

Tutorial de libSDL para la programación de videojuegos

Antonio García Alba

19 de enero de 2008

Índice general

1. Desarrollando un videojuego	1
1.1. Introducción	1
1.2. Objetivos	3
1.3. ¿Qué necesitamos?	3
1.3.1. Introducción	3
1.3.2. Herramientas para el desarrollo	3
1.3.3. Conocimientos previos	6
1.3.4. APIs para el desarrollo de videojuegos	8
1.3.5. Diseño multimedia	9
1.4. Anatomía de un Videojuego	10
1.4.1. Introducción	10
1.4.2. Estructura de un videojuego	11
1.5. El desarrollo de la “idea”. Los tipos de juegos	15
1.6. Resumen	19
2. Conociendo libSDL	21
2.1. Introducción	21
2.2. Objetivos	21
2.3. ¿Qué es libSDL?	22
2.4. ¿Qué nos proporciona SDL?	24
2.4.1. Vídeo y Gráficos	24
2.4.2. Eventos de Entrada	25
2.4.3. Sonido	25
2.4.4. Manejo del CDROM	26
2.4.5. Timers	26
2.4.6. Gestión de dispositivos	26
2.4.7. Red	27
2.5. ¿Por qué SDL?	27
2.6. Desventajas de SDL	34
2.7. El futuro de SDL	35
2.8. Recopilando	35

ÍNDICE GENERAL

3. Instalación libSDL	37
3.1. Introducción	37
3.2. Objetivos	37
3.3. Instalando SDL	37
3.3.1. Requisitos	37
3.3.2. Instalación	38
3.4. Instalando las librerías adicionales	39
3.5. ¿Dónde se instalan las librerías? Localización	40
3.6. Probando la instalación	40
3.7. Recopilando	42
4. Primeros pasos con SDL. Conociendo el entorno	43
4.1. Introducción	43
4.2. Objetivos	43
4.3. El entorno de desarrollo	43
4.4. SDL y los subsistemas	44
4.5. Compilando	45
4.5.1. La herramienta sdl-config	47
4.6. Los tipos de datos en SDL y la portabilidad	48
4.7. SDL_GetError() y SDL_WasInit()	49
4.8. Inicialización y Finalización de SDL	51
4.9. Hola Mundo	55
4.10. Trabajando con las librerías adicionales	58
4.10.1. SDL_image	58
4.10.2. SDL_ttf	58
4.10.3. SDL_mixer	59
4.10.4. SDL_net	60
4.11. Recopilando	61
5. El Subsistema de Video	63
5.1. Introducción	63
5.2. Objetivos	64
5.3. Conocimientos Previos	65
5.4. Subsistema de Vídeo	66
5.4.1. Inicializando el Subsistema de Vídeo	68
5.4.1.1. Ejemplo 1	72
5.4.2. SDL_VideoInfo. La Información del Subsistema de Video	76
5.4.2.1. Ejemplo 2	79
5.5. Conceptos y Estructuras Fundamentales	84
5.5.1. SDL_Rect. El Área Rectangular	84
5.5.2. SDL_Color. El color en SDL	89
5.5.3. SDL_Palette, La Paleta de Colores	91
5.5.4. SDL_PixelFormat. Formato de Píxel	94
5.5.5. SDL_Surface, La Superficie	98
5.5.6. SDL_Overlay	102

ÍNDICE GENERAL

5.5.6.1. Creando un Overlay	103
5.5.6.2. Destruyendo el Overlay	103
5.5.6.3. Bloqueando y Desbloqueando un Overlay	104
5.5.6.4. Dibujando un Overlay	104
5.5.7. Blitting	105
5.5.8. El Flipping y el Doble Búffer	106
5.6. Manejando Superficies	108
5.6.1. Introducción	108
5.6.2. Modificando la superficie	108
5.6.3. Primitivas Gráficas	117
5.6.4. Ejercicio 1. La recta	117
5.6.5. Ejercicio 2. El cuadrado	119
5.6.6. Ejercicio 3. La circunferencia	120
5.7. Cargando Mapas de Bits	123
5.7.1. Introducción	123
5.7.2. ¿Cómo cargar un mapa de bits?	123
5.7.3. Ejercicio 4	130
5.8. Transparencias	134
5.8.1. Introducción	134
5.8.2. ¿Para qué un color clave?	135
5.8.3. Color Key	137
5.8.4. Ejemplo 5. Aplicando el color clave	139
5.8.5. Ejercicio 5	140
5.8.6. El Canal Alpha	143
5.8.6.1. Ejemplo 6	144
5.9. Clipping	146
5.9.1. Ejemplo 7	147
5.10. Conversión de Formatos de Píxel	149
5.11. Recopilando	151
6. Captura y Gestión de Eventos	153
6.1. Introducción	153
6.2. Objetivos	154
6.3. Conocimientos previos	154
6.4. Eventos	154
6.5. Captura de eventos	156
6.5.1. Waiting	156
6.5.2. Acceso Directo al Estado de Eventos del Dispositivo	159
6.5.3. Polling	160
6.5.4. ¿Cuál es la mejor opción?	162
6.6. SDL_Event	163
6.6.1. Ejemplo 1	164
6.7. Tipos de Eventos	167
6.8. Teclado	167

ÍNDICE GENERAL

6.8.1.	Introducción	167
6.8.2.	Manejo del teclado mediante eventos	168
6.8.2.1.	Ejemplo 2	172
6.8.2.2.	Repetición de tecla	174
6.8.3.	Ejercicio 1	174
6.8.4.	Ejercicio 2	177
6.8.5.	Ejercicio 3	179
6.8.6.	Ejercicio 4	183
6.8.7.	Manejo del teclado consultando su estado	190
6.8.7.1.	Ejemplo 3	192
6.8.8.	Otras Funciones	195
6.8.9.	Ejercicio 5	195
6.9.	Ratón	199
6.9.1.	Manejo del ratón mediante eventos	199
6.9.1.1.	Ejemplo 4	201
6.9.2.	Acceso directo al estado del ratón	204
6.9.2.1.	Ejemplo 5	204
6.9.3.	Creando cursosres	206
6.9.3.1.	Ejemplo 6	209
6.10.	Dispositivos de Juegos. El Joystick	216
6.10.1.	Introducción	216
6.10.2.	Gestión del joystick mediante eventos	217
6.10.2.1.	Ejemplo 7	218
6.10.3.	Otros eventos del joystick	221
6.10.4.	Acceso directo al estado del Joystick	223
6.10.5.	Información sobre el Dispositivo de Juego	225
6.10.5.1.	Ejemplo 8	227
6.10.6.	Obteniendo el información directa de los dispositivos de juegos	228
6.10.6.1.	Ejemplo 9	230
6.10.7.	Otras funciones de estado del joystick	234
6.11.	Eventos del Sistema	235
6.11.1.	Introducción	235
6.11.2.	Evento <i>Quit</i>	236
6.11.3.	Evento <i>Video Expose</i>	236
6.11.4.	Evento de Redimensionamiento	237
6.11.5.	El Evento <i>Active</i>	238
6.11.6.	Ejercicio 6	239
6.12.	Eventos del Gestor de Ventanas	244
6.13.	Filtrando Eventos	244
6.13.1.	Ejemplo 10	246
6.14.	Enviando Eventos	252
6.15.	Recopilando	253
6.16.	Anexo. Tabla de constantes SDL para el manejo del teclado. . .	253

ÍNDICE GENERAL

7. Subsistema de Audio	257
7.1. Introducción	257
7.2. Objetivos	257
7.3. Conocimientos previos	258
7.4. El Subsistema de Audio	259
7.4.1. Inicializando el subsistema de Audio	261
7.5. Conceptos y estructuras del subsistema	262
7.6. Funciones para el manejo del Audio	264
7.6.1. Abrir, Pausar y Cerrar	264
7.6.1.1. Ejemplo 1	266
7.6.2. Bloqueando y desbloqueando	269
7.6.3. Manejando el formato WAV	270
7.7. ¿Porqué no utilizar este subsistema?	271
7.8. Recopilando	271
8. Subsistema de CDROM	273
8.1. Introducción	273
8.2. Objetivos	274
8.3. Subsistema de CDROM	274
8.3.1. Inicialización	274
8.4. Conceptos y estructuras fundamentales	275
8.5. Funciones del Subsistema de CD	277
8.5.1. Funciones Informativas	277
8.5.2. Ejemplo 1	278
8.5.3. Funciones para la reproducción del CD	279
8.5.4. Ejemplo 2	279
8.5.5. Ejemplo 3	282
8.6. Recopilando	286
9. Control del Tiempo	287
9.1. Introducción. Conocimientos Previos.	287
9.2. Objetivos	287
9.3. Funciones para el manejo del tiempo	288
9.3.1. Marcas de tiempo	288
9.3.1.1. Ejemplo 1	288
9.3.2. Pausando el Tiempo	290
9.3.2.1. Ejemplo 2	290
9.3.3. Ejercicio 1	293
9.4. Timers	295
9.4.1. Introducción	295
9.4.2. Añadiendo un temporizador	295
9.4.3. Eliminando un temporizador	296
9.4.4. Modificando el temporizador	296
9.4.5. Ejemplo 3	297
9.5. Recopilando	301

ÍNDICE GENERAL

10. Gestor de Ventanas	303
10.1. Introducción	303
10.2. Objetivos	303
10.3. Funciones para el manejo de ventanas	304
10.3.1. El Título	304
10.3.2. Icono	305
10.3.3. Ejemplo 1	306
10.3.4. Minimizando la ventana	308
10.3.5. Maximizando la ventana	308
10.3.6. Ejemplo 2	309
10.4. Convertir la entrada de usuario en exclusiva	311
10.5. Ejercicio 1	312
10.6. Capturando los Eventos del Windows Manager	315
10.7. Recopilando	316
11. SDL_image. Soporte para múltiples formatos.	317
11.1. Introducción. Conocimientos Previos	317
11.2. Objetivo	318
11.3. Compilando con SDL_image	318
11.4. Usando SDL_image	318
11.4.1. Ejemplo 1	319
11.5. Recopilando	322
12. SDL_ttf. Escritura de textos sobre superficies	325
12.1. Introducción	325
12.2. Objetivos	326
12.3. Compilando	326
12.4. Inicialización	326
12.5. Creando y Destruyendo	327
12.6. Obteniendo Información	328
12.7. Manejando las Fuentes	329
12.8. Rendering	331
12.8.1. Ejemplo 1	332
12.9. Recopilando	335
13. SDL_mixer. Gestión de sonidos.	337
13.1. Introducción. Conocimientos previos	337
13.2. Compilando con SDL_mixer	337
13.3. Inicializando la librería	338
13.4. Los Sonidos. Chunks	340
13.4.1. Cargando Chunks	341
13.4.2. Liberando Chunks	341
13.4.3. Estableciendo el Volumen	342
13.5. Canales	342
13.5.1. Asignando Canales	343

ÍNDICE GENERAL

13.5.2. Aplicando Efectos	343
13.5.3. Parando la reproducción	344
13.5.4. Ejemplo 1	345
13.5.5. Obteniendo información	349
13.5.6. Ejemplo 2	350
13.6. Grupos	355
13.6.1. Configurando los grupos	355
13.6.2. Obteniendo información	356
13.6.3. Efectos sobre los grupos	357
13.6.4. Ejemplo 3	357
13.7. Música	362
13.7.1. Cargando la Banda Sonora	363
13.7.2. Reproduciendo la Música	363
13.7.3. Obteniendo información de la Música	366
13.7.4. Ejemplo 4	367
13.8. Effects	372
13.8.1. Efectos de Posicionamiento	372
13.9. Recopilando	373
14. SDL_net. Recursos de red	375
14.1. Introducción	375
14.2. Conceptos Básicos	375
14.3. Compilando con SDL_net	378
14.4. Inicializando	379
14.5. Las Direcciones IP	379
14.5.1. Trabando con IP's	380
14.6. TCP Socket	380
14.6.1. Implementación de un servidor TCP	383
14.6.2. Implementación de un cliente TCP	386
14.7. UDP Socket	388
14.7.1. Implementando un servidor UDP	389
14.7.2. Implementando un cliente UDP	391
14.8. Sockets Genéricos. Conjuntos de Sockets	393
14.9. Recopilando	395
15. Los Sprites y los Personajes	397
15.1. Introducción	397
15.2. Objetivos	397
15.3. Sprites	397
15.4. Animando un Sprite	400
15.4.1. Qué es una animación	400
15.4.2. Nuestra primera animación	400
15.4.2.1. Implementando una animación	401
15.4.3. Animación Interna	404
15.4.4. Implementado el Control de la animación interna	406

ÍNDICE GENERAL

15.4.5. Gestionando una rejilla de imágenes	410
15.4.6. Clase Animación Interna	420
15.4.7. Animación Externa. Actualización Lógica	427
15.4.8. Implementando el control de la animación externa	428
15.4.9. Actualización Lógica	435
15.4.10 Ejercicio 1	439
15.4.11 Integrando las animaciones	443
15.5. Creando una galería	444
15.5.1. Introducción	444
15.5.2. Implementación de una galería	444
15.6. La animación interna y los autómatas	447
15.6.1. Introducción	447
15.6.2. Estados	447
15.6.3. Autómata	448
15.6.4. Implementando un autómata	450
15.6.5. Conclusión	466
15.7. Colisiones	466
15.7.1. Introducción	466
15.7.2. ¿Qué es una colisión? Detectando colisiones	467
15.7.3. Tipos de colisiones	469
15.7.4. Implementando la detección de colisiones	470
15.7.5. Detección de colisiones de superficie única	470
15.7.6. Detección de colisiones de superficies compuestas	476
15.7.7. Ejercicio 2	480
15.7.8. Ejercicio 3	482
15.7.9. Otros algoritmos de detección de colisiones	483
15.8. Recopilando	483
16. Un ejemplo del desarrollo software de un videojuego	485
16.1. Introducción	485
16.2. Conocimientos previos	485
16.3. Objetivos	486
16.4. Planteamiento informal de un videojuego	486
16.5. La historia	486
16.5.1. Introducción. Cómo contar la historia	486
16.5.2. Nuestra historia	488
16.6. Los personajes	488
16.7. Los niveles y el Editor	489
16.7.1. La creación de niveles	489
16.7.2. El editor de niveles. Los tiles	490
16.7.3. La lógica del juego	490
16.7.4. El control del tiempo	492
16.7.5. La detección de colisiones	493
16.8. ¿Por qué Programación Orientada a Objetos?	494

ÍNDICE GENERAL

16.8.1. Características de la Orientación a Objetos	495
16.9. Modelado	496
16.10 Especificación de los requisitos del sistema	497
16.10.1 Requisitos de interfaces externas	497
16.10.2 Requisitos funcionales	499
16.10.3 Requisitos de rendimiento	500
16.10.4 Restricciones de diseño	500
16.10.5 Atributos del sistema software	500
16.10.6 Otros requisitos	501
16.11 Análisis	501
16.11.1 Modelo de Casos de Uso	502
16.11.1.1 Casos de uso	502
16.11.1.2 Diagramas de casos de uso	503
16.11.1.3 Descripción de los casos de uso	503
16.11.2 Modelo conceptual de datos en UML	512
16.11.2.1 Conceptos básicos	513
16.11.2.2 Descripción diagramas de clases conceptuales .	513
16.11.3 Diagrama de clases	515
16.11.4 Modelo de comportamiento del sistema	520
16.11.4.1 Diagramas de secuencia del sistema y los Contratos de operaciones	521
16.11.4.2 Diagramas de secuencia del sistema y Contratos de las operaciones del sistema	521
16.12 Diseño del Sistema	532
16.12.1 Arquitectura del sistema software	532
16.12.2 Diseño de la capa de dominio	533
16.12.3 Diagrama de clases de diseño	534
16.12.4 Diagrama de secuencia	540
16.12.5 Diseño de los personajes	545
16.13 Implementación	548
16.13.1 La clase Apuntador	548
16.13.2 La clase Control Animacion	555
16.13.3 La clase Control Juego	559
16.13.4 La clase Control Movimiento	568
16.13.5 La clase Editor	572
16.13.6 La clase Enemigo	579
16.13.7 La clase Fuente	583
16.13.8 La clase Galeria	593
16.13.9 La clase Imagen	598
16.13.10 La clase Interfaz	605
16.13.11 La clase Item	608
16.13.12 La clase Juego	611
16.13.13 La clase Menu	615
16.13.14 La clase Musica	623

ÍNDICE GENERAL

16.13.1La clase Nivel	625
16.13.1La clase Participante	641
16.13.1La clase Protagonista	647
16.13.1La clase Sonido	655
16.13.1La clase Teclado	658
16.13.2La clase Texto	661
16.13.2La clase Universo	663
16.13.2La clase Ventana	673
16.14Recopilando	677

Índice de figuras

1.1.	Remake del popular Tetris ©	2
1.2.	Logotipo de GNU/GCC	4
1.3.	Kdevelop y Anjuta	5
1.4.	Pong vs Tenis 3D	7
1.5.	The Gimp	10
1.6.	Estructura de un videojuego	13
1.7.	Juego arcade: Tux	17
1.8.	Juego de simulación	18
1.9.	Juego de estrategia	18
2.1.	Logotipo SDL	23
2.2.	Composición de SDL	24
2.3.	Descent 2. Juego portado mediante SDL	29
2.4.	OpenGL	29
2.5.	DirectX©	30
4.1.	Hola Mundo en un entorno KDE	57
5.1.	Ejemplo de pantalla de videojuego	64
5.2.	Variaciones en la resolución de una imagen	65
5.3.	Tarjeta gráfica y monitor actuales	67
5.4.	Representación de una imagen en pantalla de 24 bpp	68
5.5.	Diagrama de inicialización del subsistema de video	72
5.6.	Superficie Rectangular	85
5.7.	Sistemas de Coordenadas	86
5.8.	Representación de las variables de <code>SDL_Rect</code>	87
5.9.	Rectángulo rotado en SDL	88
5.10.	Espacio de colores RGB	90
5.11.	Paleta de colores segura para la web	92
5.12.	Pitch de la superficie	99
5.13.	Resultado del blitting	106
5.14.	Doble Búffer	107
5.15.	Diagrama de flujo. Modificando una superficie.	109
5.16.	Obteniendo una componente de color de 8 bits.	113
5.17.	Plantilla Ejercicio 4	130

ÍNDICE DE FIGURAS

5.18. Ejemplo de personaje sin y con colorkey activado	137
6.1. Evento	154
6.2. Esperando eventos (Waiting)	157
6.3. Diagrama de flujo. Waiting	158
6.4. Acceso directo al estado del dispositivo.	159
6.5. Polling o Sondeo.	161
6.6. Distribución habitual de un teclado español.	168
6.7. Eventos y estados en el teclado.	169
6.8. Ratón.	199
6.9. Tipos de joysticks	216
6.10. Elementos de un Joystick	221
6.11. Posiciones del hat	222
6.12. Ejes del joystick	224
7.1. Frecuencia	258
7.2. Amplitud y Periodo	259
7.3. Tarjeta de sonido de gama alta.	260
7.4. Altavoces para ordenador.	261
8.1. Unidad de CD	273
10.1. Área de título de la ventana	305
11.1. Ejemplo 1	323
12.1. Anatomía de una fuente	329
14.1. Red P2P	377
14.2. Red Cliente-Servidor	378
14.3. Conexión TCP	383
14.4. Conexión UDP	388
15.1. Ejemplo de Sprite	398
15.2. Ejemplo de rejilla. Nuestro personaje principal Jacinto	405
15.3. Representación de la animación externa.	427
15.4. Ejemplo de autómata de nuestro personaje principal	449
15.5. Ejemplo de colisión.	467
15.6. Elementos de una imagen SDL.	468
15.7. Colisión entre triángulos simple.	476
15.8. Triángulo dividido en rectángulos	477
15.9. Esfera dividida en rectángulos	481
15.10. Jacinto dividido en secciones	482
16.1. División en tiles	491
16.2. Lógica del juego	492
16.3. Prototipo: Ventana Menú	498

ÍNDICE DE FIGURAS

16.4. Prototipo: Editor de Niveles	499
16.5. Diagrama de casos de uso: Videojuego de ejemplo	504
16.6. Diagrama de clases: Componentes multimedia	516
16.7. Diagrama de clases: Editando niveles	517
16.8. Diagrama de clases: Participantes	518
16.9. Diagrama de clases: Universo	519
16.10Diagrama de clases: Diagrama Global	520
16.11Diagrama de secuencia: Cargar Niveles	522
16.12Diagrama de secuencia: Siguiente Nivel	524
16.13Diagrama de secuencia: Nivel Anterior	525
16.14Diagrama de secuencia: Nuevo Nivel	526
16.15Diagrama de secuencia: Editar Nivel	527
16.16Diagrama de secuencia: Guardar Nivel	528
16.17Diagrama de secuencia: Jugar	529
16.18Diagrama de secuencia: Gestionar Niveles	531
16.19Diagrama de clases (diseño): Componentes multimedia	535
16.20Diagrama de clases (diseño): Editando niveles	536
16.21Diagrama de clases (diseño): Participantes	538
16.22Diagrama de clases (diseño): Universo	539
16.23Diagrama de secuencia (diseño): Universo	541
16.24Diagrama de secuencia (diseño): Editor	542
16.25Diagrama de secuencia (diseño): Galeria	543
16.26Diagrama de secuencia (diseño): Juego	544
16.27Diagrama de secuencia (diseño): Menu	545
16.28Diseño gráfico: Personaje Principal	546
16.29Diseño gráfico: Enemigo polvo	546
16.30Diseño gráfico: Enemigo rata	547
16.31Diseño gráfico: Tiles o bloques	547
16.32Diseño gráfico: Ítem destornillador	548
16.33Diseño gráfico: Ítem alicate	548
16.34Clase Apuntador	549
16.35Clase Control Animacion	556
16.36Clase Control Juego	560
16.37Clase Control Movimiento	569
16.38Clase Editor	573
16.39Clase Enemigo	579
16.40Clase Fuente	583
16.41Clase Galeria	593
16.42Clase Imagen	599
16.43Clase Interfaz	606
16.44Clase Item	608
16.45Clase CJuego	611
16.46Clase Menu	616
16.47Clase Musica	623

ÍNDICE DE FIGURAS

16.48Clase Nivel	626
16.49Clase Participante	642
16.50Clase Protagonista	648
16.51Clase Sonido	656
16.52Clase Teclado	658
16.53Clase Texto	661
16.54Clase Universo	664
16.55Clase Ventana	673

Índice de cuadros

3.1.	Localización de libSDL	40
4.1.	Funciones de inicialización	54
5.1.	Superficies más comunes.	100
6.1.	Eventos soportados por SDL y la estructuras que lo maneja . . .	163
6.2.	Efecto de data y mask sobre la superficie.	208
6.3.	Constantes SDLKKey comunes.	253
6.4.	Constantes modificadoras de teclado SDL.	256

ÍNDICE DE CUADROS

Capítulo 1

Desarrollando un videojuego

1.1. Introducción

Seguramente estés delante de este tutorial porque quieras desarrollar tu primer videojuego. En este capítulo vamos a presentar en qué consiste un videojuego, por dónde comenzar su realización y qué requisitos son fundamentales para afrontar el desarrollo de un videojuego. El proceso de aprendizaje en el desarrollo de videojuegos es continuo (y algunos se atreven a decir que es hasta infinito) por lo que no dejes de seguir ninguno de los capítulos del tutorial ya que es fundamental crear unos cimientos fuertes sobre los que crecer en el mundo de la programación de videojuegos.

Lo primero que nos concierne en este tutorial es marcar el límite de los conocimientos previos a la hora de desarrollar un videojuego. A parte de conocimientos técnicos, que presentaremos a continuación, es fundamental para instruirte en esta materia tener unos buenos conocimientos de inglés ya que, por suerte o por desgracia, la gran mayoría de libros, artículos... referentes a la programación, y más especialmente a la programación de videojuegos, están escritos en inglés.

Cuando se empieza en el desarrollo de videojuegos se suele pensar que el programador debe ser una especie de navaja suiza, que tiene que ser capaz de realizar todo el proceso que conlleva el desarrollo del videojuego. Veremos que esto es así pero sólo en los comienzos y que lo seguirá siendo para pequeños proyectos. Cuando estamos empezando tenemos que ser capaces de crear (o proveernos) de todo el material necesario para el desarrollo del mismo lo que hará que tengamos que invertir grandes cantidades de tiempo y esfuerzo para poner en marcha nuestros proyectos.

A la hora de realizar tareas de cierta envergadura, sobre todo en proyectos ambiciosos, en el desarrollo del videojuego converge el trabajo diseñadores, músicos, escritores, creativos... Es fundamental tener a personal especializado en cada una de las áreas que influyen en la realización de un videojuego. Como no podía de ser de otra forma nosotros vamos a centrarnos en las

1. Desarrollando un videojuego

tareas que deben de realizar los informáticos (en el amplísimo sentido de la denominación) y como aplicar técnicas de análisis, diseño e implementación que nos permitan crear un software de calidad, sin dejar de lado lo referente al mantenimiento y prueba del software.



Figura 1.1: Remake del popular Tetris ©

Para empezar en el desarrollo de videojuegos, como todo en la vida, es importante hacerlo con proyectos pequeños, con objetivos concretos y alcanzables que nos permitan cumplir con las metas que nos fijamos realizando un aprendizaje constructivo. Si empezamos con proyectos demasiado grandes seguramente no podamos acatarlos lo que provocará que abandonemos prematuramente este apasionante mundo.

