
g++ -I /usr/src/gmock -c -o gmock_main.o /usr/src/gmock/src/
gmock_main.cc
g++ -I /usr/src/gmock -c -o gmock-all.o /usr/src/gmock/src/gmock-all
.cc
g++ observable-tests.o observable.o gmock_main.o gmock-all.o -
lpthread -lgtest -o observable-tests
$ ./observable-tests
Running main() from gmock_main.cc
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from ObserverTest
[ RUN      ] ObserverTest.UpdateObserver
observable-tests.cc:11: Failure
Actual function call count doesn't match EXPECT_CALL(observer, update
())...
Expected: to be called once

```

```

Actual: never called - unsatisfied and active
[ FAILED ] ObserverTest.UpdateObserver (1 ms)
[-----] 1 test from ObserverTest (1 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (1 ms total)
[ PASSED ] 0 tests.
[ FAILED ] 1 test, listed below:
[ FAILED ] ObserverTest.UpdateObserver

1 FAILED TEST

```

Después de la compilación (líneas 2-6) se ejecuta el binario correspondiente al test (línea 7). El test falla porque se esperaba una llamada a `update()` (línea 15) y no se produjo ninguna (línea 16) de modo que la expectativa no se ha cumplido. Es lógico porque el cuerpo del método `notify()` está vacío. Siguiendo la filosofía TDD escribir el código mínimo para que la prueba pase:

Listado 5.14: Código mínimo para satisfacer la expectativa

```

1 void
2 Observable::notify(void) {
3     observers[0]->update();
4 }

```

Volvemos a ejecutar la prueba:

```

$ make test
g++ -I /usr/src/gmock -c -o observable.o observable.cc
g++ observable-tests.o observable.o gmock_main.o gmock-all.o -
    lpthread -lgtest -o observable-tests
$ ./observable-tests
Running main() from gmock_main.cc
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from ObserverTest
[ RUN      ] ObserverTest.UpdateObserver
[         OK ] ObserverTest.UpdateObserver (0 ms)
[-----] 1 test from ObserverTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (0 ms total)
[ PASSED ] 1 test.

```

La prueba pasa. Hora de escribir otra prueba. Comprobemos que `update()` no se invoca si nadie invoca `notify()`:

Listado 5.15: Prueba negativa para `Observer::update()`

```

1 TEST(ObserverTest, NeverUpdateObserver) {
2     MockObserver observer;
3     EXPECT_CALL(observer, update()).Times(0);
4
5     Observable observable;
6     observable.attach(&observer);
7 }

```

La prueba pasa. Ahora comprobemos que funciona también para dos observadores:

Listado 5.16: Prueba para notificación de dos observadores

```

1 TEST(ObserverTest, TwoObserver) {
2     MockObserver observer1, observer2;
3     EXPECT_CALL(observer1, update());
4     EXPECT_CALL(observer2, update());
5
6     Observable observable;
7     observable.attach(&observer1);
8     observable.attach(&observer2);
9
10    observable.notify();
11 }

```

Y ejecutamos la prueba:

```

$ make test
Running main() from gmock_main.cc
[====] Running 3 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 3 tests from ObserverTest
[ RUN      ] ObserverTest.UpdateObserver
[ OK       ] ObserverTest.UpdateObserver (0 ms)
[ RUN      ] ObserverTest.NeverUpdateObserver
[ OK       ] ObserverTest.NeverUpdateObserver (0 ms)
[ RUN      ] ObserverTest.TwoObserver
observable-tests.cc:30: Failure
Actual function call count doesn't match EXPECT_CALL(observer2, update
())...
    Expected: to be called once
    Actual: never called - unsatisfied and active
[ FAILED   ] ObserverTest.TwoObserver (0 ms)
[-----] 3 tests from ObserverTest (1 ms total)

[-----] Global test environment tear-down
[====] 3 tests from 1 test case ran. (1 ms total)
[ PASSED   ] 2 tests.
[ FAILED   ] 1 test, listed below:
[ FAILED   ] ObserverTest.TwoObserver

1 FAILED TEST

```

Y la segunda expectativa falla (línea) y nos la muestra en consola:

```

Actual function call count doesn t match EXPECT_CALL(observer2, update
())...

```

Implementemos `notify()` para recorrer todos los observadores:

Listado 5.17: Patrón observador con TDD: `observable.cc`

```

1 void
2 Observable::notify(void) {
3     std::for_each(observers.begin(), observers.end(),
4         std::mem_fun(&Observer::update));
5 }

```

Ejecutamos de nuevo la prueba:

```
$ make test
Running main() from gmock_main.cc
[=====] Running 3 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 3 tests from ObserverTest
[ RUN      ] ObserverTest.UpdateObserver
[         OK ] ObserverTest.UpdateObserver (0 ms)
[ RUN      ] ObserverTest.NeverUpdateObserver
[         OK ] ObserverTest.NeverUpdateObserver (0 ms)
[ RUN      ] ObserverTest.TwoObserver
[         OK ] ObserverTest.TwoObserver (0 ms)
[-----] 3 tests from ObserverTest (0 ms total)

[-----] Global test environment tear-down
[=====] 3 tests from 1 test case ran. (0 ms total)
[ PASSED ] 3 tests.
```

Todo correcto, aunque sería conveniente una prueba adicional para un mayor número de observadores registrados. También podríamos comprobar que los observadores des-registrados (*detached*) efectivamente no son invocados, etc.

Aunque los ejemplos son sencillos, es fácil ver la dinámica de TDD.

5.8. Limitaciones

Hay ciertos aspectos importantes para la aplicación en los que TDD, y el *testing* en general, tienen una utilidad limitada (al menos hoy en día). Las pruebas permiten comprobar fácilmente aspectos funcionales, pero es complejo comprobar requisitos no funcionales.

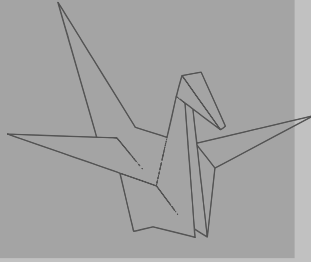


«Los tests no pueden probar la ausencia de fallos,
sólo su existencia»

Kent Beck

Puede ser complicado probar rendimiento, fiabilidad, tiempo de respuesta y otros aspectos importantes, al menos escribiendo las pruebas primero. TDD tampoco ayuda a diseñar cuestiones de carácter general como la arquitectura de la aplicación, la seguridad, la accesibilidad, el modelo de persistencia, etc. Se dice que estos detalles de diseño «no emergen» de las pruebas.

Respecto al desarrollo de videojuegos, TDD no se adapta bien al diseño y prueba de la concurrencia, y en particular es complejo probar todo lo relacionado con la representación gráfica e interacción con el usuario. A pesar de ello, los métodos ágiles también están causando un importante impacto en el desarrollo de videojuegos y la informática gráfica en general. Cada día aparecen nuevos frameworks y herramientas que hacen posible probar de forma sencilla cosas que antes se consideraban inviables.



Capítulo 6

Simulación física

Carlos González Morcillo

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 este capítulo 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 libre *Bullet*.



Figura 6.1: “Anarkanoid, el machacaladrillos sin reglas es un juego tipo *Breakout* donde la simulación física se reduce a una detección de colisiones 2D.

Cuerpo Rígido

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

6.1. Introducción

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.

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 (salvo casos específicos convenientemente documentados en la revista *Más Allá*). 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, 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 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 líneas 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 videojuego.
- 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.
- Simulación de fluidos, telas y cuerpos blandos (ver Figura 6.2).

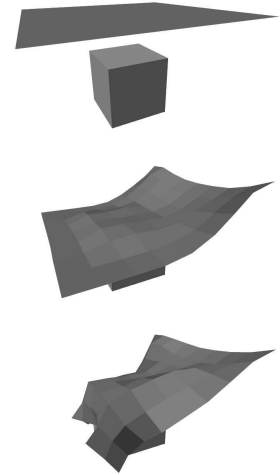


Figura 6.2: Tres instantes en la simulación física de una tela sobre un cubo. Simulación realizada con el motor Bullet.

6.1.1. 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:

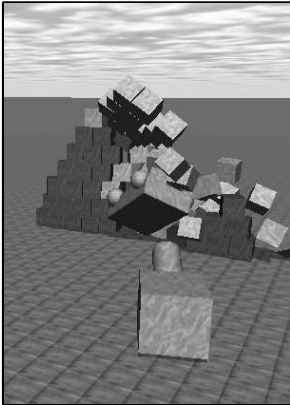


Figura 6.3: Ejemplo de simulación física con ODE (demo de la distribución oficial), que incorpora el uso de motores y diferentes geometrías de colisión.



Figura 6.4: Logotipo del motor de simulación físico estrella en el mundo del software privativo.

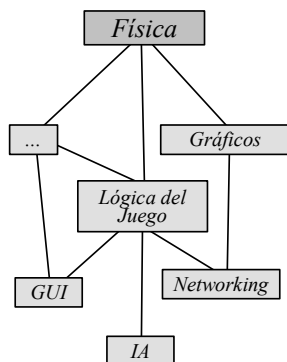


Figura 6.5: El motor de simulación física está directamente relacionado con otros módulos del videojuego. Esta dependencia conlleva una serie de dificultades que deben tenerse en cuenta.

- **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. Estudiaremos con más detalle este motor, junto con su uso en Ogre, en la Sección 6.5.
- **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). Ogre cuenta igualmente con un wrapper para utilizar este motor de simulación física.
- **PhysX.** Este motor privativo, es actualmente mantenido por NVIDIA con aceleración basada en hardware (mediante unidades específicas de procesamiento físico PPU's *Physics Processing Units* o mediante núcleos CUDA. Las tarjetas gráficas con soporte de CUDA (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. Gracias al wrapper *NxOgre* se puede utilizar este motor en Ogre.
- **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.

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 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.

6.1.2. 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:



Figura 6.6: 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.

- **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 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 por ejemplo el precálculo de la iluminación 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. En muchas ocasiones es necesario además la exportación de diferentes versiones de un mismo objeto (una versión de alta poligonalización, la versión de colisión, una versión destructible, etc).
- **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?).

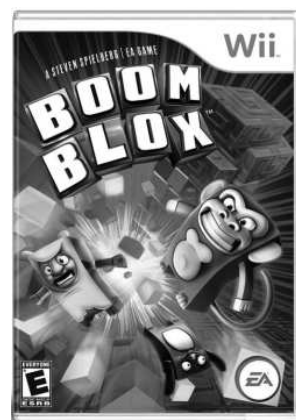


Figura 6.7: La primera incursión de Steven Spielberg en el mundo de los videojuegos fue en 2008 con *Boom Blox*, un título de Wii desarrollado por Electronic Arts con una componente de simulación física crucial para la experiencia del jugador.

6.1.3. Conceptos Básicos

A principios del siglo XVII, Isaac Newton 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. Las tres leyes pueden resumirse como:

1. Un cuerpo tiende a mantenerse en reposo o a continuar moviéndose en línea recta a una velocidad constante a menos que actúe sobre él una fuerza externa. Esta ley resume el concepto de inercia.
2. El cambio de movimiento es proporcional a la fuerza motriz aplicada y ocurre según la línea recta a lo largo de la que se aplica dicha fuerza.
3. Para cada fuerza que actúa sobre un cuerpo ocurre una reacción igual y contraria. De este modo, las acciones mutuas de dos cuerpos siempre son iguales y dirigidas en sentido opuesto.

En el estudio de la dinámica resulta especialmente interesante la segunda ley de Newton, que puede ser escrita como

$$F = m \times a \quad (6.1)$$

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.

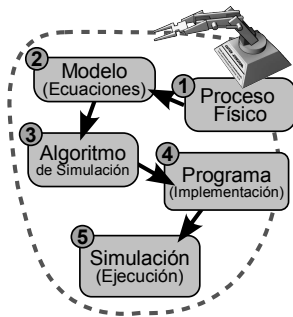


Figura 6.8: Etapas en la construcción de un motor de simulación física.



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 tiempo 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.

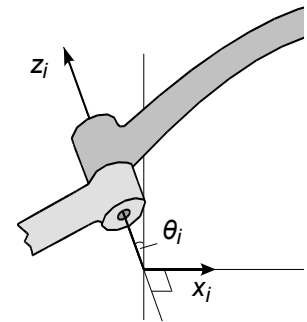


Figura 6.9: Un controlador puede intentar que se mantenga constante el ángulo θ_i formado entre dos eslabones de un brazo robótico.

En multitud de ocasiones es necesario definir restricciones que definen límites a ciertos aspectos de la simulación. Los **controladores** son elementos que generan entradas a la simulación y que tratan de controlar el comportamiento de los objetos que están siendo simulados. Por ejemplo, un controlador puede intentar mantener constante el ángulo entre dos eslabones de una cadena robótica (ver Figura 6.9).

6.2. Sistema de Detección de Colisiones

La responsabilidad principal del *Sistema de Detección de Colisiones* (SDC) es calcular cuándo *colisionan* los objetos de la escena. Para calcular esta *colisión*, los objetos se representan internamente por una forma geométrica sencilla (como esferas, cajas, cilindros...).

Además de comprobar si hubo colisión entre los objetos, el SDC se encarga de proporcionar información relevante al resto de módulos del simulador físico sobre las propiedades de la colisión. Esta información se utiliza para evitar efectos indeseables, como la penetración de un objeto en otro, y conseguir la estabilidad en la simulación cuando el objeto llega a la posición de equilibrio.

6.2.1. 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 (que cuentan con un mayor nivel de detalle).

Habitualmente se trata de simplificar al máximo la forma de colisión. Aunque el SDC soporte objetos complejos, será preferible emplear tipos de datos simples, siempre que el resultado sea aceptable. La Figura 6.10 muestra algunos ejemplos de aproximación de formas de colisión para ciertos objetos del juego.

Test de Intersección

En realidad, el *Sistema de Detección de Colisiones* puede entenderse como un módulo para realizar pruebas de intersección complejas.

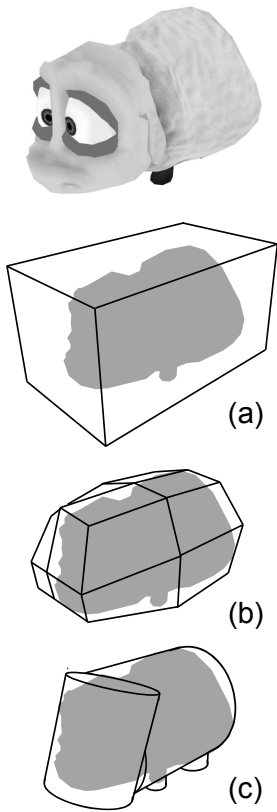


Figura 6.10: Diferentes formas de colisión para el objeto de la imagen. (a) Aproximación mediante una caja. (b) Aproximación mediante un volumen convexo. (c) Aproximación basada en la combinación de varias primitivas de tipo cilíndrico.

Formas de colisión

A cada cuerpo dinámico se le asocia habitualmente una única forma de colisión en el SDC.

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.



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 6.10, 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 son convexas. Una **forma convexa** es aquella en la que un rayo que surja desde su interior atravesará la superficie 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:

- **Esferas.** Son las primitivas más simples y eficientes; basta con definir su centro y radio (uso de un vector de 4 elementos).
- **Cajas.** 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 coordenadas global. Esto implica que si el objeto rota, como se muestra en la Figura 6.11, la aproximación de forma puede resultar de baja calidad.

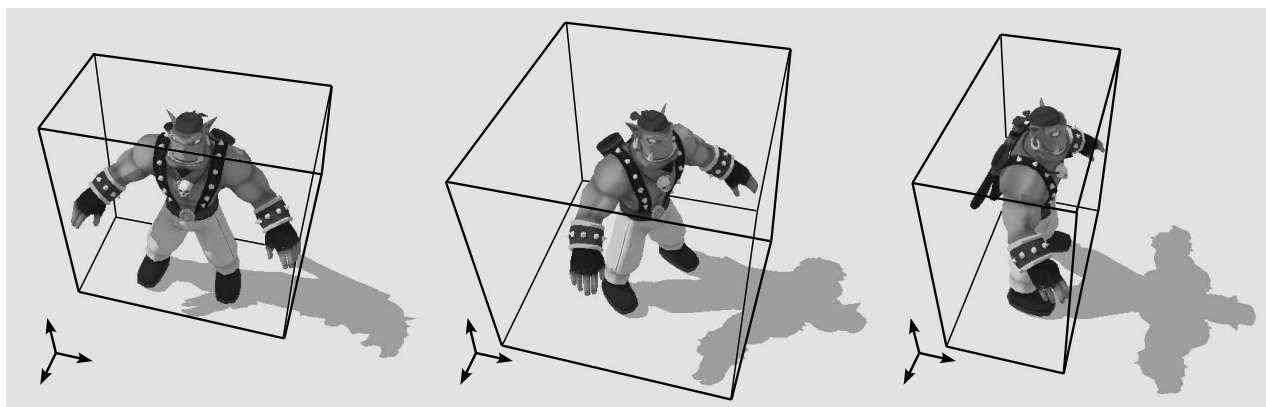


Figura 6.11: Gestión de la forma de un objeto empleando cajas límite alineadas con el sistema de referencia universal AABBs. Como se muestra en la figura, la caja central realiza una aproximación de la forma del objeto muy pobre.

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.

Por su parte, las cajas OBB (**Oriented Bounding Box**) definen una rotación relativa al sistema de coordenadas. Su descripción es muy simple y permiten calcular la colisión entre primitivas de una forma muy eficiente.

- **Cilindros.** Los cilindros son ampliamente utilizados. Se definen mediante dos puntos y un radio. Una extensión de esta forma básica es la *cápsula*, que es un cuerpo compuesto por un cilindro y dos semiesferas (ver Figura 6.12). Puede igualmente 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 convexos.** La mayoría de los SDC permiten trabajar con volúmenes convexos (ver Figura 6.10). La forma del objeto suele representarse internamente mediante un conjunto de n planos. Aunque este tipo de formas es menos eficiente que las primitivas estudiadas anteriormente, existen ciertos algoritmos como el GJK que permiten optimizar los cálculos en este tipo de formas.
- **Malla poligonal.** En ciertas ocasiones puede ser interesante utilizar mallas arbitrarias. Este tipo de superficies pueden ser abiertas (no es necesario que definan un volumen), y se construyen como mallas de triángulos. Las mallas poligonales se suelen emplear en elementos de geometría estática, como elementos del escenario, terrenos, etc. (ver Figura 6.13) Este tipo de formas de colisión son las más complejas computacionalmente, ya que el SDC debe probar con cada triángulo. Así, muchos juegos tratan

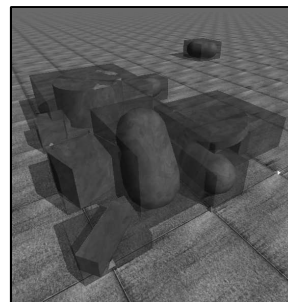
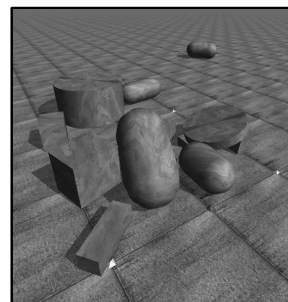


Figura 6.12: Algunas primitivas de colisión soportadas en ODE: Cajas, cilindros y cápsulas. La imagen inferior muestra los AABBs asociados a los objetos.

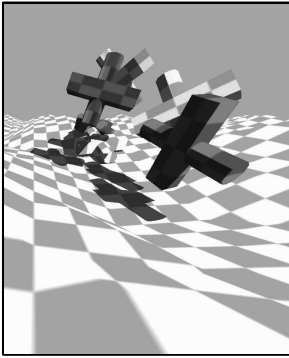


Figura 6.13: Bullet soporta la definición de mallas poligonales animadas. En este ejemplo de las demos oficiales, el suelo está animado y los objetos convexos colisionan respondiendo a su movimiento.

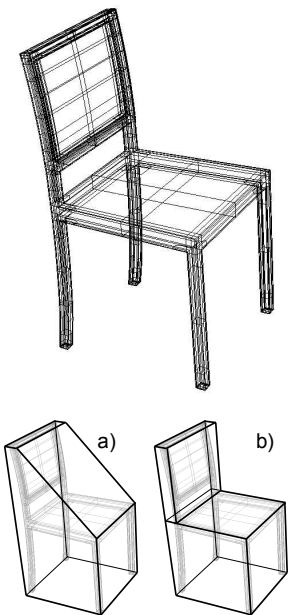


Figura 6.14: El modelo de una silla es un objeto que no se adapta bien a un volumen convexo (a). En (b) se ha utilizado una forma compuesta definiendo dos cajas.

de limitar el uso de este tipo de formas de colisión para evitar que el rendimiento se desplome.

- **Formas compuestas.** Este tipo de formas se utilizan cuando la descripción de un objeto se aproxima más convenientemente con una colección de formas. Este tipo de formas es la aproximación deseable en el caso de que tengamos que utilizar objetos cóncavos, que no se adaptan adecuadamente a volúmenes convexos (ver Figura 6.14).

6.2.2. 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 determinar si dos formas intersecan no son triviales. Por otro lado, muchos juegos requieren 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 recalcularse 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 espacial 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 Bullet, se utilizan varias capas o pasadas para detectar las colisiones. Primero suelen emplearse cajas AABB para comprobar si los objetos pueden estar potencialmente en

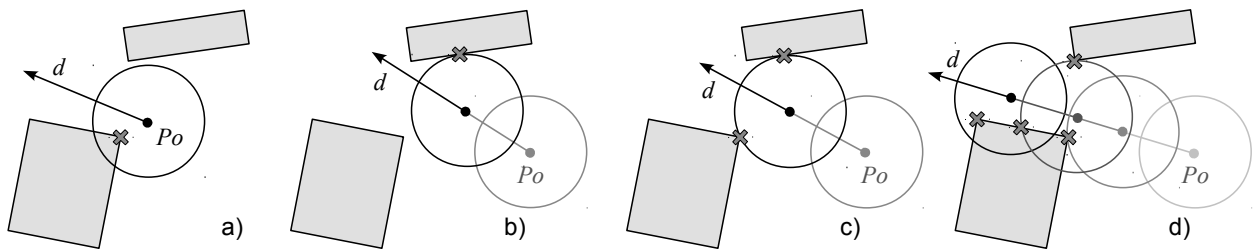


Figura 6.15: Cálculo de los puntos de colisión empleando *Shape Casting*. **a)** La forma inicialmente se encuentra colisionando con algún objeto de la escena. **b)** El SDC obtiene un punto de colisión. **c)** La forma interseca con dos objetos a la vez. **d)** En este caso el sistema calcula todos los puntos de colisión a lo largo de la trayectoria en la dirección de d .

colisión (detección de la colisión amplia). 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.

6.2.3. Preguntando al sistema...

En el módulo 2 ya estudiamos algunas de las funcionalidades que se pueden encontrar en sistemas de detección de colisiones. El objetivo es poder obtener resultados a ciertas consultas sobre el primer objeto que intersecará con un determinado rayo, si hay objetos situados en el interior de un determinado volumen, etc.

A continuación veremos dos de las principales *collision queries* que pueden encontrarse habitualmente en un SDC:

- **Ray Casting.** Este tipo de *query* requiere que se especifique un rayo, y un origen. Si el rayo interseca con algún objeto, se devolverá un punto o una lista de puntos. Como vimos, el rayo se especifica habitualmente mediante una ecuación paramétrica de modo que $p(t) = p_o + td$, siendo t el parámetro que toma valores entre 0 y 1. d nos define el vector dirección del rayo, que determinará la distancia máxima de cálculo de la colisión. P_o nos define el punto de origen del rayo. Este valor de t que nos devuelve la *query* puede ser fácilmente convertido al punto de colisión en el espacio 3D.
- **Shape Casting.** El *Shape Casting* permite determinar los puntos de colisión de una forma que viaja en la dirección de un vector determinado. Es similar al *Ray Casting*, pero en este caso es necesario tener en cuenta dos posibles situaciones que pueden ocurrir:
 1. La forma sobre la que aplicamos *Shape Casting* está inicialmente intersecando con al menos un objeto que evita que se desplace desde su posición inicial. En este caso el SDC

Uso de Rayos

El *Ray Casting* se utiliza ampliamente en videojuegos. Por ejemplo, para comprobar si un personaje está dentro del área de visión de otro personaje, para detectar si un disparo alcanza a un enemigo, para que los vehículos permanezcan en contacto con el suelo, etc.

Uso de Shape Casting

Un uso habitual de estas *queries* permite determinar si la cámara entra en colisión con los objetos de la escena, para ajustar la posición de los personajes en terrenos irregulares, etc.

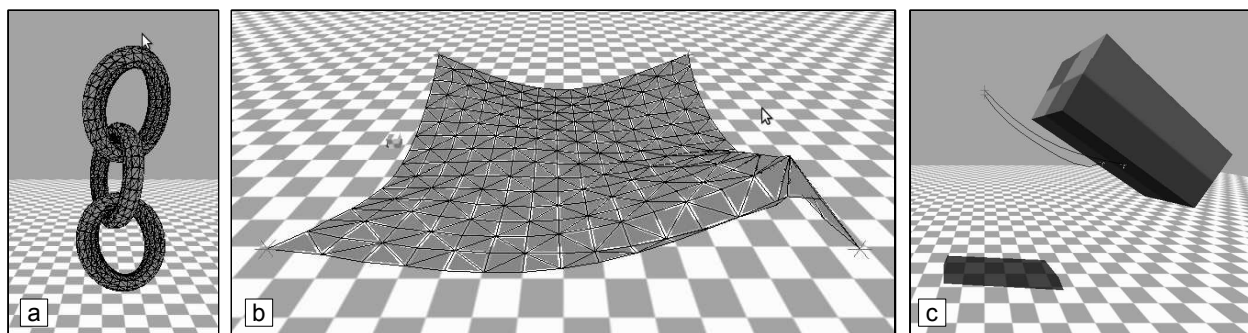


Figura 6.16: Ejemplo de simulación de cuerpos blandos con Bullet. a) Uso de *softbodies* con formas convexas. b) Simulación de una tela sostenida por cuatro puntos. c) Simulación de cuerdas.

Lista de colisiones

Para ciertas aplicaciones puede ser igualmente conveniente obtener la lista de todos los objetos con los que se interseca a lo largo de la trayectoria, como se muestra en la Figura 6.15.d).

devolverá los puntos de contacto que pueden estar situados sobre la superficie o en el interior del volumen.

2. La forma no interseca sobre ningún objeto, por lo que puede desplazarse libremente por la escena. En este caso, el resultado de la colisión suele ser un punto de colisión situado a una determinada distancia del origen, aunque puede darse el caso de varias colisiones simultáneas (como se muestra en la Figura 6.15). En muchos casos, los SDC únicamente devuelven el resultado de la primera colisión (una lista de estructuras que contienen el valor de t , el identificador del objeto con el que han colisionado, el punto de contacto, el vector normal de la superficie en ese punto de contacto, y algunos campos extra de información adicional).

6.3. Dinámica del Cuerpo Rígido

La simulación del movimiento de los objetos del juego forma parte del estudio de cómo las fuerzas afectan al comportamiento de los objetos. El módulo del motor de simulación que se encarga de la dinámica de los objetos estudia cómo cambian su posición en el tiempo. Hasta hace pocos años, los motores de simulación física se centraban en estudiar exclusivamente la *dinámica de cuerpos rígidos*¹, que permite simplificar el cálculo mediante dos suposiciones:

- Los objetos en la simulación obedecen las leyes del movimiento de Newton (estudiadas en la Sección 6.1.3). Así, no se tienen en cuenta ningún tipo de efecto cuántico ni relativista.
- Todos los objetos que intervienen en la simulación son perfectamente sólidos y no se deforman. Esto equivale a afirmar que su forma es totalmente constante.

¹Aunque en la actualidad se encuentran soportadas de una forma muy eficiente otras técnicas, como se muestran en la Figura 6.16

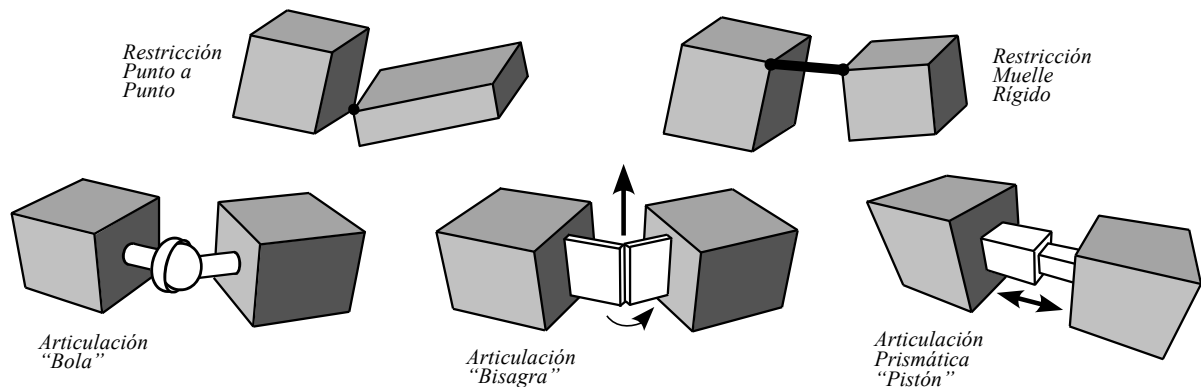


Figura 6.17: Representación de algunas de las principales restricciones que pueden encontrarse en los motores de simulación física.

En el cálculo de la variación de la posición de los objetos con el tiempo, el motor de simulación necesita resolver ecuaciones diferenciales, que cuentan como variable independiente el tiempo. La resolución de estas ecuaciones habitualmente no puede realizarse de forma analítica (es imposible encontrar expresiones simples que relacionen las posiciones y velocidades de los objetos en función del tiempo), por lo que deben usarse métodos de integración numérica.

Gracias a los métodos de *integración numérica*, es posible resolver las ecuaciones diferenciales en pasos de tiempo, de modo que la solución en un instante de tiempo sirve como entrada para el siguiente paso de integración. La duración de cada *paso* de integración suele mantenerse constante Δt .

Uno de los métodos más sencillos que se pueden emplear es el de **Euler**, suponiendo que la velocidad del cuerpo es constante durante el incremento de tiempo. El método también presenta buenos resultados cuando Δt es suficientemente pequeño. En términos generales, este método no es suficientemente preciso, y tiene problemas de convergencia y de estabilidad (el sistema se vuelve inestable, no converge y hace que la simulación *explote*). La alternativa más utilizada en la actualidad es el método de integración de **Verlet**, por su bajo error y su eficiencia computacional en la evaluación de las expresiones.

6.4. Restricciones

Las restricciones sirven para limitar el movimiento de un objeto. Un objeto sin ninguna restricción tiene 6 grados de libertad. Las restricciones se usan en multitud de situaciones en desarrollo de videojuegos, como en puertas, suspensiones de vehículos, cadenas, cuerdas, etc. A continuación enumeraremos brevemente los principales tipos de restricciones soportadas por los motores de simulación física.

- **Punto a punto.** Este es el tipo de restricciones más sencillas; los objetos están conectados por un punto. Los objetos tienen libertad de movimiento salvo por el hecho de que tienen que mantenerse conectados por ese punto.
- **Muelle rígido.** Un muelle rígido (*Stiff Spring*) funciona como una restricción de tipo punto a punto salvo por el hecho de que los objetos están separados por una determinada distancia. Así, puede verse como unidos por una *barra* rígida que los separa una distancia fija.
- **Bisagras.** Este tipo de restricción limitan el movimiento de rotación en un determinado eje (ver Figura 6.17). Este tipo de restricciones pueden definir además un límite de rotación angular permitido entre los objetos (para evitar, por ejemplo, que el codo de un brazo robótico adopte una posición imposible).
- **Pistones.** Este tipo de restricciones, denominadas en general *restricciones prismáticas*, permiten limitar el movimiento de traslación a un único eje. Estas restricciones podrían permitir opcionalmente rotación sobre ese eje.
- **Bolas.** Estas restricciones permiten definir límites de rotación flexibles, estableciendo puntos de anclaje. Sirven por ejemplo para modelar la rotación que ocurre en el hombro de un personaje.
- **Otras restricciones.** Cada motor permite una serie de restricciones específicas, como planares (que restringen el movimiento en un plano 2D), cuerdas, cadenas de objetos, *rag dolls* (definidas como una colección de cuerpos rígidos conectados utilizando una estructura jerárquica), etc...

6.5. Introducción a Bullet

Como se ha comentado al inicio del capítulo, Bullet es una biblioteca de simulación física muy utilizada, tanto en la industria del videojuego, como en la síntesis de imagen realista. Algunas de sus características principales son:

- Está desarrollada íntegramente en C++, y ha sido diseñada de modo que tenga el menor número de dependencias externas posibles.
- Se distribuye bajo licencia Zlib (licencia libre compatible con GPL), y ha sido utilizada en proyectos profesionales en multitud de plataformas, entre las que destacan PlayStation 3, Xbox 360, Wii, Linux, Windows, MacOSX, iPhone y Android.
- Cuenta con un integrador muy estable y rápido. Permite el cálculo de dinámica de cuerpo rígido, dinámicas de vehículos y diversos tipos de restricciones (bisagras, pistones, bolas, etc...).

- Permite dinámica de *SoftBodies*, como telas, cuerdas y deformación de volúmenes arbitrarios.
- Las últimas versiones permiten descargar algunos cálculos a la GPU, empleando OpenCL. La utilización de esta funcionalidad se encuentra en fase experimental, y requiere la instalación de versiones recientes de los drivers de la tarjeta gráfica.
- Está integrado en multitud de paquetes de síntesis de imagen realista (bien de forma interna o mediante plugins), como en Blender, Maya, Softimage, Houdini, Cinema4D, etc...
- Permite importar y exportar archivos Collada.

Existen multitud de videojuegos comerciales que han utilizado Bullet como motor de simulación física. Entre otros se pueden destacar *Grand Theft Auto IV* (de Red Dead Redemption), *Free Realms* (de Sony), *HotWheels* (de BattleForce), *Blood Drive* (de Activision) o *Toy Story 3 (The Game)* (de Disney).

De igual forma, el motor se ha utilizado en películas profesionales, como *Hancock* (de Sony Pictures), *Bolt* de Walt Disney, *Sherlock Holmes* (de Framestore) o *Shrek 4* (de DreamWorks).

Muchos motores gráficos y de videojuegos permiten utilizar Bullet. La comunidad ha desarrollado multitud de *wrappers* para facilitar la integración en *Ogre*, *Crystal Space*, *Irrlich* y *Blitz3D* entre otros.

En el diseño de Bullet se prestó especial atención en conseguir un motor fácilmente adaptable y modular. Tal y como se comenta en su manual de usuario, el desarrollador puede utilizar únicamente aquellos módulos que necesite (ver Figura 6.19):

- Utilización exclusiva del componente de detección de colisiones.
- Utilización del componente de dinámicas de cuerpo rígido sin emplear los componentes de *SoftBody*.
- Utilización de pequeños fragmentos de código de la biblioteca.
- Extensión de la biblioteca para las necesidades específicas de un proyecto.
- Elección entre utilizar precisión simple o doble, etc...

6.5.1. Pipeline de Físicas de Cuerpo Rígido

Bullet define el pipeline de procesamiento físico como se muestra en la Figura 6.20. El pipeline se ejecuta desde la izquierda a la derecha, comenzando por la etapa de cálculo de la gravedad, y finalizando con la integración de las posiciones (actualizando la transformación del mundo). Cada vez que se calcula un paso de la simulación `stepSimulation` en el mundo, se ejecutan las 7 etapas definidas en la imagen anterior.



Figura 6.18: Multitud de videojuegos AAA utilizan Bullet como motor de simulación física.

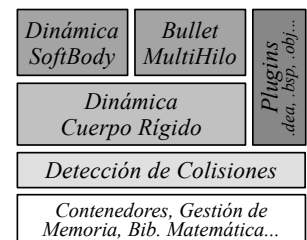


Figura 6.19: Esquema de los principales módulos funcionales de Bullet.

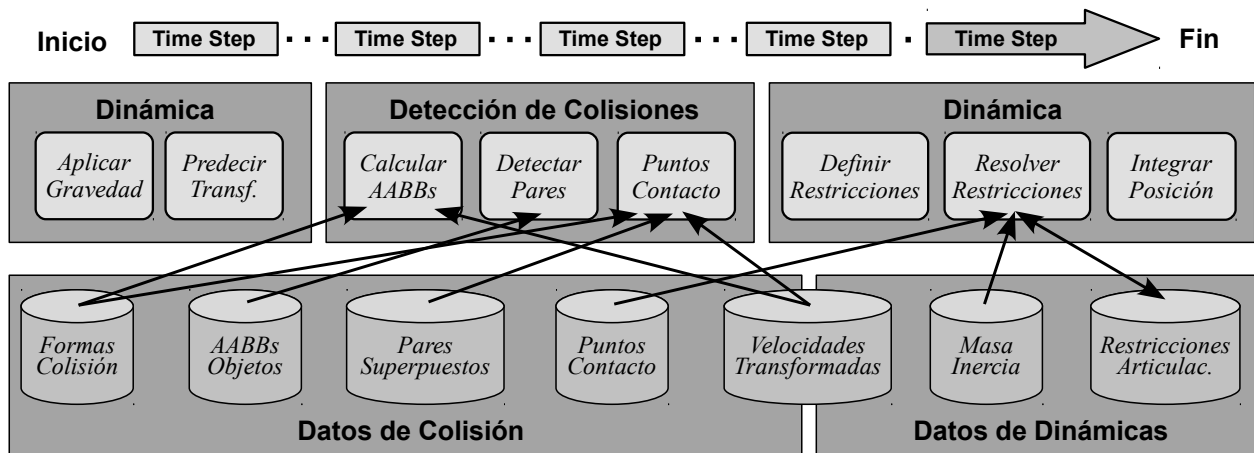


Figura 6.20: Pipeline de Bullet. En la parte superior de la imagen se describen las principales etapas computacionales, y en la parte inferior las estructuras de datos más importantes.

El Pipeline comienza **aplicando la fuerza gravedad** a los objetos de la escena. Posteriormente se realiza una **predicción de las transformaciones** calculando la posición actual de los objetos. Hecho esto se **calculan las cajas AABBs**, que darán una estimación rápida de la posición de los objetos. Con estas cajas se **determinan los pares**, que consiste en calcular si las cajas AABBs se solapan en algún eje. Si no hay ningún solape, podemos afirmar que no hay colisión. De otra forma, hay que realizar un estudio más detallado del caso. Si hay posibilidad de contacto, se pasa a la siguiente etapa de **calcular los contactos**, que calcula utilizando la forma de colisión real del objeto el punto de contacto exacto. Estos puntos de contacto se pasan a la última etapa de cálculo de la dinámica, que comienza con la **resolución de las restricciones**, donde se determina la respuesta a la colisión empleando las restricciones de movimientos que se han definido en la escena. Finalmente se **integran las posiciones** de los objetos, obteniendo las posiciones y orientaciones nuevas para este paso de simulación.

Tipos básicos

Bullet cuenta con un subconjunto de utilidades matemáticas básicas con tipos de datos y operadores definidos como *btScalar* (escalar en punto flotante), *btVector3* (vector en el espacio 3D), *btTransform* (transformación afin 3D), *btQuaternion*, *btMatrix3x3*...

Las estructuras de datos básicas que define Bullet se dividen en dos grupos: los **datos de colisión**, que dependen únicamente de la *forma del objeto* (no se tiene en cuenta las propiedades físicas, como la masa o la velocidad asociada al objeto), y los **datos de propiedades dinámicas** que almacenan las propiedades físicas como la masa, la inercia, las restricciones y las articulaciones.

En los *datos de colisión* se distinguen las **formas de colisión**, las cajas **AABBs**, los **pares superpuestos** que mantiene una lista de parejas de cajas **AABB** que se solapan en algún eje, y los **puntos de contacto** que han sido calculados como resultado de la colisión.

6.5.2. Hola Mundo en Bullet

Para comenzar, estudiaremos el “Hola Mundo” en Bullet, que definirá una esfera que cae sobre un plano. Primero instalaremos las bibliotecas, que pueden descargarse de la página web del proyecto². Desde el directorio donde tengamos el código descomprimido, ejecutamos:

