Módulo 3

Tutorial de Desarrollo Incremental

David Vallejo Fernández
david.vallejo@tegnix.com

Carlos González Morcillo
carlos.gonzalez@tegnix.com

Capítulo 3

Tutorial de Desarrollo de Videojuegos con NME

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

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

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

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

3.1. El bucle de juego

3.1.1. El bucle de renderizado

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

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

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

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

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

Listado 3.1: Esquema general de un bucle de renderizado.

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

3.1.2. Visión general del bucle de juego

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

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

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

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

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

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

Listado 3.2: Esquema general del bucle de juego.

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

3.1.3. Arquitecturas típicas del bucle de juego

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

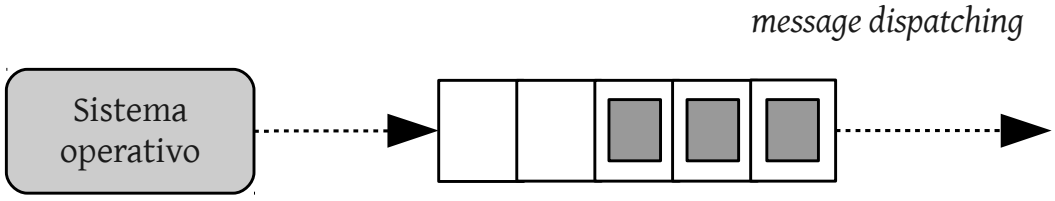

Figura 3.1: Esquema gráfico de una arquitectura basada en *message pumps*.)

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

Tratamiento de mensajes en Windows

En plataformas Windows™, los juegos han de atender los mensajes recibidos por el propio sistema operativo y dar soporte a los distintos componentes del propio motor de juego. Típicamente, en estas plataformas se implementan los denominados ***message pumps*** [5], como responsables del tratamiento de este tipo de mensajes (ver figura 3.1).

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

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

Esquemas basados en retrollamadas

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

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

Listado 3.3: Ejemplo de uso de retrollamadas con OpenGL.

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

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

Tratamiento de eventos

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

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

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

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

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

Esquema basado en estados

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

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

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

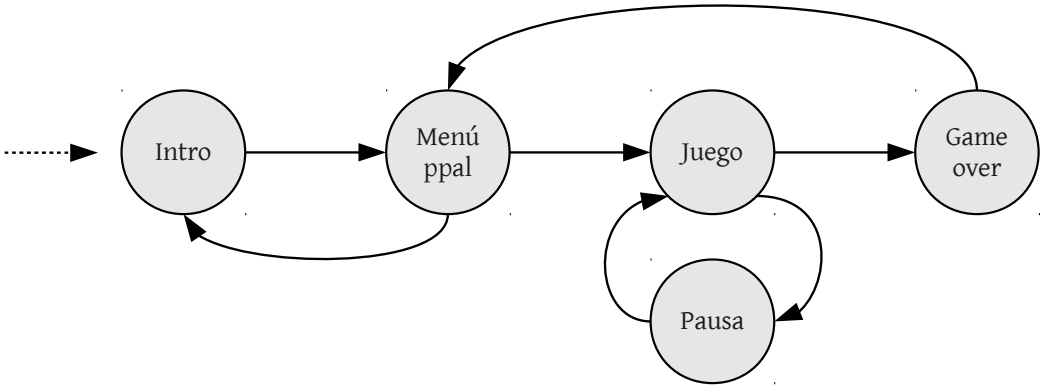

Figura 3.2: Visión general de una máquina de estados finita que representa los estados más comunes en cualquier juego.