Una de las alternativas más comunes es empezar realizando juegos parecidos a alguno ya existente que sea fácil de implementar. Esto nos hará centrarnos en cómo realizar las cosas y no en la tarea de crear una nueva historia, un nuevo modo de juego o una manera específica de interactuar con la aplicación. Podrás encontrar en la red de redes numerosos “remakes” de videojuegos clásicos como Mario Bros, Tetris, etc... realizados por programadores que empezaban su incursión en el mundo de los videojuegos. Con esto aprenderemos mucho, y si nos atrevemos a realizarle pequeñas modificaciones que aumenten su jugabilidad, la curva de aprendizaje se convertirá en exponencial.

Es muy importante que sepamos que no vamos a poder realizar un juego tipo DOOM o Unreal a las primeras de cambio, ya que estos fueron proyectos millonarios en los que estaban inmersos muchos programadores y tendríamos que dedicar parte de nuestra vida a alcanzar este nivel si partimos de cero. Nosotros trataremos aquí los videojuegos en 2D. Una recomendación personal es empezar con este tipo de videojuegos, y una vez los domines dar el salto a las tres dimensiones. En unos meses podrás empezar a realizar proyectos que ahora, seguramente, sean inimaginables.

1.2. Objetivos

1. Situarnos en el mundo de los videojuegos.
2. Conocer las herramientas necesarias para realizar un videojuego.
3. Conocer la estructura de un videojuego y los subsistemas que intervienen en él.
4. Determinar que tipo de videojuego vamos a realizar.

1.3. ¿Qué necesitamos?

1.3.1. Introducción

Antes de empezar a desarrollar un videojuego vamos a ver dónde estamos y dónde queremos llegar. Para realizar el recorrido de un punto a otro necesitamos tener claro que tenemos que saber y qué herramientas debemos de manejar para poder cumplir con el objetivo de realizar una primera aplicación gráfica de este estilo.

Estas herramientas son comunes para el desarrollo de casi cualquier tipo de aplicación y lo que haremos en presentar brevemente cada una de ellas. Es importante que hagas un esfuerzo por conocer las distintas herramientas ya que algunas serán de gran ayuda mientras que otras serán fundamentales en el proceso de desarrollo del videojuego.

1.3.2. Herramientas para el desarrollo

En esta sección vamos a ver qué herramientas necesitamos para desarrollar nuestro primer videojuego. Lo primero y fundamental es dominar algún lenguaje de programación. En este tutorial vamos a utilizar C y C++. Vamos a trabajar con SDL, además de ser el lenguaje nativo de esta librería, es el lenguaje de programación más potente para la creación de videojuegos. Existen adaptaciones de la librería a muchos otros lenguajes y una vez dominado

1. Desarrollando un videojuego

en C++ no te será complicado dar el salto al uso de la misma con otro lenguaje.

Otra razón para tomar esta desición es que el lenguaje de programación más usado en la historia del desarrollo de los videojuegos profesionales ha sido el C, aunque desde los años 90 el lenguaje C++ le ha ido ganando terreno gracias a las ventajas que añade sobre el primero, como por ejemplo la orientación a objetos, pudiendo usar todo aquello que nos resultaba ventajoso del C en C++.



Figura 1.2: Logotipo de GNU/GCC

Como vamos a programar, como no podía ser de otra forma, necesitamos un compilador. Hay compiladores libres y propietarios, gratuitos y de pago. En nuestro caso optamos por el GNU/GCC ya que es libre y gratuito y cumple con los estándares impuestos a los lenguajes de programación que vamos a utilizar. No sólo es un compilador de C/C++ sino que además nos proporciona el enlazador, el debugger, el profiler... Es decir, no estamos ante un “simple” compilador sino estamos ante una *suite* de herramientas para la compilación. Originalmente GCC significaba *GNU C Compiler* (compilador GNU para C) porque sólo compilaba lenguaje C. Posteriormente se extendió siendo capaz de compilar C++, Fortran, Ada y otros. Otra de las ventajas del compilador de GNU es que dispone de soporte para un gran número de arquitecturas lo que nos proporciona una capa de abstracción a la hora de trabajar con el lenguaje que nos facilitará la portabilidad de nuestro código.

A estas alturas ya tenemos claro que necesitamos un buen lenguaje de programación y su correspondiente compilador. Ahora llega la hora de decidirnos por un editor de textos donde escribir nuestras líneas de código. Esta es una elección personal. En el tutorial hemos *GNU Emacs* como editor de textos para escribir los ejemplos y realizar pruebas de compilación. Eres libre de utilizar el que más te guste. Eso sí, necesitas uno. En casi todas las distribuciones de Linux tienes disponible numerosos editores libres de distintas generaciones. Desde los más adorados como *Vi* o *Emacs* hasta los modernos *Anjuta* o *Kdevelop* que ofrecen entornos integrados de desarrollo que disponen

1.3. ¿Qué necesitamos?

de opciones como la navegación de clases y la posibilidad de autocompletar. Como recomendación personal te aconsejo que no te ates a ningún entorno integrado de desarrollo ya que te hará perder versatilidad como programador.

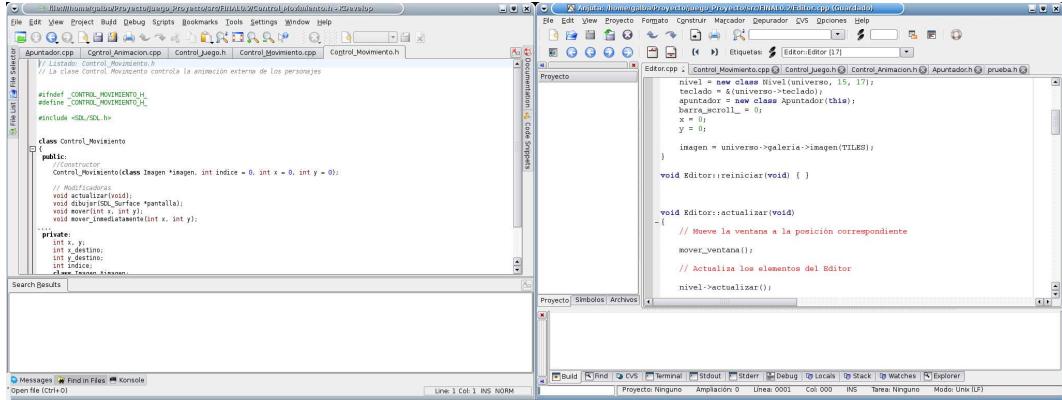


Figura 1.3: Kdevelop y Anjuta

Como depurador te aconsejo GNU/DBG con su front-end DDD. GNU/DBG es un depurador y perfilador que nos permite analizar nuestros programas en C/C++ en modo consola. Añadiéndole DDD conseguiremos poder depurar nuestro programa en un entorno más amigable. El uso de depuradores es muy importante, sobre todo, en los inicios ya que es cuando menos conocemos lo que nos tenemos entre manos y necesitamos comprobar paso a paso que hace nuestra aplicación.

La última herramienta, y la que vamos a exponer en este tutorial, es el uso de una librería que nos ayude a construir nuestro videojuego. SDL nos libera de numerosas tareas como la presentación de imágenes en pantalla y nos facilita otras muchas, como la captura de eventos o la reproducción de sonidos. SDL ha obtenido numerosos premios desde su aparición en el mundo del videojuego, es libre y gratuita... a esto se le añade numerosas virtudes de implementación, ya que, la mayoría de programadores que han trabajado en ella son profesionales con experiencia en el mundo de los videojuegos con lo que han realizado la librería sabiendo que se necesita exactamente para desarrollar un videojuego. ¿Qué más podemos pedir? Presentaremos unos criterios más sólidos sobre la elección en el transcurso del tutorial.

Una vez aprendamos a usar la librería y a crear nuestros primeros videojuegos es fundamental utilizar herramientas para controlar las versiones que vayamos generando de nuestro código así como planificadores de tareas para no dejar el desarrollo de nuestras aplicaciones al capricho del paso del tiempo.

1. Desarrollando un videojuego

Existen varias herramientas gratuitas para la planificación de actividades. Puedes utilizar *KPlato*, *TaskJuggle*, *Gantt Project*... todas de ellas alternativas libres y gratuitas. Con este tipo de herramientas pasa lo mismo que con los editores. Todas tienen sus ventajas y sus inconvenientes y el uso de una alternativa u otra se basa más en la comodidad relativa a cada usuario que en características concretas que ofrezca el programa en cuestión.

El control de versiones se puede hacer de forma manual pero es muy recomendable utilizar herramientas que nos faciliten esta gestión. Ejemplos de herramientas para el control de versiones son *CVS*, *Subversión*, *Sourcesafe*... todas ellas disponibles para su descarga ya que son libres y gratuitas. Seguramente en los primeros desarrollos que realices no necesites de una herramienta de este tipo, sin embargo si te embarcas en un proyecto al que le quieras dar continuidad será fundamental que controles los avances y cambios dentro del proyecto.

1.3.3. Conocimientos previos

En esta sección vamos a realizar una introducción a los conocimientos que son necesarios para abarcar el desarrollo de un videojuego. Serán presentados de forma general con el objetivo de obtener una vista global de lo complejo que puede llegar a ser un desarrollo de este tipo.

Incluso más importante que las herramientas son los conocimientos que tengamos de programación, o mejor dicho, conocimientos de desarrollo software. Puede que muchos de éstos nos sean necesarios cuando creamos pequeñas aplicaciones, pero en cuanto el videojuego vaya aumentando su complejidad, serán fundamentales.

Para desarrollar un videojuego debemos programar con cierta habilidad. En nuestro caso, como ya hemos comentado, vamos a utilizar los lenguajes de programación C y C++. Los conceptos que aporta el lenguaje C++ son fundamentales en la programación de videojuegos ya que proporciona aspectos, como el de clase, que facilitarán el proceso de desarrollo de una aplicación de este estilo.

Es requisito el conocer la definición y el manejo de diferentes estructuras de datos. Este aspecto es parte fundamental de los conocimientos básicos que debe tener cualquier programador. Para facilitar el uso de estructuras de datos agrupadas disponemos de la biblioteca STL que es estándar en C++. Actualmente viene incluida en todos los compiladores disponibles ya que es un estándar. Conviene aprender como utilizar las estructuras de datos que nos proporciona esta librería así como su manejo ya que nos ahorrarán mucho trabajo creando un código más ordenado y elegante.

Es fundamental tener conocimientos de ciencias, no es que tengamos que ser unos expertos en física y matemáticas pero la mayoría de los comportamientos en un videojuego se basan en funciones matemáticas. En cualquier videojuego podrás observar como se aplican, como mínimo, cierta cinemática básica en los personajes o elementos del mismo. Como todo lo anterior, en nuestros primeros proyectos, el uso de la física será básico e irá aumentando según se incremente la complejidad de nuestro videojuego. Uno de los grandes avances en el desarrollo de videojuegos es el de asemejar cada vez más el entorno donde concurre la trama del videojuego al mundo real. Si te gustan los videojuegos de deportes habrás podido observar como han ido superándose año tras año en la simulación de las condiciones físicas que rodean a dicho deporte. En los primeros tiempos de los videojuegos jugar al tenis era manejar una línea blanca que hacía rebotar un cuadrado y actualmente para empezar una partida tienes que elegir hasta el tipo de suelo donde va a transcurrir la partida según se adecúe más a tu estilo de juego.

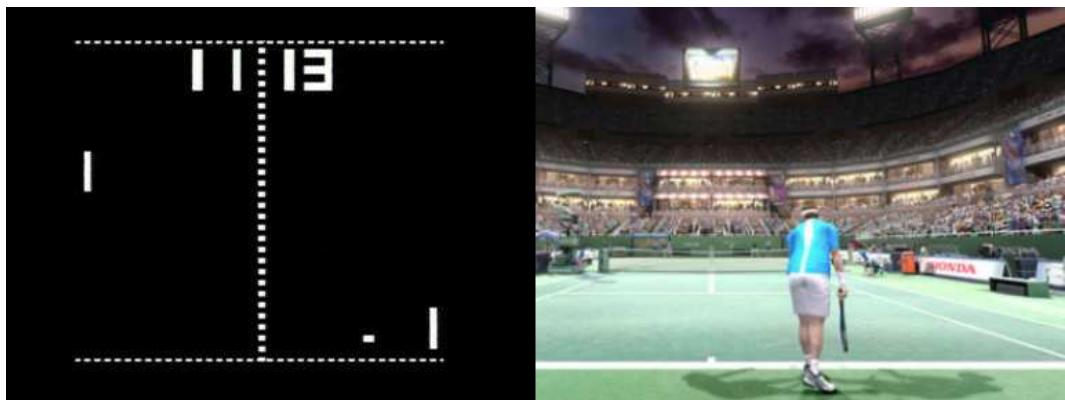


Figura 1.4: Pong vs Tenis 3D

La inteligencia artificial es el pan de cada día en el desarrollo de videojuegos. En cuanto tu aplicación alcance cierta envergadura tendrás la necesidad de incluir abundantes algoritmos de inteligencia artificial. Con estos algoritmos podremos configurar la dificultad del videojuego así como otros muchos aspectos. Según lo eficaz que sea el algoritmo, y el objetivo que queramos conseguir con él, podremos tener enemigos indestructibles o bien un videojuego asequible a casi cualquier jugador. Es importante no complicar el videojuego más allá de lo estrictamente necesario. Es fundamental que el jugador pueda superar los retos ya que si el juego es demasiado complicado el jugador se aburrirá y antes o después lo abandonará. Imagínate que estamos desarrollando un videojuego de carreras y siempre configuramos a uno de los rivales con el coche que puede alcanzar la mayor velocidad y además traza la carrera perfectamente en todas las vueltas. Será prácticamente imposible ganar a este rival lo que puede tener como consecuencia que el juego pierda

1. Desarrollando un videojuego

todo su interés.

Sería bueno, pero no es requisito, haber manejado alguna otra librería gráfica o al menos tener conocimientos básicos de programación gráfica. Si no es así no te preocunes, este tutorial está pensado para tí.

Como puedes ver hay que saber un poco de todo para desarrollar un videojuego. Por esto mismo, antes de empezar a desarrollar una aplicación de estas características, debes estar iniciado en el mundo de la informática y de la programación. Por ejemplo, si quieras programar un videojuego multi-jugador, seguramente quieras darle soporte de red. En este caso necesitarás conocimientos de redes. Es cierto que la librerías que vamos a manejar te facilitará el trabajo con la red pero si no tienes unos conocimientos básicos de redes no serás capaz de hacer que el videojuego se comporte como deseas.

Esto lo podemos hacer extensible a las distintas funcionalidades que ofrece SDL. Necesitamos unos conocimientos previos que cimenten el trabajo que vamos a realizar con los diferentes aspectos con los que nos permite interactuar la librería. SDL sólo es una herramienta para que pongas en práctica toda tu capacidad de creación.

1.3.4. APIs para el desarrollo de videojuegos

Una de las herramientas que especificamos como necesarias a la hora de desarrollar un videojuego es el uso de una librería que nos ayude en la tarea de desarrollar nuestro videojuego. Una API (*Application Programming Interface*) o interfaz de programación de aplicaciones es un conjunto de funciones que proporcionan un interfaz entre el programador y un determinado sistema. Este sistema puede ser de distintos niveles.

Nosotros hemos elegido para este tutorial la librería SDL. Esta librería nos proporciona diferentes APIs para trabajar con diferentes subsistemas que vamos a necesitar en el desarrollo de nuestro videojuego. Proporciona funciones y operaciones de dibujo en 2D, gestión de efectos, carga de imágenes, trabajo con sonido... y una lista considerable de librerías auxiliares que complementan a SDL como puede ser SDL _ net para el trabajo de comunicación dentro de las redes.

En definitiva un buen API nos facilitará el trabajo y nos permitirá avanzar más rápidamente en la construcción de nuestro videojuego. Existen otras alternativas como Allegro, Qt... En el siguiente capítulo existe un apartado dedicado a porque hemos elegido SDL y no otra librería.

1.3.5. Diseño multimedia

Un videojuego conlleva un importante proceso de diseño multimedia. No basta con conocimientos en informática, matemáticas, física, inteligencia artificial... sino que además hay que diseñar el videojuego, su idea, el desarrollo, los gráficos, el sonido...

Lo ideal sería tener una persona o un equipo de personas encargadas de cada una de estas tareas, pero no suele ser así, y aún menos cuando estamos empezando. En el momento en el que nos encontramos, que lo que queremos es realizar nuestras primeras aplicaciones, sería absurdo contratar a un equipo de profesionales que nos apoyase en nuestra tarea. Existen numerosas galerías en internet que te ofrecen contenido multimedia libre y gratuito del que puedes hacer uso para tus primeras aplicaciones. En nuestro caso vamos a realizar un videojuego en el que vamos a hacernos cargo de todo el proceso de desarrollo. Con esto buscamos poder comprender en profundidad el esfuerzo que suponer realizar un videojuego.

No está de más que seas capaz de dibujar tus propios personajes. Para esto se suele usar alguna herramienta de dibujo o de retoque fotográfico. Desde aquí te recomendamos *The Gimp*, que en la actualidad es uno de los programas de dibujo libres que están teniendo una mayor aceptación y desarrollo. Existen numerosos programas para realizar esta tarea y como en muchas ocasiones es cuestión de gusto y licencias. Normalmente cuando alguien empieza en el mundo del videojuego crea unas imágenes básicas con las que ir trabajando la lógica del videojuego que subyace bajo ellas. Cuando nos embarcamos en un proyecto de embergadura lo normal es recurrir a un diseñador gráfico para que nos cree unas imágenes con un aspecto más profesional y que serán las definitivas para nuestro videojuego.

De igual manera estarás interesado en que tu videojuego disponga de sonido y música. Tendrás que componer la banda sonora de tu videojuego así como los diferentes sonidos que se produzcan en él sino optas por hacer uso de bibliotecas libres. Un buen software para realizar esta tarea es *Audacity*, que como *Gimp*, es libre y gratuito.

Como puedes ver si estas sólo en el desarrollo del videojuego tienes que saber tocar todos los palos del contenido multimedia para las aplicaciones. Como en el desarrollo de cualquier aplicación la composición de un equipo o una comunidad ayuda mucho al desarrollo de la misma.

1. Desarrollando un videojuego

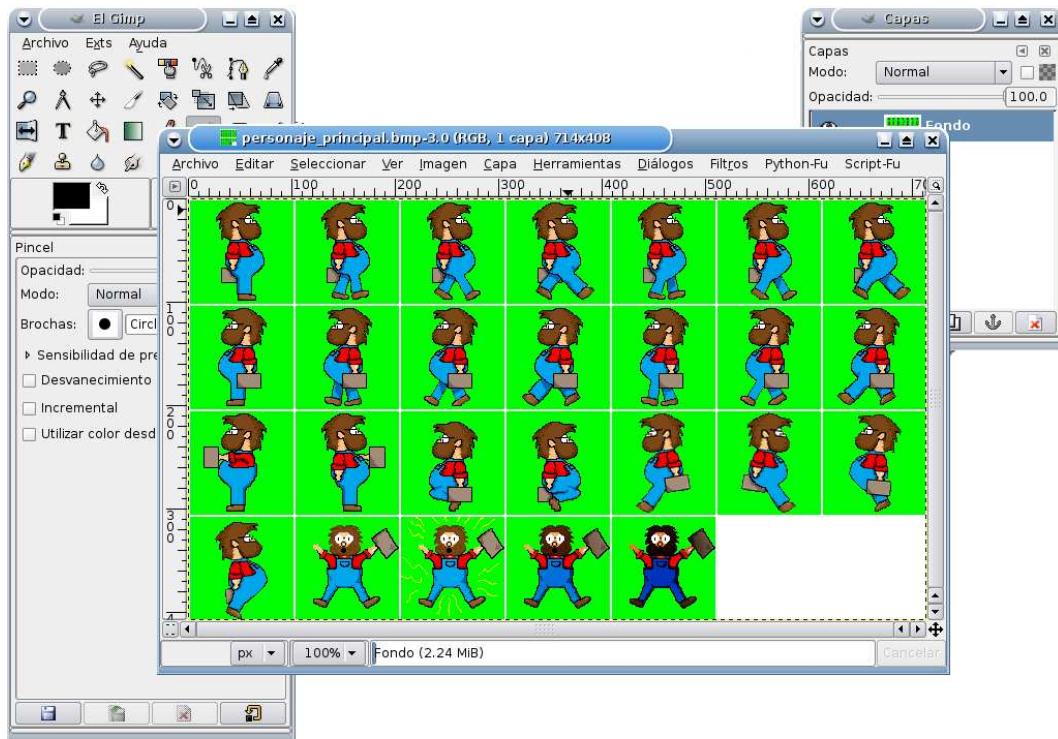


Figura 1.5: The Gimp

1.4. Anatomía de un Videojuego

1.4.1. Introducción

Cada vez que te sientas delante de un videojuego dentro del ordenador están ocurriendo un número impensable, a priori, de cosas. El proceso de mover un personaje conlleva un gran esfuerzo computacional del que solemos abstraer cuando estamos disfrutando del videojuego.

Es fundamental que el videojuego tenga una respuesta y una velocidad adecuadas. Estos parámetros y el de la dificultad definirán la jugabilidad de la aplicación. Todos los elementos de un videojuego paracen funcionar a la vez y tener vida propia pero esto es sólo una apariencia. Dentro del programa van ocurriendo cosas secuencialmente que consiguen que el juego se muestre de esta manera. Nuestro objetivo es conseguir una secuencialidad natural que hagan que el juego sea practicable.

Vamos a intentar dar en este apartado una visión general, sin entrar en detalles de implementación, de cuál debe ser la estructura de un videojuego, las partes que lo componen y qué funciones tienen.

1.4.2. Estructura de un videojuego

Como hemos comentado, para terminar con este capítulo y antes de ahondar en las características de SDL vamos a estudiar la estructura de un videojuego. Un videojuego es una aplicación diferente a los programas de gestión convencionales. Se aproxima más a procesos industriales o los que se implantan en buques o aviones para controlar su movimiento que a una aplicación de gestión.

Un videojuego debe de funcionar a tiempo real. El concepto de tiempo real es un concepto que tiende a confundirse. Cuando decimos que algo debe de funcionar a tiempo real es que debemos de obtener del sistema una respuesta en un tiempo determinado y establecido. Establecemos ciclos que no tienen por qué tener una frecuencia muy alta. Por ejemplo podemos establecer un sistema de tiempo real con intervalo de un minuto, de un milisecondo o de un año. Siempre que cumpla con la restricción de tiempo establecido estaremos hablando de un sistema en tiempo real.

El tener que satisfacer esta característica nos obligará a utilizar técnicas de programación que optimicen la respuesta que obtiene el usuario del sistema, complicando así aún más, si cabe, la elaboración del videojuego.

Durante la ejecución de la aplicación el juego estará realizando alguna tarea como la de dibujar objetos, calcular posiciones actualizando las coordenadas de los personajes, detectando colisiones, etc... todo esto continuamente independientemente de si recibimos “estímulo” por parte del usuario o no. Piénsalo bien, es totalmente lógico. Imagínate un videojuego que sólo reaccionase cuando producsemos alguna acción con el teclado o el ratón... en cuanto se nos vienen a la mente juegos complejos vemos como un *mundo* con “vida” propia envuelve a nuestro personaje, un mundo que sigue su curso pulsemos una tecla o no. Si has pensado en el clásico buscaminas, el solitario... comprenderás que la lógica que rodean a estos juegos no conllevan un trabajo adicional de la aplicación mientras que no reciban un estímulo.

Evidentemente también tenemos que estar receptivos a que se produzca alguna acción o evento del usuario y reaccionar dentro de un tiempo razonable para no mermar la jugabilidad.

Estas acciones se realizan en cada uno de los ciclos del videojuego o **game loops**. Un *game loop* es una estructura básica en forma de bucle. Este bucle se procesa varias veces por segundo en la mayoría de los casos lo que nos permite recalcular los diferentes aspectos del videojuego en un instante y que

1. Desarrollando un videojuego

el usuario no note la secuencialidad de las acciones. Este bucle contiene la esencia del juego y la convierte en la parte más importante de la ejecución del mismo.

Podemos dividir un videojuego en las siguientes partes:

1. Inicialización
2. Ciclo del videojuego o *game loop*
 - a) Gestión de la entrada
 - b) Procesamiento
 - c) Salida de datos
3. Finalización

Vamos a pasar a detallar cada una de estas partes.

Inicialización Es una parte fundamental del videojuego. Es primordial que todo lo que sea necesario durante el videojuego esté inicializado desde antes que sea utilizado. Como mínimo deberemos de inicializar al comienzo de la ejecución el subsistema de video que nos permita mostrar imágenes en la pantalla. Durante el **game loop** utilizaremos estructuras de datos que tendremos que previamente poner en marcha, librerías que configurar y lanzar, preparar el sonido del videojuego, el sistema de captura de acciones del usuario... Este es el momento de realizar todas esas tareas. No se nos debe olvidar que tenemos que establecer las posiciones iniciales de nuestros personajes en este punto ya que si no puede que aparezcan en posiciones ilegales y comenzar la aplicación con un error que posiblemente no permita su ejecución. En los diferentes apartados del tutorial referentes a los subsistemas presentaremos como inicializar los diferentes aspectos que componen el sistema.