```
cmake . -G "Unix Makefiles" -DINSTALL_LIBS=ON
make
sudo make install
```

Obviamente, es necesario tener instalado *cmake* para compilar la biblioteca. A continuación estudiaremos el código de la simulación.

Listado 6.1: Hello World en Bullet.

```
1 #include <iostream>
2 #include <btBulletDynamicsCommon.h>
3
4 int main (void) {
5     btBroadphaseInterface* broadphase = new btDbvtBroadphase();
6     btDefaultCollisionConfiguration* collisionConfiguration =
7         new btDefaultCollisionConfiguration();
8     btCollisionDispatcher* dispatcher = new btCollisionDispatcher(
9         collisionConfiguration);
10    btSequentialImpulseConstraintSolver* solver = new
11        btSequentialImpulseConstraintSolver;
12    btDiscreteDynamicsWorld* dynamicsWorld = new
13        btDiscreteDynamicsWorld(dispatcher, broadphase, solver,
14        collisionConfiguration);
15
16    // Definicion de las propiedades del mundo -----
17    dynamicsWorld->setGravity(btVector3(0, -10, 0));
18
19    // Creacion de las formas de colision -----
20    btCollisionShape* groundShape =
21        new btStaticPlaneShape(btVector3(0, 1, 0), 1);
22    btCollisionShape* fallShape = new btSphereShape(1);
23
24    // Definicion de los cuerpos rigidos en la escena -----
25    btDefaultMotionState* groundMotionState = new
26        btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),
27        btVector3(0, -1, 0)));
28    btRigidBody::btRigidBodyConstructionInfo
29        groundRigidBodyCI(0, groundMotionState, groundShape, btVector3
30        (0, 0, 0));
31    btRigidBody* gRigidBody = new btRigidBody(groundRigidBodyCI);
32    dynamicsWorld->addRigidBody(gRigidBody);
33
34    btDefaultMotionState* fallMotionState =
35        new btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),
36        btVector3(0, 50, 0)));
37    btScalar mass = 1;
38    btVector3 fallInertia(0, 0, 0);
39    fallShape->calculateLocalInertia(mass, fallInertia);
40    btRigidBody::btRigidBodyConstructionInfo fallRigidBodyCI(mass,
41        fallMotionState, fallShape, fallInertia);
42    btRigidBody* fallRigidBody = new btRigidBody(fallRigidBodyCI);
43    dynamicsWorld->addRigidBody(fallRigidBody);
```

²<http://code.google.com/p/bullet/>


```

35
36 // Bucle principal de la simulacion -----
37 for (int i=0 ; i<300 ; i++) {
38     dynamicsWorld->stepSimulation(1/60.f,10);    btTransform trans;
39     fallRigidBody->getMotionState()->getWorldTransform(trans);
40     std::cout << "Altura: " << trans.getOrigin().getY() << std::
        endl;
41 }
42
43 // Finalizacion (limpieza) -----
44 dynamicsWorld->removeRigidBody(fallRigidBody);
45 delete fallRigidBody->getMotionState(); delete fallRigidBody;
46 dynamicsWorld->removeRigidBody(gRigidBody);
47 delete gRigidBody->getMotionState(); delete gRigidBody;
48 delete fallShape; delete groundShape;
49 delete dynamicsWorld; delete solver;
50 delete collisionConfiguration;
51 delete dispatcher; delete broadphase;
52
53 return 0;
54 }

```

```

Altura: 49.9972
Altura: 49.9917
Altura: 49.9833
Altura: 49.9722
Altura: 49.9583
Altura: 49.9417
Altura: 49.9222
Altura: 49.9
Altura: 49.875
Altura: 49.8472
Altura: 49.8167
Altura: 49.7833
Altura: 49.7472
Altura: 49.7083
Altura: 49.6667
...

```

Figura 6.21: Salida por pantalla de la ejecución del *Hola Mundo* en Bullet.

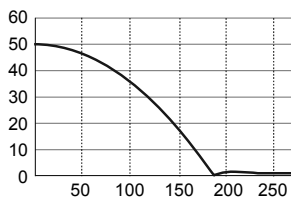


Figura 6.22: Representación de los valores obtenidos en el *Hola Mundo*.

El ejemplo anterior podría compilarse con un sencillo makefile como se muestra a continuación:

Listado 6.2: Makefile para hello world.

```

1 CXXFLAGS := `pkg-config --cflags bullet`
2 LDLIBS := `pkg-config --libs-only-l bullet`
3
4 all: HelloWorldBullet
5
6 clean:
7     rm HelloWorldBullet *~

```

Una vez compilado el programa de ejemplo, al ejecutarlo obtenemos un resultado puramente textual, como el mostrado en la Figura 6.21. Como hemos comentado anteriormente, los Bullet está diseñado para permitir un uso modular. En este primer ejemplo no se ha hecho uso de ninguna biblioteca para la representación gráfica de la escena.

Si representamos los 300 valores de la altura de la esfera, obtenemos una gráfica como muestra la Figura 6.22. Como vemos, cuando la esfera (de 1 metro de radio) llega a un metro del suelo (medido desde el centro), rebota levemente y a partir del frame 230 aproximadamente se estabiliza.

El primer `include` definido en la línea ② del programa anterior se encarga de incluir todos los archivos de cabecera necesarios para crear una aplicación que haga uso del módulo de dinámicas (cuerpo rígido, restricciones, etc...). Necesitaremos instanciar un mundo sobre el que realizar la simulación. En nuestro caso crearemos un mundo discreto, que es el adecuado salvo que tengamos objetos de movimiento muy rápido sobre el que tengamos que hacer una detección de su movimiento (en cuyo caso podríamos utilizar la clase aún experimental `btContinuousDynamicsWorld`).

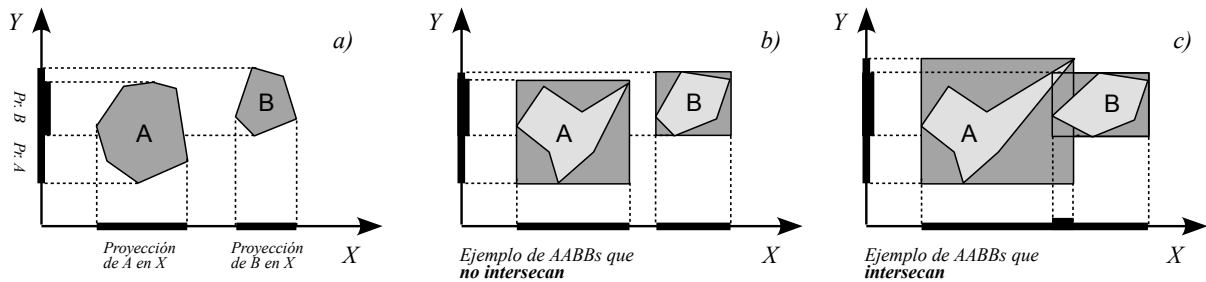


Figura 6.23: Aplicación del Teorema de los Ejes Separados y uso con cajas AABB. **a)** La proyección de las formas convexas de los objetos A y B están separadas en el eje X, pero no en el eje Y. Como podemos encontrar un eje sobre el que ambas proyecciones no intersecan, podemos asegurar que las formas no colisionan. **b)** El mismo principio aplicados sobre cajas AABB definidas en objetos cóncavos, que no intersecan. **c)** La proyección de las cajas AABB se solapa en todos los ejes. Hay un posible caso de colisión entre formas.

Creación del mundo

Como vimos en la sección 6.2.2, los motores de simulación física utilizan varias capas para detectar las colisiones entre los objetos del mundo. Bullet permite utilizar algoritmos en una primera etapa (*broadphase*) que utilizan las cajas límite de los objetos del mundo para obtener una lista de pares de cajas que pueden estar en colisión. Esta lista es una lista exhaustiva de todas las cajas límite que intersecan en alguno de los ejes (aunque, en etapas posteriores lleguen a descartarse porque en realidad no colisionan).

Muchos sistemas de simulación física se basan en el *Teorema de los Ejes Separados*. Este teorema dice que si existe un eje sobre el que la proyección de dos formas convexas no se solapan, entonces podemos asegurar que las dos formas no colisionan. Si no existe dicho eje y las dos formas son convexas, podemos asegurar que las formas colisionan. Si las formas son cóncavas, podría ocurrir que no colisionaran (dependiendo de la *suerte* que tengamos con la forma de los objetos). Este teorema se puede visualizar fácilmente en 2 dimensiones, como se muestra en la Figura 6.23.a.

El mismo teorema puede aplicarse para cajas AABB. Además, el hecho de que las cajas AABB estén perfectamente alineadas con los ejes del sistema de referencia, hace que el cálculo de la proyección de estas cajas sea muy rápida (simplemente podemos utilizar las coordenadas mínimas y máximas que definen las cajas).

La Figura 6.23 representa la aplicación de este teorema sobre cajas AABB. En el caso c) de dicha figura puede comprobarse que la proyección de las cajas se solapa en todos los ejes, por lo que tendríamos un caso potencial de colisión. En realidad, como vemos en la figura las formas que contienen dichas cajas AABB no colisionan, pero este caso deberá ser resuelto por algoritmos de detección de colisión de menor granularidad.

En la línea 5 creamos un objeto que implementa un algoritmo de optimización en una primera etapa *broadphase*. En posteriores etapas

Bullet calculará las colisiones exactas. Existen dos algoritmos básicos que implementa Bullet para mejorar al aproximación *a ciegas* de complejidad $O(n^2)$ que comprobaría toda la lista de pares. Estos algoritmos añaden nuevas parejas de cajas que en realidad no colisionan, aunque en general mejoran el tiempo de ejecución.

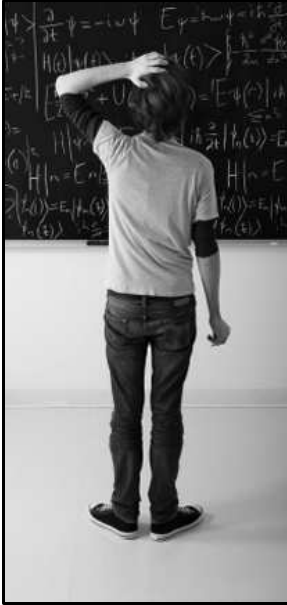


Figura 6.24: El objeto *solver* se encargará de resolver la interacción entre objetos.

Reutiliza!!

Es buena práctica reutilizar formas de colisión. Si varios objetos de la escena pueden compartir la misma forma de colisión (por ejemplo, todos los enemigos pueden gestionarse con una misma esfera de un determinado radio), es buena práctica compartir esa forma de colisión entre todos ellos.

- **Árbol AABB Dinámico.** Este algoritmo está implementado en la clase `btDbvtBroadphase`. Se construye un árbol AABB de propósito general que se utiliza tanto en la primera etapa de optimización *broadphase* como en la detección de colisiones entre *softbodies*. Este tipo de árbol se adapta automáticamente a las dimensiones del mundo, y la inserción y eliminación de objetos es más rápido que en SAP.
- **Barrido y Poda (SAP).** La implementación de *Sweep and Prune* de Bullet requiere que el tamaño del mundo sea conocido de previamente. Este método es el que tiene mejor comportamiento en mundos dinámicos donde la mayoría de los objetos tienen poco movimiento. Se implementa en el conjunto de clases `AxisSweep` (con versiones de diverso nivel de precisión).

Tras esta primera *poda*, hemos eliminado gran cantidad de objetos que no colisionan. A continuación, en las líneas [6-8] se crea un objeto de configuración de la colisión, que nos permitirá adaptar los parámetros de los algoritmos utilizados en posteriores fases para comprobar la colisión. El `btCollisionDispatcher` es una clase que permite añadir funciones de *callback* para ciertos tipos de eventos (como por ejemplo, cuando los objetos se encuentren *cerca*).

El objeto *solver* (línea [9]) se encarga de que los objetos interactúen adecuadamente, teniendo en cuenta la gravedad, las fuerzas, colisiones y restricciones. En este ejemplo se ha utilizado la versión secuencial (que implementa el método de Gauss Seidel proyectado (PGS), para resolver problemas lineales), aunque existen versiones que hacen uso de paralelismo empleando hilos.

En la línea [10] se instancia el mundo. Este objeto nos permitirá añadir los objetos del mundo, aplicar gravedad, y avanzar el paso de la simulación. En concreto, en la línea [12] se establece una de las propiedades del mundo, la gravedad, asignando un valor de 10m/s en el eje Y, por lo que se aplicará sobre ese eje la fuerza de gravedad.

Al finalizar, Bullet requiere que el usuario libere la memoria que ha utilizado explícitamente. De esta forma, a partir de la línea [42] se eliminan todos los elementos que han sido creados a lo largo de la simulación.

Hasta aquí hemos definido lo que puede ser el esqueleto básico de una aplicación mínima de Bullet. Vamos a definir a continuación los objetos que forman nuestra escena y el bucle de simulación.

Formas de Colisión

Como hemos comentado al inicio de la sección, crearemos un objeto plano que servirá como suelo sobre el que dejaremos caer una esfera. Cada uno de estos cuerpos necesita una forma de colisión, que internamente únicamente se utiliza para calcular la colisión (no tiene propiedades de masa, inercia, etc...).

Las formas de colisión no tienen una posición en el mundo; se adjuntan a los cuerpos rígidos. La elección de la forma de colisión adecuada, además de mejorar el rendimiento de la simulación, ayuda a conseguir una simulación de calidad. Bullet permite el uso de **primitivas** (que implementan algoritmos de detección de colisiones muy optimizados) o **mallas poligonales**. Las primitivas soportadas por Bullet son:

- **btSphereShape**. Esfera; la primitiva más simple y rápida.
- **btBoxShape**. La caja puede tener cualquier relación de aspecto.
- **btCylinderShape**. Cilindro con cualquier relación de aspecto.
- **btCapsuleShape**. Cápsula con cualquier relación de aspecto.
- **btConeShape**. Los conos se definen con el vértice en el (0,0,0).
- **btMultiSphereShape**. Forma convexa especial definida como combinación de esferas.
- **btCompoundShape**. No es una primitiva básica en sí, sino que permite combinar formas de cualquier tipo (tanto primitivas como formas de colisión de tipo malla que veremos a continuación). Permite obtener formas compuestas, como la que se estudió en la Figura 6.14.

Las formas de colisión de tipo malla soportadas son:

- **btConvexHull**. Este es el tipo de forma de tipo malla más rápido. Se define como una nube de vértices que forman la forma convexa más pequeña posible. El número de vértices debe ser pequeño para que la forma funcione adecuadamente. El número de vértices puede reducirse empleando la utilidad proporcionada por la clase *btShapeHull*. Existe una versión similar a este tipo llamado **btConvexTriangleMeshShape**, que está formado por caras triangulares, aunque es deseable utilizar *btConvexHull* porque es mucho más eficiente.
- **btBvhTriangleMeshShape**. Malla triangular estática. Puede tener un número considerable de polígonos, ya que utiliza una jerarquía interna para calcular la colisión. Como la construcción de esta estructura de datos puede llevar tiempo, se recomienda serializar el árbol para cargarlo rápidamente. Bullet incorpora utilidades para la serialización y carga del árbol BVH.

- **btHeightfieldTerrainShape**. Malla poligonal estática optimizada descrita por un mapa de alturas.
- **btStaticPlaneShape**. Plano infinito estático. Se especifica mediante un vector de dirección y una distancia respecto del origen del sistema de coordenadas.



Algunos consejos sobre el uso de formas de colisión en Bullet:

- Trata de utilizar las formas de colisión más eficientes: esferas, cajas, cilindros y *ConvexHull*.
- Los objetos dinámicos deben tener una forma cerrada y definida por un volumen finito. Algunas formas de colisión como los planos o las *triangleMesh* no tienen un volumen finito, por lo que únicamente pueden ser usados como cuerpos estáticos.
- Reutiliza siempre que sea posible las formas de colisión.

En la línea [16-17](#) creamos una forma de colisión de tipo plano, pasando como parámetro el vector normal del plano (vector unitario en Y), y una distancia respecto del origen. Así, el plano de colisión queda definido por la ecuación $y = 1$.

De igual modo, la forma de colisión del cuerpo que dejaremos caer sobre el suelo será una esfera de radio 1 metro ([línea 18](#)).

Una vez definidas las formas de colisión, las posicionaremos asociándolas a instancias de cuerpos rígidos. En la siguiente subsección añadiremos los cuerpos rígidos al mundo.

Cuerpos Rígidos

Para añadir cuerpos rígidos, necesitamos primero definir el concepto de **MotionState** en Bullet. Un *MotionState* es una abstracción proporcionada por Bullet para actualizar la posición de los objetos que serán dibujados en el *game loop*. Empleando *MotionStates*, Bullet se encargará de actualizar los objetos que serán representados por el motor gráfico. En la siguiente sección estudiaremos cómo trabajar con *MotionStates* en Ogre.

MotionState propio

Para implementar nuestro propio *MotionState* basta con heredar de *btMotionState* y sobrescribir los métodos `getWorldTransform` y `setWorldTransform`.

Gracias al uso de *MotionStates*, únicamente se actualiza la posición de los objetos que se han movido. Bullet se encarga además de la interpolación de movimientos, aislando al programador de esta tarea. Cuando se consulte la posición de un objeto, por defecto se devolverá la correspondiente al último paso de simulación calculado. Sin embargo, cada vez que el motor gráfico necesite redibujar la escena, Bullet se encargará de devolver la transformación interpolada.

Los *MotionStates* deben utilizarse en dos situaciones:

1. Cuando se crea un cuerpo. Bullet determina la posición inicial del cuerpo en el momento de su creación, y requiere una llamada al *MotionState*.
2. Cuando se quiera actualizar la posición del objeto.

Bullet proporciona un *MotionState* por defecto que podemos utilizar para instanciar cuerpos rígidos. Así, en la línea (21) se utiliza el *MotionState* por defecto especificando como rotación la identidad, y trasladando el origen -1 unidad en Y³.

En las líneas (22-23) se emplea la estructura *btRigidBodyConstructionInfo* para establecer la información para crear un cuerpo rígido.



Los componentes de la estructura *btRigidBodyConstructionInfo* se copian a la información del cuerpo cuando se llama al constructor. Si queremos crear un grupo de objetos con las mismas propiedades, puede crearse una única estructura de este tipo y pasarla al constructor de todos los cuerpos.

El primer parámetro es la masa del objeto. Estableciendo una masa igual a cero (primer parámetro), se crea un objeto estático (equivale a establecer una masa infinita, de modo que el objeto no se puede mover). El último parámetro es la inercia del suelo (que se establece igualmente a 0, por ser un objeto estático).

En la línea (24) creamos el objeto rígido a partir de la información almacenada en la estructura anterior, y lo añadimos al mundo en la línea (25).

La creación de la esfera sigue un patrón de código similar. En la línea (27) se crea el *MotionState* para el objeto que dejaremos caer, situado a 50 metros del suelo (línea (28)).

En las líneas (29-31) se establecen las propiedades del cuerpo; una masa de 1Kg y se llama a un método de *btCollisionShape* que nos calcula la inercia de una esfera a partir de su masa.

Bucle Principal

Para finalizar, el bucle principal se ejecuta en las líneas (36-41). El bucle se ejecuta 300 veces, llamando al paso de simulación con un intervalo de 60hz. En cada paso de la simulación se imprime la altura de la esfera sobre el suelo.

Como puede verse, la posición y la orientación del objeto dinámico se encapsulan en un objeto de tipo *btTransform*. Como se comentó anteriormente, esta información puede obtenerse a partir del *MotionS-*

³Esta traslación se realiza a modo de ejemplo para compensar la traslación de 1 unidad cuando se creó la forma de colisión del plano. El resultado sería el mismo si en ambos parámetros se hubiera puesto 0.

Tiempos!

Cuidado, ya que las funciones de cálculo de tiempo habitualmente devuelven los resultados en milisegundos. Bullet trabaja en segundos, por lo que ésta es una fuente habitual de errores.

tate asociado al *btRigidBody* a través de la estructura de inicialización *btRigidBodyConstructInfo*.

El método para avanzar un paso en la simulación (línea [38](#)) requiere dos parámetros. El primero describe la cantidad de tiempo que queremos avanzar la simulación. Bullet tiene un reloj interno que permite mantener constante esta actualización, de forma que sea independiente de la tasa de frames de la aplicación. El segundo parámetro es el número de subpasos que debe realizar bullet cada vez que se llama *stepSimulation*. Los tiempos se miden en segundos.

El primer parámetro debe ser siempre menor que el número de subpasos multiplicado por el tiempo fijo de cada paso $t_{Step} < maxSubStep \times t_{FixedStep}$.



Supongamos que queremos un tiempo fijo de simulación en cada paso de 60hz. En el mejor de los casos, nuestro videojuego tendrá una tasa de 120fps (120hz), y en el peor de los casos de 12fps. Así, t_{Step} en el primer caso será $1/120 = 0,0083$, y en el segundo $t_{Step} = 1/12 = 0,083$. Por su parte, el tiempo del paso fijo para la simulación sería $1/60 = 0,017$. Para que la expresión anterior se cumpla, en el primer caso el número de subpasos basta con 1 $0,0083 < 1 \times 0,017$. En el peor de los casos, necesitaremos que el número de pasos sea al menos de 5 para que se cumpla la expresión $0,083 < 5 \times 0,017$. Con estas condiciones tendríamos que establecer el número de subpasos a 5 para no *perder tiempo de simulación*

No olvides...

Cuando cambies el valor de los tiempos de simulación, recuerda calcular el número de subpasos de simulación para que la ecuación siga siendo correcta.

Decrementando el tamaño de cada paso de simulación se está aumentando la resolución de la simulación física. De este modo, si en el juego hay objetos que “atravesan” objetos (como paredes), es posible decrementar el *fixedTimeStep* para aumentar la resolución. Obviamente, cuando se aumenta la resolución al doble, se necesitará aumentar el número de *maxSubSteps* al doble, lo que requerirá aproximadamente el doble de tiempo de CPU para el mismo tiempo de simulación física.

Cuando se especifica un valor de *maxSubSteps* > 1 , Bullet interpolará el movimiento (y evitará al programador tener que realizar los cálculos). Si *maxSubSteps* $== 1$, no realizará interpolación.

6.6. Integración manual en Ogre

Como se ha estudiado en la sección anterior, los *MotionStates* se definen en Bullet para abstraer la representación de los *rigidBody* en el motor de dibujado. A continuación definiremos manualmente una clase `MyMotionState` que se encargará de la actualización de las entidades en Ogre.

La implementación de un *MotionState* propio debe heredar de la clase `btMotionState` de bullet, y sobrescribir los métodos `getWorldTransform` y `setWorldTransform` (por lo que se definen como virtuales). Ambos métodos toman como parámetro un objeto de la clase `btTransform`, que se utiliza para la representación interna de transformaciones de cuerpo rígido.

El siguiente listado muestra la declaración de la clase, que tiene dos variables miembro; el nodo asociado a ese *MotionState* (que tendremos que actualizar en `setWorldTransform`), y la propia transformación que devolveremos en `getWorldTransform` (línea [7]).

Listado 6.3: MyMotionState.h