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

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

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

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

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

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

3.1.4. Gestión de estados de juego con NME

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

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

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

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

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

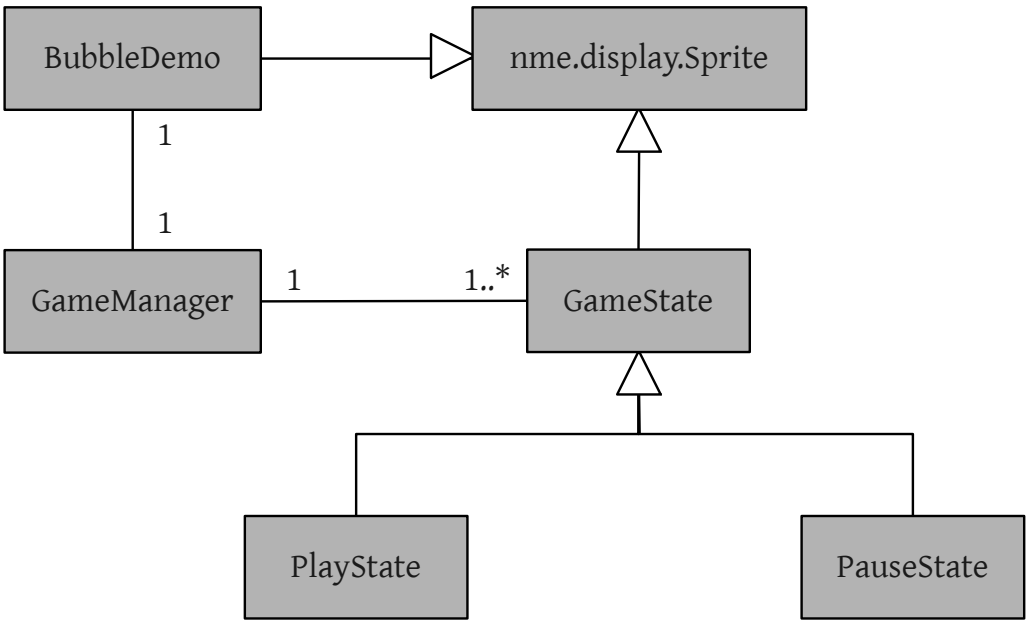

Figura 3.4: Diagrama de clases del esquema de gestión de estados de juego con NME.

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

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

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

Listado 3.4: Clase GameState.

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

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

1. Una **instancia del tipo *GameManager*** (línea 4), es decir, del tipo de la propia clase, que será la única instancia disponible de la misma. En otras palabras, la clase *GameManager* implementa el patrón de diseño *Singleton* [3] para garantizar que sólo exista una instancia disponible para dicha clase.

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

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

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

Listado 3.5: Clase GameManager. Singleton y constructor.

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

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

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

Listado 3.6: Clase GameManager. Función start().

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

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

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

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

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

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

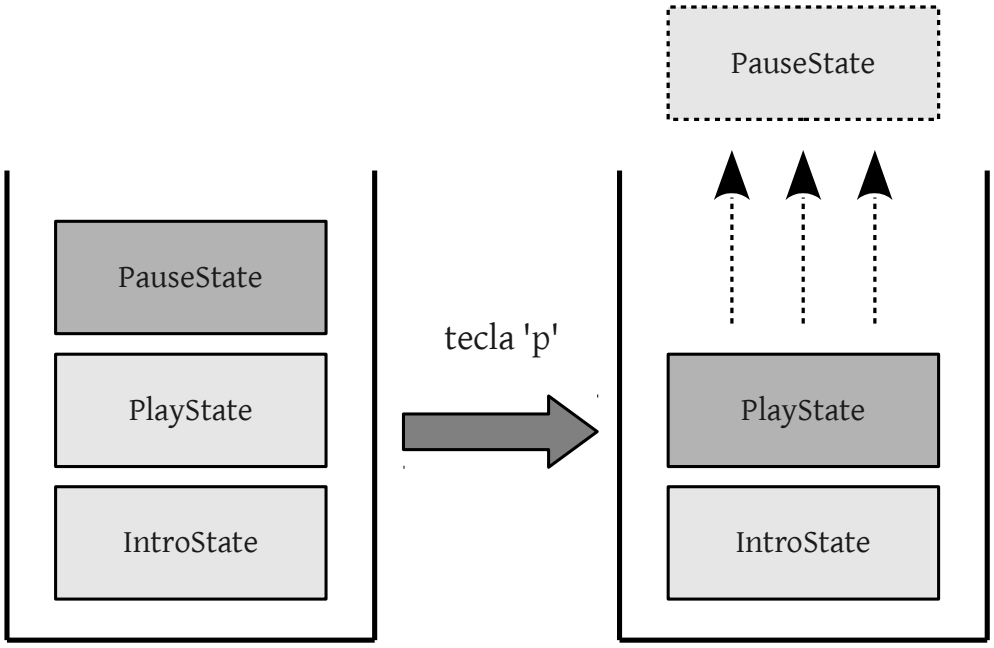

Figura 3.5: Ejemplo de actualización de la pila de estados para reanudar el juego desde el estado de *pause* (evento de tecla 'pulsación de tecla p'.)