Ciclo del videojuego o game loop es un bucle que se estará repitiendo durante el tiempo que dure el videojuego en ejecución. Normalmente este bucle se romperá cuando el usuario pulse una determinada tecla que provoque que el videojuego termine, o bien cuando el jugador pierda sus vidas y la partida haya terminado. Es el encargado de comandar en todo momento la tarea que se esté realizando. Podemos dividir el *game loop* en tres partes bien diferenciadas que tienen una fuerte interrelación entre sí:

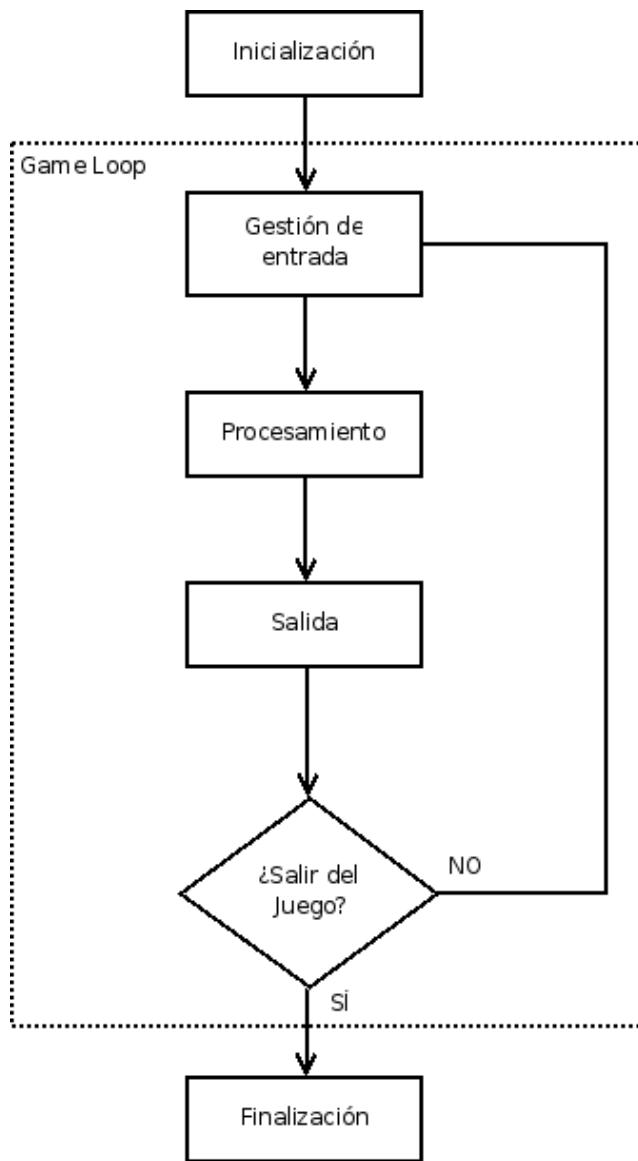


Figura 1.6: Estructura de un videojuego

Gestión de la entrada En este punto obtenemos las acciones que ha realizado el usuario sobre los dispositivos de entrada como el teclado, el ratón, el joystick... Sabremos, si se ha movido el ratón, las posiciones de origen y destino, que teclas presionó, cuales soltó... Un videojuego necesita comunicarse con el jugador mediante, al menos, un dispositivo de entrada. Este es uno de los aspectos fundamentales que tenemos que cuidar. En el capítulo dedicado a la gestión de eventos estudiaremos todos estos aspectos con detalle ya que es fundamental que la gestión de esta entrada no convierta en “injugable” a nuestra aplicación. Todo jugador necesita comunicarse con el videojuego y cada jugador tiene un dispositivo preferido para este fin,

1. Desarrollando un videojuego

por lo que es importante también el disponer de soporte para una variedad importante de este tipo de dispositivos. No nos cansaremos de repetir que cuidar la interacción del jugador con el videojuego supondrá gran parte del éxito o fracaso de nuestra aplicación.

Procesamiento En este momento reaccionaremos según haya sido el tipo de entrada dada por el jugador. Si recibimos algún tipo de entrada la procesaremos adecuadamente, tomando las decisiones según el tipo de acción que se haya realizado. Del mismo modo, y aunque no hayamos recibido entrada alguna, tendremos que procesar todo la lógica del videojuego así como los comportamientos que hayamos establecido. Por ejemplo si hemos dotado a nuestro juego de gravedad o inteligencia artificial este será el momento adecuado para aplicarla, al mismo tiempo que procesamos la entrada, mientras actualizamos lógica del videojuego... Podemos concretar como que el primer momento del procesamiento es el de reaccionar frente a la entrada producida mientras que el segundo momento de este punto es el de actualizar la lógica del videojuego.

En resumen, podemos decir que aquí reside el núcleo fundamental del videojuego y que su implementación es la que dotará a nuestro videojuego de una mayor o menor calidad.

Entre las acciones que podemos realizar respondiendo a los eventos de usuario está el procesamiento los aspectos referentes al sonido. Muchas veces no damos al sonido la importancia que se merece. Con un buen sonido y una banda sonora adecuada podemos conseguir hacer llegar al jugador sensaciones que no podemos conseguir con la simple imagen del juego. El aspecto más importante del procesamiento del sonido es la sincronización con lo que va ocurriendo en el videojuego. Un sonido a destiempo descoloca al jugador que sigue la acción del videojuego y puede ser contraproducente.

Para terminar esta referencia comentar uno de los aspectos que más adeptos va ganando. Se trata de los juegos en red. En este paso del procesamiento del videojuego también tendremos que tratar los aspectos referentes a las comunicaciones que, normalmente, se llevan a cabo mediante la programación de sockets.

La sincronización con el videojuego de los dos últimos aspectos es fundamental si queremos conseguir que estos añadidos de nuestro videojuego sean una aliciente y no un estorbo a la hora de seguir el videojuego.

Salida de datos No menos importante es la última parte del *game*

1.5. El desarrollo de la “idea”. Los tipos de juegos

loop. En este punto es donde refrescamos la pantalla con toda la información que ha sido procesada en los apartados anteriores. Mostramos los gráficos por pantalla, reproducimos los sonidos... en definitiva mostramos al usuario todo el proceso y los cálculos del paso anterior. En esta parte del ciclo del juego se realizan las operaciones de salida, tanto la gráfica, como la de sonido o la de red, estas últimas sincronizadas en el apartado anterior.

Un buen motor gráfico es fundamental. Necesitamos que el motor que soporte nuestras imágenes sea lo suficientemente potente para trabajar con nuestra aplicación. En nuestro caso SDL será más que suficiente para mostrar las imágenes así como para trabajar con los distintos aspectos que vayamos a incluir en nuestro videojuego como el sonido o el trabajo en red.

Finalización Una vez terminado el videojuego debemos de liberar todo aquellos recursos que reservamos en el primer apartado o en el transcurso del juego. Las tareas a realizar en este apartado son, básicamente, las opuestas a lo que hacíamos en el proceso de inicialización. Piensa que las imágenes, sonidos, música y demás que almacena un videojuego en memoria, si no son liberados, puede perjudicar seriamente el rendimiento del mismo para otras tareas.

1.5. El desarrollo de la “idea”. Los tipos de juegos

No sé si te habrá lo habrás notado alguna vez pero en muchas películas americanas de terror adolescente da la sensación de que el final de la película ha sido, o lo ha parecido al menos, improvisado. Después de dos horas en las que el ambiente creado en la película nos ha hecho dar algún salto que otro, que el malo haya resucitado unas diez o quince veces, todo termina de buenas a primeras.

Seguramente si recuerdas alguna película con algo más de nombre que estas “americanadas” será imposible pensar en una sensación de este estilo ya que el guión de éstas suele ser estudiado y elaborado cuidando hasta el más mínimo detalle.

Algo parecido pasa a la hora de desarrollar un videojuego. Un buen guión, una buena idea, una historia bien elaborada es fundamental para enganchar al jugador en nuestra aventura. En este aspecto prima más la originalidad del que elabora la historia ya que no existe una técnica bien definida que nos

1. Desarrollando un videojuego

proporcione un buen relato automáticamente.

El primer paso que tenemos que dar a la hora de desarrollar un videojuego es plantearnos que tipo de juego queremos a elaborar. De este planteamiento surgirán las necesidades que vamos a tener que satisfacer. Supongamos el caso que queremos desarrollar una aventura gráfica. Una aventura gráfica es un juego donde se desarrolla una historia donde el jugador tiene que ir encontrando pistas que descubren misterios que nos permiten avanzar en la historia. Este tipo de jugos depende muchas veces de pruebas de inteligencia o resolución de puzzles para avanzar, por lo que es fundamental el diseño de este tipo de enigma, de los dialogos, en definitiva el guión deben estar muy logrados para mantener la atención del usuario. Como puedes ver si nos encontramos en este caso deberemos de elaborar una historia bien fundada con diferentes caminos a seguir, pistas... una trama en toda regla. Una vez planteado necesitaremos crear los personajes, sus movimientos, saber que acciones van a realizar y como van a interactuar con el jugador. Con todo esto, y sabiendo la lógica que va a seguir el videojuego, deberemos de realizar el análisis software para un posterior diseño que implementaremos según el tipo de dispositivo o plataforma en el que vayamos a ejecutar nuestra aplicación. Por si no conoces bien este tipo de videojuegos algunos ejemplos históricos son Monkey Island o el Broken Sword.

Si decidimos desarrollar un juego arcade seguramente también necesitaremos una historia, pero no hará falta que entremos en tanto detalle como en el caso anterior. Son los primeros juegos que aparecieron en el mercado del videojuego. Se basan en medir la destreza del jugador en diferentes niveles. Uno de los aspectos fundamentales de este tipo de juegos es la rapidez, incluso más que la propia estrategia del juego. Estos juegos se suelen subdividir en niveles que el jugador debe superar cumpliendo unos ciertos objetivos. Desde el inicio se les dotó de una historia que acompañaba a la acción del juego para darle un poco más de sentido a todo lo que estaba ocurriendo en nuestra pantalla. Un tipo de juego arcade son los juegos de acción utilizando esta denominación a juegos digamos algo más violentos. Otro tipo de juego arcade muy popular son los juegos de plataformas, que no son mas que juegos que tenemos que superar dando saltos entre “plataformas” consiguiendo ciertos ítems que nos dan puntos o bien tenemos que acabar con los enemigos. Ejemplos archiconocidos de estos tipos de juegos son el Sonic, SuperMario Bros, Prince of Persia... la lista puede ser interminable. Actualmente el sector de los videojuegos arcade está copado con los juegos en tres dimensiones. Resumiendo, los tipos de arcade se centran en la acción que se está realizando más que en la historia que se desarrolla en el juego. En proyectos de gran envergadura se cuenta con un equipo de guionistas. Como seguramente no sea este nuestro caso este tipo de juego es uno de los más asequibles de realizar a la hora de introducirnos en el desarrollo de videojuegos.

1.5. El desarrollo de la “idea”. Los tipos de juegos



Figura 1.7: Juego arcade: Tux

Otro tipo muy popular de videojuegos es el de simulación. Estos juegos sumergen al usuario en algún tipo de acción que no podría realizar si no es con este simulador. Por ejemplo si jugamos a un simulador de vuelo podremos pilotar un avión dentro de un entorno que pretende que la acción sea lo más real posible. Hay muchos tipos de simuladores como deportivos, de conducción... Este tipo de juegos está consiguiendo un realismo que lleva a que sus nuevos usuarios necesiten un mayor periodo de aprendizaje debido a la complejidad que llegan a tener estos simuladores. En cuanto a los simuladores deportivos son juegos que como puedes intuir nos sumergen en un deporte determinado. Suelen ser juegos para disfrutar en grupo y requieren una gran habilidad, rapidez y precisión. El jugador tiene control total sobre el jugador o jugadores que maneje. Los algoritmos que dotan de inteligencia a los rivales son fundamentales en este tipo de juegos ya que proporcionaran que el juego sea más o menos complicado. Es importante que un juego no sea demasiado complicado ya que esto puede degenerar en que sea desesperante y que el jugador lo abandone. Estos tipos de juegos no son adecuados como un primer proyecto por su complejidad. Ejemplos de estos tipos de juegos son el Flight Simulator o el popular FIFA.

Podemos agrupar en otra categoría a los juegos de estrategia e inteligencia. Los juegos de estrategia se llaman así porque coordinan acciones con una finalidad concreta. El jugador debe de pensar la estrategia más adecuada para conseguir su objetivo, normalmente, con el menor esfuerzo posible. En la mayoría de los juegos de estrategia actuales podemos manejar más de un personaje y otras tantas actividades como manejar tropas de ataque que nos ayuden a conseguir nuestro objetivo. Dentro del tipo de juegos de inteligencia clasificamos también a aquellos juegos que podríamos definir como algo más estáticos pero que suelen ocupar horas de diversión. Se trata

1. Desarrollando un videojuego



Figura 1.8: Juego de simulación

de juegos como Sokoban, Sudoku, el mismo ajedrez o el famoso Tetris. Estos son juegos de unas características que nos permiten, si tomamos de algún libro la lógica del juego, que practiquemos el desarrollo de un videojuego sin una complejidad excesiva ya que se trataría de implementar dicha lógica y proporcionar al juego de una interfaz adecuada. Dentro de esta categoría podemos englobar también a los juegos de mesa y a los juegos de azar ya que desde el punto de vista del desarrollo tienen unas características parecidas.



Figura 1.9: Juego de estrategia

Algunos autores engloban a los juegos de rol dentro de los de estrategia. En un juego de rol manejamos un personaje creado con unas características concretas que va evolucionando según las decisiones o caminos que tome el usuario. Suelen ser juegos en las que se invierten muchas horas como los de estrategia. En estos juegos suelen existir varios objetivos que se entrelazan.

Actualmente con la posibilidad de jugar a este tipo de videojuegos en red está aumentando su popularidad. Esto permite que el mundo en el que nos movemos sea innovador e imprevisible por lo que hace al juego mucho más interesante. Debido a la inteligencia y complejidad de algoritmos que soportan este tipo de juegos tampoco los recomendamos para adentrarnos en el mundo de la programación de videojuegos.

El proceso de desarrollo de un videojuego suele ser incremental. De unas primeras versiones funcionales vamos añadiendo características que van complementando el videojuego. Tienes a tu disposición un capítulo donde se realiza paso a paso el desarrollo de un videojuego que puede servirte de guía a la hora de elaborar tus proyectos.

1.6. Resumen

Para desarrollar un videojuego hay que tener conocimiento de todas aquellas materias que queramos aportar a nuestro videojuego. Existen numerosas herramientas que nos van a facilitar el trabajo del desarrollo del mismo pero que no pueden sustituir a los conocimientos básicos que debemos de tener de las diferentes materias. No sólo la informática abarca el desarrollo de videojuegos, la creatividad y las ciencias son partes fundamentales del mismo.

La estructura de un videojuego responde a una estructura básica en forma de bucle. En cada una de las partes que se divide esta estructura tiene encomendadas unas tareas bien definidas. Aunque todos los pasos son importantes es fundamental cuidar la implementación del procesamiento de la información ya que de éste dependerá gran parte del éxito de nuestro videojuego.

Existen numerosos tipos de videojuegos. Para comenzar en el desarrollo de videojuegos es recomendable hacerlo por aquel tipo que añada menos trabajo adicional al propio hecho de implementar el videojuego en un sistema concreto. Esto nos permitirá obtener una base con la que trabajar para luego tener una vista más amplia del desarrollo de videojuego y nuestras limitaciones en este campo. No es recomendable empezar con proyectos demasiado ambiciosos ya que, seguramente, nos llevarán al fracaso y el abandono de este interesante mundo.

1. Desarrollando un videojuego

Capítulo 2

Conociendo libSDL

2.1. Introducción

Tal vez te pase como a mí y cuando lleves unos años programando sientes que te faltan herramientas para darle un aspecto visual a tus programas. Has probado varios videojuegos y no te sientes capaz de desarrollar uno con los conocimientos que tienes de programación. Un buen día me propuse a desarrollar un pequeño videojuego con el fin de adornar el resultado presentado por un autómata para una asignatura de la facultad y se abrió ante mí un mundo desconocido.

En ese momento comenzó mi aventura. Dediqué algún tiempo a buscar una buena biblioteca de desarrollo que me permitiese, sin invertir años de sacrificio, dar rienda suelta a mi imaginación. Era fundamental que no me limitase en temas como la licencia o la capacidad de la misma. Necesitaba que esta herramienta fuese gratuita y libre. Como principiante quería saber como funcionaba todo y no quería que mi trabajo en el mundo de los videojuegos comenzase teniendo que piratear esto o aquello. Una de las grandes aportaciones de mis profesores en los años de facultad fue una filosofía de trabajo legal enfrentada a “lo que se lleva” en el mundo de la informática de no prestar atención a los términos de las licencias.

Por esto, y muchos más motivos que conocerás a continuación, decidí elegir libSDL. SDL nos va a ahorrar una gran cantidad de trabajo y dolores de cabeza. Cada una de las partes que componen SDL necesitarían un tutorial para explicarlas con detalle. Nuestro objetivo es una visión más general que te permita adentrarte en el mundo del desarrollo de videojuegos.

2.2. Objetivos

El objetivo principal de este capítulo es que conozcas la librería SDL y las posibilidades que nos ofrece. Así mismo debes conocer por qué se ha elegido

2. Conociendo libSDL

esta librería y no otra.

Otro aspecto fundamental, incluso más importante que el primero, es conocer las desventajas de SDL ya que serán las limitaciones que tengamos a la hora de trabajar con esta librería. Debes saber hasta dónde puedes llegar.

2.3. ¿Qué es libSDL?

libSDL es el acrónimo de *library Simple Directmedia Layer*. *libSDL* fue creada por Sam Lantinga y su grupo, programadores de Loki Entertainment Software, para portar juegos a Linux principalmente. Sam Lantinga es un programador de amplia experiencia en el mundo de videojuegos lo que le ha permitido tener una visión específica de las necesidades de los creadores de videojuegos.

De aquí en adelante, y por economía lingüística, llamaremos a esta librería o biblioteca SDL que es como se la conoce. Del mismo modo utilizaremos el término librería como sinónimo de biblioteca aunque algunos puristas del lenguaje no estén de acuerdo con esta terminología.

SDL es una librería multimedia multiplataforma, es decir, todas las aplicaciones que desarrollemos con esta librería pueden ser compiladas, sin cambiar nada en el código, en varios sistemas diferentes como Windows, Linux, BeOS... Está diseñada para proporcionar acceso de bajo nivel al audio, teclado, ratón, joystick y a los dispositivos de video 2D. Es una librería muy utilizada en diferentes tipos de aplicaciones entre ellas se encuentra software para reproducir video MPG, emuladores y un gran número juegos y adaptaciones de juegos entre plataformas.

Podemos marcar el comienzo de los ordenadores personales allá por 1982 cuando IBM comercializó la primeras computadoras equipados con PC-DOS, versión del MS-DOS de Microsoft. En los tiempos de este sistema operativo la programación de videojuegos era muy compleja. Había que conocer en profundidad el lenguaje ensamblador y la estructura interna de la máquina, así como los elementos multimedia que pudieran poseer, como la tarjeta de video o de sonido. La principal complejidad es que la programación se hacía casi directamente sobre el hardware con todos los inconvenientes de abstracción, complejidad y portabilidad que supone esto. Resumiendo, para hacer un videojuego, fuese lo simple que fuese, hacía falta un grandísimo esfuerzo.

La aparición del sistema operativo Windows y sus versiones no terminaron de mejorar el panorama debido a un rendimiento gráfico pobre y la imposibilidad de tener un acceso directo al hardware en mucha de sus versiones.

2.3. ¿Qué es libSDL?

Además el acceso a los recursos del sistema varía de una versión a otra, de un sistema operativo a otro. Necesitamos una interfaz común para el desarrollo de videojuegos que nos permita establecernos en un entorno más “estandarizado” y con una interfaz común a los recursos del sistema. En definitiva se hizo fundamental tener una herramienta que nos proporcionase una capa de abstracción que nos permitiese dedicarnos al desarrollo del videojuego en sí liberándonos hasta cierto punto de los aspectos concretos del sistema donde fuese a ejecutarse dicha aplicación.

La empresa Loki Games salió al rescate, más concretamente un experimentado programador llamado Sam Lantinga. Ante la necesidad de realizar juegos que fuesen portables entre distintos sistemas crearon una librería multiplataforma llamada SDL, que nos proporciona acceso a los recursos del sistema donde se haga correr la aplicación mediante una interfaz independiente del sistema para el que vayamos a realizar la aplicación.

SDL soporta Linux, Windows, Windows CE, BeOS, MacOS, MacOS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX. El código de la librería nos permite trabajar en aplicaciones para AmigaOS, Dreamcast, Atari, AIX, OSF/Tru64, RISC OS, SymbianOS y OS/2 pero no estos sistemas no son soportados oficialmente.



Figura 2.1: Logotipo SDL

SDL está escrita en C pero trabaja nativamente con C++. Se puede utilizar en otros muchos lenguajes ya que existen adaptaciones que permiten hacerlo. Alguno de los lenguajes en los que se puede utilizar son Ada, C#, Eiffel, Erlang, Euphoria, Guile, Haskell, Java, Lisp, Lua, ML, Objective C, Pascal, Perl, PHP, Pico, Python, Ruby y Smalltalk.

Es importante conocer que SDL es distribuido bajo licencia GNU LGPL en su versión 2. Esta licencia permite usar SDL libremente, en programas comerciales, siempre que utilicemos un enlace dinámico a esta librería y cumplamos todos los requisitos de dicha licencia.

2. Conociendo libSDL

2.4. ¿Qué nos proporciona SDL?

SDL nos proporciona una capa de abstracción con todo lo necesario para crear aplicaciones multimedia y videojuegos. Está compuesta por subsistemas como veremos en el transcurso del tutorial. Estos subsistemas son sostenidos por diferentes APIs que nos proporcionan acceso a las diferentes partes hardware.

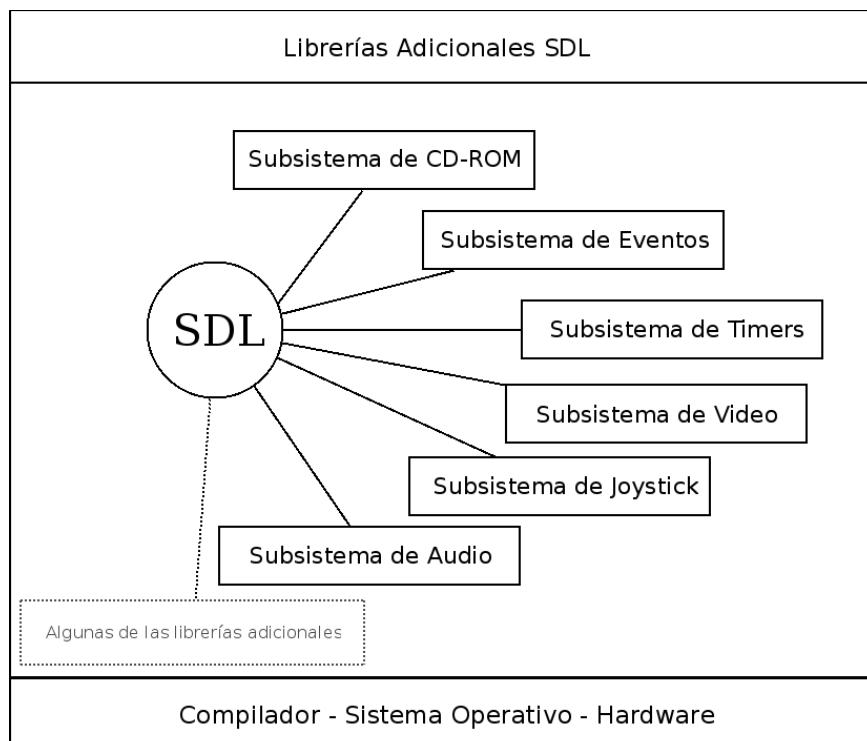


Figura 2.2: Composición de SDL

Vamos a presentar ahora algunos aspectos relevantes de los subsistemas necesarios para desarrollar un videojuego y su relación con SDL.

2.4.1. Vídeo y Gráficos

SDL proporciona una API que nos permite trabajar con los gráficos de una manera muy versátil. Podemos trabajar con píxeles en crudo, directamente, construyendo líneas y polígonos y hacer lo que queramos con ellos o bien cargar imágenes que rellenen píxeles facilitando así el trabajo de crear figuras o personajes en nuestra aplicación.

SDL se encarga de configurar los modos de video que demandemos, el modo de acceso a memoria de video... nos proporciona modos aventanados o a

2.4. ¿Qué nos proporciona SDL?

pantalla completa... Podemos escribir directamente en la memoria de video en su framebuffer. Crear superficies con canales alpha y colores clave. Volcar las superficies convirtiendo los formatos automáticamente al de destino. Permite realizar acciones aceleradas por hardware siempre que estén soportadas por nuestro sistema y muchas otras posibilidades que te presentaremos en el capítulo dedicado al subsistema de video.

No te preocupes si no entiendes ahora alguno de estos conceptos, tendremos un capítulo entero para estudiar el subsistema de vídeo.