```

1 #include <Ogre.h>
2 #include <btBulletDynamicsCommon.h>
3
4 class MyMotionState : public btMotionState {
5 protected:
6     Ogre::SceneNode* _visibleobj;
7     btTransform _pos;
8
9 public:
10    MyMotionState(const btTransform &initialpos,
11                  Ogre::SceneNode* node);
12    virtual ~MyMotionState();
13    void setNode(Ogre::SceneNode* node);
14    virtual void getWorldTransform(btTransform &worldTr) const;
15    virtual void setWorldTransform(const btTransform &worldTr);
16 };

```

La definición de la clase es directa. El siguiente listado muestra los métodos más importantes en su implementación (el destructor no tiene que eliminar el nodo; se encargará Ogre al liberar los recursos).

En las líneas [15-19] se define el método principal, que actualiza la posición y rotación del `SceneNode` en Ogre. Dado que Bullet y Ogre definen clases distintas para trabajar con Vectores y Cuaternios, es necesario obtener la rotación y posición del objeto por separado y asignarlo al nodo mediante las llamadas a `setOrientation` y `setPosition` (líneas [17] y [19]).

La llamada a `setWorldTransform` puede retornar en la línea [15] si no se ha establecido nodo en el constructor. Se habilita un método específico para establecer el nodo más adelante. Esto es interesante si se quieren añadir objetos a la simulación que no tengan representación gráfica.

btTransform

Las transformaciones de cuerpo rígido están formadas únicamente por traslaciones y rotaciones (sin escalado). Así, esta clase utiliza internamente un `btVector3` para la traslación y una matriz 3x3 para almacenar la rotación.



Figura 6.25: Fragmento del resultado de integración del “Hola Mundo” de Bullet en Ogre, empleando la clase de *MyMotionState* definida anteriormente.

Listado 6.4: MyMotionState.cpp

```

1 #include "MyMotionState.h"
2
3 MyMotionState::MyMotionState(const btTransform &initialpos,
4   Ogre::SceneNode *node) {
5   _visibleobj = node; _pos = initialpos;
6 }
7
8 void MyMotionState::setNode(Ogre::SceneNode *node)
9   { _visibleobj = node; }
10
11 void MyMotionState::getWorldTransform (btTransform &worldTr) const
12   { worldTr = _pos; }
13
14 void MyMotionState::setWorldTransform(const btTransform &worldTr){
15   if(NULL == _visibleobj) return; // Si no hay nodo, return
16   btQuaternion rot = worldTr.getRotation();
17   _visibleobj->setOrientation(rot.w(), rot.x(), rot.y(), rot.z());
18   btVector3 pos = worldTr.getOrigin();
19   _visibleobj->setPosition(pos.x(), pos.y(), pos.z());
20 }

```

Una vez creada la clase que utilizaremos para definir el *MotionState*, la utilizaremos en el “Hola Mundo” construido en la sección anterior para representar la simulación con el plano y la esfera. El resultado que tendremos se muestra en la Figura 6.25. Para la construcción del ejemplo emplearemos como esqueleto base el *FrameListener* del Módulo 2 del curso.

Listado 6.5: MyFrameListener.cpp

```

1 #include "MyFrameListener.h"
2 #include "MyMotionState.h"
3
4 MyFrameListener::MyFrameListener(RenderWindow* win,
5   Camera* cam, OverlayManager *om, SceneManager *sm) {
6   // ... Omitida parte de la inicializacion
7   _broadphase = new btDbvtBroadphase();
8   _collisionConf = new btDefaultCollisionConfiguration();
9   _dispatcher = new btCollisionDispatcher(_collisionConf);
10  _solver = new btSequentialImpulseConstraintSolver;
11  _world = new btDiscreteDynamicsWorld(_dispatcher,_broadphase,
12    _solver,_collisionConf);
13  _world->setGravity(btVector3(0,-10,0));
14  CreateInitialWorld();
15 }
16
17 MyFrameListener::~MyFrameListener() {
18  _world->removeRigidBody(_fallRigidBody);
19  delete _fallRigidBody->getMotionState();
20  delete _fallRigidBody;
21  // ... Omitida la eliminacion de los objetos
22 }
23
24 void MyFrameListener::CreateInitialWorld() {
25  // Creacion de la entidad y del SceneNode -----
26  Plane plane1(Vector3::Vector3(0,1,0), 0);
27  MeshManager::getSingleton().createPlane("p1",
28    ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane1,
29    200, 200, 1, 1, true, 1, 20, 20, Vector3::UNIT_Z);
30  SceneNode* node = _sceneManager->createSceneNode("ground");
31  Entity* groundEnt = _sceneManager->createEntity("planeEnt", "p1");

```

```

32  groundEnt->setMaterialName("Ground");
33  node->attachObject(groundEnt);
34  _sceneManager->getRootSceneNode()->addChild(node);
35
36  // Creamos las formas de colision -----
37  _groundShape = new btStaticPlaneShape(btVector3(0,1,0),1);
38  _fallShape = new btSphereShape(1);
39
40  // Creamos el plano -----
41  MyMotionState* groundMotionState = new MyMotionState(
42      btTransform(btQuaternion(0,0,0,1),btVector3(0,-1,0)), node);
43  btRigidBody::btRigidBodyConstructionInfo groundRigidBodyCI
44      (0,groundMotionState,_groundShape,btVector3(0,0,0));
45  _groundRigidBody = new btRigidBody(groundRigidBodyCI);
46  _world->addRigidBody(_groundRigidBody);
47
48  // Creamos la esfera -----
49  Entity *entity2= _sceneManager->createEntity("ball","ball.mesh");
50  SceneNode *node2= _sceneManager->getRootSceneNode()->
51      createChildSceneNode();
52  node2->attachObject(entity2);
53  MyMotionState* fallMotionState = new MyMotionState(
54      btTransform(btQuaternion(0,0,0,1),btVector3(0,50,0)), node2);
55  btScalar mass = 1;   btVector3 fallInertia(0,0,0);
56  _fallShape->calculateLocalInertia(mass,fallInertia);
57  btRigidBody::btRigidBodyConstructionInfo fallRigidBodyCI(
58      mass,fallMotionState,_fallShape,fallInertia);
59  _fallRigidBody = new btRigidBody(fallRigidBodyCI);
60  _world->addRigidBody(_fallRigidBody);
61 }
62 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
63     Real deltaT = evt.timeSinceLastFrame;
64     int fps = 1.0 / deltaT;
65
66     _world->stepSimulation(deltaT, 5);    // Actualizar fisica
67
68     _keyboard->capture();
69     if (_keyboard->isKeyDown(OIS::KC_ESCAPE)) return false;
70
71     btVector3 impulse;
72     if (_keyboard->isKeyDown(OIS::KC_I)) impulse=btVector3(0,0,-.1);
73     if (_keyboard->isKeyDown(OIS::KC_J)) impulse=btVector3(-.1,0,0);
74     if (_keyboard->isKeyDown(OIS::KC_K)) impulse=btVector3(0,0,.1);
75     if (_keyboard->isKeyDown(OIS::KC_L)) impulse=btVector3(.1,0,0);
76     _fallRigidBody->applyCentralImpulse(impulse);
77
78     // Omitida parte del codigo fuente (manejo del raton, etc...)
79     return true;
80 }
81
82 bool MyFrameListener::frameEnded(const Ogre::FrameEvent& evt) {
83     Real deltaT = evt.timeSinceLastFrame;
84     _world->stepSimulation(deltaT, 5);    // Actualizar fisica
85     return true;
86 }

```

En la implementación del *FrameListener* es necesario mantener como variables miembro el conjunto de objetos necesarios en la simulación de Bullet. Así, la implementación del constructor (líneas [4-15](#)) define los objetos necesarios para crear el mundo de simulación de Bullet (línea [11-12](#)). Estos objetos serán liberados en el destructor de la clase (ver líneas [17-22](#)). De igual modo, los dos cuerpos rígidos que intervie-

Variables miembro

Mantener los objetos como variables miembro de la clase no deja de ser una mala decisión de diseño. En la sección 6.7 veremos cómo se gestionan listas dinámicas con los objetos y las formas de colisión.

Actualización del mundo

En el ejemplo anterior, se actualiza el paso de simulación igualmente en el método *frameEnded*. Bullet se encarga de interpolar las posiciones de dibujado de los objetos. Si se elimina la llamada a *stepSimulation*, el resultado de la simulación es mucho más brusco.

nen en la simulación y sus formas asociadas son variables miembro de la clase.

El método *CreateInitialWorld* (definido en las líneas [24-60](#)) se realiza como último paso en el constructor. En este método se añaden a la escena de Ogre y al mundo de Bullet los elementos que intervendrán en la simulación (en este caso la esfera y el plano).

La creación de las entidades y los nodos para el plano y la esfera (líneas [26-34](#) y [49-51](#) respectivamente) ya han sido estudiadas en el Módulo 2 del curso. La creación de las formas para el plano y la esfera (líneas [37-38](#)) fueron descritas en el código de la sección anterior. Cabe destacar que la malla exportada en *ball.mesh* (línea [49](#)) debe tener un radio de 1 unidad, para que la forma de colisión definida en la línea [38](#) se adapte bien a su representación gráfica.

Cada objeto tendrá asociado un *MotionState* de la clase definida anteriormente, que recibirá la rotación y traslación inicial, y el puntero al nodo que guarda la entidad a representar. En el caso del plano, se define en las líneas [41-42](#), y la esfera en [52-53](#).

Por último, tendremos que añadir código específico en los métodos de retrollamada de actualización del frame. En el listado anterior se muestra el código de *frameStarted* (líneas [62-86](#)). En la línea [66](#) se actualiza el paso de simulación de Bullet, empleando el tiempo transcurrido desde la última actualización. Además, si el usuario pulsa las teclas [I](#), [J](#), [K](#) o [L](#) (líneas [71-76](#)), se aplicará una fuerza sobre la esfera. Veremos más detalles sobre la aplicación de impulsos a los objetos en el ejemplo de la sección 6.8.

Para finalizar, se muestran los flags del Makefile necesarios para integrar Bullet en los ejemplos anteriores.

Listado 6.6: Fragmento de Makefile

```

1 # Flags de compilacion -----
2 CXXFLAGS := -I $(DIRHEA) -Wall `pkg-config --cflags OGRE` `pkg-
   config --cflags bullet`
3
4 # Flags del linker -----
5 LDFLAGS := `pkg-config --libs-only-L OGRE` `pkg-config --libs-only-
   l bullet`
6 LDLIBS := `pkg-config --libs-only-l OGRE` `pkg-config --libs-only-l
   bullet` -lOIS -lGL -lstdc++

```

6.7. Hola Mundo en OgreBullet

El desarrollo de un *wrapper* completo del motor Bullet puede ser una tarea costosa. Afortunadamente existen algunas alternativas que facilitan la integración del motor en Ogre, como el proyecto *OgreBullet*. *OgreBullet* se distribuye bajo una licencia MIT libre, y es multiplataforma.

Según el autor, *OgreBullet* puede ser considerado un wrapper en versión estable. La notificación de bugs, petición de nuevos requisitos

y ejemplos se mantiene en un apartado específico de los foros de Ogre⁴. Uno de los principales problemas relativos al uso de este wrapper es la falta de documentación, por lo que en algunos casos la única alternativa es la consulta de los archivos de cabecera de la distribución.



OgreBullet puede descargarse de la página de complementos oficial de Ogre en: <http://ogreaddons.svn.sourceforge.net/viewvc/ogreaddons/trunk/ogrebulet/?view=tar>

En el siguiente ejemplo crearemos una escena donde se añadirán de forma dinámica cuerpos rígidos. Además de un puntero al *DynamicsWorld* (variable miembro *_world*), el *FrameListener* mantiene un puntero a un objeto *_debugDrawer* (línea [7]), que nos permite representar cierta información visual que facilita el depurado de la aplicación. En este primer ejemplo se activa el dibujado de las formas de colisión (línea [8]), tal y como se muestra en la Figura 6.26.

Este objeto permite añadir otros elementos que faciliten la depuración, como líneas, puntos de contacto, cajas AABBs, etc. Los métodos relativos a la representación de texto 3D en modo depuración están previstos pero aún no se encuentran desarrollados. Este objeto de depuración debe ser añadido igualmente al grafo de escena de Ogre (líneas [9-11] del siguiente listado).

La definición del mundo en *OgreBullet* requiere que se especifiquen los límites de simulación. En las líneas [14-15] se crea una caja AABB descrita por los vértices de sus esquinas que define el volumen en el que se realizará la simulación física. Este límite, junto con el vector de gravedad, permitirán crear el mundo (líneas [18-19]).

Listado 6.7: Constructor

```

1 MyFrameListener::MyFrameListener(RenderWindow* win,
2   Camera* cam, OverlayManager *om, SceneManager *sm) {
3   _numEntities = 0; // Numero de Formas instanciadas
4   _timeLastObject = 0; // Tiempo desde ultimo objeto anadido
5
6   // Creacion del modulo de debug visual de Bullet -----
7   _debugDrawer = new OgreBulletCollisions::DebugDrawer();
8   _debugDrawer->setDrawWireframe(true);
9   SceneNode *node = _sceneManager->getRootSceneNode()->
10    createChildSceneNode("debugNode", Vector3::ZERO);
11   node->attachObject(static_cast<SimpleRenderable*>(_debugDrawer));
12   // Creacion del mundo (definicion de los limites y la gravedad)
13   AxisAlignedBox worldBounds = AxisAlignedBox
14    (Vector3(-10000,-10000,-10000), Vector3(10000,10000,10000));
15   Vector3 gravity = Vector3(0, -9.8, 0);
16   _world = new OgreBulletDynamics::DynamicsWorld(_sceneManager,
17    worldBounds, gravity);
18   _world->setDebugDrawer (_debugDrawer);
19   _world->setShowDebugShapes (true); // Muestra formas debug
20   CreateInitialWorld(); // Inicializa el mundo
21 }

```



Figura 6.26: Salida del primer ejemplo con *OgreBullet*. El objeto *_debugDrawer* muestra las formas de colisión asociadas a las entidades de Ogre.

⁴Foros OgreBullet: <http://www.ogre3d.org/addonforums/viewforum.php?f=12>

El *FrameListener* mantiene dos colas de doble fin (*deque*) de punteros a los *RigidBody* (*_bodies*) y a las *CollisionShape* (*_shapes*), que facilitan la inserción y borrado de elementos de un modo más rápido que los vectores. Así, cuando se añadan objetos de forma dinámica a la escena, será necesario añadirlos a estas estructuras para su posterior liberación en el destructor de la clase. El siguiente listado muestra la implementación del destructor del *FrameListener*.

De igual modo es necesario liberar los recursos asociados al mundo dinámico y al *debugDrawer* creado en el constructor (ver líneas [16-17](#)).

Listado 6.8: Destructor

```

1 MyFrameListener::~MyFrameListener() {
2     // Eliminar cuerpos rigidos -----
3     std::deque<OgreBulletDynamics::RigidBody *>::iterator
4         itBody = _bodies.begin();
5     while (_bodies.end() != itBody) {
6         delete *itBody; ++itBody;
7     }
8     // Eliminar formas de colision -----
9     std::deque<OgreBulletCollisions::CollisionShape *>::iterator
10        itShape = _shapes.begin();
11    while (_shapes.end() != itShape) {
12        delete *itShape; ++itShape;
13    }
14    _bodies.clear(); _shapes.clear();
15    // Eliminar mundo dinamico y debugDrawer -----
16    delete _world->getDebugDrawer(); _world->setDebugDrawer(0);
17    delete _world;
18 }

```

Para añadir un objeto de simulación de *OgreBullet* debemos crear dos elementos básicos; por un lado la *CollisionShape* (líneas [14-16](#)), y por otro lado el *RigidBody* (líneas [17-18](#)).

La asociación del nodo de dibujado con el cuerpo rígido se establece en la misma llamada en la que se asocia la forma de colisión al *RigidBody*. En la clase *OgreBulletDynamics::RigidBody* existen dos métodos que permiten asociar una forma de colisión a un *RigidBody*; *setShape* y *setStaticShape*. La segunda cuenta con varias versiones; una de ellas no requiere especificar el *SceneNode*, y se corresponde con la utilizada en la línea [21](#) para añadir la forma de colisión al plano.

Añadir a las colas

Una vez añadido el objeto, deben añadirse las referencias a la forma de colisión y al cuerpo rígido en las colas de la clase (línea 24).

Listado 6.9: CreateInitialWorld

```

1 void MyFrameListener::CreateInitialWorld() {
2     // Creacion de la entidad y del SceneNode -----
3     Plane plane1(Vector3::Vector3(0,1,0), 0);
4     MeshManager::getSingleton().createPlane("p1",
5         ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane1,
6         200, 200, 1, 1, true, 1, 20, 20, Vector3::UNIT_Z);
7     SceneNode* node= _sceneManager->createSceneNode("ground");
8     Entity* groundEnt= _sceneManager->createEntity("planeEnt", "p1");
9     groundEnt->setMaterialName("Ground");
10    node->attachObject(groundEnt);
11    _sceneManager->getRootSceneNode()->addChild(node);
12
13    // Creamos forma de colision para el plano -----

```

```

14 OgreBulletCollisions::CollisionShape *Shape;
15 Shape = new OgreBulletCollisions::StaticPlaneCollisionShape
16   (Ogre::Vector3(0,1,0), 0); // Vector normal y distancia
17 OgreBulletDynamics::RigidBody *rigidBodyPlane = new
18   OgreBulletDynamics::RigidBody("rigidBodyPlane", _world);
19
20 // Creamos la forma estatica (forma, Restitucion, Friccion) ----
21 rigidBodyPlane->setStaticShape(Shape, 0.1, 0.8);
22
23 // Anadimos los objetos Shape y RigidBody -----
24 _shapes.push_back(Shape);      _bodies.push_back(rigidBodyPlane);
25 }

```

La actualización del mundo se realiza de forma similar a la estudiada anteriormente. La aplicación de ejemplo además, permite añadir objetos dinámicos cuando se pulse la tecla **B** (línea 9). A continuación veremos el código para añadir cuerpos dinámicos.

Listado 6.10: FrameStarted

```

1 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
2   Ogre::Real deltaT = evt.timeSinceLastFrame;
3   _world->stepSimulation(deltaT); // Actualizar simulacion Bullet
4   _timeLastObject -= deltaT;
5
6   _keyboard->capture();
7   if (_keyboard->isKeyDown(OIS::KC_ESCAPE)) return false;
8   if ((_keyboard->isKeyDown(OIS::KC_B)) && (_timeLastObject <= 0))
9     AddDynamicObject();
10  // Omitido el resto del cogido del metodo ...
11  return true;
12 }

```

El siguiente listado implementa la funcionalidad de añadir cajas dinámicas a la escena. Los objetos se crearán teniendo en cuenta la posición y rotación de la cámara. Para ello, se toma como vector de posición inicial el calculado como la posición de la cámara desplazada 10 unidades según su vector dirección (líneas 5-6).

Listado 6.11: AddDynamicObject

```

1 void MyFrameListener::AddDynamicObject() {
2   _timeLastObject = 0.25; // Segundos para anadir uno nuevo...
3
4   Vector3 size = Vector3::ZERO; // Tamano y posicion inicial
5   Vector3 position = (_camera->getDerivedPosition()
6     + _camera->getDerivedDirection().normalisedCopy() * 10);
7
8   // Creamos la entidad y el nodo de la escena -----
9   Entity *entity = _sceneManager->createEntity("Box" +
10     StringConverter::toString(_numEntities), "cube.mesh");
11   entity->setMaterialName("cube");
12   SceneNode *node = _sceneManager->getRootSceneNode()->
13     createChildSceneNode();
14   node->attachObject(entity);
15
16   // Obtenemos la bounding box de la entidad creada -----
17   AxisAlignedBox boundingB = entity->getBoundingBox();
18   size = boundingB.getSize();
19   size /= 2.0f; // Tamano en Bullet desde el centro (la mitad)

```

stepSimulation

La llamada a *stepSimulation* en *OgreBullet* acepta dos parámetros opcionales, el número de subpasos de simulación (por defecto a 1), y el *fixedTimeStep* (por defecto 1/60).

```

20  OgreBulletCollisions::BoxCollisionShape *boxShape = new
21  OgreBulletCollisions::BoxCollisionShape(size);
22  OgreBulletDynamics::RigidBody *rigidBox = new
23  OgreBulletDynamics::RigidBody("rigidBox" +
24  StringConverter::toString(_numEntities), _world);
25  rigidBox->setShape(node, boxShape,
26  /* Restitucion, Friccion, Masa */ 0.6, 0.6, 5.0,
27  /* Pos. y Orient. */ position , Quaternion::IDENTITY);
28  rigidBox->setLinearVelocity(
29  _camera->getDerivedDirection().normalisedCopy() * 7.0);
30  _numEntities++;
31  // Anadimos los objetos a las deques -----
32  _shapes.push_back(boxShape);  _bodies.push_back(rigidBox);
33  }

```

En el listado anterior, el nodo asociado a cada caja se añade en la llamada a *setShape* (líneas [25-27](#)). Pese a que Bullet soporta multitud de propiedades en la estructura *btRigidBodyConstructionInfo*, el wrapper se centra exclusivamente en la definición de la masa, y los coeficientes de fricción y restitución. La posición inicial y el cuaternio se indican igualmente en la llamada al método, que nos abstrae de la necesidad de definir el *MotionState*.

Las cajas se añaden a la escena con una velocidad lineal relativa a la rotación de la cámara (ver líneas [28-29](#)).

6.8. RayQueries

Al inicio del capítulo estudiamos algunos tipos de preguntas que podían realizarse al motor de simulación física. Uno de ellos eran los *RayQueries* que permitían obtener las formas de colisión que intersecaban con un determinado rayo.

Ogre? Bullet?

Recordemos que los objetos de simulación física no son conocidos por Ogre. Aunque en el módulo 2 del curso estudiamos los *RayQueries* en Ogre, es necesario realizar la pregunta en Bullet para obtener las referencias a los *RigidBody*.

Utilizaremos esta funcionalidad del SDC para aplicar un determinado impulso al primer objeto que sea *tocado* por el puntero del ratón. De igual forma, en este ejemplo se añadirán objetos definiendo una forma de colisión convexa. El resultado de la simulación (activando la representación de las formas de colisión) se muestra en la Figura 6.27.

La llamada al método *AddDynamicObject* recibe como parámetro un tipo enumerado, que indica si queremos añadir una oveja o una caja. La forma de colisión de la caja se calcula automáticamente empleando la clase *StaticMeshToShapeConverter* (línea [17](#)).



Reutiliza las formas de colisión! Tanto en el ejemplo de la sección anterior como en este código, no se reutilizan las formas de colisión. Queda como ejercicio propuesto para el lector mantener referencias a las formas de colisión (para las ovejas y para las cajas), y comprobar la diferencia de rendimiento en frames por segundo cuando el número de objetos de la escena crece.

Listado 6.12: AddDynamicObject

```

1 void MyFrameListener::AddDynamicObject (TEntity tObject) {
2 // Omitido codigo anterior del metodo -----
3 Entity *entity = NULL;
4 switch (tObject) {
5 case sheep:
6     entity = _sceneManager->createEntity("Sheep" +
7     StringConverter::toString(_numEntities), "sheep.mesh");
8     break;
9 case box: default:
10    // (Omitido) Analogamente se carga el modelo de la caja...
11 }
12
13 SceneNode *node = _sceneManager->getRootSceneNode()->
14     createChildSceneNode();
15 node->attachObject(entity);
16
17 OgreBulletCollisions::StaticMeshToShapeConverter *
18     trimeshConverter = NULL;
19 OgreBulletCollisions::CollisionShape *bodyShape = NULL;
20 OgreBulletDynamics::RigidBody *rigidBody = NULL;
21
22 switch (tObject) {
23 case sheep:
24     trimeshConverter = new
25     OgreBulletCollisions::StaticMeshToShapeConverter(entity);
26     bodyShape = trimeshConverter->createConvex();
27     delete trimeshConverter;
28     break;
29 case box: default:
30     // (Omitido) Crear bodyShape como en el ejemplo anterior...
31 }
32 rigidBody = new OgreBulletDynamics::RigidBody("rigidBody" +
33     StringConverter::toString(_numEntities), _world);
34 // Omitido resto de codigo del metodo -----
35 }

```

El objeto de la clase *StaticMeshToShapeConverter* recibe como parámetro una *Entity* de Ogre en el constructor. Esta entidad puede ser convertida a multitud de formas de colisión. En el momento de la creación, la clase reduce el número de vértices de la forma de colisión.

Cuando se pincha con el botón derecho o izquierdo del ratón sobre algún objeto de la simulación, se aplicará un impulso con diferente fuerza (definida en F , ver línea 18 del siguiente código). El método *pickBody* se encarga de obtener el primer cuerpo que colisiona con el rayo definido por la posición de la cámara y el puntero del ratón. Este método devuelve igualmente en los dos primeros parámetros el punto de colisión en el objeto y el rayo utilizado para construir el *RayQuery*.

El método *pickBody* primero obtiene el rayo utilizando la funcionalidad de Ogre, empleando las coordenadas de pantalla normalizadas (líneas 20-21). Hecho esto, se crea una *Query* que requiere como tercer parámetro la distancia máxima a la que se calculará la colisión, en la dirección del rayo, teniendo en cuenta su posición inicial (línea 4).

Si el rayo colisiona en algún cuerpo (línea 6), se devuelve el cuerpo y el punto de colisión (líneas 7-11).



Figura 6.27: Resultado de la simulación del ejemplo.

Conversor a Shape

Además de la llamada a *createConvex*, el conversor estudiado en el código anterior puede generar otras formas de colisión con *createSphere*, *createBox*, *createTrimesh*, *createCylinder* y *createConvexDecomposition* entre otras.

Listado 6.13: RayQuery en Bullet

```

1 RigidBody* MyFrameListener::pickBody (Vector3 &p, Ray &r, float x,
   float y) {
2   r = _camera->getCameraToViewportRay (x, y);
3   CollisionClosestRayResultCallback cQuery =
4     CollisionClosestRayResultCallback (r, _world, 10000);
5   _world->launchRay(cQuery);
6   if (cQuery.doesCollide()) {
7     RigidBody* body = static_cast <RigidBody *>
8       (cQuery.getCollidedObject());
9     p = cQuery.getCollisionPoint();
10    return body;
11  }
12  return NULL;
13 }
14
15 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
16 // Omitido codigo anterior del metodo -----
17 if (mbleft || mbright) { // Con botones del raton, impulso -----
18   float F = 10; if (mbright) F = 100;
19   RigidBody* body; Vector3 p; Ray r;
20   float x = posx/float(_win->getWidth()); // Pos x normalizada
21   float y = posy/float(_win->getHeight()); // Pos y normalizada
22   body = pickBody (p, r, x, y);
23
24   if (body) {
25     if (!body->isStaticObject()) {
26       body->enableActiveState ();
27       Vector3 relPos(p - body->getCenterOfMassPosition());
28       Vector3 impulse (r.getDirection ());
29       body->applyImpulse (impulse * F, relPos);
30     }
31   }
32 }
33 // Omitido resto de codigo del metodo -----
34 }

```

Para finalizar, si hubo colisión con algún cuerpo que no sea estático (líneas [24-25](#)), se aplicará un impulso. La llamada a *enableActiveState* permite activar un cuerpo. Por defecto, Bullet automáticamente desactiva objetos dinámicos cuando la velocidad es menor que un determinado umbral.

Los cuerpos desactivados en realidad están se encuentran en un estado de *dormidos*, y no consumen tiempo de ejecución salvo por la etapa de detección de colisión *broadphase*. Esta etapa automáticamente despierta a los objetos que estuvieran dormidos si se encuentra colisión con otros elementos de la escena.

Impulso

El impulso puede definirse como $\int F dt = \int (dp/dt) dt$, siendo p el momento.

En las líneas [27-29](#) se aplica un impulso sobre el objeto en la dirección del rayo que se calculó desde la cámara, con una fuerza proporcional a F . El impulso es una fuerza que actúa en un cuerpo en un determinado intervalo de tiempo. El impulso implica un cambio en el *momento*, siendo la *Fuerza* definida como el cambio en el momento. Así, el *impulso* aplicado sobre un objeto puede ser definido como la integral de la fuerza con respecto del tiempo.

El wrapper de *OgreBullet* permite definir un pequeño subconjunto de propiedades de los *RigidBody* de las soportadas en Bullet. Algunas de las principales propiedades son la Velocidad Lineal, Impulsos y

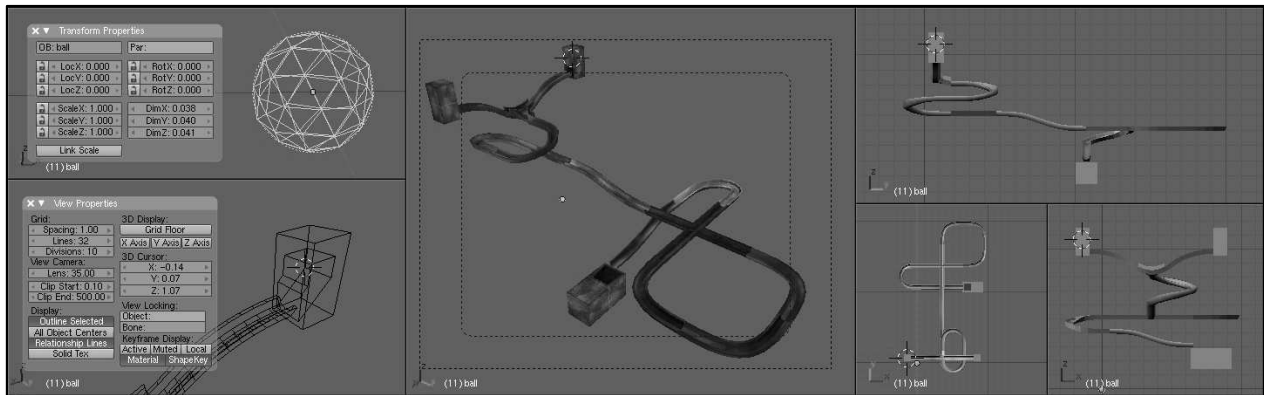


Figura 6.29: Configuración de la malla estática utilizada en el ejemplo. Es importante aplicar la escala y rotación a los objetos antes de su exportación, así como las dimensiones del objeto "ball" para aplicar los mismos límites a la *collision shape*.

Fuerzas. Si se requieren otras propiedades, será necesario acceder al objeto de la clase *btRigidBody* (mediante la llamada a *getBulletRigidBody*) y especificar manualmente las propiedades de simulación.

6.9. TriangleMeshCollisionShape

En este ejemplo se cargan dos objetos como mallas triangulares estáticas. El resultado de la ejecución puede verse en la Figura 6.28. Al igual que en el ejemplo anterior, se utiliza la funcionalidad proporcionada por el conversor de mallas, pero generando una *TriangleMeshCollisionShape* (línea [11-12](#)).



Figura 6.28: Resultado de la ejecución del ejemplo de carga de mallas triangulares.

Listado 6.14: Static Mesh

```

1 void MyFrameListener::CreateInitialWorld() {
2     // Creacion del track -----
3     Entity *entity = _sceneManager->createEntity("track.mesh");
4     SceneNode *node = _sceneManager->createSceneNode("track");
5     node->attachObject(entity);
6
7     _sceneManager->getRootSceneNode()->addChild(node);
8     OgreBulletCollisions::StaticMeshToShapeConverter *
9         trimeshConverter = new
10         OgreBulletCollisions::StaticMeshToShapeConverter(entity);
11     OgreBulletCollisions::TriangleMeshCollisionShape *trackTrimesh =
12         trimeshConverter->createTrimesh();
13
14     OgreBulletDynamics::RigidBody *rigidTrack = new
15         OgreBulletDynamics::RigidBody("track", _world);
16     rigidTrack->setShape(node, trackTrimesh, 0.8, 0.95, 0,
17         Vector3::ZERO, Quaternion::IDENTITY);
18
19     delete trimeshConverter;
20     // (Omitido) Creacion del sumidero de forma similar -----
21 }

```

Es importante consultar en la posición de los generadores de objetos en el espacio 3D (ver Figura 6.29), así como las dimensiones de los objetos que van a intervenir en la simulación. Por ejemplo, las esferas se creaban con una forma de colisión de tipo `SphereCollisionShape` de 0.02 unidades de radio porque su dimensión en el espacio 3D es de 0.04 unidades (ver Figura 6.29). De igual modo, una de las posiciones de generación es `Vector3(-0.14, 1.07, -0.07)` situada en el interior de una de las cajas.

6.10. Detección de colisiones

Una de las formas más sencillas de detectar colisiones entre objetos del mundo es iterar sobre los colectores de contactos (*contact manifold*). Los *contact manifold* son en realidad caches que contienen los puntos de contacto entre parejas de objetos de colisión. El siguiente listado muestra una forma de iterar sobre los pares de objetos en el mundo dinámico.

Listado 6.15: DetectCollisionDrain.