operaciones siguientes:

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

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

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

Listado 3.7: Clase GameManager. Función changeState().

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

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

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

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

3.1.5. *BubbleDemo*: definición de estados concretos

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

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

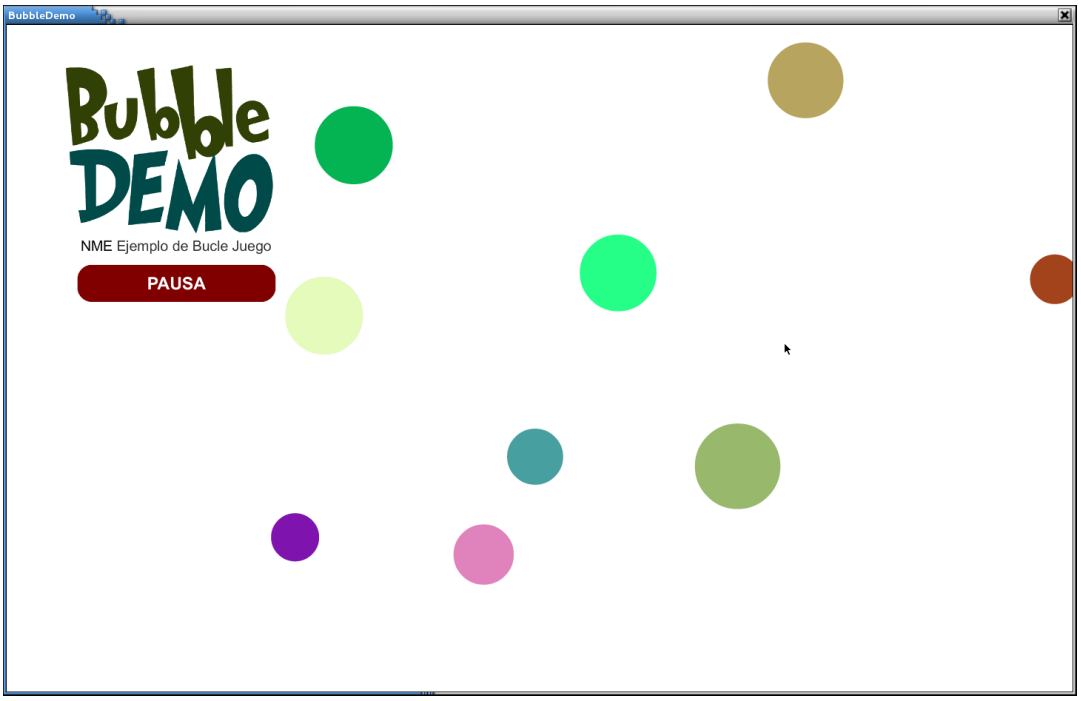

Figura 3.6: Captura de pantalla del estado de pausa de *BubbleDemo*.

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

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

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

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

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

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

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

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

La clase *PlayState*

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

Listado 3.8: Clase PlayState. Constructor.

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

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

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

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

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

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

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

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

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

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

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

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

Listado 3.10: Clase PlayState. Eventos de teclado.

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

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

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

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

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

Listado 3.11: Clase PlayState. Eventos de ratón.

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

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

Listado 3.12: Clase PlayState. Función createCircle().

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

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

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

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

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

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

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

Listado 3.14: Clase Circle. Función animate().

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

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

Listado 3.15: Clase Circle. Función impact().

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

La clase *PauseState*

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

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

Listado 3.16: Clase PauseState. Función keyPressed().

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