2.4.2. Eventos de Entrada

SDL proporciona entrada por teclado, ratón y joystick usando un modelo de entrada basado en eventos parecido al utilizado en el desarrollo de aplicaciones X11, Windows o Mac OS. La ventaja es que SDL nos proporciona una abstracción de éstos podiéndolos utilizar sin preocuparnos de en qué sistema operativo estemos desarrollando nuestra aplicación. Podremos olvidarnos de los eventos específicos producidos por cada sistema operativo que complican la tarea de escribir el código de la aplicación y anulan la posibilidad de portabilidad.

Para la gestión de la entrada de usuario SDL también nos proporciona la posibilidad de conocer el estado de un dispositivo de entrada en un momento dado sin que éste tenga que haber realizado ninguna acción. Por ejemplo podemos consultar el estado de una determinada tecla o de cierto botón del ratón cuando creamos conveniente. Este método de consulta de estados puede sernos útil a la hora de desarrollar nuestra aplicación ya que nos permite dictaminar cuándo queremos consultar el estado de los dispositivos de entrada en contraposición a la gestión de la cola de eventos.

Existe en el tutorial un extenso capítulo dedicado a la gestión de entrada en SDL.

2.4.3. Sonido

Este es el aspecto más débil de SDL. La librería proporciona un API simple para averiguar las capacidades que nos proporciona la tarjeta de sonido y lo que se necesita saber para reproducir un sonido. Podemos iniciar el subsistema de sonido a 8 o 16 bits, mono o estéreo. El soporte para reproducir un sonido real es ínfimo. Para la tarea de reproducir sonidos deberemos de utilizar librerías auxiliares que nos proporcionen una manera más comoda de trabajar sino queremos que este aspecto ocupe la mayor parte del tiempo de desarrollo de nuestro videojuego.

2. Conociendo libSDL

El motivo de que nativamente no se proporcione una API será estudiado en profundidad en el capítulo dedicado al estudio del subsistema de sonido.

2.4.4. Manejo del CDROM

SDL provee un completo API para leer y reproducir pistas en un CD-ROM. Es muy completo y no echarás en falta ninguna función para hacer uso del mismo.

2.4.5. Timers

Depende de la máquina en que ejecutemos nuestra aplicación irá, a priorio, más rápida o más lenta. Esto en aplicaciones de gestión no es algo que debamos controlar explícitamente, ya que cuanto más rápido se ejecute mejor respuesta tiene el usuario en los casos más comunes. En la programación de videojuegos la cosa cambia. El tiempo es un aspecto crítico que deberemos controlar en todo momento.

Los timers son un tipo de variable que nos permiten conocer una marca de tiempo dentro de una aplicación. Nos servirán de ayudara para realizar esta tarea de control. Debemos de conseguir que no por ejecutar el videojuego en una máquina mas potente o más moderna el videojuego se convierta en un corre-calles sin sentido. Para esto sirven los timers. Con ellos podemos controlar los tiempos en el videojuego y así poder independizar el comportamiento de la aplicación de la máquina en la que se esté ejecutando.

SDL proporciona una API sencilla, limpia y confiable que es independiente de la máquina y del sistema operativo. Los timers de SDL trabajan a una resolución de 10 milisegundos, resolución más que suficiente para controlar los tiempos de un videojuego.

2.4.6. Gestión de dispositivos

Normalmente los sistemas operativos necesitan que el programador configure aspectos sobre los dispositivos que se van a utilizar e inicializar dicho dispositivo antes de su uso. Primero deberá de consultar al sistema sobre que puede hacer con el hardware, en qué te limita el sistema operativo y qué te permite el hardware en cuestión.

Una vez utilizado el dispositivo tenemos que decirle al sistema operativo que hemos terminado de utilizarlo para dejarlo libre por si otra aplicación quiere utilizarlo. Todo este proceso y su codificación es casi igual para todos los proyectos que vayamos a realizar.

SDL nos proporciona una manera simple de descubrir que es lo que el hardware puede hacer y luego, con un par de líneas de código, podemos dejarlo todo listo para utilizar el dispositivo. Para liberar o cerrar el dispositivo las funciones que proporciona SDL son aún más simples. Además como estas líneas de código suelen ser comunes de un proyecto a otro podremos reutilizarlas.

En resumen, SDL facilita la tarea de tomar un dispositivo para poder utilizarlo y simplifica la tarea de liberarlo.

2.4.7. Red

SDL proporciona una API no nativa para trabajar con redes a bajo nivel. Esta librería adicional nos permite enviar paquetes de distinto tipo sobre el protocolo IP. Permite iniciar y controlar sockets de los tipos TCP y UDP. Esta API sólo permite controlar los aspectos que son comunes a la implementación de estos sockets en los distintos sistemas operativos que soporta. Lo mejor de esta API es que SDL se ocupa de muchas tareas, consideradas molestas, de bajo nivel que hacen preparar y manejar las conexiones sea tedioso.

El soporte de red está catalogado dentro de las librerías conocidas como adicionales en SDL. Éstas, normalmente, no son desarrolladas originalmente por Loki Games y son fruto del código abierto de SDL. En el tutorial podrás disfrutar de un capítulo dedicado exclusivamente a las librerías adicionales más importantes de SDL.

2.5. ¿Por qué SDL?

El principal motivo para elegir SDL es que es la mejor API para desarrollo de videojuegos disponible para Linux además de ser multiplataforma.

Uno de los aspectos importantes por los que hemos dirigido nuestras miradas a esta librería es su licencia. Nos permite desarrollar aplicaciones sin tener que estar pendientes de los aspectos legales que envuelven a la biblioteca. Ésta es una librería libre y tenemos la oportunidad de saber como se han implementado los diferentes aspectos de la misma lo que le proporciona, además, un valor añadido sobre nuestro aprendizaje. Así mismo, el ser libre y por su licencia, propicia un crecimiento en comunidad lo que se refleja en las características que esta librería posee.

Que la licencia sea un tema importante no quiere decir que no sea una librería potente. En los últimos años ha sido premiada varias veces como la mejor librería libre para el desarrollo de videojuegos lo que pone de manifiesto la importancia de esta librería en el mundo “libre”.

2. Conociendo libSDL

La librería SDL es multiplataforma. El fabricante nos garantiza que las aplicaciones que desarrollemos con ella funcionarán perfectamente bajo Linux, Microsoft Windows (C), BeOS... y una larga lista que ya conoces. Para que esto sea cierto tenemos que tener en cuenta algunas peculiaridades y mantener una metodología de la programación que no incurra en crear un software dependiente del sistema en el que estemos implementando dicha aplicación. De nada vale que SDL sea multiplataforma si a la hora de implementar la aplicación usamos técnicas, funciones o constantes que dependen del sistema.

En la actualidad podemos utilizar SDL para programar videojuegos en distintos tipos de dispositivos, con diferentes arquitecturas. Como puedes ver esto aumenta nuestro rango de acción. No sería lógico centrarnos en una sola plataforma cuando el mundo actual cada vez se diversifica más el universo multimedia. Podemos desarrollar aplicaciones que se ejecuten en un PC con Linux exactamente como en un MacOS si esfuerzo alguno.

Resumiendo, utilizando esta librería conseguimos no atar el desarrollo de nuestra aplicación a ningún sistema. Nos permite coger un programa que desarrollamos para Linux y compilarlo para Mac, Palms... y viceversa.

Esta librería nos provee de lo necesario para trabajar con los diferentes subsistemas “ocultándonos” miles de detalles que sólo conseguirían complicar el desarrollo de nuestra aplicación. Depende del sistema operativo que estemos corriendo en nuestro ordenador SDL se sustentará sobre una parte del mismo u otro. Por ejemplo en Microsoft Windows (C) SDL se apoya sobre DirectX, en Linux sobre X y bibliotecas auxiliares... y así en cada uno de los sistemas. Este proceso es totalmente transparente al usuario de la librería, que en este caso somos nosotros.

SDL, a diferencia de otras librerías libres para el desarrollo de videojuegos que navegan por la red, fue diseñada e implementada por un grupo de programadores de juegos experimentados y altamente cualificados. Es decir, SDL fue escrita por programadores de videojuegos para programadores de videojuegos. Una de las cualidades de SDL es su, relativamente, poca complejidad. Qué quiero decir con esto... ¿Acaso SDL lo hará todo? Ni eso, ni todo lo contrario. SDL nos proporciona las herramientas básicas sobre todo lo que tiene que hacer la librería para funcionar en cualquier máquina para programar un videojuego. Lo demás lo define uno de los autores de la biblioteca como “libre albedrío”. Tenemos las herramientas ahora el límite lo ponemos nosotros. Este razonamiento no excluye el compromiso de ser una librería suficientemente completa.

En la red podemos encontrar baterías impresionantes de grupos de pruebas que han intentado encontrar *bugs* en esta librería sin obtener, normalmente,



Figura 2.3: Descent 2. Juego portado mediante SDL



Figura 2.4: OpenGL

un resultado satisfactorio. Es una librería realmente estable. Claro está, como ya hemos comentado, una de las principales ventajas que proporciona el que sea libre es que los usuarios de la misma pueden informar de los fallos y proponer soluciones a estos fallos.

Dos de las grandes alternativas a SDL para realizar tareas parecidas son DirectX y OpenGL. No son soluciones definitivas ya que presentan diferentes problemas.

OpenGL está especializada en gráficos 3D y no proporciona acceso a otros recursos del sistema, como puede ser el teclado o el joystick, fundamentales en la programación de videojuegos. DirectX es bastante más completo ya que proporciona gráficos en 2D y 3D, entrada y salida, comunicación entre redes... el problema es que sólo es compatible con sistemas Windows además de que su uso en la programación de videojuegos es algo engorrosa si nunca se ha trabajado con las APIs de Microsoft Windows ©.

2. Conociendo libSDL

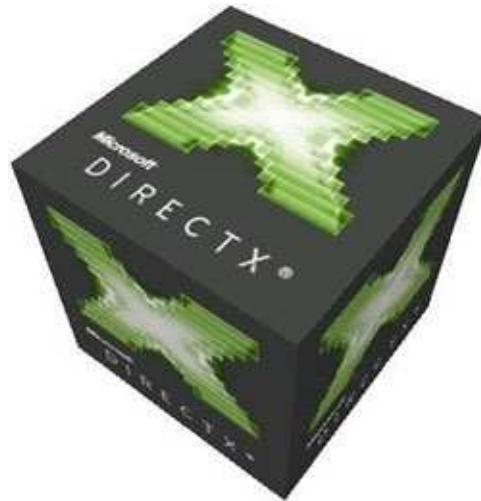


Figura 2.5: DirectX©

Por todo esto ninguna de estas dos opciones son válidas para introducirnos en el mundo de los videojuegos. Existen otras alternativas libres para iniciarse en la programación gráfica pero nos decantamos por SDL por ser la mejor de estas alternativas.

Las ventajas de SDL pueden agruparse en:

Estabilidad Una de las condiciones de la licencia de SDL es que no puede ser utilizado por empresas o particulares sin dar soporte al desarrollo de la API. Con esto se consigue la corrección de errores y la aportación de mejoras que favorecen la robustez de la API. El desarrollo de SDL es incremental de forma que se congela una versión estable mientras que se crea una versión nueva con todas las características aportadas con el fin de ser testeadas y aprobadas para una versión posterior que será considerada estable.

Simplicidad SDL ha sido diseñada para tener un API simple, que permita que haya la menor cantidad de código entre lo que quieras hacer y el resultado final. A continuación tienes un ejemplo de esta simplicidad. Este código es de una migración de código de demos comerciales del grupo *Optimum* de X11 a código SDL. Este es el código de X11:

```
;  
1 ;// Código X11  
2 ;  
3 ;int init_x (int X, int Y, int W, int H, int bpp, const char *Name)  
4 ;{  
5 ;    XPixmapFormatValues *formatList;  
6 ;    int formatCount;
```

```

7 ;     int i;
8 ;     int formatOk;
9 ;     int scanlineLength;
10 ;    XGCValues gcVal;
11 ;    unsigned long gcMask;
12 ;
13 ;    dis = XOpenDisplay ( NULL );
14 ;
15 ;    if ( dis == NULL) {
16 ;        fprintf ( stderr , "Error :\n" );
17 ;        fprintf ( stderr , " Cannot connect to Display.\n");
18 ;        exit ( 1 );
19 ;
20 ;
21 ;    screen = DefaultScreen ( dis );
22 ;    depth = DefaultDepth ( dis , screen );
23 ;    width = DisplayWidth ( dis , screen );
24 ;    height = DisplayHeight ( dis , screen );
25 ;
26 ;    winRoot = DefaultRootWindow ( dis );
27 ;    winAttr.border_pixel = BlackPixel ( dis , screen );
28 ;    winAttr.background_pixel = BlackPixel ( dis , screen );
29 ;    winMask = CWBackPixel | CWPBorderPixel;
30 ;
31 ;    formatList = XListPixmapFormats( dis, &formatCount);
32 ;
33 ;    if (formatList == NULL){
34 ;        fprintf ( stderr , " Cannot get pixmap list\n");
35 ;        exit ( 1 );
36 ;
37 ;
38 ;    formatOk=-1;
39 ;    for (i=0; i<formatCount; i++)
40 ;        if ( formatList[i].depth == depth)
41 ;            formatOk=i;
42 ;
43 ;
44 ;    if (formatOk == -1){
45 ;        fprintf ( stderr , " PROUT\n");
46 ;        exit(1);
47 ;
48 ;
49 ;    memcpy(&pixmapFormat,&formatList[formatOk] , sizeof (pixmapFormat));
50 ;    XFree(formatList);x
51 ;
52 ;    /* taille utile */
53 ;    scanlineLength = pixmapFormat.bits_per_pixel * W /8;
54 ;    /* padding eventuel */
55 ;
56 ;    if ( (scanlineLength & (pixmapFormat.scanline_pad/8 -1)) != 0){
57 ;        scanlineLength&=~(pixmapFormat.scanline_pad/8 -1);
58 ;        scanlineLength+=pixmapFormat.scanline_pad/8;
59 ;    }

```

2. Conociendo libSDL

```
60 ;     win = XCreateWindow ( dis , winRoot , X , Y , W , H , 0 , depth ,
61 ;                               InputOutput , CopyFromParent ,
62 ;                               winMask , &winAttr );
63 ;     XStoreName ( dis , win , Name );
64 ;     XSelectInput ( dis , win , KeyPressMask );
65 ;     winHint.flags = PPosition | PMinSize | PMaxSize ;
66 ;     winHint.x = X;
67 ;     winHint.y = Y;
68 ;     winHint.max_width = winHint.min_width = W;
69 ;     winHint.max_height = winHint.min_height = H;
70 ;     XSetWMNormalHints ( dis , win , &winHint );
71 ;     XCLEARWINDOW ( dis , win );
72 ;     XMapRaised ( dis , win );
73 ;     XFlush ( dis );
74 ;
75 ;     depth = DefaultDepth ( dis , screen );
76 ;     width = W;
77 ;     height = H;
78 ;
79 ;#ifdef USE_SHM
80 ;/* SHM */
81 ;xim = XShmCreateImage(dis,CopyFromParent,depth,
82 ;                      ZPixmap,0,&SHMInfo,W,H);
83 ;
84 ;if (xim == NULL){
85 ;    fprintf(stderr, " Couldnt create Ximage..\n");
86 ;    exit(-1);
87 ;}
88 ;
89 ;SHMInfo.shmid =
90 ;    shmget(IPC_PRIVATE, xim->bytes_per_line*xim->height,
91 ;           IPC_CREAT|0777);
92 ;xim->data = SHMInfo.shmaddr = (char *)shmat(SHMInfo.shmid, 0, 0);
93 ;SHMInfo.readOnly = False;
94 ;XShmAttach(dis, &SHMInfo);
95 ;XSync(dis, False);
96 ;buffer=(unsigned char *)xim->data;
97 ;
98 ;#else
99 ;
100 ;buffer =
101 ;    (unsigned char *)calloc(W*H, pixmapFormat.bits_per_pixel/8);
102 ;xim = XCreateImage ( dis , CopyFromParent , depth , ZPixmap , 0 ,
103 ;                     (char *) buffer , W , H ,
104 ;                     pixmapFormat.scanline_pad, scanlineLength);
105 ;
106 ;if (xim == NULL){
107 ;    fprintf(stderr, " Couldnt create Ximage..\n");
108 ;    exit(-1);
109 ;}
110 ;
111 ;#endif
112 ;
```

2.5. ¿Por qué SDL?

```
113 ;     gcVal.foreground = 0;
114 ;     gcVal.background = 0;
115 ;
116 ;     gcMask = GCForeground | GCBackground;
117 ;     gc = XCreateGC ( dis , win , gcMask , &gcVal );
118 ;
119 ;     if (depth==24)
120 ;         depth = pixmapFormat.bits_per_pixel;
121 ;
122 ;     return (depth);
123 ;}
```

Este es el código migrado a SDL:

```
;-----  
1 ;// Código de X11 migrado a SDL  
2 ;  
3 ;  
4 ;int init_x (int X, int Y, int W, int H, int bpp, const char *Name)  
5 ;{  
6 ;  
7 ;    int i;  
8 ;    if ( SDL_Init(SDL_INIT_VIDEO) < 0 ) {  
9 ;        fprintf ( stderr , "Erreur :\n" );  
10 ;       fprintf ( stderr ,
11 ;                   " Impossible de se connecter au Display\n" );
12 ;       exit (1);
13 ;    }  
14 ;  
15 ;    screen = SDL_SetVideoMode(W, H, bpp, SDL_SWSURFACE|SDL_HPALETTE);
16 ;  
17 ;    if ( screen == NULL ) {
18 ;        fprintf ( stderr , "Erreur :\n" );
19 ;        fprintf ( stderr ,
20 ;                   " Impossible de se connecter au Display\n" );
21 ;        exit (1);
22 ;    }  
23 ;  
24 ;    SDL_WM_SetCaption ( Name, Name );
25 ;  
26 ;    for ( i=SDL_NOEVENT; i<SDL_NUMEVENTS; ++i )
27 ;        if ( (i != SDL_KEYDOWN) && (i != SDL_QUIT) )
28 ;            SDL_EventState(i, SDL_IGNORE);
29 ;  
30 ;    depth = screen->format->BitsPerPixel;
31 ;    width = screen->w;
32 ;    height = screen->h;
33 ;    buffer = (unsigned char *)screen->pixels;
34 ;    return (depth);
35 ;}
```

Como puedes ver el código SDL es mucho mas simple de escribir y de entender.

2. Conociendo libSDL

Flexibilidad La librería es multiplataforma. Las aplicaciones que creamos van a correr en Win32, BeOS y Linux sin tener que cambiar ni una sola línea de código. Otro aspecto de la flexibilidad es que aparte de que el código es totalmente multiplataforma no tienes puedes consultar cualquier aspecto de la implementación por si te puede aportar alguna idea o quieras mejorar algo. Nada está bloqueado, puedes ver como está implementada cada una de las partes de SDL.

Resumiendo, libSDL es portable, intuitiva y estable como una roca (así la define su creador). Podemos confiar en ella y las restricciones son mínimas. Comparada con sus rivales del mundo de la programación de videojuegos es mucho más asequible en todo los sentidos para adentrarnos en el mundo de la programación de videojuegos. Su potencial ha conseguido que hoy en día puedan ejecutarse juegos como *Civilization* o *Descent 2* en máquinas Linux.

Tenemos una herramienta profesional asequible que nos va a permitir dar mucha guerra en el mundo del videojuego.

2.6. Desventajas de SDL

Uno de los principales problemas de esta librería, y una de los principales aspectos por los que creamos este tutorial, es la escasa documentación existente. Tendremos que tener cierta habilidad para leer cientos de líneas de código en programas de ejemplo para buscar aquello que nos hace falta. Uno de los objetivos de este tutorial es que establezcas un punto de partida de conocimiento y no navegues a la deriva en tus inicios. Actualmente existen proyectos de ampliación y traducción de la documentación. Una vez te hagas con la experiencia necesaria puedes aportar tus conocimientos a los grupos encargados de realizar estas tareas.

Algunos programadores consideran la licencia LGPL como un obstáculo para la creación de videojuegos con esta librería. Desde nuestra tribuna lo consideramos como una ventaja que va a permitir que todo el mundo que quiera pueda tener acceso libre a SDL.

Esta biblioteca tampoco nos permite realizar juegos en tres dimensiones. Esta afirmación no es del todo cierta ya que se complementa con OpenGL para crear este tipo de videojuegos que serían temario de otro tutorial. De todas formas al no posibilitarlo intrínsecamente podemos considerarlo como desventaja ante aquellos que quieran realizar videojuegos de este tipo.

2.7. El futuro de SDL

SDL no termina aquí. Existen numerosos proyectos para mejorar esta librería a los que también puedes contribuir. En la actualidad se trabaja en conceptos menos comunes como el rediseñado de la arquitectura para permitir múltiples pantallas y entradas.

El desarrollo, sin eliminar la característica que la hace portable, está actualmente orientado a dotar a sistemas Linux de una mayor potencia en el mundo del videojuego. En la página oficial de SDL puedes seguir el estado actual de los proyectos que se llevan a cabo.

2.8. Recopilando

Si eres un programador experimentado SDL proporciona una biblioteca clara que puede ahorrarnos mucho tiempo en el desarrollo de nuestra aplicación sin perder estabilidad y que además es portable entre varias máquinas y sistemas operativos.

Si no eres un programador experimentado la librería SDL también tiene mucho que aportarte. SDL te permite centrarte en la tarea de desarrollo del juego, obviando los detalles del sistema operativo, y además puedes empezar a utilizarla sin tener que gastar ni un sólo euro en licencias de software a cambio de tu aportación como programador al desarrollo de la librería.

Con SDL, una máquina corriendo tu distribución Linux favorita, el compilador GCC que vamos a usar en el tutorial y tu editor libre favorito puedes, perfectamente, desarrollar un videojuego que no necesitaremos portar luego a diferentes sistemas operativos, ya que será compatible desde su creación. Podremos compartir nuestro videojuego con los usuarios de los sistemas operativos más comunes sin ningún problema.

Resumiendo, en la actualidad SDL es la librería de mayor potencial para la programación de videojuegos multiplataforma. Nos permite ahorrar mucho tiempo y permitiéndonos llegar a una gran audiencia gracias a su naturaleza multiplataforma.

2. Conociendo libSDL

Capítulo 3

Instalación libSDL

3.1. Introducción

El proceso de instalación de la librería SDL no es traumático. Vamos a centrarnos en la instalación para sistemas operativos **GNU/Linux** y el compilador **GCC** ya que sobre ellos vamos a desarrollar el tutorial.

Vamos a instalar también todas las librerías adicionales que necesitaremos para el seguimiento del curso y así dar por cerrado la etapa de instalación de librerías.

3.2. Objetivos

1. Instalar la librería SDL.
2. Instalar las librerías adicionales de SDL.
3. Comprobar la instalación de dichas librerías.

3.3. Instalando SDL

Algunas distribuciones GNU/Linux incluyen las bibliotecas SDL y todo lo necesario entre sus archivos de instalación. Como no es el caso más común vamos a explicar que necesitamos para instalar las librerías de SDL en nuestro ordenador con un sistema GNU/Linux compilando el código fuente.

3.3.1. Requisitos

Lo primero que necesitamos, incluso antes de SDL, es un compilador de C/C++. En nuestro caso hemos elegido el compilador GCC de GNU. Este requisito está cubierto en la mayoría de las distribuciones GNU/Linux. Para verificar si tenemos instalado el compilador y que versión estamos usando utilizamos el comando:

3. Instalación libSDL

```
gcc --version
```

En nuestro caso obtenemos, además de información sobre la licencia del compilador, el siguiente mensaje:

```
gcc (GCC) 4.1.2
```

Esto significa que todo ha ido OK. Vamos a instalar libSDL y en el siguiente apartado trataremos el tema de las librerías adicionales.

3.3.2. Instalación

Lo primero que debemos hacer es descomprimir el tarball (fichero .tar preparado para ser compilado e instalado) que contiene la librería. Este fichero lo tienes disponible en la carpeta **Instalación** del material del tutorial en la subcarpeta libSDL. Se trata del fichero **SDL-1.2.11.tar.gz**.

Como podrás ver vamos a instalar la versión 1.2.11 que es la versión de SDL más actual cuando realizamos este tutorial. Para descomprimir este fichero tendrás que dar la siguiente orden **tar** que permite descomprimir ficheros con compresión gzip:

```
tar -xvzf SDL-1.2.11.tar.gz
```

Esto creará una carpeta del mismo nombre del fichero que hemos descomprimido. En esta carpeta tienes un fichero **README** que te guía en la instalación de la librería según el sistema operativo que vayamos a utilizar. Los pasos de instalación más comunes son los siguientes.

Una vez situados en esta nueva carpeta, desde nuestro terminal, tecleamos las siguientes órdenes:

```
>./configure  
>make  
>make install
```

Así instalaremos la librería en un sistema operativo GNU/Linux que como ya sabes es el sistema operativo elegido a utilizar durante el tutorial. La última orden es imprescindible que la ejecutes como superusuario o no se podrá llevar a cabo la instalación.

Es importante para tu formación que conozcas que conseguimos con estas tres órdenes.

3.4. Instalando las librerías adicionales

Con `./configure` conseguimos que se cree un *makefile* adaptado a nuestro equipo. El script que contiene hace comprobaciones de dependencias, como el tipo de compilador que disponemos y la versión, para crear este archivo de tipo `make`.