```

1 void MyFrameListener::DetectCollisionDrain() {
2   btCollisionWorld *bulletWorld=_world->getBulletCollisionWorld();
3   int numManifolds=bulletWorld->getDispatcher()->getNumManifolds();
4
5   for (int i=0;i<numManifolds;i++) {
6     btPersistentManifold* contactManifold =
7       bulletWorld->getDispatcher()->getManifoldByIndexInternal(i);
8     btCollisionObject* obA =
9       static_cast<btCollisionObject*>(contactManifold->getBody0());
10    btCollisionObject* obB =
11      static_cast<btCollisionObject*>(contactManifold->getBody1());
12
13    Ogre::SceneNode* drain = _sceneManager->getSceneNode("drain");
14
15    OgreBulletCollisions::Object *obDrain =
16      _world->findObject(drain);
17    OgreBulletCollisions::Object *obOB_A = _world->findObject(obA);
18    OgreBulletCollisions::Object *obOB_B = _world->findObject(obB);
19
20    if ((obOB_A == obDrain) || (obOB_B == obDrain)) {
21      Ogre::SceneNode* node = NULL;
22      if ((obOB_A != obDrain) && (obOB_A)) {
23        node = obOB_A->getRootNode(); delete obOB_A;
24      }
25      else if ((obOB_B != obDrain) && (obOB_B)) {
26        node = obOB_B->getRootNode(); delete obOB_B;
27      }
28      if (node) {
29        std::cout << node->getName() << std::endl;
30        _sceneManager->getRootSceneNode()->
31          removeAndDestroyChild (node->getName());
32      }
33    }
34  }
35 }

```



Figura 6.30: Ejemplo de detección de colisiones empleando colectores de contactos.

En la línea [2](#) se obtiene el puntero directamente a la clase *btCollisionWorld*, que se encuentra oculta en la implementación de *OgreBullet*. Con este puntero se accederá directamente a la funcionalidad de *Bullet* sin emplear la clase de recubrimiento de *OgreBullet*. La clase *btCollisionWorld* sirve a la vez como interfaz y como contenedor de las funcionalidades relativas a la detección de colisiones.

Mediante la llamada a *getDispatcher* (línea [3](#)) se obtiene un puntero a la clase *btDispatcher*, que se utiliza en la fase de colisión *broadphase* para la gestión de pares de colisión. Esta clase nos permite obtener el número de *colectores* que hay activos en cada instante. El bucle de las líneas [5-34](#) se encarga de iterar sobre los colectores. En la línea [6-7](#) se obtiene un puntero a un objeto de la clase *btPersistentManifold*. Esta clase es una implementación de una caché persistente mientras los objetos colisionen en la etapa de colisión *broadphase*.

OgreBullet...

El listado anterior muestra además cómo acceder al objeto del mundo de *bullet*, que permite utilizar gran cantidad de métodos que no están implementados en *OgreBullet*.



Los puntos de contacto se crean en la etapa de detección de colisiones fina (*narrow phase*). La cache de *btPersistentManifold* puede estar vacía o contener hasta un máximo de 4 puntos de colisión. Los algoritmos de detección de la colisión añaden y eliminan puntos de esta caché empleando ciertas heurísticas que limitan el máximo de puntos a 4. Es posible obtener el número de puntos de contacto asociados a la cache en cada instante mediante el método *getNumContacts()*.

La cache de colisión mantiene punteros a los dos objetos que están colisionando. Estos objetos pueden obtenerse mediante la llamada a métodos *get* (líneas [8-11](#)).

La clase *CollisionsWorld* de *OgreBullet* proporciona un método *findObject* que permite obtener un puntero a objeto genérico a partir de un *SceneNode* o un *btCollisionObject* (ver líneas [15-18](#)).

La última parte del código (líneas [20-32](#)) comprueba si alguno de los dos objetos de la colisión son el sumidero. En ese caso, se obtiene el puntero al otro objeto (que se corresponderá con un objeto de tipo esfera creado dinámicamente), y se elimina de la escena. Así, los objetos en esta segunda versión del ejemplo no llegan a añadirse en la caja de la parte inferior del circuito.



Otros mecanismos de colisión. En la documentación de Bullet se comentan brevemente otros mecanismos que pueden utilizarse para la detección de colisiones, como los objetos de la clase *btGhostObject*. Los objetos de esta clase pueden tener asociadas llamadas de *callback* de modo que se invoquen automáticamente cuando los objetos se solapan en la etapa de detección de colisiones mediante el test de cajas AABB.

6.11. Restricción de Vehículo

En esta sección estudiaremos cómo utilizar un tipo de restricción específica para la definición de vehículos. *OgreBullet* cuenta con abstracciones de alto nivel que trabajan internamente con llamadas a las clases derivadas *btRaycastVehicle*, que permiten convertir un cuerpo rígido en un vehículo.

A continuación estudiaremos algunos fragmentos de código empleados en el siguiente ejemplo para la construcción del vehículo. Queda como ejercicio propuesto añadir obstáculos y elementos de interacción en la escena empleando mallas triangulares estáticas.



Figura 6.31: Ejemplo de definición de un vehículo en OgreBullet.

Listado 6.16: Fragmento de MyFrameListener.h

```

1 OgreBulletDynamics::WheeledRigidBody *mCarChassis;
2 OgreBulletDynamics::VehicleTuning *mTuning;
3 OgreBulletDynamics::VehicleRayCaster *mVehicleRayCaster;
4 OgreBulletDynamics::RaycastVehicle *mVehicle;
5 Ogre::Entity *mChassis;
6 Ogre::Entity *mWheels[4];
7 Ogre::SceneNode *mWheelNodes[4];
8 float mSteering;
```

En el anterior archivo de cabecera se definen ciertas variables miembro de la clase que se utilizarán en la definición del vehículo. *mCarChassis* es un puntero a una clase que ofrece OgreBullet para la construcción de vehículos con ruedas. La clase *VehicleTuning* es una clase de cobertura sobre la clase *btVehicleTuning* de Bullet que permite especificar ciertas propiedades del vehículo (como la compresión, suspensión, deslizamiento, etc).

VehicleRayCaster es una clase que ofrece un interfaz entre la simulación del vehículo y el *RayCasting* (usado para localizar el punto de contacto entre el vehículo y el suelo). La clase *RaycastVehicle* es una clase de cobertura sobre la clase base de Bullet *btRaycastVehicle*. Las líneas [5-7] definen los nodos y entidades necesarias para el chasis y las ruedas del vehículo. Finalmente, la variable *Steering* de la línea [8] define la dirección del vehículo.

A continuación estudiaremos la definición del vehículo en el método *CreateInitialWorld* del *FrameListener*. La línea [1] del siguiente listado

define el vector de altura del chasis (elevación sobre el suelo), y la altura de conexión de las ruedas en él (línea 2) que será utilizado más adelante. En la construcción inicial del vehículo se establece la dirección del vehículo como 0.0 (línea 3).

Las líneas 5-9 crean la entidad del chasis y el nodo que la contendrá. La línea 10 utiliza el vector de altura del chasis para posicionar el nodo del chasis.

Listado 6.17: Fragmento de CreateInitialWorld (I).

```

1  const Ogre::Vector3 chassisShift(0, 1.0, 0);
2  float connectionHeight = 0.7f;
3  mSteering = 0.0;
4
5  mChassis = _sceneManager->createEntity("chassis", "chassis.mesh");
6  SceneNode *node = _sceneManager->getRootSceneNode()->
   createChildSceneNode ();
7
8  SceneNode *chassisnode = node->createChildSceneNode();
9  chassisnode->attachObject (mChassis);
10 chassisnode->setPosition (chassisShift);

```

El chasis tendrá asociada una forma de colisión de tipo caja (línea 1). Esta caja formará parte de una forma de colisión compuesta, que se define en la línea 2, y a la que se añade la caja anterior desplazada según el vector *chassisShift* (línea 3).

En la línea 4 se define el cuerpo rígido del vehículo, al que se asocia la forma de colisión creada anteriormente (línea 6). En la línea 9 se establecen los valores de suspensión del vehículo, y se evita que el vehículo pueda desactivarse (línea 8), de modo que el objeto no se «dormirá» incluso si se detiene durante un tiempo continuado.

Listado 6.18: Fragmento de CreateInitialWorld (II).

```

1  BoxCollisionShape* chassisShape = new BoxCollisionShape(Ogre::
   Vector3(1.f,0.75f,2.1f));
2  CompoundCollisionShape* compound = new CompoundCollisionShape();
3  compound->addChildShape(chassisShape, chassisShift);
4  mCarChassis = new WheeledRigidBody("carChassis", _world);
5  Vector3 CarPosition = Vector3(0, 0, -15);
6  mCarChassis->setShape (node, compound, 0.6, 0.6, 800, CarPosition,
   Quaternion::IDENTITY);
7  mCarChassis->setDamping(0.2, 0.2);
8  mCarChassis->disableDeactivation();

```

En el siguiente fragmento se comienza definiendo algunos parámetros de tuning del vehículo (línea 1). Estos parámetros son la rigidez, compresión y amortiguación de la suspensión y la fricción de deslizamiento. La línea 5 establece el sistema de coordenadas local del vehículo mediante los índices derecho, superior y adelante.

Las líneas 7 y 8 definen los ejes que se utilizarán como dirección del vehículo y en la definición de las ruedas.

El bucle de las líneas 10-16 construye los nodos de las ruedas, cargando 4 instancias de la malla «*wheel.mesh*».



El ejemplo desarrollado en esta sección trabaja únicamente con las dos ruedas delanteras (índices 0 y 1) como ruedas motrices. Además, ambas ruedas giran de forma simétrica según la variable de dirección *mSteering*. Queda propuesto como ejercicio modificar el código de esta sección para que la dirección se pueda realizar igualmente con las ruedas traseras, así como incorporar otras opciones de motricidad (ver Listado de *FrameStarted*).

Listado 6.19: Fragmento de CreateInitialWorld (III).

```

1 mTuning = new VehicleTuning(20.2, 4.4, 2.3, 500.0, 10.5);
2 mVehicleRayCaster = new VehicleRayCaster(_world);
3 mVehicle = new RaycastVehicle(mCarChassis, mTuning,
    mVehicleRayCaster);
4
5 mVehicle->setCoordinateSystem(0, 1, 2);
6
7 Ogre::Vector3 wheelDirectionCS0(0,-1,0);
8 Ogre::Vector3 wheelAxleCS(-1,0,0);
9
10 for (size_t i = 0; i < 4; i++) {
11     mWheels[i] = _sceneManager->createEntity("wheel"+i,"wheel.mesh");
12     mWheels[i]->setCastShadows(true);
13
14     mWheelNodes[i] = _sceneManager->getRootSceneNode()->
        createChildSceneNode();
15     mWheelNodes[i]->attachObject(mWheels[i]);
16 }

```

El siguiente fragmento de listado se repite para cada rueda, calculando el punto de conexión en función del ancho de cada rueda y la altura de conexión. Este punto es pasado al método *addWheel*, junto con información relativa a ciertas propiedades físicas de cada rueda. La variable *isFrontWheel* (ver línea ③) indica si la rueda añadida forma parte del conjunto de ruedas delanteras (en este caso, únicamente las dos primeras ruedas tendrán esta variable a *true* en el momento de creación).

Listado 6.20: Fragmento de CreateInitialWorld (IV).

```

1 Ogre::Vector3 connectionPointCS0 (1-(0.3*gWheelWidth),
    connectionHeight, 2-gWheelRadius);
2
3 mVehicle->addWheel(mWheelNodes[0], connectionPointCS0,
    wheelDirectionCS0, wheelAxleCS, gSuspensionRestLength,
    gWheelRadius, isFrontWheel, gWheelFriction, gRollInfluence);

```


Finalmente el método de callback del *FrameStarted* se encarga de modificar la fuerza que se aplica sobre el motor del vehículo cuando se utilizan los cursores superior e inferior del teclado. De igual modo, empleando los cursores izquierdo y derecho del teclado se modifica la dirección del vehículo (ver líneas [11-15](#)).

Listado 6.21: Fragmento de FrameStarted.

```
1 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
2     // Omitido el código anterior...
3     mVehicle->applyEngineForce (0,0);
4     mVehicle->applyEngineForce (0,1);
5
6     if (_keyboard->isKeyDown(OIS::KC_UP)) {
7         mVehicle->applyEngineForce (gEngineForce, 0);
8         mVehicle->applyEngineForce (gEngineForce, 1);
9     }
10
11    if (_keyboard->isKeyDown(OIS::KC_LEFT)) {
12        if (mSteering < 0.8) mSteering+=0.01;
13        mVehicle->setSteeringValue (mSteering, 0);
14        mVehicle->setSteeringValue (mSteering, 1);
15    }
16
17    // Omitido el resto del código...
18 }
```

6.12. Determinismo

El determinismo en el ámbito de la simulación física puede definirse de forma intuitiva como la posibilidad de «repetición» de un mismo comportamiento. En el caso de videojuegos esto puede ser interesante en la repetición de una misma jugada en un videojuego deportivo, o en la ejecución de una misma simulación física en los diferentes ordenadores de un videojuego multijugador. Incluso aunque el videojuego siga un enfoque con cálculos de simulación centrados en el servidor, habitualmente es necesario realizar ciertas interpolaciones del lado del cliente para mitigar los efectos de la latencia, por lo que resulta imprescindible tratar con enfoques deterministas.

Para lograr determinismo es necesario lograr que la simulación se realice *exactamente* con los mismos datos de entrada. Debido a la precisión en aritmética en punto flotante, es posible que $v \times 2 \times dt$ no de el mismo resultado que $v \times dt + v \times dt$. Así, es necesario emplear *el mismo* valor de dt en todas las simulaciones. Por otro lado, utilizar un dt fijo hace que no podamos representar la simulación de forma independiente de las capacidades de la máquina o la carga de representación gráfica concreta en cada momento. Así, nos interesa tener lo mejor de ambas aproximaciones; por un lado un tiempo fijo para conseguir el determinismo en la simulación, y por otro lado la gestión con diferentes tiempos asociados al framerate para lograr independencia de la máquina.

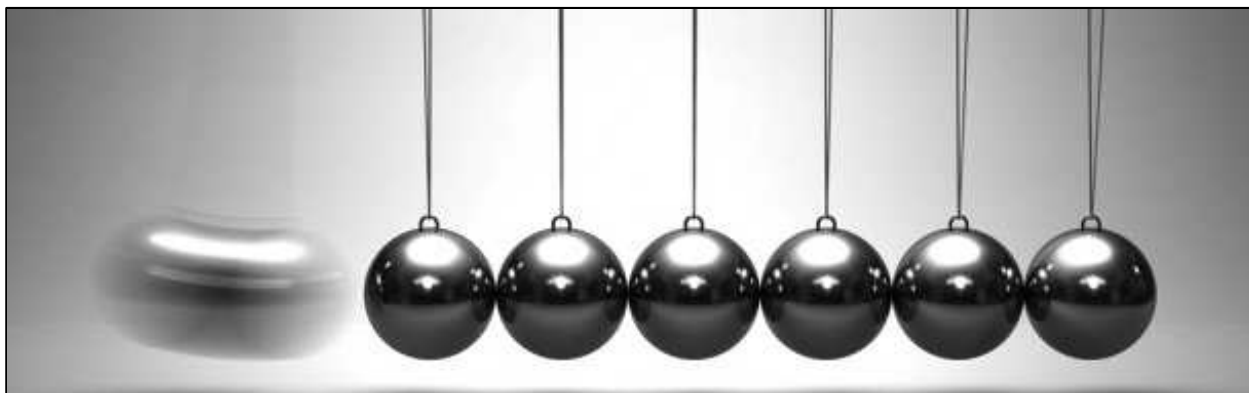


Figura 6.32: La gestión del determinismo puede ser un aspecto crítico en muchos videojuegos. El error de determinismo rápidamente se propaga haciendo que la simulación obtenga diferentes resultados.

Una posible manera de realizar la simulación sería la siguiente: el motor de simulación física se ejecuta por adelantado en intervalos de tiempo discretos dt , de modo que se mantengan los incrementos del motor gráfico con un intervalo adecuado. Por ejemplo, si queremos tener 50fps y la simulación física se ejecuta a 100fps, entonces tendríamos que ejecutar dos veces la simulación física por cada despliegue gráfico.

Esto es correcto con esos cálculos sencillos, pero ¿qué ocurre si queremos dibujar a 200fps?. En ese caso tendríamos que ejecutar la mitad de veces el simulador físico, pero no podemos calcular por adelantado un valor de dt . Además, podría ocurrir que no existiera un múltiplo cómodo para sincronizar el motor de simulación física y el motor de despliegue gráfico.

La forma de resolver el problema pasa por cambiar el modo de pensar en él. Podemos pensar que el motor de render produce tiempo, y el motor de simulación física tiene que consumirlo en bloques discretos de un tamaño determinado.



Puede ayudar pensar que el motor de render *produce* chunks de tiempo discreto, mientras que el motor de simulación física los consume.

A continuación se muestra un sencillo *game loop* que puede emplearse para conseguir determinismo de una forma sencilla.

Los tiempos mostrados en este pseudocódigo se especifican en milisegundos, y se obtienen a partir de una hipotética función *getMilliseconds()*.

La línea ① define *TickMs*, una variable que nos define la velocidad del reloj interno de nuestro juego (por ejemplo, 32ms). Esta variable no tiene que ver con el reloj de Bullet. Las variables relativas al reloj de

simulación física describen el comportamiento independiente y asíncrono del reloj interno de Bullet (línea [2](#)) y el reloj del motor de juego (línea [3](#)).

Listado 6.22: Pseudocódigo física determinista.

```

1  const unsigned int TickMs 32
2  unsigned long time_physics_prev, time_physics_curr;
3  unsigned long time_gameclock;
4
5  // Inicialmente reseteamos los temporizadores
6  time_physics_prev = time_physics_curr = getMilliseconds();
7  time_gameclock = getMilliseconds();
8
9  while (1) {
10     video->renderOneFrame();
11     time_physics_curr = getMilliseconds();
12     mWorld->stepSimulation(((float) (time_physics_curr -
13                             time_physics_prev))/1000.0, 10);
14     time_physics_prev = time_physics_curr;
15     long long dt = getMilliseconds() - time_gameclock;
16
17     while (dt >= TickMs) {
18         dt -= TickMs;
19         time_gameclock += TickMs;
20         input->do_all_your_input_processing();
21     }
22 }
```

Como se indica en las líneas [6-7](#), inicialmente se resetean los temporizadores. El pseudocódigo del bucle principal del juego se resume en las líneas [9-21](#). Tras representar un frame, se obtiene el tiempo transcurrido desde la última simulación física (línea [11](#)), y se avanza un paso de simulación en segundos (como la llamada al sistema lo obtiene en milisegundos y Bullet lo requiere en segundos, hay que dividir por 1000).

Por último, se actualiza la parte relativa al reloj de juego. Se calcula en *dt* la diferencia entre los milisegundos que pasaron desde la última vez que se actualizó el reloj de juego, y se dejan pasar (en el bucle definido en las líneas [17-21](#)) empleando ticks discretos. En cada tick consumido se procesan los eventos de entrada.

6.13. Escala de los Objetos

Como se ha comentado en secciones anteriores, Bullet asume que las unidades de espacio se definen en metros y el tiempo en segundos. El movimiento de los objetos se define entre 0.05 y 10 unidades. Así, la escala habitual para definir los pasos de simulación suelen ser 1/60 segundos. Si los objetos son muy grandes, y se trabaja con la gravedad por defecto ($9,8m/s^2$), los objetos parecerán que se mueven a cámara lenta. Si esto ocurre, muy probablemente tengamos un problema relativo a la escala de los mismos.

Una posible solución puede pasar por aplicar una escala al mundo de la simulación. Esto equivale a utilizar un conjunto diferente de

unidades, como centímetros en lugar de metros. Si se seleccionan con cuidado, esto puede permitir realizar simulaciones más realistas. Por ejemplo, si queremos diseñar un videojuego de billar, escalamos el mundo en un factor de 100, de modo que 4 unidades equivaldrán a 4cm (diámetro de las bolas de billar).

6.14. Serialización

La serialización de objetos en Bullet es una característica propia de la biblioteca que no requiere de ningún plugin o soporte adicional. La serialización de objetos presenta grandes ventajas relativas al precálculo de formas de colisión complejas. Para guardar un mundo dinámico en un archivo `.bullet`, puede utilizarse el siguiente fragmento de código de ejemplo:

Listado 6.23: Ejemplo de Serialización.

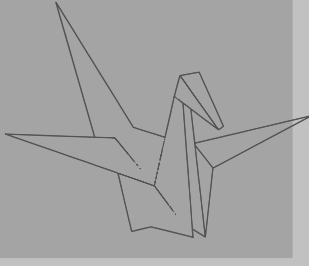
```
1 btDefaultSerializer*    serializer = new btDefaultSerializer();
2 dynamicsWorld->serialize(serializer);
3
4 FILE* file = fopen("testFile.bullet","wb");
5 fwrite(serializer->getBufferPointer(),serializer->
   getCurrentBufferSize(),1, file);
6 fclose(file);
```

Aunque lo más sencillo es serializar un mundo completo, es igualmente posible serializar únicamente algunas partes del mismo. El formato de los archivos `.bullet` soporta la serialización parcial de elementos empleando chunks independientes.

En la posterior carga de los archivos `.bullet`, se debe utilizar la cabecera de `BulletWorldImporter`, creando un objeto de esa clase. El constructor de esa clase requiere que se le especifique el mundo dinámico sobre el que creará los objetos que fueron serializados. El uso del importador puede resumirse en el siguiente fragmento de código:

Listado 6.24: Importación de datos serializados.

```
1 #include "btBulletWorldImporter.h"
2 btBulletWorldImporter* f = new btBulletWorldImporter(_dynWorld);
3 f->loadFile("testFile.bullet");
```



Capítulo 7

Gestión de Widgets

César Mora Castro

Como se ha visto a lo largo del curso, el desarrollo de un videojuego requiere tener en cuenta una gran variedad de disciplinas y aspectos: gráficos 3D o 2D, música, simulación física, efectos de sonido, jugabilidad, eficiencia, etc. Uno de los aspectos a los que se les suele dar menos importancia, pero que juegan un papel fundamental a la hora de que un juego tenga éxito o fracase, es la *interfaz de usuario*. Sin embargo, el mayor inconveniente es que la mayoría de los motores gráficos no dan soporte para la gestión de *Widgets*, y realizarlos desde cero es un trabajo más costoso de lo que pueda parecer.

En este capítulo se describe de forma general la estructura y las guías que hay que tener en cuenta a la hora de diseñar y desarrollar una interfaz de usuario. Además, se explicará el uso de *CEGUI*, una biblioteca de gestión de *Widgets* para integrarlos en motores gráficos.

Eficiencia y diseño

Para desarrollar un videojuego, tan importante es cuidar la eficiencia y optimizarlo, como que tenga un diseño visualmente atractivo.

7.1. Interfaces de usuario en videojuegos

Las interfaces de usuario específicas de los videojuegos deben tener el objetivo de crear una sensación positiva, que consiga la mayor inmersión del usuario posible. Estas interfaces deben tener lo que se denomina *flow*. El *flow*[19] es la capacidad de atraer la atención del usuario, manteniendo su concentración, la inmersión dentro de la trama del videojuego, y que consiga producir una experiencia satisfactoria.

Cada vez se realizan estudios más serios sobre cómo desarrollar interfaces que tengan *flow*, y sean capaces de brindar una mejor experiencia al usuario. La principal diferencia con las interfaces de usuario

de aplicaciones no orientadas al ocio, es que estas centran su diseño en la usabilidad y la eficiencia, no en el impacto *sensorial*. Si un videojuego no consigue atraer y provocar sensaciones positivas al usuario, este posiblemente no triunfará, por muy eficiente que sea, o por muy original o interesante que sea su trama.

A continuación se detalla una lista de aspectos a tener en cuenta que son recomendables para aumentar el *flow* de un videojuego, aunque tradicionalmente se han considerado *perjudiciales* a la hora de diseñar interfaces tradicionales según las reglas de interacción persona-computador:

- *Mostrar la menor cantidad de información posible*: durante el transcurso del juego, es mejor no sobrecargar la interfaz con una gran cantidad de información. En la gran mayoría de videojuegos, toda esta configuración se establece *antes* de comenzar el juego (por ejemplo, en el *Menú*), por lo que la interfaz queda menos sobrecargada y distrae menos al usuario. Incluso el usuario puede tener la opción de mostrar menos información aún si lo desea.
- *Inconsistencia de acciones*: en algunos casos, es posible que se den inconsistencias en las acciones dependiendo del contexto del personaje. Por ejemplo, el botón de saltar cuando el personaje está en tierra puede ser el de nadar si de repente salta al agua.

Es importante mantener un número reducido de teclas (tanto si es por limitaciones de la plataforma, como una videoconsola, como para hacer la usabilidad más sencilla al usuario). Por lo tanto, hay que conseguir agrupar las acciones en los botones según su naturaleza. Por ejemplo, un botón lleva a cabo acciones con objetos y personajes (hablar, abrir una puerta), y otros movimientos de desplazamiento (saltar, escalar). Esto aumentará la intuitividad y la usabilidad del videojuego, y por lo tanto, su *flow*.

- *Dificultar los objetivos al usuario*: una de las reglas de oro de la interacción persona-computador es prevenir al usuario de cometer errores. Sin embargo, en los videojuegos esta regla puede volverse contradictoria, pues en la mayoría de los casos el usuario busca en los videojuegos un sentimiento de satisfacción que se logra por medio de la superación de obstáculos y desafíos.

Por lo tanto, es también de vital importancia conseguir un equilibrio en la dificultad del juego, que no sea tan difícil como para frustrar al usuario, pero no tan fácil como para que resulte aburrido. En este aspecto la interfaz juega un papel de mucho peso.

Se ha visto cómo el caso particular de las interfaces de los videojuegos puede contradecir reglas que se aplican en el diseño de interfaces de usuario clásicas en el campo de la interacción persona-computador. Sin embargo, existen muchas recomendaciones que son aplicables a ambos tipos de interfaces. A continuación se explican algunas:

- *Mantener una organización intuitiva*: es importante que el diseño de los *menús* sean intuitivos. En muchos casos, esta falta de organización crea confusión innecesaria al usuario.

- *Ofrecer una legibilidad adecuada:* en algunos casos, darle demasiada importancia a la estética puede implicar que la legibilidad del texto de las opciones o botones se vea drásticamente reducida. Es importante mantener la funcionalidad básica.
- *Esperas innecesarias:* en multitud de videojuegos, el usuario se ve forzado a esperar a que una determinada película o animación se reproduzca de forma completa, sin poder omitirla. Incluso en el caso en que pueda omitirse, el usuario ha debido esperar previamente a la carga del clip de vídeo para después poder omitirla. Este tipo de inconvenientes reduce notablemente el *flow* de la aplicación.

Existen multitud de formas en las que el usuario realiza esperas innecesarias, y que aumenta su frustración. Por ejemplo, si quiere volver a repetir una acción, tiene que volver a confirmar uno por uno todos los parámetros, aunque estos no cambien. Este es el caso de los juegos de carreras, en el que para volver a repetir una carrera es necesario presionar multitud botones e incluso esperar varios minutos.

- *Ayuda en línea:* muchas veces el usuario necesita consultar el manual durante el juego para entender alguna característica de él. Sin embargo, este manual es documentación extensa que no rompe con la inmersión del videojuego. Es conveniente que se proporcione una versión *suave*, acorde con la estética y que muestre la información estrictamente necesaria.

En general, es importante tener en cuenta cuatro puntos importantes:

1. **Intuitividad:** cuán fácil es aprender a usar una interfaz de un videojuego.
2. **Eficiencia:** cuán rápido se puede realizar una tarea, sobretodo si es muy repetitiva.
3. **Simplicidad:** mantener los controles y la información lo más minimalista posible.
4. **Estética:** cuán sensorialmente atractiva es la interfaz.

Una vez vistas algunas guías para desarrollar interfaces de usuario de videojuegos atractivas y con *flow*, se va a describir la estructura básica que deben tener.

Existe una estructura básica que un videojuego debe seguir. No es buena práctica comenzar el juego directamente en el “terreno de juego” o “campo de batalla”. La estructura típica que debe seguir la interfaz de un videojuego es la siguiente:

En primer lugar, es buena práctica mostrar una *splash screen*. Este tipo de pantallas se muestran al ejecutar el juego, o mientras se carga algún recurso que puede durar un tiempo considerable, y pueden ser

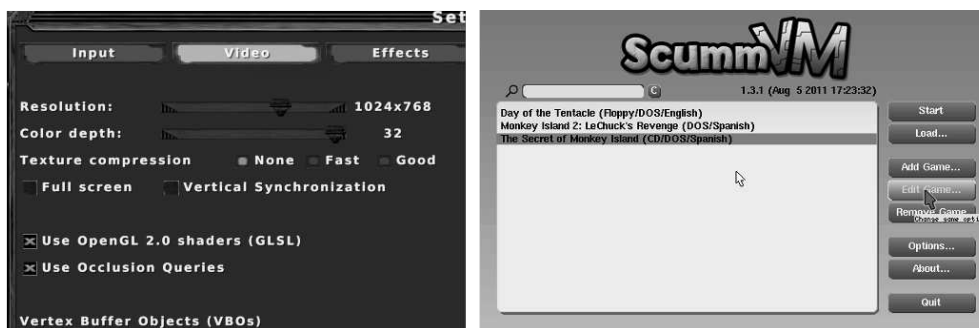


Figura 7.2: Extracto del menú de configuración de Nexuiz (izquierda), e interfaz de ScummVM (derecha).

usadas para mostrar información sobre el juego o sobre sus desarrolladores. Suelen mostrarse a pantalla completa, o de menor tamaño pero centradas. La Figura 7.1 muestra la *splash screen* del juego *free orion*.

Las otros dos elementos de la estructura de un videojuego son el Menú y el HUD. Suponen una parte muy importante de la interfaz de un juego, y es muy común utilizar *Widgets* en ellos. A continuación se analizarán más en detalle y se mostrarán algunos ejemplos.

7.1.1. Menú

Todos los videojuegos deben tener un *Menú* desde el cual poder elegir los modos de juego, configurar opciones, mostrar información adicional y otras características. Dentro de estos menús, es muy frecuente el uso de *Widgets* como botones, barras deslizantes (por ejemplo, para configurar la resolución de pantalla), listas desplegables (para elegir idioma), o *check buttons* (para activar o desactivar opciones). Por eso es importante disponer de un buen repertorio de *Widgets*, y que sea altamente personalizable para poder adaptarlos al estilo visual del videojuego.

En la Figura 7.2 se puede apreciar ejemplos de interfaces de dos conocidos juegos *open-source*. La interfaz de la izquierda, correspondiente al juego *Nexuiz*, muestra un trozo de su diálogo de configuración, mientras que la de la derecha corresponde a la interfaz de *ScummVM*. En estos pequeños ejemplos se muestra un número considerable de *Widgets*, cuyo uso es muy común:

- *Pestañas*: para dividir las opciones por categorías.
- *Barras de desplazamiento*: para configurar opciones que pueden tomar valores muy numerosos.
- *Radio buttons*: parámetros que pueden tomar valores excluyentes entre ellos.
- *Check buttons*: activan o desactivan una opción.



Figura 7.1: Ejemplo de *Splash Screen* durante la carga del juego *FreeOrion*.

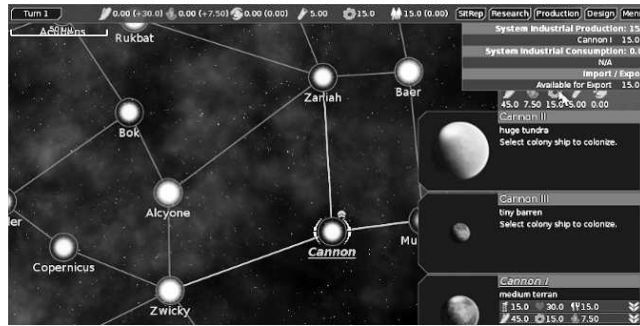


Figura 7.3: Screenshot del HUD del juego *FreeOrion*.

7.1.2. HUD

En relación a las interfaces de los videojuegos, concretamente se denomina *HUD* (del inglés, *Head-Up Display*), a la información y elementos de interacción mostrados durante el propio transcurso de la partida. La información que suelen proporcionar son la vida de los personajes, mapas, velocidad de los vehículos, etc.

En la Figura 7.3 se muestra una parte de la interfaz del juego de estrategia ambientada en el espacio *FreeOrion*. La interfaz utiliza elementos para mostrar los parámetros del juego, y también utiliza *Widgets* para interactuar con él.

Como se puede intuir, el uso de estos *Widgets* es muy común en cualquier videojuego, independientemente de su complejidad. Sin embargo, crear desde cero un conjunto medianamente funcional de estos es una tarea nada trivial, y que roba mucho tiempo de la línea principal de trabajo, que es el desarrollo del propio videojuego.

Una vez que se ha dado unas guías de estilo y la estructura básicas para cualquier videojuego, y se ha mostrado la importancia de los *Widgets* en ejemplos reales, se va a estudiar el uso de una potente biblioteca que proporciona estos mismos elementos para distintos motores gráficos, para que el desarrollo de *Widgets* para videojuegos sea lo menos problemático posible.

CEGUI's mission

Como dice en su página web, "CEGUI está dirigido a desarrolladores de videojuegos que deben invertir su tiempo en desarrollar buenos juegos, no creando subsistemas de interfaces de usuario".

7.2. Introducción CEGUI

CEGUI[2] (*Crazy Eddie's GUI*) es una biblioteca *open source* multiplataforma que proporciona entorno de ventanas y *Widgets* para motores gráficos, en los cuales no se da soporte nativo, o es muy deficiente. Es orientada a objetos y está escrita en C++.



CEGUI es una biblioteca muy potente en pleno desarrollo, por lo que está sujeta a continuos cambios. Todas las características y ejemplos descritos a lo largo de este capítulo se han creado utilizando la versión actualmente estable, la **0.7.x**. No se asegura el correcto funcionamiento en versiones anteriores o posteriores.

CEGUI es muy potente y flexible. Es compatible con los motores gráficos OpenGL, Direct3D, Irrlicht y Ogre3D.

De la misma forma que Ogre3D es únicamente un motor de rendering, CEGUI es sólo un motor de gestión de *Widgets*, por lo que el renderizado y la gestión de eventos de entrada deben ser realizadas por bibliotecas externas.

En sucesivas secciones se explicará cómo integrar CEGUI con las aplicaciones de este curso que hacen uso de Ogre3D y OIS. Además se mostrarán ejemplos prácticos de las características más importantes que ofrece.

7.2.1. Inicialización

La arquitectura de CEGUI es muy parecida a la de Ogre3D, por lo que su uso es similar. Está muy orientado al uso de *scripts*, y hace uso del patrón *Singleton* para implementar los diferentes subsistemas. Los más importantes son:

- `CEGUI::System`: gestiona los parámetros y componentes más importantes de la biblioteca.
- `CEGUI::WindowManager`: se encarga de la creación y gestión de las *windows* de CEGUI.
- `CEGUI::SchemeManager`: gestiona los diferentes esquemas que utilizará la interfaz gráfica.
- `CEGUI::FontManager`: gestiona los distintos tipos de fuentes de la interfaz.

Estos subsistemas ofrecen funciones para poder gestionar los diferentes recursos que utiliza CEGUI. A continuación se listan estos tipos de recursos:

- *Schemes*: tienen la extensión *.scheme*. Definen el repertorio (o esquema) de *Widgets* que se utilizarán. También indica qué scripts utilizará de otros tipos, como por ejemplo el *ImageSet*, las *Fonts* o el *LookNFeel*.

CEGUI y Ogre3D

A lo largo de estos capítulos, se utilizará la fórmula CEGUI/Ogre3D/OIS para proveer interfaz gráfica, motor de rendering y gestión de eventos a los videojuegos, aunque también es posible utilizarlo con otras bibliotecas (ver documentación de CEGUI).

- *Imageset*: tienen la extensión *.imageset*. Define cuáles serán la imágenes de los elementos de la interfaz (punteros, barras, botones, etc).
- *LookNFeel*: tienen la extensión *.looknfeel*. Define el *comportamiento visual* de cada uno de los *Widgets* para distintas acciones, por ejemplo, cómo se muestran cuando se pasa el puntero del ratón o cuando se presionan,
- *Fonts*: tienen la extensión *.font*. Cada uno de los scripts define un tipo de fuente junto con propiedades específicas como su tamaño.
- *Layouts*: tienen la extensión *.layout*. Cada script define clases de *ventanas* concretas, con cada uno de sus elementos. Por ejemplo, una ventana de chat o una consola.

Según se avance en el capítulo se irá estudiando más en profundidad cuál es el funcionamiento y la estructura de estos recursos.

En el siguiente código se muestran los primeros pasos para poder integrar CEGUI con Ogre3D, y de qué forma se inicializa.

Listado 7.1: Inicialización de CEGUI para su uso con Ogre3D.

```
1 #include <CEGUI.h>
2 #include <RendererModules/Ogre/CEGUIOgreRenderer.h>
3
4 CEGUI::OgreRenderer* renderer = &CEGUI::OgreRenderer::
    bootstrapSystem();
5
6 CEGUI::Scheme::setDefaultResourceGroup("Schemes");
7 CEGUI::Imageset::setDefaultResourceGroup("Imagesets");
8 CEGUI::Font::setDefaultResourceGroup("Fonts");
9 CEGUI::WindowManager::setDefaultResourceGroup("Layouts");
10 CEGUI::WidgetLookManager::setDefaultResourceGroup("LookNFeel");
```



Este método de inicializar el *renderer* utilizando el *bootstrapSystem* fue introducido a partir de la versión 0.7.1. Para inicializar CEGUI en versiones anteriores, es necesario referirse a su documentación.

En las líneas 1 y 2 se insertan las cabeceras necesarias. La primera incluye la biblioteca general, y en la segunda se indica de forma concreta que se va a utilizar el motor gráfico Ogre3D. En la línea 4 se inicializa CEGUI para ser utilizado con Ogre3D. Además es necesario indicar dónde estarán los recursos que utilizará la interfaz gráfica, tanto los scripts que utiliza, como las fuentes o las imágenes.

Dependiendo de la distribución que se utilice, estos recursos pueden venir con el paquete del repositorio o no. En este ejemplo vamos a considerar que debemos descargarlos aparte. Se pueden conseguir directamente descargando el código fuente de CEGUI desde su

página[2]. Las distribuciones que los proporcionan, suelen situarlos en `/usr/share/CEGUI/` o `/usr/local/share/CEGUI/`

Para que CEGUI pueda encontrar los recursos, es necesario añadir estos grupos al fichero `resources.cfg` de Ogre.

Listado 7.2: Contenido del fichero `resources.cfg`.

```

1 [General]
2 FileSystem=media
3 [Schemes]
4 FileSystem=media/schemes
5 [Imagesets]
6 FileSystem=media/imagesets
7 [Fonts]
8 FileSystem=media/fonts
9 [Layouts]
10 FileSystem=media/layouts
11 [LookNFeel]
12 FileSystem=media/looknfeel

```

Y las opciones que hay que añadir al `Makefile` para compilar son:

Listado 7.3: Flags de compilación y de enlazado de CEGUI.

```

1 #Flags de compilado
2 CXXFLAGS += `pkg-config --cflags CEGUI-OGRE`
3
4 #Flags de enlazado
5 LDFLAGS += `pkg-config --libs-only-L CEGUI-OGRE`
6 LDLIBS += `pkg-config --libs-only-l CEGUI-OGRE`

```

Es importante incluir los *flags* de compilado, en la línea 2, para que encuentre las cabeceras según se han indicado en el ejemplo de inicialización.

Esto es sólo el código que inicializa la biblioteca en la aplicación para que pueda comenzar a utilizar CEGUI como interfaz gráfica, todavía no tiene ninguna funcionalidad. Pero antes de empezar a añadir *Widgets*, es necesario conocer otros conceptos primordiales.

7.2.2. El Sistema de Dimensión Unificado

El posicionamiento y tamaño de los distintos *Widgets* no es tan trivial como indicar los valores absolutos. Puede ser deseable que un *Widget* se repositone y redimensione si el *Widget* al que pertenece se redimensiona, por ejemplo.

CEGUI utiliza lo que denomina el Sistema de Dimensión Unificado (*Unified Dimension System*). El elemento principal de este sistema es:

`CEGUI::UDim(scale, offset)`

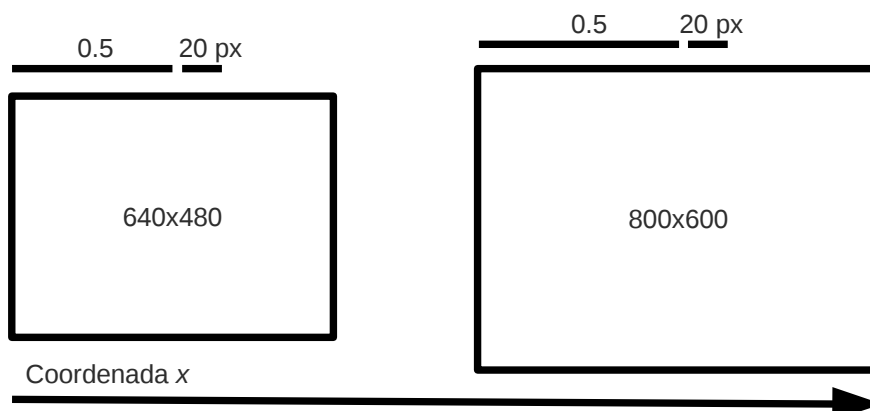


Figura 7.4: Ejemplo del funcionamiento de *UDim*.

(left, top)



(right, bottom)

Figura 7.5: Área rectangular definida por *URect*.

Indica la posición en una dimensión. El primer parámetro indica la posición relativa, que toma un valor entre 0 y 1, mientras que el segundo indica un desplazamiento absoluto en píxeles.

Por ejemplo, supongamos que posicionamos un *Widget* con *UDim* 0.5,20 en la dimensión *x*. Si el ancho de la pantalla fuese 640, la posición sería $0.5 \cdot 640 + 20 = 340$, mientras que si el ancho fuese 800, la posición sería $0.5 \cdot 800 + 20 = 420$. En la Figura 7.4 se aprecian los dos ejemplos de forma gráfica. De esta forma, si la resolución de la pantalla cambia, por ejemplo, el *Widget* se reposicionará y se redimensionará de forma automática.

Teniendo en cuenta cómo expresar el posicionamiento en una única dimensión utilizando *UDim*, se definen dos elementos más.

Para definir un punto o el tamaño de un *Widget* se usa:

```
CEGUI::UVector2(UDim x, UDim y)
```

que está compuesto por dos *UDim*, uno para la coordenada *x*, y otro para la *y*, o para el ancho y el alto, dependiendo de su uso.

El segundo elemento, se utiliza para definir un área rectangular:

```
CEGUI::URect(UDim left, UDim top, UDim right, UDIM bottom)
```

Como muestra la Figura 7.5, los dos primeros definen la esquina superior izquierda, y los dos últimos la esquina inferior derecha.

7.2.3. Detección de eventos de entrada

Puesto que CEGUI es únicamente un motor de gestión de *Widgets*, tampoco incorpora la detección de eventos de entrada, por lo que es necesario *inyectárselos* desde otra biblioteca. En este caso, se aprovechará la que ya se ha estudiado: *OIS*.

Considerando que se utiliza *OIS* mediante *callbacks* (en modo *buffered*), hay que añadir las siguientes líneas para enviar a CEGUI la pulsación y liberación de teclas y de los botones del ratón.

Listado 7.4: Inyección de eventos de pulsación y liberación de teclas a CEGUI.

```

1 bool MyFrameListener::keyPressed(const OIS::KeyEvent& evt)
2 {
3     CEGUI::System::getSingleton().injectKeyDown(evt.key);
4     CEGUI::System::getSingleton().injectChar(evt.text);
5
6     return true;
7 }
8
9 bool MyFrameListener::keyReleased(const OIS::KeyEvent& evt)
10 {
11     CEGUI::System::getSingleton().injectKeyUp(evt.key);
12
13     return true;
14 }
15
16 bool MyFrameListener::mousePressed(const OIS::MouseEvent& evt, OIS
17     ::MouseButtonID id)
18 {
19     CEGUI::System::getSingleton().injectMouseButtonDown(
20         convertMouseButton(id));
21     return true;
22 }
23
24 bool MyFrameListener::mouseReleased(const OIS::MouseEvent& evt, OIS
25     ::MouseButtonID id)
26 {
27     CEGUI::System::getSingleton().injectMouseButtonUp(
28         convertMouseButton(id));
29     return true;
30 }

```

Además, es necesario convertir la forma en que identifica *OIS* los botones del ratón, a la que utiliza CEGUI, puesto que no es la misma, al contrario que sucede con las teclas del teclado. Para ello se ha escrito la función `convertMouseButton()`:

Listado 7.5: Función de conversión entre identificador de botones de ratón de OIS y CEGUI.

```

1 CEGUI::MouseButton MyFrameListener::convertMouseButton(OIS::
2     MouseButtonID id)
3 {
4     CEGUI::MouseButton ceguiId;
5     switch(id)
6     {
7         case OIS::MB_Left:
8             ceguiId = CEGUI::LeftButton;
9             break;
10        case OIS::MB_Right:
11            ceguiId = CEGUI::RightButton;
12            break;
13        case OIS::MB_Middle:
14            ceguiId = CEGUI::MiddleButton;
15            break;

```

```

15     default:
16         ceguiId = CEGUI::LeftButton;
17     }
18     return ceguiId;
19 }

```

Por otro lado, también es necesario decir a CEGUI cuánto tiempo ha pasado desde la detección del último evento, por lo que hay que añadir la siguiente línea a la función `frameStarted()`:

Listado 7.6: Orden que indica a CEGUI el tiempo transcurrido entre eventos.

```

1 CEGUI::System::getSingleton().injectTimePulse(evt.
    timeSinceLastFrame)

```

Hasta ahora se ha visto el funcionamiento básico de *CEGUI*, los tipos básicos de scripts que define, la inicialización, el sistema de posicionamiento y dimensionado que utiliza e incluso como enviarle eventos de entrada. Una vez adquiridos estos conocimientos, es momento de crear la primera aplicación de *Ogre* que muestre un *Widget* con funcionalidad, como se describirá en la siguiente sección.

7.3. Primera aplicación

En esta sección se van a poner en práctica los primeros conceptos descritos para crear una primera aplicación. Esta aplicación tendrá toda la funcionalidad de *Ogre* (mostrando a *Sinbad*), y sobre él un botón para salir de la aplicación.



Es importante tener en cuenta que en CEGUI, **todos los elementos son *Windows***. Cada uno de los *Windows* puede contener a su vez otros *Windows*. De este modo, pueden darse situaciones raras como que un botón contenga a otro botón, pero que en la práctica no suceden.

Listado 7.7: Código de la función `createGUI()`

```

1 void MyApp::createGUI()
2 {
3     renderer = &CEGUI::OgreRenderer::bootstrapSystem();
4     CEGUI::Scheme::setDefaultResourceGroup("Schemes");
5     CEGUI::Imageset::setDefaultResourceGroup("Imagesets");
6     CEGUI::Font::setDefaultResourceGroup("Fonts");
7     CEGUI::WindowManager::setDefaultResourceGroup("Layouts");
8     CEGUI::WidgetLookManager::setDefaultResourceGroup("LookNFeel");
9
10    CEGUI::SchemeManager::getSingleton().create("TaharezLook.scheme")
11    ;
12    CEGUI::System::getSingleton().setDefaultFont("DejaVuSans-10");

```

```

12  CEGUI::System::getSingleton().setDefaultMouseCursor("TaharezLook"
13      , "MouseArrow");
14  //Creating GUI Sheet
15  CEGUI::Window* sheet = CEGUI::WindowManager::getSingleton().
16      createWindow("DefaultWindow", "Ex1/Sheet");
17  //Creating quit button
18  CEGUI::Window* quitButton = CEGUI::WindowManager::getSingleton().
19      createWindow("TaharezLook/Button", "Ex1/QuitButton");
20  quitButton->setText("Quit");
21  quitButton->setSize(CEGUI::UVector2(CEGUI::UDim(0.15,0),CEGUI::
22      UDim(0.05,0)));
23  quitButton->setPosition(CEGUI::UVector2(CEGUI::UDim(0.5-0.15/2,0)
24      ,CEGUI::UDim(0.2,0)));
25  quitButton->subscribeEvent(CEGUI::PushButton::EventClicked,
26      CEGUI::Event::Subscriber(&MyFrameListener::quit,
27      _frameListener));
28  sheet->addChildWindow(quitButton);
29  CEGUI::System::getSingleton().setGUISheet(sheet);
30  }

```

La función `createGUI()` se encarga de la inicialización de la interfaz gráfica y de la creación de los elementos que contendrá. Como se explicó en la Sección 7.2.1, de las líneas 3-8 se indica que se quiere utilizar Ogre3D como motor de rendering, y se indica a CEGUI dónde están los distintos recursos.

En la línea 10 se crea el esquema que se va a utilizar. Se recuerda que un esquema definía el conjunto de *Widgets* que se podrán utilizar en la aplicación, junto a los tipos de letras, apariencia o comportamiento visual. Es como la elección del *tema* de la interfaz. El fichero de script *TaharezLook.scheme* debe de encontrarse en algún lugar del que CEGUI tenga constancia. Como se ha definido en el fichero *resources.cfg*, los esquemas (*Schemes*) deben estar en *media/schemes*. Desde su página web se pueden descargar otros ejemplos, como el esquema *Vanilla*. Más adelante se analizará brevemente el contenido de estos scripts, para poder ajustar la interfaz gráfica a la estética del videojuego.

En las líneas 11 y 12 se definen algunos parámetros por defecto. En el primer caso, el tipo de letra predeterminada, y en el segundo el cursor, ambos elementos definidos en *TaharezLook.scheme*.

Los *Widgets* de la interfaz gráfica se organizan de forma jerárquica, de forma análoga al grafo de escena Ogre. Cada *Widget* (o *Window*) debe pertenecer a otro que lo contenga. De este modo, debe de haber un *Widget* “padre” o “raíz”. Este *Widget* se conoce en CEGUI como *Sheet* (del inglés, *hoja*, refiriéndose a la hoja en blanco que contiene todos los elementos). Esta se crea en la línea 15. El primer parámetro indica el tipo del *Window*, que será un tipo genérico, *DefaultWindow*. El segundo es el nombre que se le da a ese elemento. Con *Ex1* se hace referencia a que es el primer ejemplo (*Example1*), y el segundo es el nombre del elemento.

El siguiente paso es crear el botón. En la línea 18 se llama al *WindowManager* para crear un *Window* (hay que recordar que en CEGUI todo es un *Window*). El primer parámetro indica el tipo, definido en el esquema escogido, en este caso un botón. El segundo es el nombre del elemento, para el cual se sigue el mismo convenio de nombrado que para el *Sheet*.

Convenio de nombrado

CEGUI no exige que se siga ningún convenio de nombrado para sus elementos. Sin embargo, es altamente recomendable utilizar un convenio jerárquico, utilizando la barra “/” como separador.

Después se indican algunos parámetros del botón. En la línea 19 se indica el texto del botón, utilizando la fuente predeterminada del sistema que se indicó en la inicialización.

En la línea 20 se indica el tamaño. Como se explicó en la Sección 7.2.2, para el tamaño se utiliza un *UVector2*, que contiene dos valores, ancho y alto. Por lo tanto, el ancho de este botón siempre será 0.15 del ancho del *Sheet*, y el alto será 0.05.

Para indicar la posición del *Widget*, se opta por centrarlo horizontalmente. Como se puede ver en la línea 21, para la posición en la dimensión *x* se indica la mitad del ancho del *Sheet*, 0.5, menos la mitad del ancho del *Widget*, es decir, $0.15/2$ (el tamaño se acaba de indicar en la línea 20). Esto se debe a que el posicionamiento toma como punto de referencia la esquina superior izquierda del *Widget*. Para la posición en el eje *y* se ha optado por un 0.2 del alto del *Sheet*.

Hasta el momento se ha creado el *Sheet* que contendrá todo el conjunto de *Widgets*, un botón con el texto “Quit”, y con un tamaño y apariencia determinado. Ahora se le va a asociar un comportamiento para cuando se pulse. Para asociar comportamientos, es necesario suscribir las funciones que implementan el comportamiento a los elementos. En la línea 22 se asocia ese comportamiento. El primer parámetro indica a qué tipo de acción se asocia el comportamiento, y en el segundo qué función se ejecuta. Existen una extensa lista de acciones a las que se pueden asociar comportamientos, como por ejemplo:

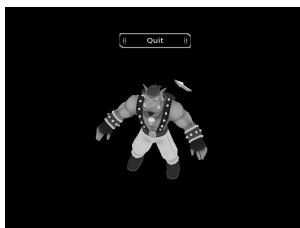


Figura 7.6: Screenshot de la primera aplicación de ejemplo.

- MouseClicked
- MouseEnters
- MouseLeaves
- EventActivated
- EventTextChanged
- EventAlphaChanged
- EventSized

Como se ha estudiado anteriormente, al igual que pasa con el grafo de escena de Ogre3D, cada *Window* de CEGUI debe tener un padre. En la línea 25 se asocia el botón al *Sheet*, y por último, en la línea 26 se indica a CEGUI cuál es el *Sheet* que debe mostrar.

A continuación se muestra la definición de la función que implementa el comportamiento. Como se puede apreciar, simplemente cambia el valor de una variable booleana que controla la salida de la aplicación. Es importante tener en cuenta que no todas las funciones pueden ser utilizadas para implementar el comportamiento de los elementos. En su *signatura*, deben de tener como valor de retorno *bool*, y aceptar un único parámetro del tipo *const CEGUI::EventArgs& e*

Listado 7.8: Función que implementa el comportamiento del botón al ser pulsado.

```
1 bool MyFrameListener::quit(const CEGUI::EventArgs &e)
2 {
3     _quit = true;
4     return true;
5 }
```

En la Figura 7.6 se puede ver una captura de esta primera aplicación.

Ya que se ha visto cómo inicializar *CEGUI* y cuál es su funcionamiento básico, es momento de comenzar a crear interfaces más complejas, útiles, y atractivas.

7.4. Tipos de Widgets

Para comenzar a desarrollar una interfaz gráfica con *CEGUI* para un videojuego, primero es necesario saber cuál es exactamente el repertorio de *Widgets* disponible. Como se ha estudiado en secciones anteriores, el repertorio como tal está definido en el esquema escogido. Para los sucesivos ejemplos, vamos a utilizar el esquema *TaharezLook*, utilizado también en la primera aplicación de inicialización. *CEGUI* proporciona otros esquemas que ofrecen otros repertorios de *Widgets*, aunque los más comunes suelen estar implementados en todos ellos. Otros esquemas que proporciona *CEGUI* son *OgreTray*, *VanillaSkin* y *WindowsLook*.

El *script* del esquema define además la apariencia y el comportamiento visual de los *Widgets*. Para cambiar la apariencia visual, no es necesario crear un fichero esquema desde cero (lo que sería una ardua tarea). Basta con cambiar el *script* de los *ImageSet* y de los *Fonts*. Esto se estudiará con más profundidad en la Sección 7.8.

A continuación se muestra una pequeña lista de los *Widgets* más importantes definidos en el esquema *TaharezLook*:

- Button
- Check Box
- Combo Box
- Frame Window

- List Box
- Progress Bar
- Slider
- Static Text
- etc

El siguiente paso en el aprendizaje de CEGUI es crear una interfaz que bien podría servir para un juego, aprovechando las ventajas que proporcionan los *scripts*.

7.5. Layouts

Los *scripts* de layouts especifican qué *Widgets* habrá y su organización para una ventana específica. Por ejemplo, un *layout* llamado *chatBox.layout* puede contener la estructura de una ventana con un *editBox* para insertar texto, un *textBox* para mostrar la conversación, y un *button* para enviar el mensaje. De cada *layout* se pueden crear tantas instancias como se desee.

No hay que olvidar que estos ficheros son *xml*, por lo que deben seguir su estructura. La siguiente es la organización genérica de un recurso *layout*:

Listado 7.9: Estructura de un *script layout*.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <GUILayout>
3   <Window Type="WindowType" Name="Window1">
4     <Property Name="Property1" Value="Property1Value"/>
5     <Property Name="Property2" Value="Property2Value"/>
6     <!-- This is a comment -->
7     <Window Type="WindowType" Name="Window1/Window2">
8       <Property Name="Property1" Value="Property1Value"/>
9       <Property Name="Property2" Value="Property2Value"/>
10    </Window>
11    <!-- ... --!>
12  </Window>
13 </GUILayout>

```

En la línea 1 se escribe la cabecera del archivo *xml*, lo cual no tiene nada que ver con CEGUI. En la línea 2 se abre la etiqueta *GUILayout*, para indicar el tipo de *script* y se cierra en la línea 13.

A partir de aquí, se definen los *Windows* que contendrá la interfaz (¡en CEGUI todo es un *Window*!). Para declarar uno, se indica el tipo de *Window* y el nombre, como se puede ver en la línea 3. Dentro de él, se especifican sus propiedades (líneas 4 y 5). Estas propiedades pueden indicar el tamaño, la posición, el texto, la transparencia, etc. Los

tipos de *Widgets* y sus propiedades se definían en el esquema escogido, por lo que es necesario consultar su documentación específica. En la Sección 7.6 se verá un ejemplo concreto.

En la línea 6 se muestra un comentario en *xml*.

Después de la definición de las propiedades de un *Window*, se pueden definir más *Windows* que pertenecerán al primero, ya que los *Widgets* siguen una estructura jerárquica.

7.6. Ejemplo de interfaz

El siguiente es el código de un *script layout* que define una ventana de configuración, con distintos *Widgets* para personalizar el volumen, la resolución, o el puerto para utilizar en modo *multiplayer*. Además incorpora un botón para aplicar los cambios y otro para salir de la aplicación. En la Figura 7.7 se muestra el resultado final.

Listado 7.10: Ejemplo de *layout* para crear una ventana de configuración.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <GUILayout >
3   <Window Type="TaharezLook/FrameWindow" Name="Cfg" >
4     <Property Name="Text" Value="Cfguration Window" />
5     <Property Name="TitlebarFont" Value="DejaVuSans-10" />
6     <Property Name="TitlebarEnabled" Value="True" />
7     <Property Name="UnifiedAreaRect" Value="
8       {{0.133,0},{0.027,0},{0.320,300},{0.127,300}}" />
9     <!-- Sonud parameter -->
10    <Window Type="TaharezLook/StaticText" Name="Cfg/SndText" >
11      <Property Name="Text" Value="Sonud Volume" />
12      <Property Name="UnifiedAreaRect" Value="
13        {{0.385,0},{0.0316,0},{0.965,0},{0.174,0}}" />
14    </Window>
15    <Window Type="TaharezLook/Spinner" Name="Cfg/SndVolume" >
16      <Property Name="Text" Value="Sonud Volume" />
17      <Property Name="StepSize" Value="1" />
18      <Property Name="CurrentValue" Value="75" />
19      <Property Name="MaximumValue" Value="100" />
20      <Property Name="MinimumValue" Value="0" />
21      <Property Name="UnifiedAreaRect" Value="
22        {{0.0598,0},{0.046,0},{0.355,0},{0.166,0}}" />
23    </Window>
24    <!-- Fullscreen parameter -->
25    <Window Type="TaharezLook/StaticText" Name="Cfg/FullScrText" >
26      <Property Name="Text" Value="Fullscreen" />
27      <Property Name="UnifiedAreaRect" Value="
28        {{0.385,0},{0.226,0},{0.965,0},{0.367,0}}" />
29    </Window>
30    <Window Type="TaharezLook/Checkbox" Name="Cfg/FullscrCheckbox"
31      >
32      <Property Name="UnifiedAreaRect" Value="
33        {{0.179,0},{0.244,0},{0.231,0},{0.370,0}}" />
34    </Window>
35    <!-- Port parameter -->
36    <Window Type="TaharezLook/StaticText" Name="Cfg/PortText" >
37      <Property Name="Text" Value="Port" />

```

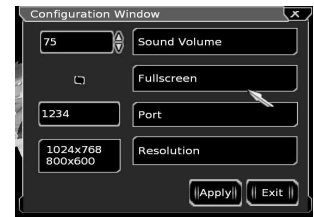


Figura 7.7: Resultado de la ventana de configuración del ejemplo.