Una vez finalizado este proceso deberemos de ejecutar la orden `make`. En este paso lo que hacemos es compilar la librería adaptada a nuestro sistema y arquitectura con lo que nos ahorraremos problemas de compatibilidad ya que estarán creadas específicamente para nuestro sistema.

El último paso y no menos importante, `make install`, deberemos de ejecutarlo como `ROOT` o superusuario, ya que la librería se va a copiar a un directorio cuyos permisos de escritura están limitados para la mayoría de los usuarios.

3.4. Instalando las librerías adicionales

El método de instalación de las librerías adicionales es análogo al de la propia `SDL`. Hay que tener en cuenta varios aspectos. Antes de instalar la librería `SDL_image` tenemos que tener que prever que necesita de ciertas bibliotecas auxiliares. Lo primero que vamos a hacer es proceder a su instalación.

Estas bibliotecas auxiliares son `libpng`, `libjpeg` y `zlib`. Dentro de la carpeta `Instalacion/Librerias Adicionales` del material puedes encontrar una subcarpeta llamada `PreSDL_image` donde están dichas librerías. Como puedes observar son tres ficheros `.tar.gz`. El proceso de instalación es idéntico al de `libSDL`:

1. Descomprimimos con `tar -xvzf <nombre_fichero>`
2. Nos situamos en la carpeta donde se extrajo el contenido del fichero.
3. Una vez en ella ejecutaremos las órdenes `./configure; make; make install`

Repitiendo este proceso con los archivos de `SDL_image`, `SDL_mixer`, `SDL_ttf` y `SDL_net` conseguiremos tener todas las librerías auxiliares que vamos a necesitar instaladas.

Todas estas bibliotecas tienen detrás un soporte oficial. Seguramente cuando estés realizando este tutorial podrás encontrar en la página oficial de cada una de las librerías una versión mejorada de las mismas. Con las que te ofrecemos en el material del curso te será suficiente para poder completar el tutorial pero te recomiendo encarecidamente que a la hora de realizar tus proyectos cuentes con la última versión estable de estas librerías que seguramente, a parte de corregir algún fallo interno, se les haya dotado de una potencia mayor.

3. Instalación libSDL

3.5. ¿Dónde se instalan las librerías? Localización

Como referencia, y aunque normalmente no es necesario saber estas localizaciones, vamos a presentar la localización de cada uno de los componentes de las librerías SDL en nuestro sistema GNU/Linux.

Localización	Contenido
/usr/local/include/SDL	Ficheros de cabecera
/usr/local/lib	Librerías
/usr/local/bin	Utilidad sdl-config
/usr/local/man	Páginas de manuales

Cuadro 3.1: Localización de libSDL.

Si tienes algún problema de enlazado con las librerías lo primero que debes comprobar es si estas localizaciones se encuentran en la ruta donde busca las librerías el enlazador. En el caso de que no fuera así añadir la ruta a la variable definida con este fin que no es otra que `LD_LIBRARY_PATH`. Comprueba primero que estas librerías estén en estas localizaciones. De no ser así probablemente estén en el mismo directorio pero omitiendo la carpeta `local`. En este caso añade esta nueva ruta al `LD_LIBRARY_PATH` vuelve a realizar la prueba.

`sdl-config` es una utilidad de SDL que nos proporciona información sobre varios aspectos de esta librería. Podemos consultar la versión, los parámetros que debemos de pasar al compilador para que pueda hacer uso de la librería... En el apartado de compilación haremos uso de ella para que puedas aprovechar sus capacidades y conocer las posibilidades que ofrece.

3.6. Probando la instalación

Vamos a probar si hemos instalado correctamente las librerías. La primera prueba a realizar es la comprobación de que la instalación de la biblioteca SDL ha sido efectuada correctamente. Para ello utilizaremos el comando `sdl-config`.

Si introducimos en una terminal la orden `sdl-config --version` el sistema devolverá la versión de la librería SDL instalada. En mi caso, que es el del material del tutorial, la versión 1.2.11. Si nos devuelve el valor correspondiente a la versión instalada es que todo habrá funcionado correctamente.

3.6. Probando la instalación

En el caso de que el sistema no encuentre la orden podemos tener dos problemas. El primero y más lógico es que tengamos que revisar la instalación de SDL. El segundo y menos frecuente es que no tengamos el directorio `/usr/local/bin` en el PATH del sistema por lo que tendremos que acudir a dicha carpeta para ejecutar el programa.

La segunda prueba que vamos a realizar consiste en coger un ejemplo sintácticamente correcto y compilarlo para comprobar si enlaza bien contra las librerías que hemos instalado. El ejemplo que vamos a probar es el siguiente:

```
;;
1 ;// Listado: test1.c
2 ;
3 ;// No te preocunes por no entender este código
4 ;// Sólo compilalo para comprobar que enlaza correctamente
5 ;
6 ;#include <SDL/SDL.h>
7 ;#include <stdio.h>
8 ;
9 ;
10 ;int main() {
11 ;
12 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
13 ;        fprintf(stderr, "No podemos inicializar SDL: %s\n", SDL_GetError());
14 ;        exit(1);
15 ;    }
16 ;    else {
17 ;
18 ;        fprintf(stdout, "Hemos inicializado SDL\n");
19 ;        atexit(SDL_Quit);
20 ;    }
21 ;
22 ;
23 ;    return 0;
24 ;}
```

Para compilar el ejemplo debes de introducir la siguiente orden la consola del sistema:

```
g++ -o test test1.c -lSDL
```

Con lo que conseguiremos crear un ejecutable *test* si la compilación fue correcta. En el siguiente capítulo estudiaremos como compilar un programa que utilice las librerías auxiliares de SDL. Por el momento nos vamos a conformar con enlazar estas librerías con el programa de ejemplo que nos servirá para comprobar si han sido instaladas correctamente. Para enlazarlas y crear el ejecutable introducimos la siguiente orden:

```
g++ -o test test1.c -lSDL -lSDL_image -lSDL_ttf -lSDL_mixer
```

3. Instalación libSDL

`-lSDL_net`

Con esta orden indicamos al compilador que enlace las librerías adicionales que hemos instalado. Si no recibimos ningún mensaje de error es que la compilación se habrá realizado correctamente. En cada uno de los capítulos tendremos que compilar ejemplos pero crearemos *makefiles* para que no se convierta la compilación en una tarea tediosa y repetitiva.

3.7. Recopilando

En este capítulo hemos realizado la tarea de instalar las librerías para preparar nuestro entorno de trabajo. Una vez instaladas hemos realizado una prueba con un pequeño ejemplo para comprobar que todo se realizó correctamente.

Ahora ya podemos avanzar en nuestro tutorial y compilar los ejemplos que se nos proporciona con el material.

Capítulo 4

Primeros pasos con SDL.

Conociendo el entorno

4.1. Introducción

A estas alturas del tutorial podemos decir que estamos preparados para comenzar a trabajar con la librería SDL. Sentirás curiosidad por entender el ejemplo que expusimos en el último apartado del capítulo anterior. En él aparecían funciones que no eran nativas del lenguaje de programación, en este caso C.

En este capítulo daremos los primeros pasos con SDL para que te vayas familiarizando con la librería.

4.2. Objetivos

Los objetivos de este capítulo son muy concisos:

1. Conocer el entorno de desarrollo a utilizar.
2. Conocer los distintos subsistemas disponibles en SDL.
3. Aprender a compilar un programa.
4. Iniciar SDL y sus librerías adicionales para que puedan ser usadas en C/C++.

4.3. El entorno de desarrollo

En esta sección debería hablarte de las cualidades de un u otro entorno de desarrollo integrado y el porqué nos hemos decidido por uno en concreto. Ante esto una gran objeción. Desde este tutorial no somos partidarios de ligarnos a ninguno de ellos por lo que te aconsejamos que tú tampoco lo hagas. No es necesario un IDE de programación para el seguimiento del tutorial pero

4. Primeros pasos con SDL. Conociendo el entorno

puedes usar el que más te guste.

El primer requisito que debe de cumplir nuestro entorno es ser legal. Yo decidí utilizar *GNU Emacs* como editor de textos, *GNU GCC* como compilador y *Linux* como sistema operativo. Estas tres cosas más las librerías *SDL* compiladas e instaladas son más que suficientes para seguir este curso. Puedes seguir el tutorial desde un sistema libre o propietario, que sea de pago o gratuito, pero por favor sé legal. Puedes usar *Kdevelop*, *Anjuta*... el que te sea más cómodo, cualquiera nos vale.

4.4. SDL y los subsistemas

Un subsistema engloba a una parte específica del hardware con un propósito específico. En *SDL* es una parte de la librería que nos ofrece soporte a diferentes partes del hardware. En el capítulo anterior veíamos las diferentes APIs que ofrece *SDL*, cada una de esas APIs responden ante un subsistema ya que los elementos que las componen tienen un objetivo común.

Todos estos subsistemas comprenden la parte principal del temario de este tutorial. Como fueron introducidos en el capítulo anterior vamos a ofrecer una vista rápida de cada uno de ellos para que conozcas las posibilidades que te ofrece *SDL*.

Los subsistemas soportados por *SDL* son:

Subsistema de Video Engloba todo lo referente al hardware de video y lo referente a los gráficos.

Subsistema de Audio Es el encargado de la reproducción de sonidos y de la gestión del hardware destinado al audio.

Subsistema de Gestión de Eventos Es la base de la interactividad con el usuario y con ciertos aspectos del sistema. Con este subsistema sabemos, por ejemplo, que desea hacer el usuario.

Joysticks Al no existir un estándar este subsistema nos permite un mayor control sobre los dispositivos de juegos.

CD-ROM Nos permite controlar la reproducción de CD-Audio.

Timers Nos permite sincronizar nuestro videojuego para que tenga la misma respuesta en diferentes tipos de sistemas.

Otros Además de todos estos subsistemas existen diferentes extensiones que nos facilitarán el trabajo con SDL como `SDL_image`, `SDL_ttf`... Además de añadir funcionalidad, como `SDL_net` para el manejo de redes o `SDL_mixer` que nos proporciona que el subsistema de audio sea realmente manejable.

Como ya hemos comentado profundizaremos en el estudio de cada uno de estos subsistemas. Cuando realices tus aplicaciones podrás dividir el desarrollo en tareas teniendo como discriminante el subsistema que vayas a utilizar. Por ejemplo en un primer momento puedes empezar a desarrollar la parte gráfica, luego, en una segunda tarea proporcionar a ese entorno gráfico de manejabilidad mediante eventos. Una vez depurada estas dos fases adentrarte en añadir audio a la aplicación... Con esta división en tareas podrás centrarte exclusivamente en un subsistema a la vez lo que te permitirá tener un mayor control sobre tu código.

En uno de los capítulos finales del tutorial desarrollaremos un videojuego paso por paso. Podrás tomarlo como guía la hora de empezar con el desarrollo de tu propio videojuego.

4.5. Compilando

Si has seguido los pasos de este tutorial para instalar la librería SDL ya habrás conseguido compilar tu primer programa SDL. En este apartado vamos a profundizar más en la semántica de las órdenes que introduciste así como en la utilización de las diferentes opciones de `sdl-config` que nos pueden ayudar a la hora de realizar la compilación.

El compilador elegido para el desarrollo del curso es el compilador GCC de GNU que nos proporciona la capacidad de poder construir nuestros proyectos implementados en C y en C++. Más que un compilador es una *suite* de compilación ya que realiza más tareas que el “simple” compilado. Por esto y porque es una de las herramientas libres por excelencia nos hemos decidido por esta suite. Aún así puedes usar cualquier compilador de C/C++.

Vamos a partir de las órdenes lanzadas en el capítulo anterior para estudiar el porqué de ejecutar dichas ordenes. Aunque se suponen ciertos conocimientos de programación y como consecuencia de compilación no queremos dejar nada en el tintero. La primera orden fue:

```
g++ -o test test1.c -lSDL
```

Como ya sabes `g++` es la orden que nos permite construir nuestro programa con el compilador GNU de C++. Cuando llamamos a este compilador con `g++` o `gcc` normalmente realizamos el preprocesado, la compilación y el

4. Primeros pasos con SDL. Conociendo el entorno

enlazado. Mediante el uso de opciones podemos parar este proceso en un punto intermedio. GCC acepta opciones, nombres de ficheros y operandos. Acepta varias opciones pero éstas no pueden ser agrupadas y tienen que ser especificadas por separado. Cuando vayamos haciendo uso de las opciones iremos explicando en qué consiste cada una de ellas ya que no es objetivo de este tutorial un profundo estudio del compilador, analizando sólamente lo necesario.

En nuestra orden le pasamos como opción `-o` que nos permite especificar el nombre del fichero ejecutable que se va a crear después de compilar y enlazar el programa. El nombre de nuestro fichero ejecutable será `test`. Sólo hemos necesitado un fichero por lo que en la misma orden vamos a compilar y enlazar el programa. El fichero a compilar es `test1.c`.

Finalmente aparece en la orden `-lSDL`. Esta es la parte del comando que nos permite indicarle al compilador que debe enlazar nuestro fichero con la librería SDL. Si pruebas a intentar compilar el programa de prueba sin esta opción recibirás un mensaje de error como el siguiente:

```
In function ‘main’:  
test1.c:(.text+0x19): undefined reference to ‘SDL_Init’  
test1.c:(.text+0x27): undefined reference to ‘SDL_GetError’  
test1.c:(.text+0x4a): undefined reference to ‘SDL_Quit’  
test1.c:(.text+0x73): undefined reference to ‘SDL_SetVideoMode’  
test1.c:(.text+0x81): undefined reference to ‘SDL_GetError’  
test1.c:(.text+0xb1): undefined reference to ‘SDL_WM_SetCaption’  
test1.c:(.text+0xd2): undefined reference to ‘SDL_PollEvent’  
collect2: ld devolvió el estado de salida 1
```

Puedes ver todos los errores que nos muestra el compilador son referencias sobre funciones que, como puedes observar por el prefijo, pertenecen a SDL. La solución: indicarle al compilador que debe de enlazar contra las librerías de SDL mediante `-lSDL`.

El proceso de compilación no se complica mucho más por añadir la biblioteca SDL. Para cada una de las librerías adicionales de SDL existe una forma equivalente de indicar al compilador que las utilice en el proceso de enlazado. Vamos a observar la segunda orden que ejecutamos en el capítulo anterior, se trata de:

```
g++ -o test test1.c -lSDL -lSDL_image -lSDL_ttf -lSDL_mixer  
-lSDL_net
```

Como puedes ver hemos añadido `-lSDL_image` `-lSDL_ttf` `-lSDL_mixer` `-lSDL_net` con esto indicamos que queremos enlazar con todas las librerías

auxiliares que instalamos también en el capítulo anterior. Si no necesitamos alguna de estas librerías adicionales lo mejor es no incluirlas en el momento de compilación ya que haría crecer el tamaño de nuestro fichero ejecutable sin sentido alguno.

En el caso de que necesitamos compilar un fichero fuente SDL pero no enlazarlo el proceso es el mismo que para cualquier aplicación escrita en C/C++. La orden desde consola es la siguiente:

```
g++ -c mi_fichero.cpp
```

Con esto conseguiremos compilar el fichero objeto para luego enlazarlo. `g++` compilará el fichero y como resultado obtendremos el objeto del fichero fuente. Estos ficheros son fácilmente reconocibles ya que poseen, normalmente, la extensión `.o`

Es una buena práctica especificar al compilador las banderas propias de SDL. Con este fin vamos a estudiar la utilidad `sdl-config`.

4.5.1. La herramienta `sdl-config`

SDL proporciona una herramienta de consulta que nos va a ser útil a la hora de realizar la compilación de nuestro proyecto. Se trata de `sdl-config`. Si introduces esta orden en un terminal y pulsas enter podrás observar que hay disponibles varias opciones. Vamos a detallar las más relevantes:

`sdl-config --version` Nos muestra la versión de la librería SDL instalada en el equipo.

`sdl-config --cflags` Indica las banderas u opciones que le podemos indicar al compilador de C/C++ cuando utilicemos SDL. Normalmente trabajaremos con `makefiles` por lo que será interesante iniciar la variable `CXXFLAGS` con el valor devuelto por este comando. Es una buena práctica que el compilador conozca estas banderas a la hora de crear un fichero objeto.

`sdl-config --libs`

`sdl-config --static-libs` Estas dos opciones nos permiten indicarle al compilador las librerías sobre las que se debe de enlazar nuestro programa SDL así como la ruta donde están las mismas.

Haciendo uso de esta utilidad podemos compilar nuestro fichero fuente de la siguiente manera:

4. Primeros pasos con SDL. Conociendo el entorno

```
g++ -o test test1.c `sdl-config --cflags --libs`
```

Esta utilidad nos permite no tener que recordar que librerías y que banderas tenemos que añadir a la línea de compilación. Recopilando, cuando queramos obtener un fichero objeto la orden a ejecutar es:

```
g++ -c `sdl-config -cflags` mifuente.c
```

Cuando queramos enlazar nuestros ficheros objeto utilizaremos la siguiente orden:

```
g++ -o miaplicacion mifuente.o `sdl-config --libs`  
<bibliotecas adicionales>
```

4.6. Los tipos de datos en SDL y la portabilidad

Antes de empezar a presentar funciones debes conocer algunas peculiaridades sobre los tipos de datos en SDL. No podemos dejar pasar la oportunidad de aclarar conceptos antes de seguir caminando con este tutorial.

Como ya sabemos SDL es multiplataforma. Necesitamos que los tipos de datos que utilicemos sean iguales independientemente del sistema en el que desarrollemos nuestra aplicación. Seguramente queramos compilarla en otro entorno y es importante que lo tengamos todo controlado. Por ejemplo el tipo de datos *int* puede tener rangos diferentes según el sistema donde compilemos nuestra aplicación o en versiones de compilador diferentes. Esto es debido a que se reservan diferentes números de bits para la representación de este tipo de dato. Para que nuestro código sea portable y no tengamos comportamientos no deseados estos bits tienen que ser los mismos sea cual sea el sistema al que portemos el código.

Con el fin de conseguir esta unicidad SDL redefine estos datos según el número de bytes que necesitemos y si el dato debe de tener signo o no. La forma de identificar las redeficiones de los tipos de datos es muy simple como vamos a ver. Estas rediciones siguen una regla básica. Observa la siguiente palabra:

GtttBB

En G indicamos si la variable que vamos a definir es con signo (S) o sin signo (U). En la parte que referenciamos aquí con ttt indicamos el tipo de datos que vamos a utilizar, es decir, si el tipo de dato a declarar o definir es un *int*, *float*... Como puedes ver el que hayamos puesto tres t no condiciona el número de caracteres de esta parte de la definición. El último de los campos se

4.7. `SDL_GetError()` y `SDL_WasInit()`

refiere al número de bits que necesitamos para este tipo de dato. No podemos poner cualquier número si no sólo aquellos que sean múltiplos de 8, desde el mismo 8 hasta el 64, es decir que para el número de bits los valores posibles son 8, 16, 32 y 64.

Ninguno de los campos que hemos expuesto son omitibles. Así si queremos definir una variable entera de 32 bits sin signo lo haríamos de la siguiente manera:

```
Uint32 mi_variable;
```

Si por ejemplo queremos definir un entero de 16 bits con signo pues bastaría con que el tipo de datos fuese `Sint16`. Estos tipos tendrán las mismas características sea cual sea el sistema en el que vayamos a portar nuestro código. Como puedes ver esta redefinición de datos no tiene una sintaxis compleja y va a ser de mucha utilidad en el desarrollo de nuestras aplicaciones.

La mayoría de los parámetros y tipos de datos más complejos definidos en SDL se construyen sobre este tipo de datos. Lógico, ya que garantiza la homogeneidad de la que hace gala para los diferentes sistemas.

Con esta notación podemos utilizar fácilmente tipos de datos ajustados a nuestras necesidades en todo momento lo que es un punto a favor de SDL. Veamos un ejemplo por si todavía no has visto la ventaja que supone tener definidos este tipo de datos. Imagínate que desarrollas tu aplicación sobre un sistema que define el tipo `int` con 32 bits. Ahora decides portar tu aplicación a un sistema Linux con un compilador que reserva para los enteros de tipo `int` sólo 16 bits. Puede darse el caso de que en un primer momento usases valores que ahora mismo estuviesen fuera de rango para el sistema al que portas la aplicación. El esfuerzo de los creadores para que SDL fuese portable cae en saco roto. La solución es bastante clara en vez de utilizar `int` en el código original, y si necesitamos 32 bits, es definir nuestras variables de este tipo con `Sint32` y solucionaríamos este problema.

4.7. `SDL_GetError()` y `SDL_WasInit()`

Vamos a presentar en esta sección dos funciones que nos van a ser de gran ayuda a la hora de realizar nuestras aplicaciones. Sin más pasamos a detallarlas.

La función `SDL_WasInit()` determina que subsistema o subsistemas están inicializados. El prototipo de la función es el siguiente:

```
Uint32 SDL_WasInit(Uint32 flags);
```

Esta función toma un sólo parámetro, un entero de 32 bits sin signo, y devuelve un valor de las mismas características. Esta función actúa de

4. Primeros pasos con SDL. Conociendo el entorno

la siguiente manera. Le pasamos como parámetro una de las banderas que utilizamos para inicializar la librería SDL, por ejemplo `SDL_INIT_AUDIO`. Ahora bien si la función nos devuelve el valor 0 es que dicho subsistema no ha sido inicializado, pero si nos devuelve el valor que le hemos pasado como parámetro, en nuestro caso `SDL_INIT_AUDIO` el significado será que el subsistema ha sido inicializado y que está disponible en dicho instante.

Si, por ejemplo, le pasamos otro valor combinado, como puede ser `SDL_INIT_AUDIO | SDL_INIT_VIDEO` la función puede devolver tres valores diferentes. El primero de ellos 0, que significará que ninguno de los dos subsistemas ha sido inicializado. Si devuelve `SDL_INIT_AUDIO | SDL_INIT_VIDEO` tendremos los dos subsistemas inicializados. Ahora bien, la función `SDL_WasInit()` tiene la capacidad de discernir si ha sido inicializado sólo uno de los subsistemas que ha recibido como parámetro, por lo que devolverá el valor de la constante del subsistema que, habiendo sido pasado como parámetro de la función, esté inicializado. Así si el subsistema de audio no está inicializado y el de video sí, la función devolverá el valor `SDL_INIT_VIDEO`.

Esto nos permite comprobar si tenemos disponible un subsistema o si debemos de inicializarlo.

La función `SDL_GetError()` es fundamental para el programador SDL. Nos muestra información del último error producido en SDL. Proporciona un mensaje de error para cualquier función fallida de SDL. El prototipo de la función es:

```
char *SDL_GetError(void);
```

Como puedes ver devuelve una cadena de caracteres que podemos mostrar en pantalla, normalmente por consola, para poder revisar cual ha sido el último error interno de SDL. Esta cadena es asignada estáticamente y no debe ser liberada por el usuario en ningún momento.

Cada vez que realicemos una llamada a una función que informe de un estado de error deberemos de realizar la comprobación, mediante una estructura selectiva, de existencia de error teniendo que hacer uso de esta función para informar del mismo. Las funciones en SDL suelen devolver el valor -1 si se ha producido un error en ellas pero esto no es suficiente en la mayoría de los casos. No se proporciona más información que el propio error. Ahí es donde sale a nuestra ayuda `SDL_GetError()` aportándonos información adicional.

Cuando utilicemos esta función en nuestro código y no ejecutemos nuestra aplicación desde consola se creará un fichero `stdout.txt` o `stderr.txt` en el directorio de trabajo desde donde hayamos lanzado la aplicación. Se creará

4.8. Inicialización y Finalización de SDL

un fichero u otro dependiendo de donde hayamos direccionado el error. En nuestro caso usaremos la función *fprintf()* de C o *cerr* en C++ para indicar que dicho mensaje debe ir a la salida estándar de errores. Como puedes suponer dichos ficheros contendrán los errores ocurridos en la ejecución del programa.

Este trozo de código ilustra las líneas anteriores:

```
1 ;_____
2 // En C
3 ;
4 ;if(funcion() != OK) {
5 ;
6     fprintf(stderr, "Error en funcion: %s", SDL_GetError());
7     exit(1);
8 ;
9 ;
10 // En C++
11 ;
12 ;if(funcion() != OK) {
13 ;
14     cerr << "Error en funcion(): " << SDL_GetError() << endl;
15     exit(1);
16 ;
17 ;}_____
```

Esta función te será de mucha utilidad durante el desarrollo de tus aplicaciones. No dudes en utilizarla ya que te ahorrará muchos quebraderos de cabeza.

Puede que estés deseando que dejemos de dar vueltas por elementos “accesorios” de SDL y nos pongamos mano a la obra con los subsistemas. Piensa que esto que estamos viendo ahora mismo es realmente importante y nos ayudará a situarnos el tiempo que estemos realizando aplicaciones en SDL.

4.8. Inicialización y Finalización de SDL

En esta sección vamos a mostrar como preparar nuestro código para que acepte funciones y tipos de datos SDL. Este proceso es fundamental ya que será necesario para todas y cada uno de las aplicaciones y ejemplos que creemos con esta librería.