```

32     <Property Name="UnifiedAreaRect" Value="
33         {{0.385,0},{0.420,0},{0.9656,0},{0.551,0}}" />
34 </Window>
35 <Window Type="TaharezLook/Editbox" Name="Cfg/PortEditbox" >
36     <Property Name="Text" Value="1234" />
37     <Property Name="MaxTextLength" Value="1073741823" />
38     <Property Name="UnifiedAreaRect" Value="
39         {{0.0541,0},{0.417,0},{0.341,0},{0.548,0}}" />
40     <Property Name="TextParsingEnabled" Value="False" />
41 </Window>
42 <!-- Resolution parameter -->
43 <Window Type="TaharezLook/StaticText" Name="Cfg/ResText" >
44     <Property Name="Text" Value="Resolution" />
45     <Property Name="UnifiedAreaRect" Value="
46         {{0.385,0},{0.60,0},{0.965,0},{0.750,0}}" />
47 </Window>
48 <Window Type="TaharezLook/ItemListBox" Name="Cfg/ResListBox" >
49     <Property Name="UnifiedAreaRect" Value="
50         {{0.0530,0},{0.613,0},{0.341,0},{0.7904,0}}" />
51     <Window Type="TaharezLook/ListBoxItem" Name="Cfg/Res/Item1">
52     <Property Name="Text" Value="1024x768"/>
53 </Window>
54     <Window Type="TaharezLook/ListBoxItem" Name="Cfg/Res/Item2">
55     <Property Name="Text" Value="800x600"/>
56 </Window>
57 <!-- Exit button -->
58 <Window Type="TaharezLook/Button" Name="Cfg/ExitButton" >
59     <Property Name="Text" Value="Exit" />
60     <Property Name="UnifiedAreaRect" Value="
61         {{0.784,0},{0.825,0},{0.968,0},{0.966,0}}" />
62 </Window>
63 <!-- Apply button -->
64 <Window Type="TaharezLook/Button" Name="Cfg/ApplyButton" >
65     <Property Name="Text" Value="Apply" />
66     <Property Name="UnifiedAreaRect" Value="
67         {{0.583,0},{0.825,0},{0.768,0},{0.969,0}}" />
68 </Window>
69 </Window>
70 </GUILayout>

```

Para comenzar, la línea 2 define el tipo de *script*, como se ha explicado en la Sección anterior. El tipo de *Window* que contendrá al resto es un *TaharezLook/FrameWindow*, y se le ha puesto el nombre *Cfg*. Es importante que el tipo del *Window* indique el esquema al que pertenece, por eso se antepone *TaharezLook/*. A partir de aquí se añaden el resto de *Widgets*.

En total se han añadido 10 *Widgets* más a la ventana *Cfg*:

- Una etiqueta (*StaticText*) con el texto “Sound Volume” (llamada “*Cfg/SndText*” - línea 9) y un *Spinner* para indicar el valor (llamado “*Cfg/SndVolume*” - línea 13).
- Una etiqueta con el texto “Fullscreen” (línea 22) y un *Checkbox* para activarlo y desactivarlo (línea 26).
- Una etiqueta con el texto “Port”(línea 30) y un *EditBox* para indicar el número de puerto (línea 34).
- Una etiqueta con el texto “Resolution” (línea 41) y un *ItemListBox* para elegir entre varias opciones (45).

- Un botón (*Button*) con el texto “Exit” para terminar la aplicación (línea 55).
- Un botón con el texto “Apply” para aplicar los cambios (línea 60).

Cada uno de los *Windows* tiene unas propiedades para personalizarlos. En la documentación se explican todas y cada una de las opciones de los *Window* en función del esquema que se utilice.



Para indicar el valor de una propiedad en un fichero de *script* cualquiera de CEGUI, se indica en el campo *Value*, y siempre entre comillas, ya sea un número, una cadena o una palabra reservada.

Estas son algunas de las propiedades utilizadas:

- *Text*: indica el texto del *Widget*. Por ejemplo, la etiqueta de un botón o el título de una ventana.
- *UnifiedAreaRect*: es uno de los más importantes. Indica la posición y el tamaño, mediante un objeto *URect*, de un *Window* **relativo a su padre**. Como se indicó en secciones anteriores, se trata de cuatro *UDims* (cada uno de ellos con un factor de escala relativo y un offset), para indicar la esquina superior izquierda del rectángulo (los dos primeros *UDims*), y la inferior derecha.

Es importante tener en cuenta que si al *Widget* hijo se le indica que ocupe todo el espacio (con el valor para la propiedad de $\{\{0,0\},\{0,0\},\{1,0\},\{1,0\}\}$), ocupará todo el espacio del *Window* al que pertenezca, no necesariamente toda la pantalla.

- *TitlebarFont*: indica el tipo de fuente utilizado en el título de la barra de una *FrameWindow*.
- *TitlebarEnabled*: activa o desactiva la barra de título de una *FrameWindow*.
- *CurrentValue*: valor actual de un *Widget* al que haya que indicárselo, como el *Spinner*.
- *MaximumValue* y *MinimumValue*: acota el rango de valores que puede tomar un *Widget*.

Cada *Widget* tiene sus propiedades y uso especial. Por ejemplo, el *Widget* utilizado para escoger la resolución (un *ItemListBox*), contiene a su vez otros dos *Window* del tipo *ListBoxItem*, que representan cada una de las opciones (líneas 47 y 50).

Para poder añadir funcionalidad a estos *Widgets* (ya sea asociar una acción o utilizar valores, por ejemplo), se deben recuperar desde código mediante el *WindowManager*, con el mismo nombre que se ha especificado en el *layout*. Este *script* por tanto se utiliza únicamente

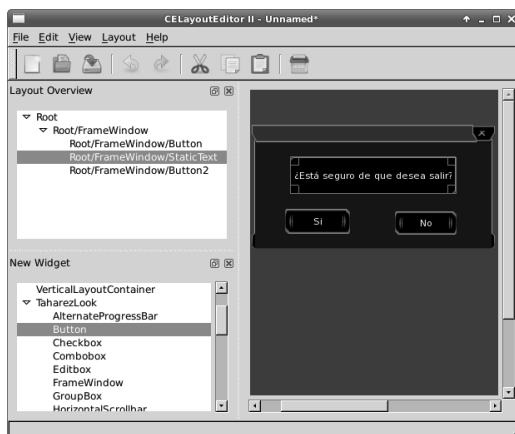


Figura 7.8: Interfaz de CEGUI Layout Editor II.

para cambiar la organización y apariencia de la interfaz, pero no su funcionalidad.

En este ejemplo concreto, sólo se ha añadido una acción al botón con la etiqueta “Exit” para cerrar la aplicación.

7.7. Editores de *layouts* gráficos

Este método para diseñar interfaces de usuario directamente modificando los ficheros *.layout* puede ser muy costoso. Existen un par de aplicaciones gráficas para el diseño de las interfaces. La primera *CEGUI Layout Editor* es muy avanzada, pero se ha abandonado. Actualmente se está desarrollando un editor nuevo y actualizado, *CEGUI Layout Editor 2*, disponible desde el repositorio mercurial de CEGUI.

Para compilar el editor, es necesario recompilar CEGUI con soporte para Python, pues es necesaria la biblioteca *PyCEGUI*. Para compilarla en Ubuntu 11.04 y 11.10 se puede seguir un tutorial disponible en la web de CEGUI[2], en el apartado *HowTo*, el tutorial “Build PyCEGUI from source for Linux”.

En una Debian inestable, bastó con descargar el código fuente de CEGUI, versión 0.7.6, y compilar el proyecto:

```
./configure
make && sudo make install && sudo ldconfig
```



Es importante estar atento a que en la salida del *./configure*, al mostrar el resumen, en la sección de “Scripting” ponga “Building Python extension module(s): yes”

También es necesario instalar las dependencias:

```
aptitude install python-qt4 python-qt4-gl
```

Para descargar la última versión del editor, instalar *mercurial*, y clonar el repositorio con:

```
hg clone http://
  crayzedsgui.hg.sourceforge.net/hgroot/crayzedsgui/CELayoutEditorII
  CELayourEditorII
```

Por último, para ejecutarlo:

```
cd layouteditor
python CELayoutEditorII.py
```

En la Figura 7.8 se muestra la interfaz de *CEGUI Layout Editor II*. No hay que olvidar que esta herramienta está todavía en desarrollo, por lo que es posible que no implemente correctamente toda la funcionalidad, o no se instale de la misma forma en todas las distribuciones.

7.8. Scripts en detalle

En esta Sección se va a ver la estructura del resto de *scripts* que se utilizan para construir la interfaz. Es importante conocerlos para poder cambiar la apariencia y poder adaptarlas a las necesidades artísticas del proyecto.

7.8.1. Scheme

Los esquemas contienen toda la información para ofrecer *Widgets* a una interfaz, su apariencia, su comportamiento visual o su tipo de letra.

Listado 7.11: Estructura de un *script scheme*.

```
1 <?xml version="1.0" ?>
2 <GUIScheme Name="TaharezLook">
3   <ImageSet Filename="TaharezLook.imageset" />
4   <Font Filename="DejaVuSans-10.font" />
5   <LookNFeel Filename="TaharezLook.looknfeel" />
6   <WindowRendererSet Filename="CEGUIFalagardWRBase" />
7   <FalagardMapping WindowType="TaharezLook/Button"
8     TargetType="CEGUI/PushButton" Renderer="Falagard/Button"
9     LookNFeel="TaharezLook/Button" />
10  <FalagardMapping WindowType="TaharezLook/Checkbox"
11    TargetType="CEGUI/Checkbox" Renderer="Falagard/
12    ToggleButton" LookNFeel="TaharezLook/Checkbox" />
13 </GUIScheme>
```

Al igual que sucedía con los *layout*, comienzan con una etiqueta que identifican el tipo de *script*. Esta etiqueta es *GUIScheme*, en la línea 2.

De las líneas 3-6 se indican los *scripts* que utilizará el esquema de los otros tipos. Qué conjunto de imágenes para los botones, cursores y otro tipo de *Widgets* mediante el *Imageset* (línea 3), qué fuentes utilizará mediante un *Font* (línea 4), y el comportamiento visual de los *Widgets* a través del *LookNFeel* (línea 5). Además se indica qué sistema de renderizado de *skin* utilizará (línea 6). CEGUI utiliza *Falagard*.

El resto se dedica a declarar el conjunto de *Widgets*. Para ello realiza un mapeado entre el *Widget* que habrá disponible en el esquema (por ejemplo, "TaharezLook/Button", en la línea 7), y los que ofrece CEGUI. Además, por cada uno se indican varios parámetros.

De este *script* usualmente se suele cambiar los *Imageset* y los *Font* para cambiar la apariencia de la interfaz, y suministrar los desarrollados de forma propia.

7.8.2. Font

Describen los tipos de fuente que se utilizarán en la interfaz.

Listado 7.12: Estructura de un *script font*.

```
1 <?xml version="1.0" ?>
2 <Font Name="DejaVuSans-10" Filename="DejaVuSans.ttf" Type="FreeType
  " Size="10" NativeHorzRes="800" NativeVertRes="600" AutoScaled=
  "true"/>
```

En este ejemplo, sólo se define un tipo de fuente, en la línea 2. Además de asignarle un nombre para poder usarla con CEGUI, se indica cuál es el fichero *tff* que contiene la fuente, y otras propiedades como el tamaño o la resolución. Dentro de este fichero se puede añadir más de una fuente, añadiendo más etiquetas del tipo *Font*.

7.8.3. Imageset

Contiene las verdaderas imágenes que compondrán la interfaz. Este recurso es, junto al de las fuentes, los que más sujetos a cambio están para poder adaptar la apariencia de la interfaz a la del videojuego.

Listado 7.13: Estructura de un *script imageset*.

```
1 <?xml version="1.0" ?>
2 <Imageset Name="TaharezLook" Imagefile="TaharezLook.tga"
  NativeHorzRes="800" NativeVertRes="600" AutoScaled="true">
3   <Image Name="MouseArrow" XPos="138" YPos="127" Width="31"
  Height="25" XOffset="0" YOffset="0" />
4 </Imageset>
```

CEGUI almacena las imágenes que componen la interfaz como un conjunto de imágenes. De este modo, CEGUI sólo trabaja con un archivo de imagen, y dentro de ella debe saber qué porción corresponde

a cada elemento. En la Figura 7.9 podemos ver el archivo de imagen que contiene toda la apariencia del esquema *TaharezLook* y *OgreTray*.

En la línea 2 del *script*, se indica el nombre del conjunto de imágenes y cuál es el archivo de imagen que las contendrá, en este caso, *TaharezLook.tga*. A partir de ese archivo, se define cada uno de los elementos indicando en qué región de *TaharezLook.tga* se encuentra. En la línea 3 se indica que el cursor del ratón (“MouseArrow”) se encuentra en el rectángulo definido por la esquina superior izquierda en la posición (138, 127), y con unas dimensiones de 31x25.

De esta forma, se puede diseñar toda la interfaz con un programa de dibujo, unirlos todos en un archivo, e indicar en el *script image-set* dónde se encuentra cada una. Para ver a qué más elementos se les puede añadir una imagen, consultar la referencia, o estudiar los *imageset* proporcionados por CEGUI.

7.8.4. LookNFeel

Estos tipos de *scripts* son mucho más complejos, y los que CEGUI proporciona suelen ser más que suficiente para cualquier interfaz. Aún así, en la documentación se puede consultar su estructura y contenido.

7.9. Cámara de Ogre en un Window

Una característica muy interesante que se puede realizar con CEGUI es mostrar lo que capta una cámara de Ogre en un *Widget*. Puede ser interesante para mostrar mapas o la vista trasera de un coche de carreras, por ejemplo.

Como se aprecia en la Figura 7.10, la aplicación mostrará a Sinbad de la forma habitual, pero además habrá una ventana que muestra otra vista distinta, y ésta a su vez contendrá un botón para salir de la aplicación.

A continuación se muestra el fichero del *layout*. Después de haber visto en la Sección 7.5 cómo funcionan los *layouts*, este no tiene ninguna complicación. Se trata de una única *FrameWindow* (línea 3) llamada “CamWin”, con el título “Back Camera” y una posición y tamaño determinados. Esta a su vez contiene dos *Windows* más: uno del tipo *StaticImage* (línea 8), que servirá para mostrar la imagen de la textura generada a partir de una cámara secundaria de Ogre, para tener ese segundo punto de vista; y uno del tipo *Button* (línea 11), con el texto “Exit”.

El único aspecto resaltable de este código es que el *Widget* que muestra la imagen (“CamWin/RTTWindow”) ocupa todo el area de su padre (su propiedad *UnifiedAreaRect* vale $\{\{0,0\},\{0,0\},\{1,0\},\{1,0\}\}$). Esto quiere decir que ocupará toda la ventana “CamWin”, que no toda la pantalla.

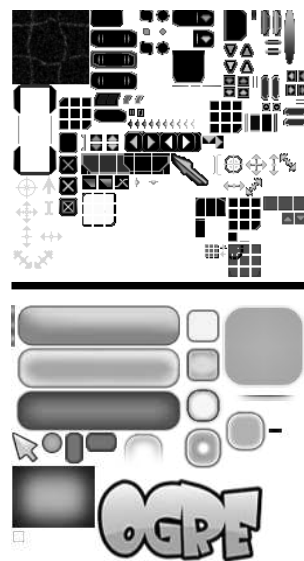


Figura 7.9: Matriz de imágenes que componen la interfaz del esquema *TaharezLook* (arriba), y *OgreTray* (abajo).



Figura 7.10: Screenshot de la aplicación de ejemplo.

Listado 7.14: Layout del ejemplo.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <GUILayout>
3   <Window Type="TaharezLook/FrameWindow" Name="CamWin" >
4     <Property Name="Text" Value="Back Camera" />
5     <Property Name="TitlebarFont" Value="DejaVuSans-10" />
6     <Property Name="TitlebarEnabled" Value="True" />
7     <Property Name="UnifiedAreaRect" Value="
8       {{0.6,0},{0.6,0},{0.99,0},{0.99,0}}" />
9     <Window Type="TaharezLook/StaticImage" Name="CamWin/RTTWindow"
10      >
11       <Property Name="UnifiedAreaRect" Value="
12         {{0,0},{0,0},{1,0},{1,0}}" />
13     </Window>
14     <Window Type="TaharezLook/Button" Name="ExitButton" >
15       <Property Name="Text" Value="Exit" />
16       <Property Name="UnifiedAreaRect" Value="
17         {{0.01,0},{0.01,0},{0.25,0},{0.15,0}}" />
18     </Window>
19 </GUILayout>

```

Una vez tenemos la organización de la interfaz, es necesario dotarla de funcionalidad. A continuación se muestran las modificaciones que hay que añadir para que se pueda renderizar una cámara de Ogre en el *Widget*.

Listado 7.15: Inicialización de la textura y del *Widget* que la mostrará.

```

1 Ogre::Camera* _camBack = _sceneManager->createCamera("BackCamera");
2 _camBack->setPosition(Ogre::Vector3(-5,-20,20));
3 _camBack->lookAt(Ogre::Vector3(0,0,0));
4 _camBack->setNearClipDistance(5);
5 _camBack->setFarClipDistance(10000);
6 _camBack->setAspectRatio(width / height);
7
8 Ogre::TexturePtr tex = _root->getTextureManager()->createManual(
9   "RTT",
10  Ogre::ResourceGroupManager::
11    DEFAULT_RESOURCE_GROUP_NAME,
12  Ogre::TEX_TYPE_2D,
13  512,
14  512,
15  0,
16  Ogre::PF_R8G8B8,
17  Ogre::TU_RENDERTARGET);
18 Ogre::RenderTexture* rtex = tex->getBuffer()->getRenderTarget();
19
20 Ogre::Viewport* v = rtex->addViewport(_camBack);
21 v->setOverlaysEnabled(false);
22 v->setClearEveryFrame(true);
23 v->setBackgroundColour(Ogre::ColourValue::Black);
24
25 CEGUI::Texture& guiTex = renderer->createTexture(tex);
26
27 CEGUI::ImageSet& imageSet = CEGUI::ImageSetManager::getSingleton().
28   create("RTTImageSet", guiTex);
29 imageSet.defineImage("RTTImage",
30   CEGUI::Point(0.0f,0.0f),
31   CEGUI::Size(guiTex.getSize().d_width, guiTex.getSize())

```

```

        .d_height),
31         CEGUI::Point(0.0f,0.0f));
32
33 CEGUI::Window* ex1 = CEGUI::WindowManager::getSingleton().
    loadWindowLayout("render.layout");
34
35 CEGUI::Window* RTTWindow = CEGUI::WindowManager::getSingleton().
    getWindow("CamWin/RTTWindow");
36
37 RTTWindow->setProperty("Image",CEGUI::PropertyHelper::imageToString
    (&imageSet.getImage("RTTImage")));
38
39 //Exit button
40 CEGUI::Window* exitButton = CEGUI::WindowManager::getSingleton().
    getWindow("ExitButton");
41 exitButton->subscribeEvent(CEGUI::PushButton::EventClicked,
42     CEGUI::Event::Subscriber(&MyFrameListener::quit,
43     _framelistener));
44 //Attaching layout
45 sheet->addChildWindow(ex1);
46 CEGUI::System::getSingleton().setGUISheet(sheet);

```

Esta parte supone más código de Ogre que de CEGUI.

Puesto que el objetivo es mostrar en un *Widget* lo que está capturando una cámara, el primer paso es crearla. De las líneas 1 a las 6 se crea una *Ogre::Camera* de forma convencional. Se indica la posición, hacia dónde mira, los planos de corte *Near* y *Far*, y el *aspect ratio*.

Por otro lado, hay que crear la textura en la que se volcará la imagen de la cámara. Para ello se utiliza la función `createManual()` de *TextureManager*, en las líneas 8 a la 16. La textura se llama “RTT”, tendrá un tamaño de 512x512, y será del tipo *Ogre::TU_RENDERTARGET*, para que pueda albergar la imagen de una cámara.

El siguiente paso es crear el *ViewPort*. En la línea 18 se obtiene el objeto *RenderTexture* a partir de la textura manual creada. En la línea 20 se obtiene el objeto *ViewPort* a partir del *RenderTexture* y utilizando la cámara que se quiere mostrar. En este caso, *_camBack*. A este *ViewPort* se le indica que no dibuje los *Overlays* (línea 21), aunque podría hacerlo sin problemas, y que se actualice en cada frame (línea 22). Además se establece el color de fondo en negro (línea 23).

Hasta ahora se ha creado la textura *tex* de Ogre que es capaz de actualizarse con la imagen capturada por una cámara, también de Ogre, para tal efecto. El siguiente paso es preparar CEGUI para mostrar imagen en uno de sus *Widgets*, y que además esa imagen la obtenga de la textura de Ogre.

En la línea 25 se crea la textura *guiTex* de CEGUI. El objeto *render* específico para Ogre proporciona la función `createTexture()`, que la crea a partir de una de Ogre.

Como se vio en la Sección 7.8.3, CEGUI está diseñado para tratar las distintas imágenes como porciones de un array de imágenes (ver Figura 7.9).

De este modo, primeramente hay que crear un *Imageset* a partir de la textura que devuelve Ogre (ya en formato de CEGUI) en la línea 27. A este conjunto de imágenes se le ha llamado “RTTImageset”. Después,

hay que identificar qué porción corresponde a la textura de la cámara de ese *Imageset*. En este caso, es la textura completa, por lo que en la línea 28 se define la imagen con el nombre “RTTImage”. El primer parámetro es el nombre, el segundo la esquina superior izquierda de la porción que define la imagen (se indica el 0,0), el tercero el tamaño de la porción, que corresponde al tamaño de la textura, y el cuarto un *offset*.

Ya se ha conseguido obtener una imagen de CEGUI que se actualizará cada frame con lo que capturará la cámara. Lo único que falta es recuperar los *Window* definidos en el *layout* e indicar que el *Widget* “CamWin/RTTWindow” muestre dicha textura.

En la línea 33 se carga el *layout* como se hizo en anteriores ejemplos. Es importante hacerlo antes de comenzar a recuperar los *Windows* definidos en el *layout*, porque de lo contrario no los encontrará.

Se recupera el *Window* que mostrará la imagen (del tipo *StaticImage*), llamado “CamWin/RTTWindow”, en la línea 35. En la siguiente línea, la 37, se indica en la propiedad “Image” de dicho *Window* que utilice la imagen “RTTImage” del conjunto de imágenes.

Con esto ya es suficiente para mostrar en el *Widget* la imagen de la cámara. Por último se añade la funcionalidad al botón de salida, en las líneas 40 y 41, como en anteriores ejemplos, y se añade el *layout* al *Sheet* (línea 45) y se establece dicho *Sheet* por defecto (línea 46).

7.10. Formateo de texto

Una característica muy versátil que ofrece CEGUI es la de proporcionar formato a las cadenas de texto mediante el uso de *tags* (etiquetas). Este formato permite cambiar el color, tamaño, alineación o incluso insertar imágenes. A continuación se describe el funcionamiento de estas etiquetas.

7.10.1. Introducción

El formato de las etiquetas utilizadas son el siguiente:

```
[ tag-name='value' ]
```

Estas etiquetas se insertan directamente en las cadenas de texto, y funcionan como estados. Es decir, si se activa una etiqueta con un determinado color, se aplicará a todo el texto que le preceda a no ser que se cambie explícitamente con otra etiqueta.

A continuación se muestran los diferentes aspectos que se pueden personalizar con esta técnica. Al final se muestra una aplicación de ejemplo que implementa su funcionamiento.



Si se quiere mostrar como texto la cadena “[Texto]” sin que lo interprete como una etiqueta, es necesario utilizar un carácter de escape. En el caso concreto de C++ se debe anteponer “\” únicamente a la primera llave, de la forma “\\[Texto]”

7.10.2. Color

Para cambiar el color del texto, se utiliza la etiqueta “colour”, usada de la siguiente forma:

```
[colour='FFFF0000']
```

Esta etiqueta colorea el texto que la siguiese de color rojo. El formato en el que se expresa el color es *ARGB* de 8 bits, es decir '*AARRGGBB*'. El primer parámetro expresa la componente de transparencia alpha, y el resto la componente RGB.

7.10.3. Formato

Para cambiar el formato de la fuente (tipo de letra, negrita o cursiva, por ejemplo) es más complejo ya que se necesitan los propios ficheros que definan ese tipo de fuente, y además deben estar definidos en el *script Font*.

Estando seguro de tener los archivos *.font* y de tenerlos incluidos dentro del fichero del esquema, se puede cambiar el formato utilizando la etiqueta:

```
[font='Arial-Bold-10']
```

Esta en concreto corresponde al formato en negrita del tipo de letra Arial, de tamaño 10. El nombre que se le indica a la etiqueta es el que se especificó en el *.scheme*.

7.10.4. Insertar imágenes

Insertar una imagen dentro de una cadena, al estilo de los emoticonos, es sencillo. La estructura de la etiqueta es la siguiente:

```
[imageset='set:<imageset> image:<image>']
```

Una vez más, CEGUI trata las imágenes individuales como parte de un *Imageset*, por lo que hay que indicarle el conjunto, y la imagen.

Aprovechando el *Imageset* que utiliza la interfaz, “TaharezLook”, se va a mostrar una de ellas, por ejemplo la equis para cerrar la ventana. Echando un vistazo al *xml*, se puede identificar que la imagen que se

corresponde con la equis se llama “CloseButtonNormal”. La etiqueta que habría que utilizar sería la siguiente:

```
[ image='set:TaharezLook image=CloseButtonNormal' ]
```

Además, existe otra etiqueta poder cambiar el tamaño de las imágenes insertadas. El formato de la etiqueta es el siguiente:

```
[ image-size='w:<width_value> h:<height_value>' ]
```

El valor del ancho y del alto se da en píxeles, y debe ponerse antes de la etiqueta que inserta la imagen. Para mostrar las imágenes en su tamaño original, se deben poner los valores de *width* y *height* a cero.

7.10.5. Alineamiento vertical

Cuando un texto contiene distintos tamaños de letra, es posible configurar el alineamiento vertical de cada parte. El alto de una línea concreta vendrá definido por el alto del texto con mayor tamaño. El resto de texto, con menor tamaño, podrá alinearse verticalmente dentro de ese espacio.

Los tipos de alineamiento vertical disponibles son:

- *top*: lo alinea hacia arriba.
- *bottom*: lo alinea hacia abajo.
- *center*: lo centra verticalmente.
- *stretch*: lo *estira* verticalmente para ocupar todo el alto.

El formato de la etiqueta es:

```
[ vert-alignment='<tipo_de_alineamiento>' ]
```

7.10.6. Padding

El *padding* consiste en reservar un espacio alrededor del texto que se desee. Para definirlo, se indican los píxeles para el *padding* izquierdo, derecho, superior e inferior. Así, además de el espacio que ocupe una determinada cadena, se reservará *como un margen* el espacio indicado en el *padding*. Viendo la aplicación de ejemplo se puede apreciar mejor este concepto.

El formato de la etiqueta es:

```
[ padding='l:<left_padding> t:<top_padding> r:<right_padding> b:<bottom_padding>' ]
```

Para eliminar el *padding*, utilizar la etiqueta con los valores a 0.

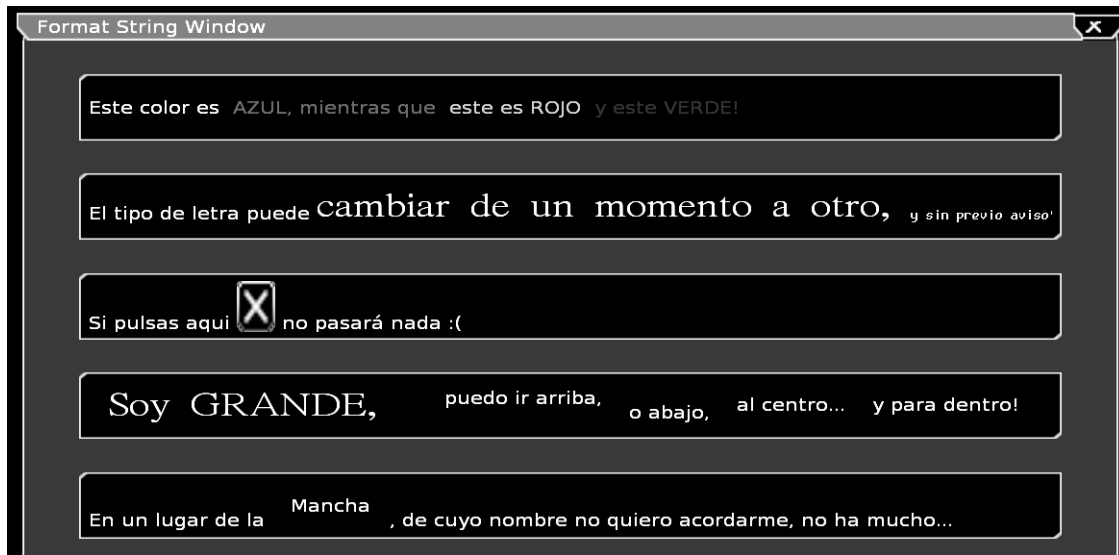


Figura 7.11: Ejemplos de uso del formateo de cadenas.

7.10.7. Ejemplo de texto formateado

El siguiente es un ejemplo que muestra algunas de las características que se han descrito. En la Figura 7.11 se aprecia el acabado final. El siguiente es el *layout* utilizado:

Listado 7.16: *Layout* de la aplicación.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <GUILayout>
3   <Window Type="TaharezLook/FrameWindow" Name="FormatWin">
4     <Property Name="Text" Value="Format String Window"/>
5     <Property Name="TitlebarEnabled" Value="True"/>
6     <Property Name="UnifiedAreaRect" Value="
7       {{0.05,0},{0.05,0},{0.95,0},{0.95,0}}"/>
8     <!-- Static Text -->
9     <Window Type="TaharezLook/StaticText" Name="FormatWin/Text1">
10      <Property Name="UnifiedAreaRect" Value="
11        {{0.05,0},{0.05,0},{0.95,0},{0.15,0}}"/>
12    </Window>
13    <!-- Other Static Text ... -->
14    <!-- Exit Button -->
15    <Window Type="TaharezLook/Button" Name="FormatWin/ExitButton">
16      <Property Name="Text" Value="Exit" />
17      <Property Name="UnifiedAreaRect" Value="
18        {{0,0},{0.95,0},{1,0},{1,0}}"/>
19    </Window>
20  </Window>
21 </GUILayout>

```

Y el siguiente listado muestra cada una de las cadenas que se han utilizado, junto a las etiquetas:

Listado 7.17: Código con los tags de formateo.

```

1 "Este color es [colour='FFFF0000'] AZUL, mientras que [colour='
  FF00FF00'] este es ROJO [colour='FF0000FF'] y este VERDE!"
2
3 "El tipo de letra puede [font='Batang-26'] cambiar de un momento a
  otro, [font='fcp-16'] y sin previo aviso!"
4
5 "Si pulsas aqui [image-size='w:40 h:55'] [image='set:TaharezLook
  image:CloseButtonNormal'] no pasara nada :("
6
7 "[font='Batang-26'] Soy GRANDE, [font='DejaVuSans-10'] [vert-
  alignment='top'] puedo ir arriba, [vert-alignment='bottom'] o
  abajo, [vert-alignment='centre'] al centro..."
8
9 "En un lugar de la [padding='l:20 t:15 r:20 b:15'] Mancha [padding='l
  :0 t:0 r:0 b:0'], de cuyo nombre no quiero acordarme, no ha
  mucho..."

```

La primera cadena (línea 1) utiliza las etiquetas del tipo *colour* para cambiar el color del texto escrito a partir de ella. Se utilizan los colores rojo, verde y azul, en ese orden.

La segunda (línea 3) muestra cómo se pueden utilizar las etiquetas para cambiar totalmente el tipo de fuente, siempre y cuando estén definidos los recursos *.font* y estos estén reflejados dentro el *.scheme*.

La tercera (línea 5) muestra una imagen insertada en medio del texto, y además redimensionada. Para ello se utiliza la etiqueta de redimensionado para cambiar el tamaño a 40x55, y después inserta la imagen "CloseButtonNormal", del conjunto "TaharezLook"

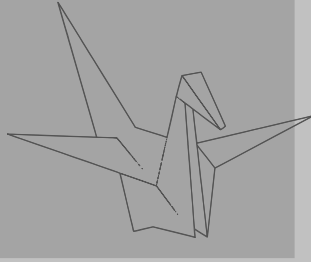
La cuarta (línea 7) muestra una cadena con un texto ("Soy GRANDE") de un tipo de fuente con un tamaño 30, y el resto con un tamaño 10. Para el resto del texto, sobra espacio vertical, por lo que se utiliza la etiqueta *vertical-alignment* para indicar dónde posicionarlo.

Por último, la quinta cadena (línea 9), utiliza *padding* para la palabra "Mancha". A esta palabra se le reserva un *margen* izquierdo y derecho de 20 píxeles, y un superior e inferior de 15.

7.11. Características avanzadas

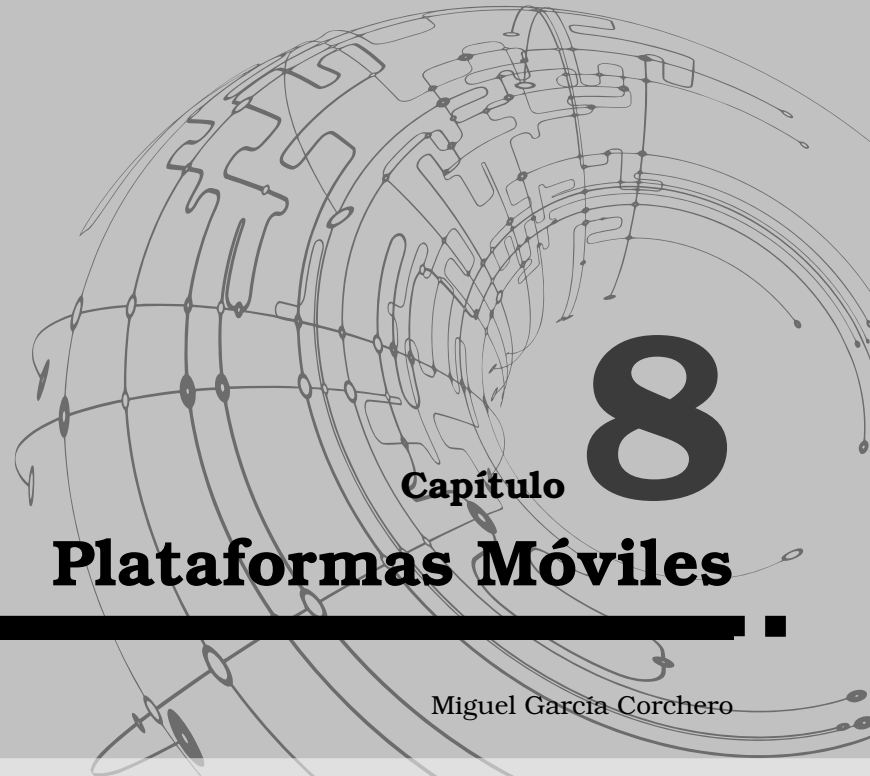
CEGUI es una biblioteca muy potente y flexible que puede ser utilizada junto a muchas otras para crear efectos visualmente muy impactantes. Algunas características avanzadas que se han implementado son efectos de ventanas, como transparencia y aspecto gelatinoso, o incluso incrustar un navegador dentro de una ventana.

Para aprender estas características y más, en su página existen muchos manuales, y dispone de una muy buena documentación [2].



Desarrollo de Videojuegos

3 Técnicas Avanzadas



8

Capítulo

Plataformas Móviles

Miguel García Corchero

Motores de videojuego

Existen otros motores de videojuegos más utilizados en la industria como CryEngine o Unreal Engine y tienen más funcionalidades que Unity3D pero también es más complejo trabajar con ellos.

Un **motor de videojuegos** es un término que hace referencia a una serie de **herramientas** que permiten el diseño, la creación y la representación de un videojuego. La funcionalidad básica de un motor es proveer al videojuego renderización, gestión de físicas, colisiones, scripting, animación, administración de memoria o gestión del sonidos entre otras cosas.

En este capítulo se trata el caso específico del motor de videojuegos **Unity3D** y se realizará un repaso superficial por la forma de trabajar con un motor de videojuegos mientras se realiza un videojuego de ejemplo para dispositivos móviles con el sistema operativo **iOS** o **Android**.

El videojuego será un *shoot'em up* de aviones con vista cenital.



Figura 8.1: Visualización del modelo 3D del jugador.

8.1. Método de trabajo con un motor de videojuegos

8.1.1. Generación de contenido externo al motor

- **Diseño del videojuego:** Esta fase suele hacerse con papel y boli. En ella definiremos las mecánicas necesarias a implementar, haremos bocetos de los personajes o situaciones implicadas y creamos listas de tareas a realizar asignando diferentes prioridades.
- **Generación del material gráfico:** El videojuego necesitará gráficos, texturas, fuentes, animaciones o sonidos. Este material se conoce como assets. Podemos utilizar diferentes programas de

modelado y texturizado 3D o 2D para esta tarea, ya que todos los assets son exportados a un formato de entrada de los que reconoce el motor gráfico.

8.1.2. Generación de contenido interno al motor

- **Escritura de scripts:** La escritura de scripts se realiza con un editor de texto externo a las herramientas del motor gráfico pero se considera contenido íntimamente relacionado del motor de videojuego.
- **Escritura de shaders:** Los shaders también se escriben con un editor externo.
- **Importación de assets:** Uno de los pasos iniciales dentro del entorno integrado de Unity3D es añadir al proyecto todo el material generado anteriormente y ajustar sus atributos; como formatos, tamaños de textura, ajuste de propiedades, calculo de normales, etc.
- **Creación de escenas:** Crearemos una escena por cada nivel o conjunto de menús del videojuego. En la escena estableceremos relaciones entre objetos y crearemos instancias de ellos.
- **Creación de prefabs:** Los *prefabs* son agrupaciones de objetos que se salvan como un objeto con entidad propia.
- **Optimización de la escena:** Uno de los pasos fundamentales se lleva a cabo al final del desarrollo de la escena y es la optimización. Para ello emplearemos técnicas de lightmapping y occlusion culling con las herramientas del entorno.



Con Unity3D no tenemos que preocuparnos de la gestión de las físicas, colisiones, renderización o controles a bajo nivel. Nos dedicamos únicamente a la programación de scripts y shaders.

8.2. Creación de escenas

Una escena está constituida por instancias de objetos de nuestros assets y las relaciones entre ellos. Podemos considerar una escena como la serialización de el objeto escena. Este objeto contiene jerárquicamente otros objetos que son almacenados cuando se produce esa serialización, para posteriormente cargar la escena con todas esas instancias que contendrán los mismos valores que cuando fueron almacenados. Hay **dos enfoques** diferentes a la hora de crear escenas:

Scripts

Los scripts que utilizamos se pueden escribir en los lenguajes C#, Javascript o BOO.

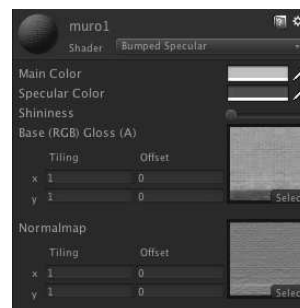


Figura 8.2: Visualización de un shader de normal mapping.

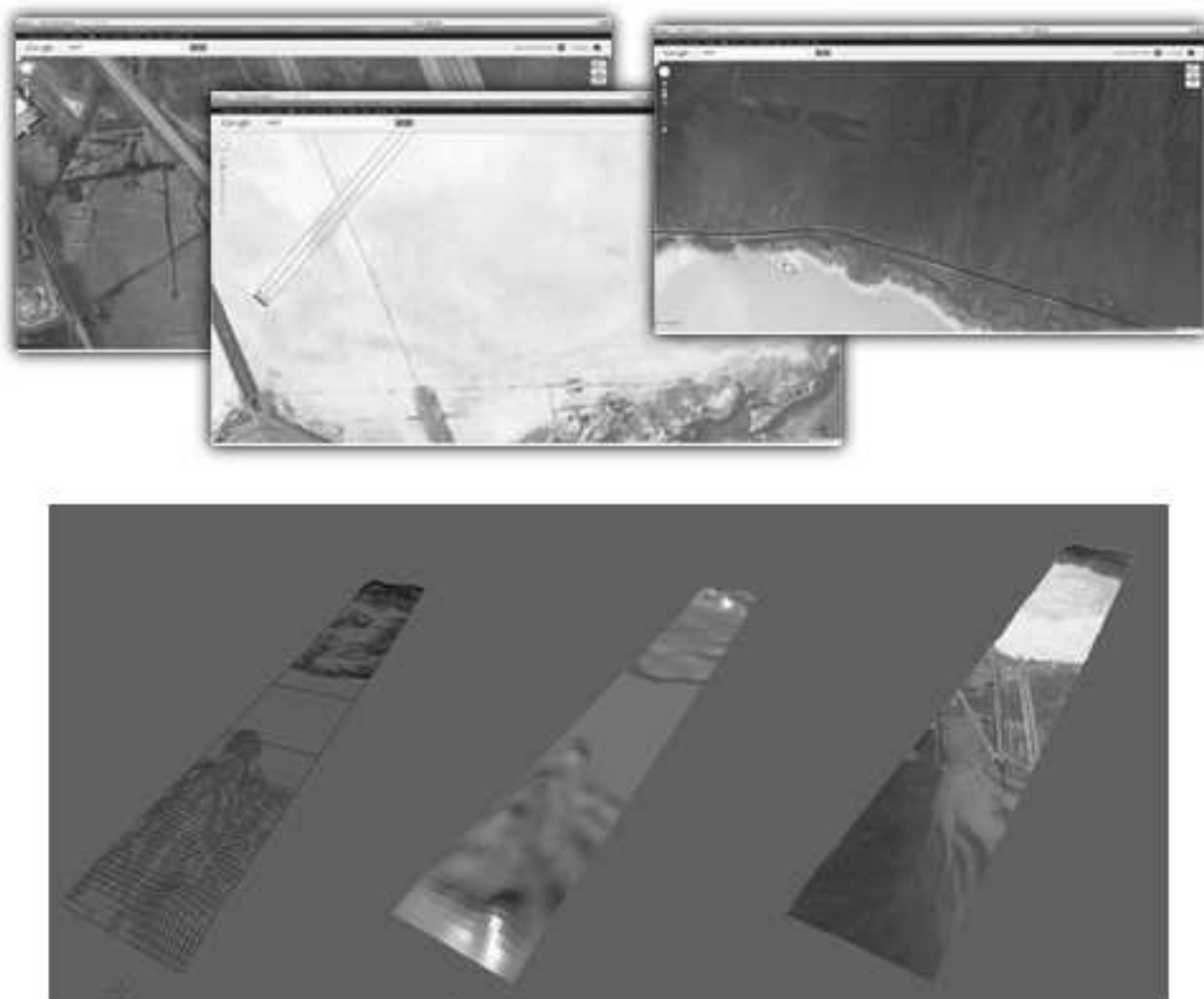


Figura 8.3: En nuestro ejemplo se han utilizado capturas de pantalla de google map para obtener texturas de terreno y se han mapeado sobre un plano en Blender. Posteriormente se ha utilizado el modo scuplt para dar relieve al terreno y generar un escenario tridimensional.

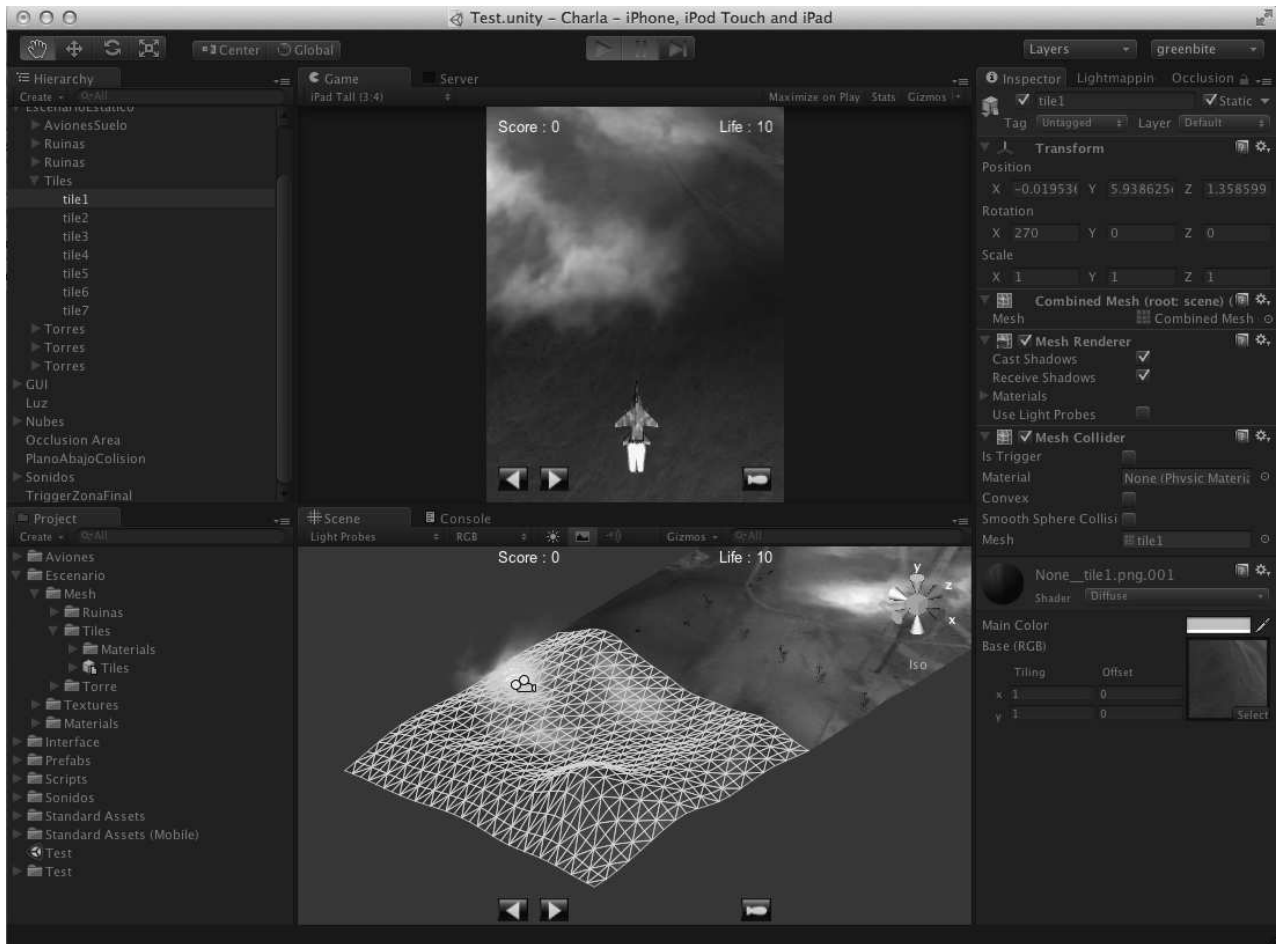


Figura 8.4: Interface de Unity3D. Dividida en las vistas más utilizadas: Jerarquía de escena, Assets del proyecto, Vista del videojuego, Vista de escena y Vista de propiedades del asset seleccionado.

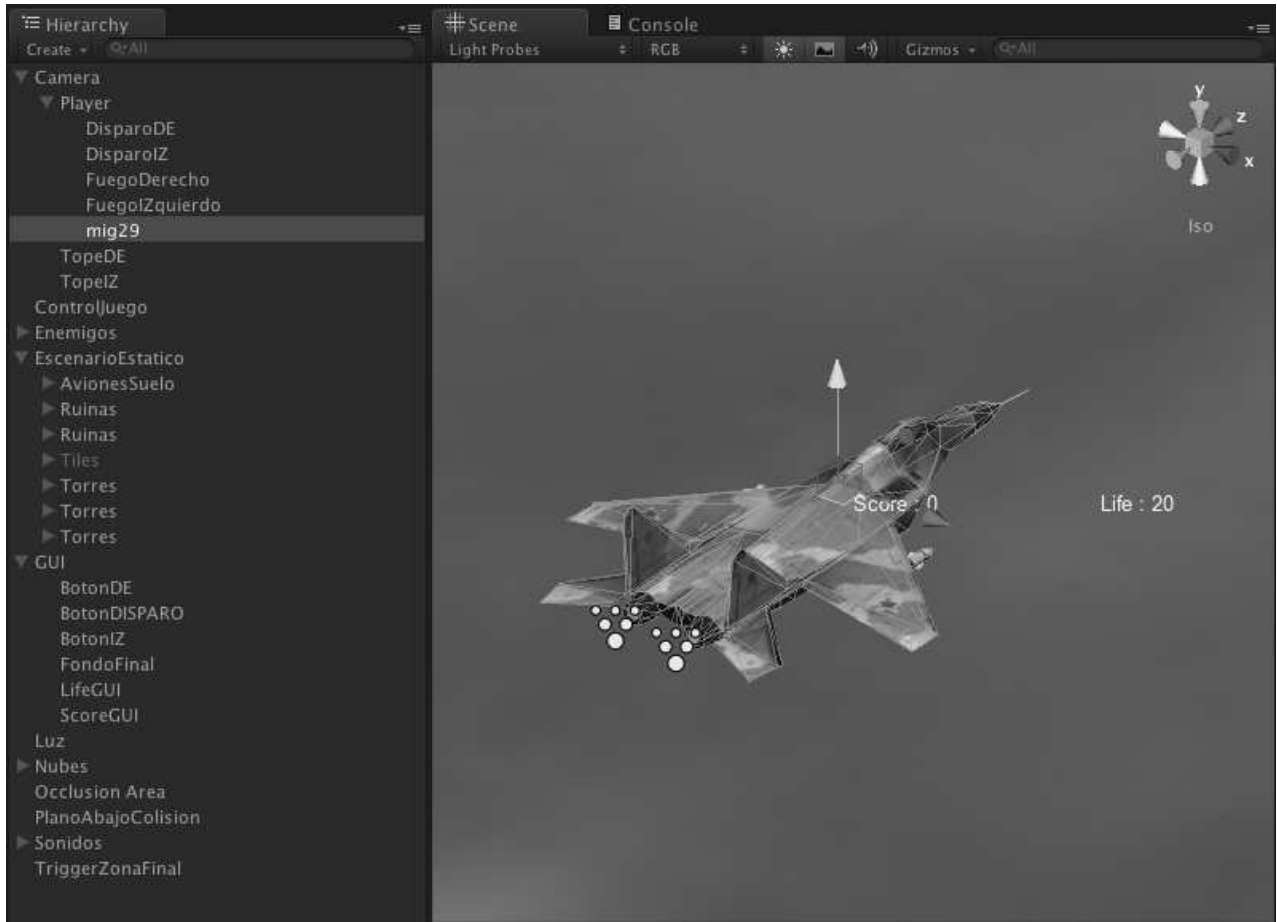


Figura 8.6: Para la escena de nuestro ejemplo hemos añadido el modelo 3D del escenario, los enemigos, el jugador y hemos establecido las relaciones de jerarquía necesarias entre estos elementos.



Figura 8.5: Visualización de la jerarquía de assets de nuestro proyecto.

1. **Una escena por cada nivel del juego:** Utilizaremos este enfoque cuando cada nivel tenga elementos diferentes. Podrán repetirse elementos de otros niveles, pero trataremos que estos elementos sean prefabs para que si los modificamos en alguna escena, el cambio se produzca en todas.
2. **Una única escena con elementos modificados dinámicamente:** Puede que en nuestro videojuego todos los niveles tengan el mismo tipo de elementos pero lo diferente sea la dificultad o el número de enemigos, en este caso podemos crear una única escena pero variar sus condiciones dinámicamente en función de en que nivel estemos.

Como las escenas pueden cargarse desde otra escena. Podemos realizar un cambio de escena cuando se ha llegado al final de un determinado nivel por ejemplo. Este cambio de escena mantendrá en memoria los elementos comunes como texturas o modelos 3D o soni-



Figura 8.7: El nodo Camera contiene otros nodos de manera jerárquica.

dos que pertenezcan a los dos escenas, la escena que se descarga y la escena que se carga, por lo que dependiendo del caso esta carga suele ser bastante rápida. También podemos realizar los menús en una escena y desde ahí cargar la escena del primer nivel del juego.

8.3. Creación de *prefabs*

Como se ha descrito en el apartado anterior, cada escena contiene instancias de objetos de nuestros Assets. Cada uno de los objetos de nuestra escena es un nodo, y cada nodo puede contener jerárquicamente a otros nodos.

Podemos agrupar esa jerarquía y darle un nombre propio para después serializarla e instanciarla en el futuro. A ese concepto se le conoce con el nombre de *prefab*. Podemos crear tantos *prefabs* como queramos a partir de jerarquías de objetos de una escena y son una parte fundamental para entender el método de trabajo con un motor.

En nuestra escena hemos creado *prefabs* para cada uno de los tipos de enemigos, y también hemos creado *prefabs* para el disparo, una explosión y un efecto de partículas de llamas.

Uso de *prefabs*

Cuando un elemento se repita en los diferentes niveles o en la misma escena debe de ser un *prefab*. De esta forma sólo se tiene una referencia de ese objeto y es óptimo en rendimiento y organización de assets.

8.4. Programación de *scripts*

Una *script* es un fichero de código que contiene instrucciones sobre el comportamiento de un determinado actor de nuestra escena. Podemos añadir uno o varios *scripts* a cada uno de los elementos de nuestra escena y además los *scripts* tienen la posibilidad de hacer referencia a estos objetos o *scripts* de otros objetos.

En los *scripts* podemos utilizar las clases y API's que nos proporciona el motor de videojuegos. Algunas de estas clases son:

Documentación

Podemos consultar la documentación de cada una de las API's para los tres lenguajes de scripting desde la página oficial de Unity3D



Figura 8.8: Prefabs de nuestro videojuego de ejemplo.



Figura 8.9: En nuestro modelo del helicóptero se ha separado la hélice del resto del modelo para poder añadirle este script de movimiento.

- **GameObject:** Esta clase tiene información sobre el objeto. Todos los nodos de una escena son GameObjects.
- **Transform:** Esta clase representa la posición, rotación y escala de un elemento en el espacio tridimensional.
- **AudioSource:** Esta clase almacena un sonido y permite gestionar su reproducción.
- **Texture 2D:** Esta clase contiene una textura bidimensional.

Los scripts tienen algunos métodos especiales que podemos implementar como:

- **Update:** Este método es invocado por el motor gráfico cada vez que el objeto va a ser renderizado.
- **Start:** Este método es invocado por el motor gráfico cuando se instancia el objeto que contiene a este script.

8.4.1. Algunos scripts básicos

Algunos de nuestros enemigos son helicópteros. Podemos añadir el siguiente script a el objeto de las hélices para que realice una rotación sobre su eje perpendicular.

Listado 8.1: HeliceHelicoptero.js

```

1 #pragma strict
2
3 public var delta = 4.0;
4
5 function Update () {
6     //Rotar la helice en el eje y
7     transform.Rotate(0,Time.deltaTime * delta,0);
8 }

```

En nuestro escenario se han añadido nubes modeladas mediante planos y con una textura de una nube con transparencias. Para darle mayor realismo a este elemento se va a programar un script para mover las coordenadas u,v del mapeado de este plano provocando que la textura se mueve sobre el plano y simulando un movimiento de nubes.

Listado 8.2: Nube.js

```

1 #pragma strict
2
3 public var delta = 0.1;
4 public var moveFromLeftToRight : boolean = false;
5 private var offset : float = 0.0;
6
7 function Update () {
8     //Mover la coordenada u o v de la textura de el material del
9     //objeto que contiene este script
10    if (!moveFromLeftToRight) {
11        renderer.material.SetTextureOffset ("_MainTex", Vector2(
12            offset,0));
13    } else {
14        renderer.material.SetTextureOffset ("_MainTex", Vector2(0,
15            offset));
16    }
17    offset+=Time.deltaTime * delta;
18    if (offset>1.0) { offset-=1.0; }
19 }

```

En los scripts se utiliza el valor de Time.deltaTime para interpolar el valor de otros elementos con respecto al tiempo que ha pasado desde el último frame renderizado. De esta forma el videojuego va igual de rápido en todas las máquinas, pero se visualizará de manera más fluida en máquinas más rápidas debido a que al renderizar mayor número de frames por segundo se producirán más posiciones intermedias de cada uno de los valores que dependen de Time.deltaTime.

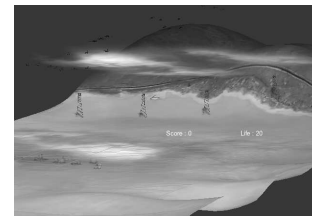


Figura 8.10: Visualización de las nubes.



Figura 8.11: Trigger de la zona final.

8.4.2. Triggers

Un trigger es una porción del espacio definida por un objeto geométrico como una caja o una esfera que utilizaremos para colocar sobre la escena y de esta forma saber cuando un determinado objeto entra o sale de una zona concreta. De esta forma podremos invocar diferentes comportamientos en ese momento. En nuestra escena utilizaremos un trigger para determinar cuando hemos llegado al enemigo final.

Listado 8.3: TriggerFinal.js

```

1 #pragma strict
2
3 function OnTriggerEnter (other : Collider) {
4     if (other.tag=="Player"){
5         //Si entra un objeto dentro de el trigger y el objeto es el
           jugador, pasarle el mensaje a ControlJuego de que hemos
           llegado a la parte final
6         var ControlJuegoPointer : Transform = (GameObject.
           FindWithTag("ControlJuego")).transform;
7         var ControlJuegoStatic : ControlJuego = (
           ControlJuegoPointer.GetComponent("ControlJuego") as
           ControlJuego);
8         ControlJuegoStatic.EntrarEnZonaFinal();
9     }
10 }
```

8.4.3. Invocación de métodos retardada

A veces necesitaremos programar un evento para que pase trascurrido un determinado tiempo. En nuestro ejemplo esto ocurre con la explosión, que invocaremos su destrucción 5 segundos después de haberse instanciado.

VARIABLES PÚBLICAS

Las variables públicas nos permitirán modificar esos parámetros desde el interface de Unity3D cuando tengamos el objeto que contiene el script seleccionado.

Listado 8.4: DestruirPasadoUnTiempo.js

```

1 public var timeOut = 1.0;
2
3 function Start(){
4     //Realizar una llamada al método destruir pasados los segundos
           de timeOUT
5     Invoke ("Destruir", timeOut);
6 }
7
8 function Destruir(){
9     //Destruir el objeto que contiene este script
10    DestroyObject (gameObject);
11 }
```

8.4.4. Comunicación entre diferentes scripts

La mayoría de los scripts se comunicarán con otros scripts más complejos. Para ello hay que crear un puntero al objeto que contiene

el script y utilizarlo para a su vez crear un puntero a la instancia de el script de ese objeto. Después, sobre el puntero de esa instancia de script, podremos invocar los métodos definidos en el mismo.

En el siguiente script que controla el disparo se utiliza este procedimiento de comunicación para invocar un método de el script Sonidos, encargado de la reproducción de los sonidos y músicas.

Listado 8.5: Disparo.js

```

1 public var delta = 8.0;
2 public var timeOut = 5.0;
3 public var enemigo : boolean;
4
5 function Start() {
6     //Invocar PlaySonidoDisparo del script Sonidos del objeto
7     //Sonidos
8     var SonidosPointer : Transform = (GameObject.FindWithTag("
9     Sonidos")).transform;
10    var SonidosStatic : Sonidos = (SonidosPointer.GetComponent("
11    Sonidos") as Sonidos);
12    SonidosStatic.PlaySonidoDisparo();
13    Invoke ("Destruir", timeOut);
14 }
15
16 function Update () {
17     //Actualizar la posición del disparo
18     if (enemigo){
19         transform.position.z-=Time.deltaTime * delta*0.85;
20     } else {
21         transform.position.z+=Time.deltaTime * delta;
22     }
23 }
24 function OnCollisionEnter(collision : Collision) {
25     Destruir();
26 }
27
28 function Destruir() {
29     DestroyObject (gameObject);
30 }

```

Sonido 3D

Es posible reproducir sonido en una posición del espacio para que el motor de juego calcule la atenuación, reberberación o efecto doppler del mismo.

Listado 8.6: Sonidos.js

```

1 #pragma strict
2
3 var SonidoDisparo : AudioSource;
4 var SonidoExplosionAire : AudioSource;
5 var SonidoExplosionSuelo : AudioSource;
6 var SonidoVuelo : AudioSource;
7 var MusicaJuego : AudioSource;
8 var MusicaFinal : AudioSource;
9
10 //Realizamos una fachada para que los demás objetos invoquen la
11 //reproducción de sonidos o música
12
13 function PlaySonidoExplosionSuelo(){
14     SonidoExplosionSuelo.Play();
15 }
16
17 function PlaySonidoExplosionAire(){
18     SonidoExplosionAire.Play();
19 }

```

```

19
20 function PlaySonidoDisparo(){
21     SonidoDisparo.Play();
22 }
23
24 function PlayMusicaJuego(){
25     MusicaJuego.Play();
26 }
27
28 function StopMusicaJuego(){
29     MusicaJuego.Stop();
30 }
31
32 function PlayMusicaFinal(){
33     MusicaFinal.Play();
34 }
35
36 function StopMusicaFinal(){
37     MusicaFinal.Stop();
38 }
39
40 function PlaySonidoVuelo(){
41     SonidoVuelo.Play();
42 }

```

Autómatas finitos

La gran mayoría de comportamientos de los actores de un videojuego pueden ser modelados como un autómata finito determinista.

Límite de FPS

En dispositivos móviles el límite de frames por segundo está ajustado por defecto a 30 FPS pero podemos cambiarlo modificando el atributo `Application.targetFrameRate`.

8.4.5. Control del flujo general de la partida

Normalmente se suele utilizar un script que controla el flujo general de la partida. Este script se utiliza como nexo de unión entre el resto de los scripts y se le pasarán mensajes por ejemplo cuando ha terminado la partida. Podemos modelar el comportamiento de este script como si de un autómata se tratara.

Listado 8.7: ControlJuego.js

```

1 #pragma strict
2
3 public var velocidadCamara :float = 2.0;
4 public var enZonaFinal : boolean = false;
5 public var Camara : Transform;
6 public var ScoreGUI : GUIText;
7 public var LifeGUI : GUIText;
8
9 public var BotonIZ : GUITexture;
10 public var BotonDE : GUITexture;
11 public var BotonDISPARO : GUITexture;
12
13 public var FondoFinal : GUITexture;
14 public var TexturaFinalBien : Texture2D;
15 public var TexturaFinalMal : Texture2D;
16
17 private var SonidosStatic : Sonidos;
18 private var Score : int;
19
20 function Awake(){
21     //Hacemos que el juego corra a 60 FPS como máximo
22     Application.targetFrameRate = 60;
23 }
24
25 function Start(){

```