Los primero que debemos hacer cuando vamos a escribir nuestro programa SDL es incluir el fichero de cabecera de la biblioteca. SDL se instala en una ruta donde el enlazador va a comprobar si existe la librería a la hora de realizar el enlazado de nuestro programa con dicha librería. Para incluir el fichero

4. Primeros pasos con SDL. Conociendo el entorno

de cabecera sólo debemos añadir esta línea junto a los demás “*includes*” de nuestro programa:

```
#include <SDL/SDL.h>
```

Si has optado por seguir el curso sobre otro sistema operativo distinto a GNU/Linux debes de comprobar que la ruta donde instalaste la librería está incluida en el *library path* del sistema. Con este `include` nuestra suite de compilación `GCC` podrá comprobar los problemas de sintaxis en las funciones y tipos de datos propios de `SDL`.

Una vez incluido el fichero de cabecera debemos de iniciar la utilización de la librería. En cada uno de los apartados del tutorial encontrarás un recordatorio de cómo inicializar el subsistema específico que se esté tratando en dicho apartado.

`SDL` proporciona una función que realiza la tarea de inicializar la librería con unas determinadas características. Es la función:

```
int SDL_Init(Uint32 flags);
```

Esta función recibe como parámetro un entero sin signo de 32 bits y devuelve un 0 si la inicialización fue correcta. En el caso de no serla devuelve el valor -1. El parámetro *flags* puede tomar distintos valores según los subsistemas que queramos inicializar. Estos valores vienen definidos en `SDL` por unas constantes que son:

- `SDL_INIT_VIDEO`: Inicializa el subsistema de video.
- `SDL_INIT_AUDIO`: Inicializa el subsistema de audio.
- `SDL_INIT_TIMER`: Inicializa el subsistema de *timers*.
- `SDL_INIT_CDROM`: Inicializa el subsistema de CD-ROM.
- `SDL_INIT_JOYSTICK`: Inicializa el subsistema de joystick.
- `SDL_INIT_EVERYTHING`: Inicializa todos los subsistemas.
- `SDL_INIT_NOPARACHUTE`: Prepara a `SDL` para capturar señales de error fatal.

A la hora de inicializar el sistema podemos combinar varios de estos subsistemas en la misma llamada. Estas constantes definen valores de bit. Para combinarlas haremos uso del operador `|`. Por ejemplo si queremos iniciar el sistema con los subsistemas de audio, video y CD-ROM pasaríamos como parámetro a la función `SDL_Init` el valor:

4.8. Inicialización y Finalización de SDL

```
SDL_INIT_AUDIO | SDL_INIT_VIDEO | SDL_INIT_CDROM
```

Quedando la llamada a función como *SDL_Init(SDL_INIT_AUDIO | SDL_INIT_VIDEO | SDL_INIT_CDROM)*. Si queremos activar todo solamente pasaremos como parámetro *SDL_INIT_EVERYTHING*.

Como decíamos en el apartado anterior, es una buena costumbre hacer uso de la función *SDL_GetError()* en las llamadas a funciones que nos devuelven un posible estado de error. *SDL_Init()* es una de estas funciones. Para comprobar si la inicialización se ha realizado correctamente podemos utilizar el siguiente esquema:

```
;  
1 ;// Ejemplo de inicialización de subsistemas  
2 ;// Inicializamos video y audio  
3 ;  
4 ;if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {  
5 ;  
6 ;    fprintf(stderr, "Error en SDL_Init(): %s", SDL_GetError());  
7 ;    exit(1);  
8 ;  
9 ;};
```

En este código inicializamos los subsistemas de video y audio a la vez. Si existe algún problema informamos a través de la salida estándar de errores y terminamos el programa. Como puedes ver se trata de código C, pero válido en C++. Como comentamos en el primer capítulo iremos haciendo una introducción progresiva al desarrollo en C++.

La tarea de inicializar el sistema sólo se puede realizar una vez. Si necesitamos inicializar otro subsistema en el transcurso de nuestra aplicación podemos hacerlo con la función *int SDL_InitSubSystem(Uint32 flags)*; de la misma manera que lo hacíamos con la función principal. Por ejemplo si queremos inicializar el subsistema de audio después de haber llamando ya a la función *SDL_Init()* la sintaxis correcta sería:

```
SDL_InitSubSystem(SDL_INIT_AUDIO);
```

Esta función devolvería el correspondiente valor que nos permite verificar si la inicialización fue correcta. Igual que necesitamos inicializar SDL para comenzar a trabajar con ella, tenemos que cerrarla una vez que hayamos concluido de utilizar la librería. Con este objetivo SDL proporciona la función:

```
void SDL_Quit(void);
```

Se encarga de cerrar todos los subsistemas de SDL además de liberar todos los recursos ocupados por la librería que fueron reservados inicialmente

4. Primeros pasos con SDL. Conociendo el entorno

mediante *SDL_Init()*. Como ya hemos dicho siempre tiene que ser llamada antes de salir del programa. Como puedes observar la función ni devuelve ni recibe ningún parámetro por lo que la hace ideal para pasarla como parámetro a la función *atexit()*. Por si no conocías la función *atexit()* recibe un puntero a función que ejecutará a la terminación del programa.

Introduciendo en nuestro código la sentencia *atexit(SDL_Quit);*, siempre que sea después de *SDL_Init()* claro está, podremos olvidarnos de realizar la llamada a la función de terminación en puntos concretos del código. Esta es una solución válida para realizar pequeños programas. Para un código más avanzado debemos de cerrar la librería manualmente cuando ya no sea necesaria.

Tal como pasaba con la inicialización de SDL podemos cerrar un sólo subsistema en un momento dado. Para ello deberemos de hacer uso de la función:

```
void SDL_QuitSubSystem(Unit32 flags);
```

Por ejemplo si quisieramos desactivar el subsistema de audio introduciríamos en nuestro código la llamada a la función como *SDL_QuitSubsystem(SDL_INIT_AUDIO);*. Como puedes observar la función recibe las mismas banderas que la función de inicialización de la librería.

En la siguiente tabla tienes un resumen de las funciones que acabamos de presentar:

Función	Descripción
<i>SDL_Init()</i>	Inicializa SDL con uno o más subsistemas.
<i>SDL_InitSubSystem()</i>	Inicializa un subsistema de SDL en particular. Sólo podemos usar esta función después de haber usado <i>SDL_Init()</i> .
<i>SDL_Quit()</i>	Termina todos los subsistemas de SDL.
<i>SDL_QuitSubSystem()</i>	Termina un subsistema de SDL en particular.
<i>SDL_WasInit()</i>	Nos permite comprobar que subsistemas están inicializados actualmente.
<i>SDL_GetError()</i>	Devuelve el último error interno informado por SDL.

Cuadro 4.1: Funciones de inicialización.

Llegados a este punto sabemos iniciar la librería, así como compilar con ella. Es hora de, con ayuda de un par de funciones más realizar nuestro primer ejemplo con SDL.

4.9. Hola Mundo

Es muy difícil definir que tipo de programa es un “Hola mundo” en SDL. Por simplicidad vamos a intentar poner en marcha SDL inicializando el subsistema de video en modo ventana. Colocaremos en esa ventana el nombre de nuestra aplicación y daremos por finalizado nuestro ejemplo. Vamos a utilizar código C en este ejemplo, según vayamos avanzando en el tutorial iremos introduciendo un mayor número de líneas de código C++ y mostrando las equivalencias en C para no desviarnos de nuestro objetivo principal.

El código que responde a esta especificación es:

```
;  
1 ;// Listado: prueba.cpp  
2 ;// Hola Mundo  
3 ;  
4 ;#include <stdio.h>  
5 ;#include <SDL/SDL.h> // Incluimos la librería SDL  
6 ;  
7 ;  
8 ;int main() {  
9 ;  
10 ;    SDL_Surface *pantalla; // Definimos una superficie  
11 ;    SDL_Event evento; // Definimos una variable de eventos  
12 ;  
13 ;  
14 ;    // Inicializamos SDL  
15 ;  
16 ;    if(SDL_Init(SDL_INIT_VIDEO)) {  
17 ;  
18 ;        // En caso de error  
19 ;  
20 ;        fprintf(stderr, "Error al inicializar SDL: %s\n",  
21 ;                    SDL_GetError());  
22 ;        exit(1);  
23 ;  
24 ;    }  
25 ;  
26 ;    atexit(SDL_Quit); // Al salir, cierra SDL  
27 ;  
28 ;    // Establecemos el modo de pantalla  
29 ;  
30 ;    pantalla = SDL_SetVideoMode(640, 480, 0, SDL_ANYFORMAT);  
31 ;  
32 ;    if(pantalla == NULL) {  
33 ;  
34 ;        // Si no hemos podido inicializar la superficie  
35 ;  
36 ;        fprintf(stderr, "Error al crear la superficie: %s\n",  
37 ;                    SDL_GetError());  
38 ;        exit(1);
```

4. Primeros pasos con SDL. Conociendo el entorno

```
39 ;
40 ;    }
41 ;
42 ;    // Personalizamos el título de la ventana
43 ;
44 ;    SDL_WM_SetCaption("HOLA MUNDO", NULL);
45 ;
46 ;
47 ;    // Bucle infinito
48 ;
49 ;    for(;;) {
50 ;
51 ;        // Consultamos los eventos
52 ;
53 ;        while(SDL_PollEvent(&evento)) {
54 ;
55 ;            if(evento.type == SDL_QUIT) // Si es de salida
56 ;                return 0;
57 ;        }
58 ;    }
59 ;
60 ;}
```

Para compilar el ejemplo sólo debes de introducir en la consola, una vez situado en el directorio donde guardes el fichero de prueba la orden:

```
g++ -o test prueba.cpp `sdl-config -cflags -libs`
```

Con lo que crearemos un ejecutable llamado *test* resultado de la compilación. Si ejecutas el programa observarás una ventana cuyo título será “Hola Mundo”. Vamos a estudiar que hemos hecho en esta primera prueba.

Lo primero que hacemos en la función principal es definir dos variables que utilizaremos con posterioridad. La primera es de tipo superficie (*SDL_Surface*). Este tipo de variables es muy común en el tratamiento de gráficos en SDL. Se estudiará en profundidad en el capítulo que trata sobre el subsistema de video. Esta superficie es la principal de la aplicación que mostrará todos aquellos gráficos que maquetemos en ella. Para realizar esta tarea existen varias técnicas, que como hemos indicado, estudiaremos más adelante.

La segunda variable que creamos es la de tipo evento (*SDL_Event*). Mediante esta variable vamos a controlar una acción del usuario que nos permita dar por finalizada la ejecución de la aplicación. También estudiaremos los eventos en profundidad en un capítulo posterior.

El siguiente paso en el código es iniciar la librería SDL con el subsistema de video. Para ello, como puedes observar utilizamos la función *SDL_Init()*. Esta

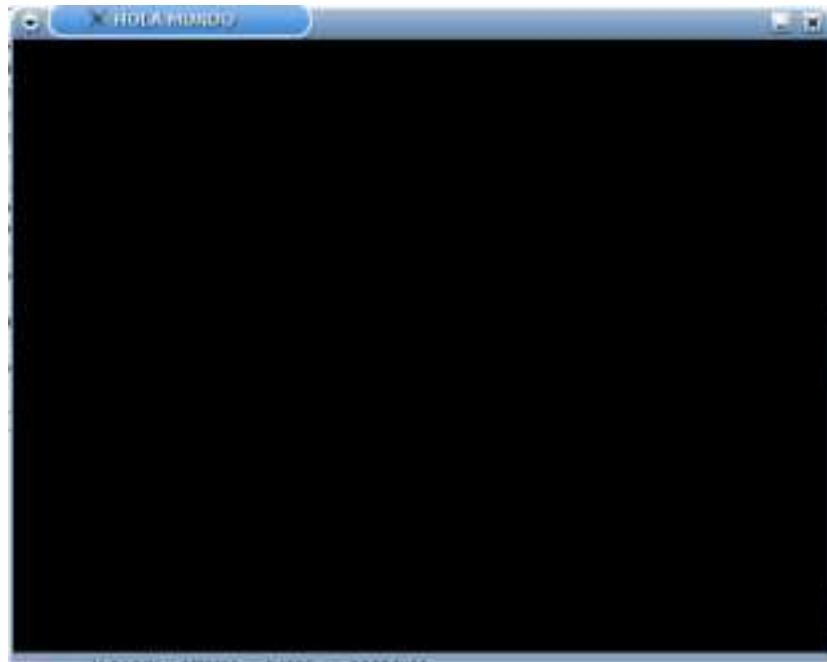


Figura 4.1: Hola Mundo en un entorno KDE

parte del código es fundamental ya que será común en toda las aplicaciones que realicemos especificando qué subsistemas queremos que se inicialicen con la librería. El hecho de realizar la inicialización dentro del campo de condición de un *if* simplemente es para comprobar que sea correcta, y si no lo es, poder mostrar el correspondiente mensaje en la salida estándar de errores.

Lo siguiente que nos encontramos en el código es la llamada a la función *atexit()* que recibe como parámetro la función *SDL_Quit*. Esto produce que a la salida de la aplicación sea cerrada la librería SDL y sus subsistemas.

Es el momento de establecer el modo de video. Para esto sirve la función *SDL_SetVideoMode()*. Esta función devuelve la superficie que mostraremos en pantalla. Esta función y los conceptos asociados a ellas serán estudiados con profundidad. Una vez establecido el modo de video comprobamos que se haya realizado correctamente.

Nuestro "Hola mundo" se basa en la personalización de una ventana. Con ese objetivo realizamos la llamada a la función *SDL_WM_SetCaption("HOLA MUNDO", NULL)*. Esta función establece en el título de la ventana el primer parámetro que recibe, mientras que el segundo, en este caso a *NULL*, sirve para especificar el ícono para la aplicación. Como todas las demás funciones de este ejemplo será vista con más profundidad en capítulos posteriores.

El último trozo de código es conocido como *game loop*. El *game loop* de

4. Primeros pasos con SDL. Conociendo el entorno

un videojuego es un bucle que se repite indefinidamente en el que se realizan varias acciones. Las acciones más comunes son las de responder a las acciones producidas por el usuario, actualizar la lógica del juego, así como los gráficos... En este caso nuestro bucle sólo reacciona al evento de salida de la aplicación. Por el momento esto es todo lo que debes de saber sobre el código del ejemplo.

4.10. Trabajando con las librerías adicionales

Cada una de las librerías adicionales SDL que utilicemos en nuestro código tienen que ser inicializadas y cerradas individualmente. El proceso es análogo al utilizado con SDL con la particularización de las funciones que realizan esta tarea en las diferentes librerías.

En esta sección vamos a ver como podemos realizar esta tarea para cada una de las librerías. Incluiremos un pequeño recordatorio en los capítulos donde tratamos estas librerías.

4.10.1. SDL_image

Preparar el código para hacer uso esta librería es un proceso muy sencillo. Simplemente necesitamos especificar en nuestro fichero fuente que vamos a utilizarla mediante su fichero de cabecera. Para esto añadimos `#include <SDL/SDL_image.h>`

A parte de esto debes recordar especificar mediante `-lSDL_image` al compilador que quieras que esta librería sea enlazada por nuestra aplicación.

4.10.2. SDL_ttf

Para utilizar esta librería auxiliar tenemos que seguir unos pasos que son comunes a la mayoría de las librerías auxiliares que utilizamos con SDL. El primer paso es incluir su cabecera en los ficheros fuentes donde vayamos a utilizar funciones o tipos de datos definidas en esta biblioteca. Para ello añadimos `#include <SDL/SDL_ttf.h>` en nuestro código fuente.

El segundo paso es el de inicializar la librería. De esto se hace cargo la propia librería que nos proporciona una función que en este caso es:

```
int TTF_Init(void);
```

Esta función no recibe ningún parámetro y devuelve 0 si la inicialización fue correcta. En el caso de que ocurriese algún error la función devolverá el

4.10. Trabajando con las librerías adicionales

valor -1. Al terminar de utilizar la librería deberemos de cerrarla. Para realizar esta tarea tenemos otra función cuyo prototipo es:

```
void TTF_Quit(void);
```

Como puedes ver esta función no recibe ni devuelve ningún valor lo que la hace compatible con la función `atexit()` lo que nos permitirá, si hacemos uso de ella, descuidarnos de la tarea de cerrar dicha librería.

Un esquema de proceso de uso de esta librería podría ser el siguiente:

```
;-----  
1 ;// ... otros includes ...  
2 ;  
3 ;#include <SDL/SDL_ttf.h>  
4 ;  
5 ;// ... código ...  
6 ;  
7 ;SDL_Init(SDL_INIT_VIDEO);  
8 ;  
9 ;  
10 ;// Inicializamos SDL_ttf  
11 ;if(!TTF_Init() < 0) {  
12 ;  
13 ;    fprintf(stderr, "No se puedo inicializar SDL_ttf\n");  
14 ;    exit(1);  
15 ;  
16 ;}  
17 ;  
18 ;atexit(TTF_Quit());  
19 ;  
20 ;// ... código donde utilizamos SDL_ttf ...  
;
```

Recuerda que tienes que indicar al compilador el uso de esta librería mediante `-lSDL_ttf` cuando quieras construir la aplicación.

4.10.3. SDL_mixer

Para utilizar esta librería auxiliar tenemos que seguir los pasos comunes a la mayoría de las librerías auxiliares que utilizamos con SDL. El primero es incluir su cabecera en los ficheros fuente donde vayamos a utilizar funciones o tipos de datos definidas en la biblioteca en cuestión. Para ello añadimos `#include <SDL/SDL_mixer.h>` a nuestro código fuente donde hagamos uso de esta librería.

El segundo paso es el de inicializar la librería. De esto se hace cargo la propia librería mediante una función que en el caso de *SDL_mixer* es:

```
int Mix_OpenAudio(int frequency, Uint16 format, int channels, int  
                  chunksizes);
```

4. Primeros pasos con SDL. Conociendo el entorno

Esta función recibe varios parámetros. El nombre que recibe cada uno de ellos es descriptivo pero, si aún así no conoces los conceptos que representan, no te impacientes ya que hay un capítulo entero donde desarrollamos la utilización de esta librería. La función devuelve 0 si la inicialización fue correcta. En el caso de que ocurriese algún error la función devolverá el valor -1. Al terminar de utilizar la librería deberemos de cerrarla para ello tenemos otra función cuyo prototipo es:

```
void Mix_CloseAudio(void);
```

Como puedes ver esta función no recibe ni devuelve ningún valor lo que la hace compatible con la función *atexit()* lo que nos permitirá, siempre que hagamos uso de ella, no tener que preocuparnos de cerrar dicha librería. Un esquema de proceso de uso de esta librería es análogo al presentado en la inicialización-cierre de *SDL_ttf*.

Recuerda que tienes que indicar al compilador el uso de esta librería mediante `-lSDL_mixer` cuando quieras construir la aplicación.

4.10.4. **SDL_net**

Para hacer uso de esta librería tenemos que realizar unos pasos que te van a resultar, a estas alturas, familiares. El primero de ellos es incluir el fichero de cabecera de la librería en los ficheros fuente donde necesitemos de la utilización de funciones y tipos de datos definidos en *SDL_net*. Para incluir el fichero utilizamos `#include <SDL/SDL_net.h>`

El siguiente paso es inicializar la propia librería antes de hacer uso de ella. Para esto la librería proporciona la función:

```
int SDLNet_Init();
```

Esta función no recibe ningún parámetro y devuelve -1 en caso de que exista algún error. Si la inicialización ha sido correcta devuelve el valor 0. Una vez inicializada la librería podemos hacer uso de ella. SDL debe estar inicializada antes de realizar una llamada a esta función.

Cuando la aplicación llegue a su fin, o bien, no necesitemos más de esta librería debemos de cerrarla. Para ello existe otra función propia de *SDL_net*. Se trata de:

```
void SDLNet_Quit();
```

Como puedes ver esta función no recibe ni devuelve ningún valor. Esto la hace compatible con la función *atexit()* lo que nos permitirá, siempre que

hagamos uso de ella, no tener que preocuparnos de cerrar dicha librería. Esta función se encarga de finalizar la API para los servicios de red y realiza la limpieza de los recursos utilizados. Después de la llamada a esta función se cierran todos los sockets y no se permite el uso de ninguna función de la biblioteca *SDL_net*.

Claro está, para poder volver a utilizar funciones de *SDL_net* después de la llamada a esta función habría que llamar de nuevo a *int SDLNet_Init()*.

4.11. Recopilando

En este capítulo hemos dado los primeros pasos con *SDL*. Han sido un poco pesados porque detallan tareas, como la de inicialización, que van a ser comunes a todas las aplicaciones *SDL* que desarrollemos.

Una vez establecido el entorno de desarrollo se han presentado los subsistemas de *SDL* en un segundo contacto, ya que forman gran parte del contenido del tutorial.

Hemos detallados los pasos necesarios para compilar cualquier programa *SDL* con la ayuda de la herramienta *sdl-config*.

SDL presenta características, funciones y tipos de datos para que nuestro código sea totalmente portable, así como funciones auxiliares que nos sirven de ayuda a la hora de desarrollar nuestra aplicación.

La inicialización y finalización de *SDL* y las distintas librerías auxiliares es fundamental para el correcto funcionamiento de nuestro programa.

Hemos implementado, compilado y probado nuestro primer programa "Hola Mundo".

Capítulo 5

El Subsistema de Video

5.1. Introducción

En la mayoría de videojuegos el componente visual está entre los más importantes de su desarrollo. Para una persona corriente más del setenta por ciento de la información que recibe lo hace a través de la vista. Esto hace que el subsistema de vídeo sea una parte fundamental del desarrollo de videojuegos y por tanto de SDL. Este subsistema es usado para crear y manejar gráficos en dos dimensiones, como ocurre con DirectDraw ©o Qt. Para crear gráficos tridimensionales hay que recurrir a otras herramientas como OpenGL la cual es soportada desde SDL.

Un juego atractivo gráficamente incita a ser probado, utilizado y disfrutado más que otro, que aunque pueda tener un mejor comportamiento, tenga más descuidado el aspecto gráfico. Todos conocemos videojuegos que su jugabilidad y diversión compensan un mejorable diseño gráfico lo que no le resta ese componente adictivo que tienen muchos juegos de los años 80. En la actualidad el procesamiento gráfico ha conocido grandes avances. Se producen unos juegos de una calidad gráfica inimaginable hace unos años. Esto provoca un aumento de carga de trabajo importante al proceso de crear videojuegos. Actualmente el crear cierto tipo de videojuegos tiene más tareas comunes con la realización de una película que con el desarrollo software. Piensa que hay personas que invierten cientos de euros en que su máquina tenga una potencia gráfica descomunal sólo para disfrutar de los juegos a la más alta definición.

El videojuego de la figura 5.1 es un claro ejemplo de un juego que ha evolucionado mucho en relativamente poco tiempo. Es el archiconocido Frozen Bubble. Este juego ha experimentado unas mejoras gráficas paralelas a las mejoras de los entornos gráficos de los sistemas operativos Linux para escritorio.

Es fundamental que conozcamos la manera de interactuar con SDL para establecer modos de vídeo que nos permitan mostrar nuestro videojuego de forma óptima, la manera de cargar diferentes gráficos e imágenes en pantalla,

5. El Subsistema de Video



Figura 5.1: Ejemplo de pantalla de videojuego

así como dibujar figuras básicas en la ventana de nuestra aplicación. Todas estas tareas están soportadas por el concepto de superficie que estudiaremos en esta unidad.

Nuestra destreza manejando esta sección de la API de SDL será determinante a la hora de desarrollar un videojuego.

5.2. Objetivos

Los objetivos de este capítulo son:

1. Saber inicializar el subsistema de vídeo.
2. Elegir y establecer el mejor modo de vídeo para nuestra aplicación.
3. Conocer las diferentes estructuras que soportan el subsistema de vídeo y todo lo que engloba.
4. Comprender el concepto de superficie así como su creación y manejo.
5. Manejar la carga de imágenes en pantalla así como los funciones que actúan sobre ellas.
6. Aprender a dibujar figuras básicas en pantalla.

5.3. Conocimientos Previos

Antes de entrar en materia con el subsistema de video es importante que repasemos algunos conceptos básicos sobre las imágenes y su representación en un ordenador. Algunos de estos conceptos serán recordados cuando se asocien a algún tipo de dato o acción en SDL.

Una imagen puede ser representada como un conjunto de puntos. Cada uno de esos puntos tienen un color asociado. El número de puntos que representa una imagen puede variar según la definición que queramos obtener. Con cuantos más puntos sea representada una imagen más calidad tendrá y más espacio necesitaremos para almacenarla. Este aumento de tamaño provocará un aumento de tiempo de computación para que pueda ser mostrada. Esto se traduce en un mayor consumo de espacio de disco y que las transferencias de datos que hace que la imagen se nos muestre en la pantalla de nuestro ordenador estén más cargadas al tener más datos que intercambiar. Por este motivo es fundamental trabajar con una resolución idónea, buscando el equilibrio entre espacio y calidad. En la imagen 5.2 podemos ver el efecto que produce un cambio de resolución en una imagen.

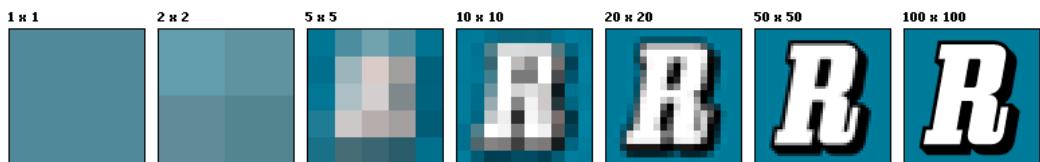


Figura 5.2: Variaciones en la resolución de una imagen

Estos puntos a los que hacemos referencia se conocen como **píxeles** y determinan la resolución de una imagen. Los monitores tienen una limitación en cuanto al número de píxeles que pueden mostrar. Esta limitación es un máximo ya que siempre podrá mostrar un número menor de píxeles. Cada uno de estos píxeles tendrá asociado un color para los que se destinará un determinado número de bits (**bits por píxel** o bpp) que determinarán la calidad, medida en profundidad de color, de la imagen. Los valores más comunes para el bpp son:

- 8 bits (256 colores)
 - Debemos usar una paleta para establecer los distintos 256 posibles colores.
- 16 bits (65,536 colores o Highcolor)
 - Existen distintas combinaciones para el uso de estos 16 bits. Las más comunes son:

5. El Subsistema de Video

- 5 bits para rojo, 6 bits para verde y 5 para azul. Se utiliza esta distribución porque el ojo humano distingue más colores verdes que otros.
- 4 bits para rojo, 4 bits para verde, 4 para azul y 4 para transparencias. Este es un modelo más equitativo.
- 24 bits (16,777,216 colores o Truecolor)
 - Este es el modelo RGB puro. Se destinan 8 bits para cada color que cubren todo el espectro.
- 32 bits (16,777,216 colores)
 - Utiliza RGB más 8 bits destinados a canales *alpha* o transparencias.

Depende del sistema en el que trabajamos y para el que vaya destinado la aplicación definirán unas resoluciones de trabajo con una profundidad de color bien determinada. Como puedes suponer cuanto mayor sea el número de bits por píxel mayor será el tamaño de la imagen. Este es otro factor a considerar ya que necesitaremos la cantidad de bits indicada por el bpp para almacenar un sólo píxel. SDL nos proporciona funciones que nos permiten establecer estos modos de video así como consultar su compatibilidad con el sistema.

Un concepto importante es el de framebuffer que aparecerá varias veces en el tutorial. El framebuffer no es más que la zona de memoria que se corresponde con la imagen que mostrará el sistema en pantalla. Cada píxel tendrá una concordancia exacta en pantalla en una determinada posición. El formato del framebuffer suele ser un vector para simplificar la escritura sobre él. Al ser un vector unidimensional, a la hora de querer modificar un píxel dado por unas coordenadas (x, y) tendremos que realizar una conversión de escala.

5.4. Subsistema de Vídeo

El subsistema de vídeo es la parte del sistema que nos permite interactuar con los dispositivos de vídeo. Generalmente el hardware de vídeo está compuesto por una tarjeta gráfica y un monitor o pantalla de visualización.

Existen numerosos tipos de tarjetas gráficas en el mercado pero todas ellas tienen características comunes que nos permiten trabajar con ellas independientemente del fabricante o del tipo de chip que monten dichas tarjetas. Poseen una unidad de procesamiento de gráficos y una memoria, ya sea compartida o dedicada, donde almacenar las estructuras o imágenes que se deben mostrar por pantalla.

El número de pantallas existentes en el mercado también es muy amplio pero no es algo que nos afecte a la hora de desarrollar un videojuego

5.4. Subsistema de Vídeo

actualmente a no ser que lo hagamos para un dispositivo específico. En otros tiempos conocer la cantidad de colores que podía manejar un subsistema de vídeo era fundamental para diseñar cualquier aplicación para dicho sistema. Hoy en día la mayoría de los ordenadores que tenemos en casa son capaces de manejar el conocido como color real o **true color** a unas resoluciones que son prácticamente estándar.

Los principales aspectos que tenemos que tener en cuenta de estos dispositivos son dos:



Figura 5.3: Tarjeta gráfica y monitor actuales

- La primera es la resolución de ambos dispositivos. Siempre deberemos trabajar a una resolución que sea admisible por el hardware en cuestión. Sería absurdo realizar un videojuego con un tamaño de ventana de 5000×5000 para un ordenador personal actual. Algo más avanzado el capítulo presentaremos las resoluciones más habituales.

El concepto de resolución de pantalla es bastante simple. Se trata del número de píxeles que puede ser mostrada por pantalla. Viene presentada, normalmente, por la ecuación $x \times y$ donde x hace referencia al número de columnas de píxeles e y indica el número de filas de píxeles. Puedes observar esta referencia en la figura 5.4

- La segunda es la posibilidad de crear superficies en la memoria de la tarjeta gráfica. Algunas tienen la memoria integradas en la placa base del ordenador comparten memoria con la CPU por lo que emularan este comportamiento aunque almacenen el contenido en la memoria principal del sistema. El poder almacenar en dicha memoria las superficies que

5. El Subsistema de Video

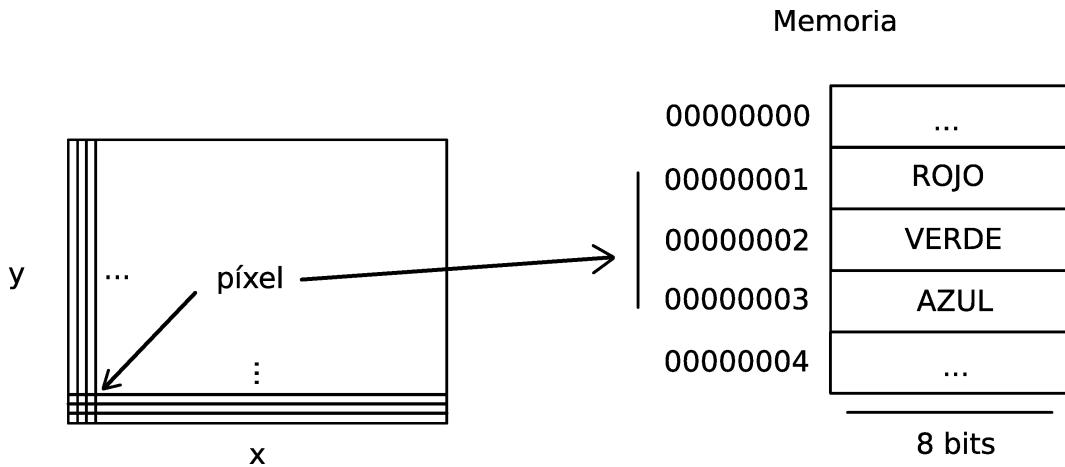


Figura 5.4: Representación de una imagen en pantalla de 24 bpp

queramos mostrar es muy importante ya que concederá un mayor rendimiento a nuestra aplicación. Esta memoria es conocida como RAMDAC.

5.4.1. Inicializando el Subsistema de Vídeo

Para poder utilizar el subsistema de vídeo debemos inicializarlo. Para ello, como ya hemos visto, debemos de pasar como parámetro a la función *SDL_Init()*, que es la encargada de poner en marcha la librería, la correspondiente macro del subsistema, en nuestro caso *SDL_INIT_VIDEO*.

Seguidamente tenemos que establecer el formato en el que vamos a mostrar la información gráfica, es decir, el modo de vídeo. Al establecer el modo de video creamos la superficie principal. Esta superficie será la única mostrada por pantalla, la más importante de la aplicación donde volcaremos todo aquello que querremos mostrar. La librería SDL proporciona una función que simplifica esta tarea. El prototipo de dicha función es la siguiente:

```
SDL_Surface *SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags);
```

Como puedes ver la función devuelve un puntero del tipo *SDL_Surface*. *Surface*, en catalán superficie, es el elemento básico en SDL para la manipulación de gráficos. En una superficie podremos dibujar, cargar gráficos y actualizar las áreas donde realicemos modificaciones, copiar o volcar otros gráficos... La superficie devuelta por esta función será la superficie principal que será en la que debamos colocar todo aquello que queramos que se muestre en nuestra pantalla. En el siguiente apartado trataremos las superficies con más detalle.

Esta función recibe varios parámetros de entrada:

- Parámetros que establecen el tamaño de la pantalla. El valor adecuado para estos parámetros depende principalmente del medio en el que vayamos a ejecutar nuestra aplicación. Tomar esta decisión es muy importante ya que deberemos relativizar todos los elementos del videojuego al tamaño de pantalla que hayamos definido.
 - *width*: Establece la anchura de la superficie principal o ventana de la aplicación.
 - *height*: Establece la altura de la superficie principal o ventana de la aplicación.
- *bpp*: Mediante este parámetro establecemos la profundidad de color. La profundidad de color no es más que la cantidad de bits de información que queremos dedicar a representar la información de color de cada píxel dentro de una imagen. Indicamos a SDL el número de bits por píxel (*bpp*: bits per pixel) con el que vamos a trabajar en nuestra aplicación. Los valores típicos son 8, 16, 24 y 32. El modo de 8 bits proporciona 256 colores mientras que 24 bits es color real. El número de colores que se pueden utilizar con los diferentes modos se obtienen con la fórmula 2^{bpp} . Los 8 bits entre los 24 de color real y 32 se destinan a canales alfa. Los canales alfa es un tipo de información especial de las imágenes que trataremos en uno de los subapartados.
- *flags*: Es un campo de 32 bits entero, sin signo, que nos permite establecer diferentes características del modo de vídeo que vamos a configurar. Sus posibles valores son:
 - **SDL_SWSURFACE**: La superficie de vídeo a crear se almacenará en la memoria principal del ordenador. Si no se especifica nada las superficies serán creadas en esta memoria por defecto.
 - **SDL_HWSURFACE**: La superficie de vídeo a crear se almacenará en la memoria de vídeo.
 - **SDL_ASYNCBLIT**: Establece el modo de blit asíncrono. Es decir, habilita el uso de transferencia asíncrona de bloques a la superficie de visualización. Sirve para mejorar el rendimiento en computadoras con más de un procesador. Esta opción puede provocar la disminución de rendimiento en máquinas monoprocesador. El concepto de blit será estudiado con posterioridad.

5. El Subsistema de Video

- **SDL_ANYFORMAT:** Esta bandera debe ser usada cuando creamos la superficie principal de nuestra aplicación en una ventana dentro de un gestor de ventanas. En caso de que el modo que queremos inicializar no esté disponible inicializa el mejor modo posible de profundidad de color (bpp) dentro de los disponibles para nuestra aplicación, pero puede ocurrir que el mejor modo para SDL no sea el mejor para nuestro videojuego. Es un método de adaptación al entorno donde se ejecute nuestro proyecto. Si la profundidad de color de nuestra aplicación no está disponible en el entorno en el que estamos trabajando SDL emulará una superficie con sombras (shadow surface). Utilizando esta bandera evitaremos este efecto ya que provoca que SDL utilice la superficie de vídeo independientemente de la profundidad.
- **SDL_HWPALLETTE:** Proporciona a SDL acceso exclusivo a la paleta de color. Si no se usa esta bandera no siempre se podrán obtener los colores utilizando las funciones *SDL_SetColors()* y *SDL_SetPalette()* de manejo de paleta de colores. Es obvio, si no se tiene acceso exclusivo puede que en un momento dado no estén disponibles.
- **SDL_DOUBLEBUF:** Activa el uso de doble búffer. Esta opción sólo es válida junto a **SDL_HWSURFACE**. Para actualizar la pantalla utilizaremos la función *SDL_Flip()* que nos permitirá intercambiar los búfferes. Este concepto será explicado dentro de unas líneas. El en caso de no estar activada esta opción la función *SDL_Flip()* realizará una llamada a *SDL_UpdateRect()* que actualizará toda la superficie que reciba como parámetro. El doble búffer evita la aparición de efectos no deseados provocados por la carga de proceso del sistema en un momento dado. Este concepto será estudiado con más detalle en este mismo capítulo.
- **SDL_FULLSCREEN:** Intenta activar el modo a pantalla completa. Sin este argumento la superficie principal será mostrada en una ventana del administrador de ventanas del sistema.
- **SDL_OPENGL:** Crea un contexto OpenGL en la superficie creada. Se deben establecer los atributos OpenGL previamente (*SDL_GL_SetAttribute()*).
- **SDL_OPENGLBLIT:** Crea un contexto OpenGL en la superficie creada, permitiendo que SDL haga el renderizado de dos dimensiones, es decir permite operaciones de volcado normales de SDL.

- **SDL_RESIZABLE**: Permite que la ventana dónde hemos creado la superficie pueda cambiar de tamaño. Desaconsejable en muchos casos. Si el usuario cambia la ventana de tamaño se generará el evento `SDL_VIDEORESIZE` y se puede llamar de nuevo a la función `SDL_SetVideoMode()` para cambiar al nuevo tamaño la superficie con la que estamos trabajando. Un cambio de tamaño en la ventana de nuestro videojuego puede suponer tener que actualizar mucho de los cálculos posicionales realizados.
- **SDL_NOFRAME**: Si creamos la superficie en una ventana esta opción nos ofrece la posibilidad de eliminar el borde de la misma, así como la barra de título y otras decoraciones. Cuando establecemos el modo a pantalla completa se activa esta bandera automáticamente.

Para poder combinar varios de los valores compatibles de este parámetro debe usarse el operador `|`. Estas constantes están definidas de a nivel de bit realizando una operación lógica `OR` o suma lógica para combinarlas.

El último parámetro de la función modifica el campo *flags* de las variables de tipo estructura `SDL_Surface`. Esta es la única forma de hacerlo ya que este campo, como veremos en el apartado de estudio del manejo de superficies, es de sólo de lectura.

Si no se consigue establecer el modo de vídeo la función devolverá el valor `NULL`, lo que nos permitirá comprobar el éxito de la llamada a función. Dependiendo del tipo de videojuego que estemos desarrollando deberemos decidir si utilizar un modo de ventana en el escritorio o bien optar por la pantalla completa.

Como puedes observar en el estudio de los parámetros, en SDL, es tan fácil crear una aplicación a pantalla completa como en una ventana, ya que se utiliza la misma función `SDL_SetVideoMode()` para crear los dos tipos de aplicaciones.

En el caso de optar por el modo de ventana es una buena práctica definir la profundidad de color a 0 y activar la bandera `SDL_ANYFORMAT` lo que permitirá a SDL integrar nuestra aplicación en el entorno de escritorio con el número de bits por píxel que tenga configurado dicho entorno, aunque siempre podamos forzar el modo que más nos convenga. En el caso de desarrollar una aplicación diseñada a pantalla completa deberemos de especificar todos los parámetros con las características que hayamos decidido dotar a nuestra aplicación.

5. El Subsistema de Video

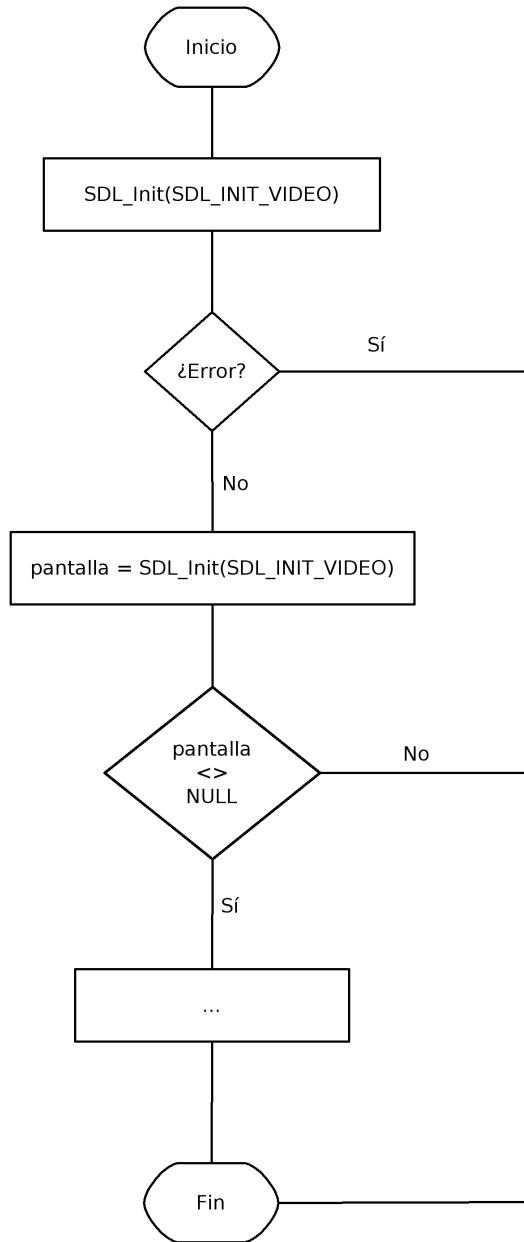


Figura 5.5: Diagrama de inicialización del subsistema de video

5.4.1.1. Ejemplo 1

Con el siguiente ejemplo vamos a comprobar la compatibilidad de nuestro sistema con las distintas opciones que admite la configuración del modo de vídeo. En este ejemplo solicitamos a SDL que establezca distintos modos de vídeo para saber si nuestro sistema es compatible con ellos. El elegir uno u otro dependerá de la aplicación a desarrollar.

En todos ellos hemos configurado una resolución de 640 píxeles de ancho

5.4. Subsistema de Vídeo

por 480 de alto con una profundidad de color de 24 bits conocida como true color o color verdadero. El fichero de cabecera es el siguiente:

```
1 ;_____
2 // Listado: compatibilidad_video.h
3 ;
4 ;#ifndef _COMPATIBILIDAD SDL_
5 ;#define _COMPATIBILIDAD SDL_
6 ;
7 ;// Precondición: Subsistema de video de SDL activo.
8 ;// Postcondición: Realiza comprobaciones de compatibilidad de
9 ;// SDL con varias opciones de inicialización.
10 ;
11 ;void compatibilidad_video_sdl(int w, int h, int bpp);
12 ;
13 ;
14 ;#endif
;
```

El código de la implementación de la función *compatibilidad_video_sdl()* es

```
1 ;_____
2 // Listado: compatibilidad_video.c
3 ;// Comprueba la compatibilidad del sistema con los modos de video de libsdl
4 ;// mediante el método ensayo-error
5 ;
6 ;#include <stdio.h>
7 ;#include <SDL/SDL.h>
8 ;#include "compatibilidad_video.h"
9 ;
10 ;void compatibilidad_video_sdl(int w, int h, int bpp)
11 ;{
12 ;
13 ;    // Nuestro "rectángulo" gráfico con la información de video a mostrar;
14 ;
15 ;    SDL_Surface *pantalla;
16 ;
17 ;    // Vamos a probar los diferentes parametros de SetVideoMode
18 ;
19 ;    // Almacenando la superficie en memoria principal a w x h x bpp
20 ;
21 ;    pantalla = SDL_SetVideoMode(w, h, bpp, SDL_SWSURFACE);
22 ;
23 ;    if(pantalla == NULL)
24 ;        printf("SDL_SWSURFACE %d x %d x %d no compatible. Error: %s\n",
25 ;               w, h, bpp, SDL_GetError());
26 ;    else
27 ;        printf("Compatible con SDL_SWSURFACE %d x %d x %d\n", w, h, bpp);
28 ;
29 ;    // Almacenando la superficie en memoria de video a w x h x bpp
30 ;
31 ;    pantalla = SDL_SetVideoMode(w, h, bpp, SDL_HWSURFACE);
```

5. El Subsistema de Video

```
32 ;
33 ;     if(pantalla == NULL)
34 ;         printf("SDL_HWSURFACE %d x %d x %d no compatible. Error: %s\n",
35 ;                 w, h, bpp, SDL_GetError());
36 ;     else {
37 ;
38 ;         printf("Compatible con SDL_HWSURFACE %d x %d x %d\n", w, h, bpp);
39 ;
40 ;         // ¿Es compatible con el doble búfer? Sólo con HWSURFACE
41 ;         pantalla = SDL_SetVideoMode(w, h, bpp, SDL_HWSURFACE | SDL_DOUBLEBUF);
42 ;         if(pantalla == NULL)
43 ;             printf("SDL_DOUBLEBUF %d x %d x %d no compatible. Error: %s\n",
44 ;                   w, h, bpp, SDL_GetError());
45 ;         else
46 ;             printf("Compatible con SDL_DOUBLEBUF %d x %d x %d\n", w, h, bpp);
47 ;
48 ;
49 ;         // Blit asíncrono para mejorar rendimiento en máquinas multiprocesador
50 ;         pantalla = SDL_SetVideoMode(w, h, bpp, SDL_ASYNCBLIT);
51 ;         if(pantalla == NULL)
52 ;             printf("SDL_ASYNCBLIT %d x %d x %d no compatible. Error: %s\n",
53 ;                   w, h, bpp, SDL_GetError());
54 ;         else
55 ;             printf("Compatible con SDL_ASYNCBLIT %d x %d x %d\n", w, h, bpp);
56 ;
57 ;         // Forzamos los bpp en modo ventana
58 ;         pantalla = SDL_SetVideoMode(w, h, bpp, SDL_ANYFORMAT);
59 ;         if(pantalla == NULL)
60 ;             printf("SDL_ANYFORMAT %d x %d x %d no compatible. Error: %s\n",
61 ;                   w, h, bpp, SDL_GetError());
62 ;         else
63 ;             printf("Compatible con SDL_ANYFORMAT %d x %d x %d\n", w, h, bpp);
64 ;
65 ;
66 ;         // Acceso exclusivo a la paleta de color
67 ;         pantalla = SDL_SetVideoMode(w, h, bpp, SDL_HWPALLETTE);
68 ;         if(pantalla == NULL)
69 ;             printf("SDL_HWPALLETTE %d x %d x %d no compatible. Error: %s\n",
70 ;                   w, h, bpp, SDL_GetError());
71 ;         else
72 ;             printf("Compatible con SDL_HWPALLETTE %d x %d x %d\n", w, h, bpp);
73 ;
74 ;
75 ;         // Modo a pantalla completa
76 ;         pantalla = SDL_SetVideoMode(w, h, bpp, SDL_FULLSCREEN);
77 ;         if(pantalla == NULL)
78 ;             printf("SDL_FULLSCREEN %d x %d x %d no compatible. Error: %s\n",
79 ;                   w, h, bpp, SDL_GetError());
80 ;         else
81 ;             printf("Compatible con SDL_FULLSCREEN %d x %d x %d\n", w, h, bpp);
82 ;
83 ;
84 ;         // Crear un contexto OpenGL en la superficie
```

5.4. Subsistema de Vídeo

```
85 ;     pantalla = SDL_SetVideoMode(w, h, bpp, SDL_OPENGL);
86 ;     if(pantalla == NULL)
87 ;         printf("SDL_OPENGL %d x %d x %d no compatible. Error: %s\n",
88 ;                w, h, bpp, SDL_GetError());
89 ;     else
90 ;         printf("Compatible con SDL_OPENGL %d x %d x %d\n", w, h, bpp);
91 ;
92 ;
93 ; // Crear un contexto OpenGL en la superficie y
94 ; // permitir renderizado opengl
95 ; pantalla = SDL_SetVideoMode(w, h, bpp, SDL_OPENGLBLIT);
96 ; if(pantalla == NULL)
97 ;     printf("SDL_OPENGLBLIT %d x %d x %d no compatible. Error: %s\n",
98 ;            w, h, bpp, SDL_GetError());
99 ; else
100 ;     printf("Compatible con SDL_OPENGLBLIT %d x %d x %d\n", w, h, bpp);
101 ;
102 ; // Permite que la superficie principal pueda cambiarsele el tamaño
103 ; pantalla = SDL_SetVideoMode(w, h, bpp, SDL_RESIZABLE);
104 ; if(pantalla == NULL)
105 ;     printf("SDL_RESIZABLE %d x %d x %d no compatible. Error: %s\n",
106 ;            w, h, bpp, SDL_GetError());
107 ; else
108 ;     printf("Compatible con SDL_RESIZABLE %d x %d x %d\n", w, h, bpp);
109 ;
110 ; pantalla = SDL_SetVideoMode(w, h, bpp, SDL_NOFRAME);
111 ; if(pantalla == NULL)
112 ;     printf("SDL_NOFRAME %d x %d x %d no compatible. Error: %s\n",
113 ;            w, h, bpp, SDL_GetError());
114 ; else
115 ;     printf("Compatible con SDL_NOFRAME %d x %d x %d\n", w, h, bpp);
116 ;
117 ;
118 ;};
```

La lógica del listado se basa en el método ensayo-error. Vamos probando cada una de las opciones de inicialización de SDL y comprobamos si se ha realizado correctamente o bien existe algún problema. Esto nos permitirá conocer mediante fuerza bruta que opciones son compatibles con nuestro sistema. Es un método incremental, partiendo de los modos más básicos, hasta los que incluyen las opciones más complejas.

En el listado se trata de definir una variable de tipo *SDL_Surface* pantalla que será la que inicialicemos con los distintos modos. Esta variable irá almacenando las superficies devueltas por los diferentes modos de inicialización y si en algún momento contiene el valor **NULL** sabremos que dicho modo no es compatible con nuestro sistema.

Una vez estudiadas las diferentes estructuras manejadas en SDL se procederá a una elaboración más lógica de este primer ejemplo. Como puedes

5. El Subsistema de Video

obsevar la aplicación está íntegramente escrita en C.

Como primer ejercicio del tutorial te propongo que transcribas el programa del ejemplo a C++ utilizando todas las características posibles de las que este te proporciona. Es decir, utiliza `iostream` en vez de `stdio.h`, `cout` en vez de `printf...` así podrás practicar un poco de C++ básico antes de entrar con ejemplos más complejos.