```
26 //Obtenemos el puntero a Sonidos y ajustamos algunos valores
    iniciales
27 var SonidosPointer : Transform = (GameObject.FindWithTag("
    Sonidos")).transform;
28 SonidosStatic = (SonidosPointer.GetComponent("Sonidos") as
    Sonidos);
29 SonidosStatic.PlayMusicaJuego();
30 SonidosStatic.PlaySonidoVuelo();
31 ScoreGUI.text="Score : "+Score;
32 }
33
34 function Update () {
35     if (enZonaFinal && velocidadCamara>0.0){
36         //Si estamos en la zona final paramos el movimiento de
            manera gradual
37         velocidadCamara*=0.95;
38         if (velocidadCamara<0.1) { velocidadCamara=0; }
39     }
40
41     if (velocidadCamara>0.0){
42         //Movemos la cámara en su componente z para hacer scroll
43         Camara.position.z+=Time.deltaTime * velocidadCamara;
44     }
45 }
46
47 function EntrarEnZonaFinal(){
48     //Se ha entrado en el trigger de la zona final
49     enZonaFinal=true;
50
51     SonidosStatic.StopMusicaJuego();
52     SonidosStatic.PlayMusicaFinal();
53 }
54
55 function FinDeJuegoGanando(){
56     //Fin de partida cuando hemos completado la misión
57     FondoFinal.texture = TexturaFinalBien;
58     Restart();
59 }
60
61 function FinDeJuegoPerdiendo(){
62     //Fin de partida cuando hemos fallado la misión
63     FondoFinal.texture = TexturaFinalMal;
64     Restart();
65 }
66
67 function AddScore(valor : int){
68     //Añadimos puntos, por lo que hay que hacer la suma y
        actualizar el texto
69     Score+=valor;
70     ScoreGUI.text="Score : "+Score;
71 }
72
73 function Restart(){
74     //Ocultamos los textos y botones
75     LifeGUI.enabled=false;
76     ScoreGUI.enabled=false;
77     BotonDISPARO.enabled=false;
78     BotonIZ.enabled=false;
79     BotonDE.enabled=false;
80     FondoFinal.enabled=true;
81
82     //Esperamos 5 segundos y hacemos un reload de la escena
83     yield WaitForSeconds (5);
84     Application.LoadLevel(Application.loadedLevel);
85 }
```

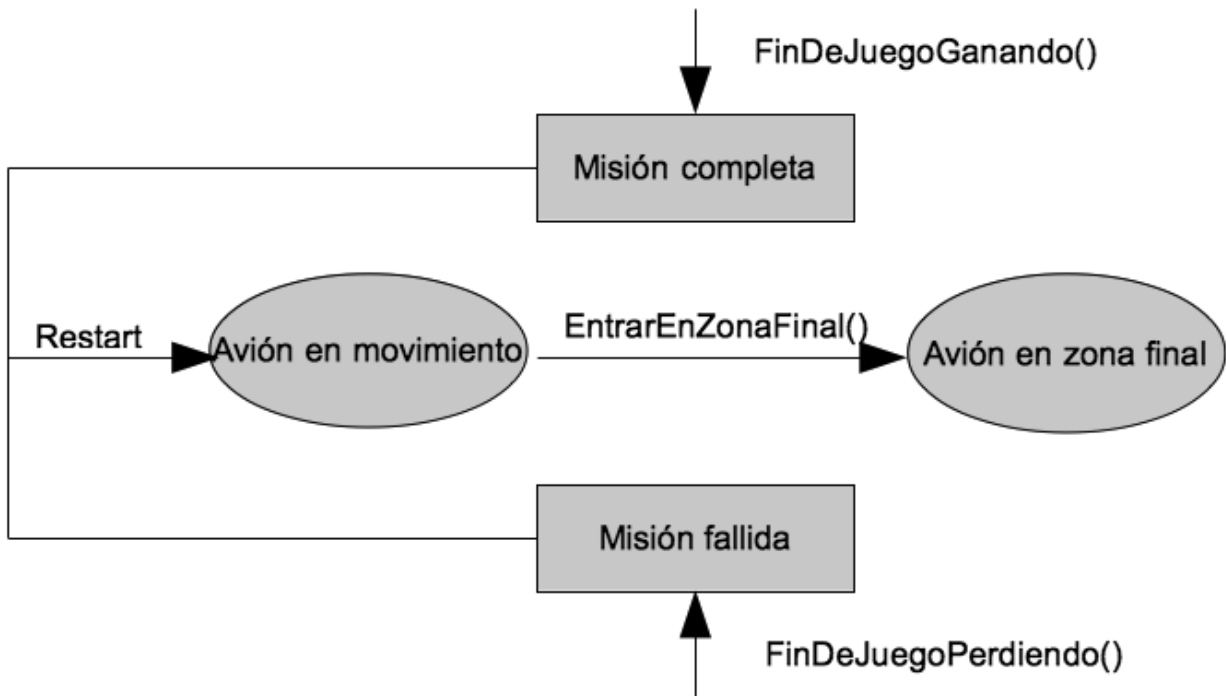


Figura 8.12: Diagrama de control de juego.

Retardos

Utilizaremos la instrucción `WaitForSeconds` para introducir retardos en los scripts.

Detección de colisiones

Cuando un objeto tridimensional tiene añadidos los elementos `collider` y `rigidbody` permite detectar colisiones mediante el método `OnCollisionEnter`.

8.4.6. Programación de enemigos

Vamos a tener un único script para definir el comportamiento de todos los enemigos, incluidos el enemigo final. En este script vamos a definir una serie de atributos públicos que después ajustaremos con unos valores específicos para cada uno de los prefabs de enemigos.

Cada enemigo vendrá determinado un rango de disparo y un tiempo de recarga, y modificando estos parámetros crearemos enemigos más peligrosos que otros. Estos valores influirán en el cálculo de puntuación que proporciona ese enemigo concreto al ser destruido.

Listado 8.8: Enemigo.js

```

1 #pragma strict
2
3 public var explosionPrefab : Transform;
4 public var llamasPrefab : Transform;
5 public var disparoPrefab : Transform;
6 public var player : Transform;
7 public var rangoDisparo : float;
8 public var tiempoRecarga = 0.5;
9 public var jefeFinal : boolean = false;
10

```

```

11 private var siguienteTiempoDisparo = 0.0;
12 private var enRangoDisparo : boolean=true;
13
14 private var llamasInstancia : Transform;
15 private var cantidadRotationCaida : Vector3 = Vector3(0,0,0);
16
17 function Update () {
18     if (transform.gameObject.rigidbody.useGravity){
19         //Si el enemigo está callendo, el avión rota sobre sus ejes
20         //porque entra en barrena
21         transform.Rotate(Time.deltaTime * cantidadRotationCaida.x,
22             Time.deltaTime * cantidadRotationCaida.y,Time.deltaTime
23             * cantidadRotationCaida.z);
24     } else {
25         if (player!=null){
26             var distancia : float = transform.position.z-player.
27                 position.z;
28             if (distancia<=rangoDisparo && distancia>0) {
29                 //Si estamos en rango de disparo el avión dispara
30                 //al frente
31                 if (Time.time > siguienteTiempoDisparo){
32                     siguienteTiempoDisparo = Time.time +
33                         tiempoRecarga;
34                     Instantiate(disparoPrefab, transform.position,
35                         Quaternion.identity);
36                 }
37             }
38         }
39     }
40 }
41
42 function OnCollisionEnter(collision : Collision) {
43     //Determinar posición y rotación del punto de contacto de la
44     //colisión
45     var contact : ContactPoint = collision.contacts[0];
46     var rot : Quaternion = Quaternion.FromToRotation(Vector3.up,
47         contact.normal);
48     var pos : Vector3 = contact.point;
49
50     var SonidosPointer : Transform = (GameObject.FindWithTag("
51         Sonidos")).transform;
52     var SonidosStatic : Sonidos = (SonidosPointer.GetComponent("
53         Sonidos") as Sonidos);
54
55     if (transform.gameObject.rigidbody.useGravity || collision.
56         collider.tag=="Player"){
57         //Si estamos callendo y hemos vuelto a colisionar entonces
58         //explota
59         var ControlJuegoPointer : Transform = (GameObject.
60             FindWithTag("ControlJuego")).transform;
61         var ControlJuegoStatic : ControlJuego = (
62             ControlJuegoPointer.GetComponent("ControlJuego") as
63             ControlJuego);
64
65         SonidosStatic.PlaySonidoExplosionSuelo();
66
67         //Instanciamos la explosión final en la posición del
68         //impacto
69         Instantiate(explosionPrefab, pos, rot);
70         if (llamasInstancia!=null){
71             Destroy (llamasInstancia.gameObject);
72         }
73
74         if (jefeFinal) {
75             ControlJuegoStatic.AddScore(500);
76         }
77     }
78 }

```



```

59         ControlJuegoStatic.FinDeJuegoGanando();
60     } else {
61         var cantidadScore : float = (rangoDisparo * (1.0/
62             tiempoRecarga))*5;
63         ControlJuegoStatic.AddScore(cantidadScore);
64     }
65     //Eliminamos el objeto enemigo
66     Destroy (transform.gameObject);
67 } else if (collision.collider.tag=="Disparo"){
68     //Si no estamos callendo y hemos sido tocados por un
69     disparo, empezamos a caer y a arder
70     SonidosStatic.PlaySonidoExplosionAire();
71     //Instanciamos llamas para la posición del impacto y las
72     añadimos jerárquicamente al enemigo
73     llamasInstancia=Instantiate(llamasPrefab, transform.
74         position, Quaternion.identity);
75     llamasInstancia.parent = transform;
76     //Activamos la gravedad del rigidBody del objeto
77     transform.gameObject.rigidbody.useGravity=true;
78     //Calculamos la cantidad de movimiento en caída para los
79     ejes de manera aleatoria
80     cantidadRotationCaída.x=Random.Range(0, 20.0);
81     cantidadRotationCaída.y=Random.Range(0, 20.0);
82     cantidadRotationCaída.z=Random.Range(0, 20.0);
83 }

```

8.4.7. Programación del control del jugador

El jugador controlará su avión con los botones: izquierda, derecha y disparo. Además hay que tener en cuenta que cuando el jugador colisiona con un disparo enemigo o un enemigo debe reducir su vida, y cuando esta llega a cero explotar.

Teclas de control

Aunque el juego final se controle mediante botones virtuales dibujados sobre la pantalla táctil del dispositivo, también permitiremos su control con un teclado para cuando probemos el juego en el emulador integrado en Unity3D.

Listado 8.9: Player.js

```

1 #pragma strict
2
3 public var explosionPrefab : Transform;
4 public var disparoPrefab : Transform;
5 public var posicionDisparoIZ : Transform;
6 public var posicionDisparoDE : Transform;
7 public var tiempoRecarga = 0.5;
8 public var cantidadMovimiento = 0.1;
9 public var camara : Transform;
10 public var vida : int = 5;
11 public var LifeGUI : GUIText;
12 public var topeIZ : Transform;
13 public var topeDE : Transform;
14
15 private var siguienteTiempoDisparo = 0.0;
16 private var anteriorDisparoIZ : boolean = false;
17 private var botonIzquierda : boolean = false;
18 private var botonDerecha : boolean = false;
19 private var botonDisparo : boolean = false;
20

```

```

21 function Start(){
22     //Inicializamos el marcador de vida con el valor de vida
        inicial
23     LifeGUI.text="Life : "+vida;
24 }
25
26 function Update() {
27     if ((botonDisparo || Input.GetButton("Fire1")) && Time.time >
        siguienteTiempoDisparo){
28         //Si hay que disparar, instanciamos prefabs de disparo en
            las posiciones alternativamente izquierda y derecha de
            el avión del jugador
29         siguienteTiempoDisparo = Time.time + tiempoRecarga;
30         if (anteriorDisparoIZ){
31             Instantiate(disparoPrefab, posicionDisparoDE.position,
                posicionDisparoDE.rotation);
32         } else {
33             Instantiate(disparoPrefab, posicionDisparoIZ.position,
                posicionDisparoIZ.rotation);
34         }
35         anteriorDisparoIZ=!anteriorDisparoIZ;
36     }
37
38     if (botonIzquierda || Input.GetButton("Left")){
39         //Si hay moverse a la izquierda se actualiza la posición
            del jugador
40         //También se mueve un poco la cámara para simular un poco
            de efecto parallax
41         if (transform.position.x>topeIZ.position.x) {
42             transform.position.x-= Time.deltaTime *
                cantidadMovimiento;
43             camara.position.x-= Time.deltaTime * cantidadMovimiento
                /2;
44         }
45     } else if (botonDerecha || Input.GetButton("Right")) {
46         //Si hay moverse a la derecha se actualiza la posición del
            jugador
47         //También se mueve un poco la cámara para simular un poco
            de efecto parallax
48         if (transform.position.x<topeDE.position.x) {
49             transform.position.x+= Time.deltaTime *
                cantidadMovimiento;
50             camara.position.x+= Time.deltaTime * cantidadMovimiento
                /2;
51         }
52     }
53 }
54
55 function OnCollisionEnter(collision : Collision) {
56     if (collision.collider.tag=="DisparoEnemigo" || collision.
        collider.tag=="Enemigo"){
57         //Si el jugador colisiona con un disparo o un enemigo la
            vida disminuye
58         vida--;
59         LifeGUI.text="Life : "+vida;
60
61         if (vida<=0){
62             //Si la vida es 0 entonces acaba la partida
63             var ControlJuegoPointer : Transform = (GameObject.
                FindWithTag("ControlJuego")).transform;
64             var ControlJuegoStatic : ControlJuego = (
                ControlJuegoPointer.GetComponent("ControlJuego") as
                ControlJuego);
65             ControlJuegoStatic.FinDeJuegoPerdiendo();
66

```

```

67         //Reproducimos sonido de explosión
68         var SonidosPointer : Transform = (GameObject.
           FindWithTag("Sonidos")).transform;
69         var SonidosStatic : Sonidos = (SonidosPointer.
           GetComponent("Sonidos") as Sonidos);
70         SonidosStatic.PlaySonidoExplosionSuelo();
71
72         //Instanciamos un prefab de explosión en la posición
           del avión del jugador
73         Instantiate(explosionPrefab, transform.position,
           Quaternion.identity);
74         //Eliminamos el avión del jugador
75         Destroy(gameObject);
76     }
77 }
78 }
79
80
81 //Métodos para controlar la pulsación de los botones virtuales
82
83 function ActivarBotonIzquierda() {
84     botonIzquierda=true;
85 }
86
87 function ActivarBotonDerecha() {
88     botonDerecha=true;
89 }
90
91 function ActivarBotonDisparo() {
92     botonDisparo=true;
93 }
94
95 function DesactivarBotonIzquierda() {
96     botonIzquierda=false;
97 }
98
99 function DesactivarBotonDerecha() {
100    botonDerecha=false;
101 }
102
103 function DesactivarBotonDisparo() {
104    botonDisparo=false;
105 }

```

8.4.8. Programación del *interface*

OnGUI

El método OnGui será llamado cuando se redimensiona la ventana o se cambia de resolución de modo que se calcule la posición de los elementos con respecto a las dimensiones de la pantalla para que siempre estén bien colocados.

Utilizaremos botones dibujados sobre la pantalla táctil para controlar el videojuego. Para colocar cada uno de los botones virtuales utilizaremos un script que en función del valor del atributo tipoGUI lo posicionará en una zona determinada de la pantalla.

Listado 8.10: ColocarGUI.js

```

1 #pragma strict
2
3 enum TipoGUI { Life, Score, BotonLeft, BotonRight, BotonShoot };
4 public var tipoGUI : TipoGUI;
5
6 function OnGUI () {
7     // Hacemos que el ancho del botón ocupe un 10 por ciento

```

```

 8 // Alto del botón mantiene la proporción respecto a la imagen
 9 var anchoBoton : float = Screen.width*0.1;
10 var altoBoton : float = anchoBoton * 94.0/117.0;
11 var margen : int = 10;
12
13 //Dependiendo del tipo de guiTexture o guiText; colocamos
14 switch(tipoGUI){
15     case tipoGUI.Life:
16         guiText.pixelOffset = Vector2 (Screen.width/2 - 55,
17             Screen.height/2 - margen);
18         break;
19     case tipoGUI.Score:
20         guiText.pixelOffset = Vector2 (-Screen.width/2 + margen
21             , Screen.height/2 - margen);
22         break;
23     case tipoGUI.BotonLeft:
24         guiTexture.pixelInset = Rect (-Screen.width/2 + margen,
25             -Screen.height/2 + margen, anchoBoton, altoBoton);
26         break;
27     case tipoGUI.BotonRight:
28         guiTexture.pixelInset = Rect (-Screen.width/2 +
29             anchoBoton+ 2*margen, -Screen.height/2 +margen,
30             anchoBoton, altoBoton);
31         break;
32     case tipoGUI.BotonShoot:
33         guiTexture.pixelInset = Rect (Screen.width/2 -
34             anchoBoton - margen, - Screen.height/2 + margen,
35             anchoBoton, altoBoton);
36         break;
37 }
38 }

```

Para darle funcionalidad a estos botones utilizaremos un único script, que en función de el valor de el atributo tipoGUI se comportará de un modo u otro cuando se pulse.

Listado 8.11: BotonGUI.js

```

1 #pragma strict
2
3 public var tipoGUI : TipoGUI;
4 public var Boton : GUITexture;
5 public var TextureON : Texture2D;
6 public var TextureOFF : Texture2D;
7
8 private var wasClicked : boolean = false;
9 private var PlayerStatic : Player;
10
11 function Update() {
12     //Recorre los toques de pantalla
13     for (var touch : Touch in Input.touches) {
14         if (Boton.HitTest (touch.position)){
15             //Si algún toque está dentro de la zona del botón
16             if (touch.phase == TouchPhase.Began) {
17                 //Activar el botón cuando comienza el toque
18                 wasClicked = true;
19                 Activate();
20             } else if (touch.phase == TouchPhase.Ended || touch.
21                 phase == TouchPhase.Canceled) {
22                 //Desactivar el botón cuando comienza el toque
23                 wasClicked = false;
24                 Deactivate();
25             }
26         }
27     }
28 }

```

Touch

El objeto touch representa un toque sobre la pantalla y contiene información de si se está tocando o soltado, además de la posición x e y donde se realizó el toque.

```
25     }
26   }
27 }
28
29 function Activate() {
30   //Ponemos la textura botón pulsado
31   Boton.texture= TextureON;
32   //Dependiendo del tipo de botón que pulsamos enviamos el
33   //mensaje correspondiente a Player
34   switch(tipoGUI){
35     case tipoGUI.BotonLeft:
36       PlayerStatic.ActivarBotonIzquierda();
37       break;
38     case tipoGUI.BotonRight:
39       PlayerStatic.ActivarBotonDerecha();
40       break;
41     case tipoGUI.BotonShoot:
42       PlayerStatic.ActivarBotonDisparo();
43       break;
44   }
45 }
46
47 function Deactivate() {
48   //Ponemos la textura botón sin pulsar
49   Boton.texture= TextureOFF;
50   //Dependiendo del tipo de botón que soltamos enviamos el
51   //mensaje correspondiente a Player
52   wasClicked = false;
53   switch(tipoGUI){
54     case tipoGUI.BotonLeft:
55       PlayerStatic.DesactivarBotonIzquierda();
56       break;
57     case tipoGUI.BotonRight:
58       PlayerStatic.DesactivarBotonDerecha();
59       break;
60     case tipoGUI.BotonShoot:
61       PlayerStatic.DesactivarBotonDisparo();
62       break;
63   }
64 }
65
66 function Start () {
67   //Obtenemos el puntero a Player y ajustamos algunos valores
68   //iniciales
69   var PlayerPointer : Transform = (GameObject.FindWithTag("Player"
70   ")).transform;
71   PlayerStatic = (PlayerPointer.GetComponent("Player") as Player)
72   ;
73   wasClicked = false;
74   Boton.texture= TextureOFF;
75 }
```

8.5. Optimización

El motor gráfico nos proporciona dos herramientas imprescindibles para optimizar nuestros videojuegos: *lightmapping* y *occlusion culling*. Aplicando estas técnicas reduciremos mucho la carga de renderizado y nos permitirá que nuestros videojuegos puedan correr a buena velocidad en dispositivos de poca potencia gráfica como smartphones o tablets.

8.5.1. *Light mapping*

Esta técnica consiste en calcular previamente las sombras que reciben y provocan los objetos estáticos de las escenas para generar unos mapas de sombreado que se aplican mediante multitextura sobre la maya de los modelos 3D. Los modelos sobre los que se aplica esta técnica son renderizados como polígonos con textura sin ningún sombreado, lo que evita el cálculo de iluminación de la maya, ahorrando mucho tiempo de computo.



Figura 8.14: Ejemplo de las sombras que proyectan sobre el terreno los modelos tridimensionales de unas ruinas colocados en la escena.

8.5.2. *Occlusion culling*

Esta técnica consiste en calcular previamente desde todas las posibles posiciones que puede tomar la cámara que objetos son visibles y cuales son ocluidos por otros. Después se utiliza esta información en tiempo de renderizado para no representar los objetos que después no van a ser visibles.

El cálculo de todas las posiciones que puede tomar la cámara se hace discretizando el espacio mediante una matriz tridimensional de la cual podremos elegir el nivel de granularidad. De este modo se calcula que objetos estáticos deberán más tarde ser renderizados cuando la cámara se encuentre en esta región del espacio.

La combinación de esta técnica junto con otras como frustum culling permitirán que podamos tener escenas con millones de polígonos,

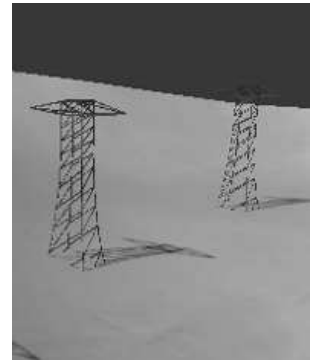


Figura 8.13: Visualización de la sombra provocada por una torre de electricidad.



Figura 8.15: Imagen de lo que visualiza el jugador cuando el motor gráfico está descartando objetos para no ser renderizados cuando la cámara pasa por la sección que se puede contemplar en la figura 8.17.

pero en cada frame de renderizado sólo se representará una pequeña fracción de estos polígonos.



Figura 8.16: Ejemplo de mapa generados mediante esta técnica que después se mapeará sobre el modelo 3D de la escena. Este mapa de sombreado es de la zona que se aprecia en la figura 8.17.



Figura 8.18: Imagen del enemigo final.

8.6. Resultado final

El resultado final es una pieza jugable de unos dos minutos y medio de duración que podría ser un nivel de un videojuego shoot em up de aviones con vista cenital.

Este videojuego está listo para compilarse para dispositivos con iOS o Android y funcionar en terminales de gama media-baja, pudiendo alcanzar los 60 FPS en terminales de gama alta.



Figura 8.17: Imagen del resultado final.

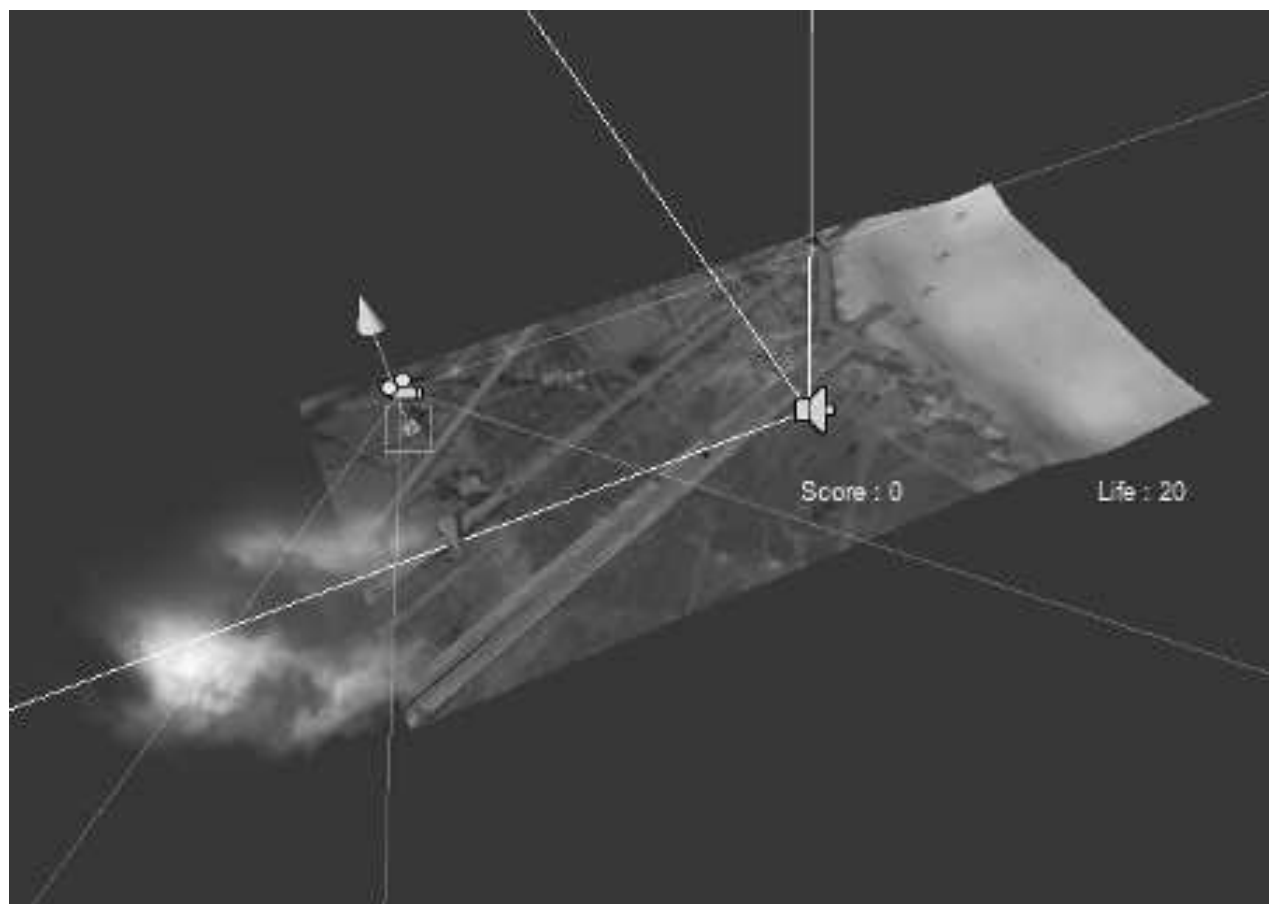


Figura 8.19: En nuestro ejemplo se ha dividido el escenario en porciones para poder aplicar la técnica, de este modo en un momento determinado sólo se renderiza una pequeña parte del escenario.

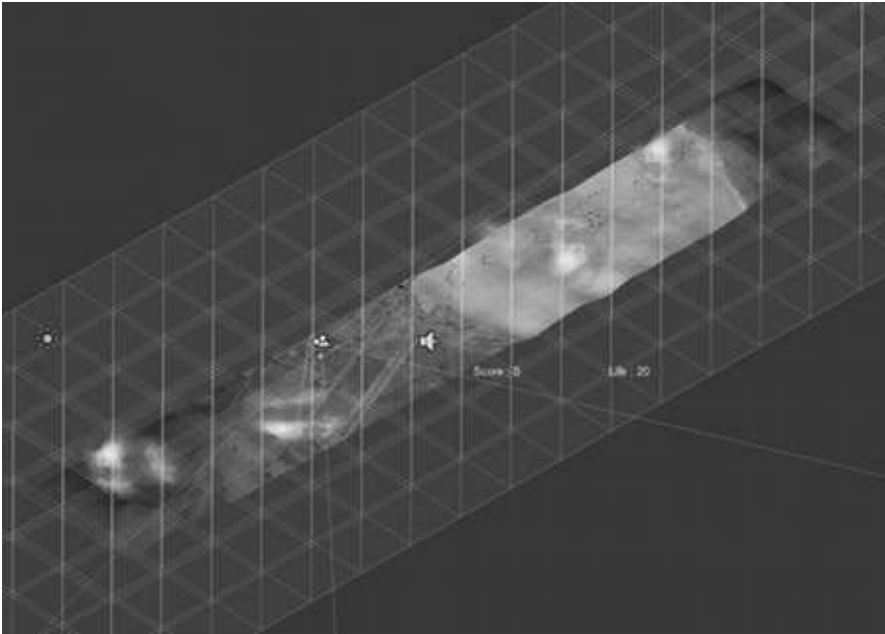


Figura 8.20: Nivel de granularidad elegido para la matriz de discretización del espacio en nuestro ejemplo.

Bibliografía

- [1] www.boost.org.
- [2] www.cegui.org.uk.
- [3] www.swig.org.
- [4] ISO/IEC 9241. *ISO/IEC 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability*. ISO, 1998.
- [5] Advanced Micro Devices, Inc., Disponible en línea en http://support.amd.com/us/Processor_TechDocs/31116.pdf. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors*, Apr. 2010.
- [6] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.
- [7] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional. Addison-Wesley Professional, 1999.
- [8] E Bethke. *Game Development and Production*. Wordware Publishing, 2003.
- [9] Carlos Ble. *Diseño ágil con TDD*. 2009.
- [10] D. Bulka and D. Mayhew. *Efficient C++, Performance Programming Techniques*. Addison-Wesley, 1999.
- [11] James O. Coplien. Curiously recurring template patterns. *C++ Report*, February 1995.
- [12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, third edition edition, 2009.
- [13] A. Dix. *Human computer interaction*. Prentice Hall, 1993.
- [14] K. Flood. Game unified process (gup).

-
- [15] J. L. González. *Jugabilidad: Caracterización de la Experiencia del Jugador en Videojuegos*. PhD thesis, Universidad de Granada, 2010.
- [16] T. Granollers. *MPIu+a. Una metodología que integra la ingeniería del software, la interacción persona-ordenador y la accesibilidad en el contexto de equipos de desarrollo multidisciplinares*. PhD thesis, Universitat de Lleida, 2004.
- [17] Intel Corporation, Disponible en línea en <http://www.intel.com/content/dam/doc/manual/>. *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3B: System Programming Guide, Part 2*, Mar. 2012.
- [18] ISO/IEC. Working Draft, Standard for Programming Language C++. Document number N3242=11-0012., Feb. 2011.
- [19] D. Johnson and J. Wiles. Effective affective user interface design in games. *Ergonomics*, 46(13-14):1332–1345, 2003.
- [20] N.M. Josuttis. *The C++ standard library: a tutorial and handbook*. C++ programming languages. Addison-Wesley, 1999.
- [21] C Keith. Agile game development tutorial. In *Game Developers Conference*, 2007.
- [22] C Keith. *Agile Game Development with Scrum*. Addison-Wesley Professional, 2010.
- [23] Donald E. Knuth. Structured Programming with *go to* Statements. *ACM Computing Surveys*, 6(4), Dec. 1974.
- [24] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice Hall, 2009.
- [25] 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.
- [26] David R. Musser and Alexander A. Stepanov. Generic programming. In *Symbolic and Algebraic Computation: International symposium ISSAC 1988*, pages 13–25, 1988.
- [27] J. Nielsen. *Usability Engineering*. AP Professional, 1993.
- [28] NNG. *User Experience - Our Definition*. Nielsen Norman Group.
- [29] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Revised Fourth Edition*. Morgan Kaufmann, 4th edition edition, 2012.
- [30] R. Rouse III. *Game Design: Theory and Practice*. Wordware Publishing, 2001.
- [31] W.W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, 1970.

-
- [32] E. Salen, K.; Zimmerman. *Rules of Play: Game Design Fundamentals*. The MIT Press, 2003.
 - [33] D. Schmidt. Acceptor-connector: an object creational pattern for connecting and initializing communication services, 1997.
 - [34] B. Shneiderman. Universal usability. *Communications of the ACM*, pages 84–91, 2000.
 - [35] D. Sikora. Incremental game development.
 - [36] B. Stroustrup. *The C++ programming language*. Addison-Wesley, 2000.
 - [37] I. Takeuchi, H.; Nonaka. *Scrum: The new product development game*, 1986.

Este libro fue maquetado mediante el sistema de composición de textos \LaTeX utilizando software del proyecto GNU.

Ciudad Real, a 10 de Julio de 2012.