5.4.2. `SDL_VideoInfo`. La Información del Subsistema de Video

Es una buena práctica antes de lanzar un videojuego consultar al sistema por el mejor modo de vídeo disponible y mostrar todos nuestros gráficos debidamente escalados por hardware manteniendo un aspecto correcto. Desafortunadamente, aunque es posible hacerlo, es una tarea trivial ni eficiente. Seguramente en unos cuantos años la historia cambiará ya que actualmente hay grandes grupos de desarrollo realizando algoritmos y formatos de imagen que pueden ser escalados sin un consumo de recursos relevante con unos resultados más que aceptables. El escalado de mapas de bits siempre produce una pérdida de calidad.

Como habrás podido comprobar la forma de probar la compatibilidad del sistema de vídeo de nuestro primer ejemplo no nos ofrece información mas allá de la misma prueba. Ahora que conoces las distintas opciones con las que se puede establecer el modo de vídeo te vamos a presentar la estructura que almacena las capacidades del sistema, más concretamente, las características que soporta nuestra tarjeta de vídeo. Esto nos será muy útil a la hora de desarrollar nuestras aplicaciones aunque en muchos de nuestros ejemplos al ser básicos optaremos por usar una configuración compatible con casi la totalidad de los sistemas.

SDL posee una función que nos permite obtener información del sistema de vídeo sobre el que estamos trabajando, esta información es de solo lectura, claro está. Si todavía no hemos inicializado un modo de vídeo nos devuelve el mejor modo disponible a criterio de SDL. La función es la siguiente:

```
SDL_VideoInfo *SDL_GetVideoInfo(void);
```

Como puedes observar no recibe parámetros, como por otra parte es lógico. Devuelve un puntero a una estructura `SDL_VideoInfo` que contiene información de sólo lectura. Este puntero no necesita ser mantenido ya que de eso se encarga SDL. La estructura `SDL_VideoInfo` está definida de la siguiente manera:

```

1 ;_____
1 ;typedef struct {
2 ;    Uint32 hw_available:1;
3 ;    Uint32 wm_available:1;
4 ;    Uint32 blit_hw:1;
5 ;    Uint32 blit_hw_CC:1;
6 ;    Uint32 blit_sw:1;
7 ;    Uint32 blit_sw_CC:1;
8 ;    Uint32 blit_sw_A:1;
9 ;    Uint32 blit_fill;
10 ;   Uint32 video_mem;
11 ;   SDL_PixelFormat *vfmt;
12 ;} SDL_VideoInfo;
;
```

En esta estructura es donde se almacena toda la información necesaria para conocer las capacidades de nuestro sistema respecto al hardware de video. Es fundamental conocer el significado de cada uno de estos campos. La mayoría de los miembros de esta estructura son banderas en formato de bit funcionando como una variable booleana, donde 0 denota que la característica no está disponible y 1 representa que nuestro sistema puede ofrecer dicha característica. Como puedes observar los nombres de los componentes de la estructura son algo crípticos por lo que vamos a estudiar que significa cada uno de ellos:

- *hw_available*: Especifica si podemos almacenar superficies en la memoria de la tarjeta de vídeo.
- *wm_available*: Indica si hay una gestor de ventanas disponible. El manejador de ventanas es otro subsistema dentro de SDL que se encarga de establecer la configuración para la ventana, como puede ser el modo aventanado (windowed) o a pantalla completa (fullscreen)
- *blit_hw*: Muestra si el blitting hardware - hardware está acelerado. Es decir, especifica si está disponible la aceleración para hacer volcados entre superficies que se encuentran en la memoria de vídeo.
- *blit_hw_CC*: Especifica si el blitting con transparencias hardware - hardware está acelerado, es decir como *blit_hw* pero con transparencias.
- *blit_hw_A*: Indica si el blitting con alpha hardware - hardware está acelerado, es decir como *blit_hw* pero con superficies con canales alpha.
- *blit_sw*: Indica si en blitting software - hardware está acelerado. Especifica si está disponible la aceleración para hacer volcados de la memoria principal a la memoria de vídeo.
- *blit_sw_CC*: Indica si el blitting con transparencias software - hardware está acelerado. Idéntico a *blit_sw* pero con partes transparentes.

5. El Subsistema de Video

- *blit_sw_A*: Indica si está acelerado el blitting con alpha software - hardware. Cómo *blit_sw* pero con superficies alpha.
- *blit_fill*: Indica si está acelerado el relleno de color.
- *video_mem*: Cantidad de memoria total en Kilobytes.
- *vfmt*: Puntero a la estructura que contiene el píxel format del sistema gráfico. Esta estructura *SDL_PixelFormat* contiene el formato de pixel de la tarjeta de vídeo. Llamando a *SDL_GetVideoInfo()* antes de *SDL_SetVideoMode()* conseguiremos obtener en dicha estructura el mejor modo de vídeo disponible.

Para recordar que significa cada uno de los crípticos campos puedes utilizar la siguiente técnica. *hw* y *sw* significa simplemente hardware y software, *CC* abrevia color clave (*color key*) y *A* representa alpha.

Ahora bien puede ser interesante conocer el nombre del driver de vídeo que estamos utilizando. SDL proporciona una función que nos permite conocer el driver de vídeo instalado en nuestro sistema. El prototipo de la función es el siguiente:

```
char *SDL_VideoDriverName(char *namebuf, int maxlen);
```

Esta función recibe dos parámetros, el primero es un puntero a una cadena de caracteres donde se guardará el nombre del driver. El segundo corresponde a la cantidad máxima de caracteres que queremos utilizar para almacenar dicho nombre, incluido el carácter \0. Si no se inicia previamente el subsistema de vídeo la función devuelve **NULL**, de lo contrario almacena el nombre del driver.

Para establecer el modo de vídeo idóneo para ejecutar una aplicación desarrollada con SDL en un sistema nos puede resultar muy útil la siguiente función:

```
SDL_Rect **SDL_ListModes(SDL_PixelFormat *format, Uint32 flags);
```

Esta función devuelve una lista en forma de vector de elementos *SDL_Rect**. Contiene los modos de vídeo ordenados de mayor a menor, empezando por el que tiene un mayor número de píxeles llegando hasta el más básico. Los parámetros que recibe son, primero, el formato de pixel de vídeo que podemos consultar mediante el campo *vmft* de la estructura a la que apunta el valor devuelto por la función *SDL_GetVideoInfo()* o bien puede ser **NULL** si no queremos especificarlo. El segundo parámetro es una combinación de banderas como las estudiadas anteriormente en la definición de tipo de la superficie *SDL_Surface*. De la misma forma la función devuelve **NULL** si no hay modos disponibles para un formato de píxeles en particular o -1 si cualquier modo es válido.

Lo más normal es que diseñemos nuestro videojuego para que se ejecute a pantalla completa. Una vez tengas decidida la resolución de pantalla y la profundidad de color que quieras utilizar podemos comprobar si nuestro sistema soporta este modo de vídeo en particular usando la función:

```
int SDL_VideoModeOK(int width, int height, int bpp, Uint32 flags);
```

Esta función recibe como parámetros las dimensiones del modo a consultar width (ancho) y height (alto), el número de bits por pixel y las banderas de las características del modo. Si el modo no está soportado por el sistema devuelve el valor 0, de lo contrario devuelve el número de bits por píxel. Es una buena práctica comprobar si un modo está disponible antes de inicializarlo.

Vamos a realizar una pequeña aplicación que nos muestre con qué técnicas y capacidades es compatible nuestro sistema. Esto es muy importante porque nos permitirá trabajar de una forma a la hora de desarrollar los videojuegos.

5.4.2.1. Ejemplo 2

Veamos la definición de la función que nos permite trabajar con la información del hardware de video:

```
;  
1 // Listado: sdl_videoinfo.h  
2 // Función que muestra la compatibilidad del sistema con las tecnologías  
3 // utilizadas por la SDL  
4 ;  
5 ifndef _SDL_VIDEOINFO_  
6 define _SDL_VIDEOINFO_  
7 ;  
8 // Precondición: El subsistema de video SDL debe estar activado.  
9 // Postcondición: Obtiene información de las estructuras de SDL video info  
10 // y la muestra por pantalla.  
11 ;  
12 void sdl_videoinfo(void);  
13 ;  
14 endif  
;
```

Aquí puedes observar la implementación de la función:

```
;  
1 // Listado: sdl_videoinfo  
2 //  
3 // Implementación de la función sdl_videoinfo()  
4 ;  
5 #include <SDL/SDL.h>  
6 #include "sdl_videoinfo.h"  
7 ;  
8 void sdl_videoinfo(void)  
9 {
```

5. El Subsistema de Video

```
10 ;
11 ;     const SDL_VideoInfo *propiedades;
12 ;     SDL_Surface *pantalla;
13 ;     SDL_Rect **modos;
14 ;
15 ;     //Variables auxiliares
16 ;     char driver[20];
17 ;     int maxlen = 20;
18 ;     int i = 0;
19 ;
20 ;     // Obtenemos la información del sistema de video
21 ;     propiedades = SDL_GetVideoInfo();
22 ;     if(propiedades == NULL) {
23 ;         fprintf(stderr, "No se pudo obtener la información %s\n",
24 ;                 SDL_GetError());
25 ;         exit(1);
26 ;     }
27 ;
28 ;     // Obtenemos los modos de video disponibles
29 ;     modos = SDL_ListModes(NULL, SDL_HWSURFACE);
30 ;
31 ;     printf("\n\n == MODOS DE VIDEO DISPONIBLES == \n");
32 ;
33 ;     // Comprobamos que métodos están disponibles
34 ;     if(modos == (SDL_Rect **)0)
35 ;         printf("No existen modos disponibles \n");
36 ;     else if(modos == (SDL_Rect **) -1)
37 ;         printf("Todos los modos disponibles \n");
38 ;     else {
39 ;         printf("Lista de modos disponibles\n");
40 ;         for(i = 0; modos[i]; i++)
41 ;             printf("%d x %d\n", modos[i]->w, modos[i]->h);
42 ;     }
43 ;
44 ;     // Comprobamos que el modo a seleccionar sea compatible
45 ;     if(SDL_VideoModeOK(640, 480, 24, SDL_SWSURFACE) == 0) {
46 ;         fprintf(stderr, "Modo no soportado: %s\n", SDL_GetError());
47 ;         exit(1);
48 ;     }
49 ;
50 ;
51 ;     // Una vez comprobado establecemos el modo de video
52 ;     pantalla = SDL_SetVideoMode(640, 480, 24, SDL_SWSURFACE);
53 ;     if(pantalla == NULL)
54 ;         printf("SDL_SWSURFACE 640x480x24 no compatible. Error: %s\n",
55 ;               SDL_GetError());
56 ;
57 ;
58 ;     // Obtenemos información del driver de video
59 ;     printf("\n\n == INFORMACIÓN DRIVER VIDEO == \n");
60 ;     SDL_VideoDriverName(driver, maxlen);
61 ;
62 ;     if(driver == NULL) {
```

5.4. Subsistema de Vídeo

```
63 ;         fprintf(stderr, "No se puede obtener nombre driver video\n");
64 ;         exit(1);
65 ;
66 ;
67 ;         printf("Driver: %s\n", driver);
68 ;
69 ;
70 ;         // Obtenemos información sobre las capacidades de nuestro
71 ;         // sistema respecto a SDL
72 ;         printf("\n == INFORMACION SDL_INFO == \n\n");
73 ;         if(propiedades->hw_available == 1)
74 ;             printf("HW Compatible\n");
75 ;         else
76 ;             printf("HW no compatible\n");
77 ;
78 ;         if(propiedades->wm_available == 1)
79 ;             printf("Hay un manejador de ventanas disponible\n");
80 ;         else
81 ;             printf("No hay un manejador de ventanas disponible\n");
82 ;
83 ;         if(propiedades->blit_hw == 1)
84 ;             printf("El blitting hardware - hardware está acelerado\n");
85 ;         else
86 ;             printf("El blitting hardware - hardware NO está acelerado\n");
87 ;
88 ;         if(propiedades->blit_hw_CC == 1) {
89 ;             printf("El blitting con transparencias hardware - hardware ");
90 ;             printf("está acelerado\n");
91 ;         }
92 ;         else {
93 ;             printf("El blitting con transparencias hardware - hardware ");
94 ;             printf("NO está acelerado\n");
95 ;         }
96 ;
97 ;         if(propiedades->blit_sw == 1)
98 ;             printf("El blitting software - \
99 ;                   hardware está acelerado.\n");
100 ;        else
101 ;            printf("El blitting software - hardware NO está acelerado. \n");
102 ;
103 ;        if(propiedades->blit_sw_CC == 1) {
104 ;            printf("El blitting software - hardware con transparencias");
105 ;            printf(" está acelerado\n");
106 ;        }
107 ;        else {
108 ;            printf("El blitting software - hardware con transparencias");
109 ;            printf(" NO está acelerado\n");
110 ;        }
111 ;
112 ;        if(propiedades->blit_sw_A == 1)
113 ;            printf("El blitting software - \
114 ;                  hardware con alpha está acelerado\n");
115 ;        else
```

5. El Subsistema de Video

```
116 ;         printf("El blitting software - \
117 ;                     hardware con alpha NO está acelerado\n");
118 ;
119 ;     if(propiedades->blit_fill == 1)
120 ;         printf("El rellenado de color está acelerado\n");
121 ;     else
122 ;         printf("El rellenado de color NO está acelerado\n");
123 ;
124 ;     printf("La memoria de video tiene %f MB\n", (float) propiedades->video_mem);
125 ;
126 ;}
```

Para probar las funciones de estos ejemplos creamos un sencillo programa principal:

```
;_____
1 ;// Listado: main.c
2 ;//
3 ;// Programa de prueba, compatibilidad
4 ;// del sistema e información SDL video_info
5 ;
6 ;#include <stdio.h>
7 ;#include <SDL/SDL.h>
8 ;#include "compatibilidad_video.h"
9 ;#include "sdl_videoinfo.h"
10 ;
11 ;
12 ;void datos_configuracion(int *w, int *h, int *bpp);
13 ;
14 ;
15 ;int main()
16 ;{
17 ;    int opcion;
18 ;
19 ;    int h, w, bpp;
20 ;
21 ;    atexit(SDL_Quit);
22 ;
23 ;    do {
24 ;
25 ;        // Iniciamos SDL
26 ;        if(SDL_Init(SDL_INIT_VIDEO) < 0){
27 ;            fprintf(stderr, " No se pudo iniciar SDL: %s\n",
28 ;                    SDL_GetError());
29 ;            exit(1);
30 ;        }
31 ;
32 ;
33 ;        printf("\n 1.Compatibilidad Video SDL\n");
34 ;        printf(" 2.Información SDL_VideoInfo.\n");
35 ;        printf(" 3.Salir\n");
36 ;        printf("Elija una opción: ");
37 ;        scanf("%d", &opcion);
```

```

38 ;
39 ;     switch(opcion) {
40 ;
41 ;         case 1:
42 ;             datos_configuracion(&w, &h, &bpp);
43 ;             compatibilidad_video_sdl(w, h, bpp);
44 ;             SDL_Quit();
45 ;             break;
46 ;
47 ;         case 2: sdl_videoinfo(); SDL_Quit(); break;
48 ;
49 ;         case 3: SDL_Quit(); break;
50 ;
51 ;         default: printf("\nOpción no válida\n");
52 ;
53 ;     }
54 ;
55 ; } while(opcion != 3);
56 ;
57 ;
58 ; return 0;
59 ;}
60 ;
61 ;// Función para inicializar los datos para comprobar la compatibilidad
62 ;
63 ;void datos_configuracion(int *w, int *h, int *bpp) {
64 ;
65 ;     printf("\nIntroduce la anchura en píxeles de la pantalla: ");
66 ;     scanf("%d", w);
67 ;
68 ;     printf("Introduce la altura en píxeles de la pantalla: ");
69 ;     scanf("%d", h);
70 ;
71 ;     printf("Introduce la profundidad de color: ");
72 ;     scanf("%d", bpp);
73 ;
74 ;     printf("\n"); // Para el formato
75 ;
76 ;}
;
```

En este listado obtenemos las propiedades de nuestro hardware de vídeo mediante *SDL_GetVideoInfo()*. Seguidamente obtenemos una lista de los modos compatibles e inicializamos el modo de vídeo a uno de los más simples disponibles comprobando previamente que sea soportado mediante *SDL_VideoModeOK()*. Esta comprobación, como puedes observar, es redundante ya que tenemos una lista de los modos disponibles.

Obtenemos el nombre del driver que estamos utilizando mediante *SDL_VideoDriverName()* y lo mostramos por pantalla, así como un estudio de la estructura que obtuvimos con las propiedades del hardware de vídeo. Mostramos en cada caso si es compatible con una determinada característica o

5. El Subsistema de Video

no. Para terminar obtenemos el número de megabytes del que dispone nuestra tarjeta de vídeo.

Puede que todavía no hayas comprendido alguno de los conceptos que hemos expuesto en el temario. Era necesario que conocieses las posibilidades que ofrece el hardware de video antes de estudiar estos conceptos ya que te harán tener una vista más amplia sobre los aspectos que estamos trabajando. A continuación presentamos estos conceptos.

5.5. Conceptos y Estructuras Fundamentales

En esta sección presentamos diferentes conceptos y estructuras que son de fundamental conocimiento para el seguimiento del tutorial. En SDL existen siete estructuras con un formato bien definido con un propósito específico. En este apartado vamos a estudiar cada una de ellas. Por supuesto no todas estas estructuras tienen la misma frecuencia de uso, ahondaremos en las más utilizadas.

El concepto se superficie es el concepto fundamental de esta sección. Aunque sería lógico que lo presentasemos en primer lugar hemos decidido tener primero una visión de todos los conceptos que tiene como base el concepto de superficie.

5.5.1. *SDL_Rect*. El Área Rectangular

En el subsistema de vídeo de SDL, siendo de naturaleza de dos dimensiones, se trabaja copiando bloques rectangulares de píxeles de una superficie a otra. Esta es una forma común de trabajar con imágenes que SDL ha adoptado como buena y nos ofrece su implementación.

Aunque la tendencia va cambiando los videojuegos son creados mediante imágenes de tipo mapa de bit mostrados en una ventana, o a pantalla completa, envueltas por una área rectangular de su mismo tamaño. Si tenemos un dibujo de un personaje trabajaremos con una superficie rectangular que rodea al personaje tan grande como sea el lienzo donde esté almacenada la imagen.

Como es evidente necesitamos una estructura que contenga este área rectangular y toda librería orientada al diseño de videojuegos en dos dimensiones proporciona dicha estructura. En SDL la estructura encargada de almacenar dicha área es *SDL_Rect*.

SDL_Rect es un tipo definido por una estructura de la siguiente forma:

```
1 ;  
1 typedef struct {
```

5.5. Conceptos y Estructuras Fundamentales

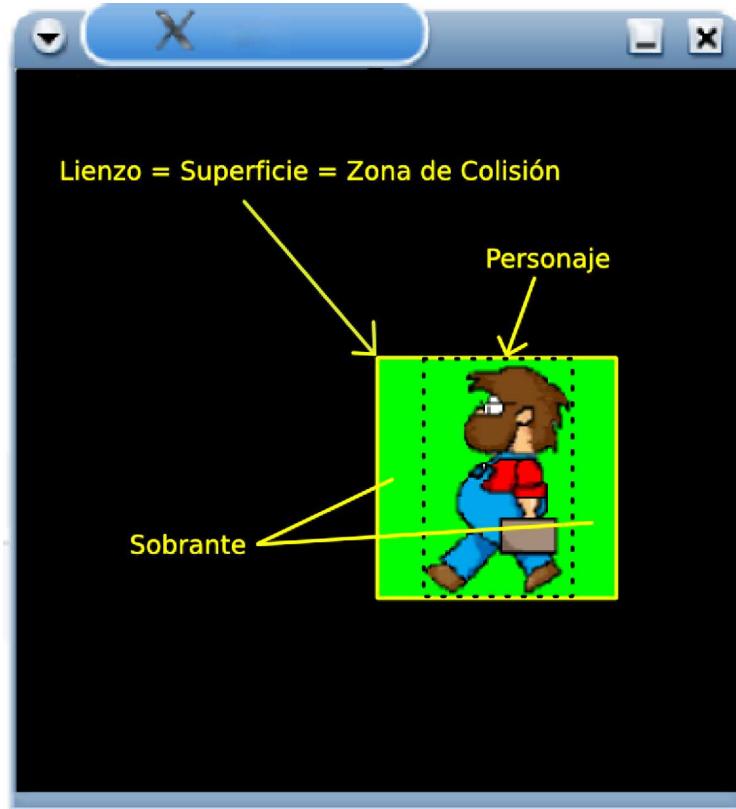


Figura 5.6: Superficie Rectangular

```
2 ;     Sint16 x, y;  
3 ;     Uint16 w, h;  
4 ;} SDL_Rect;  
;
```

Las variables x e y hacen referencia a la posición en píxeles sobre la pantalla o superficie principal donde queremos posicionar nuestra superficie. Son variables enteras con signo ya que la posición, en un momento dado, podría ser negativa y estar fuera de la pantalla visible. El rango de estas variables llega desde $-32,768$ hasta $32,767$ lo que es más que suficiente para el tipo de dato que representa esta variable. La posición (x, y) tiene como referencia la esquina superior izquierda de la superficie principal que es la que comandará la posición del mismo en la ventana o pantalla.

En el sistema de coordenadas que utiliza SDL es diferente al sistema cartesiano de coordenadas que estamos acostumbrados a utilizar. La posición $(0, 0)$ se establece en la esquina superior izquierda de la superficie o pantalla principal. Cualquier valor positivo sobre x o y supone un desplazamiento hacia la derecha, en el caso de x , o hacia abajo, en el caso de y en la superficie en cuestión. Un valor negativo representa una posición fuera de la pantalla. Puedes observar las diferencias entre estos dos sistemas de coordenadas en la figura 5.7

5. El Subsistema de Video

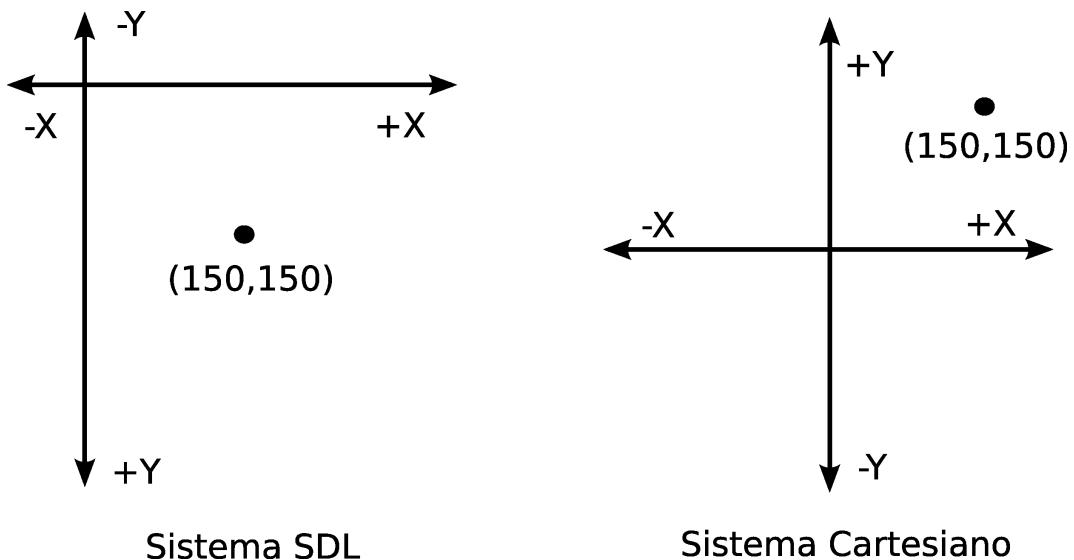


Figura 5.7: Sistemas de Coordenadas

Las variables w y h almacenan el ancho y alto en píxeles de la superficie a la que envuelve la superficie rectangular. Estas variables son enteras sin signo, de 16 bits, ya que el tamaño de una superficie no puede ser negativo. El rango de estas variables se comprende entre 0 y 65,535 más que suficiente, de nuevo, para caracterizar el ancho y alto de una superficie rectangular.

El almacenar el ancho y el alto de la superficie facilita la tarea de mover una superficie en 2D. En otras APIs se almacena la posición de la esquina superior izquierda, como en SDL, y la esquina inferior derecha, en vez de la anchura y altura, lo que provoca que para mover un rectángulo por la pantalla hay que actualizar cuatro valores en vez de dos. El dotar de movimiento a una superficie es una tarea muy común en la programación de videojuegos lo que provoca que actualizar cuatro valores, en vez de dos, sea una sobrecarga excesiva e innecesaria para el sistema.

Veamos algunas cuestiones interesantes. ¿Cómo saber qué puntos están dentro de un rectángulo? Seguramente en algún momento dentro del código que desarrolles necesitarás saber si cierto punto está contenido dentro de un rectángulo. Como puedes observar en la figura 5.8 nuestro rectángulo abarca la superficie comprendida desde la posición (x, y) hasta la $(x + w, y + h)$. Las otras dos esquinas vienen dadas por $(x, y + h)$ la esquina que se encuentra abajo a la izquierda y $(x + w, y)$ la que se corresponde con la esquina superior derecha. Con estos datos puedes deducir cuál es la respuesta a la pregunta que abría este párrafo.

```
1 ;// Comprueba si (x1, y1) estĂ;n dentro del rectĂ;ngulo rect
```

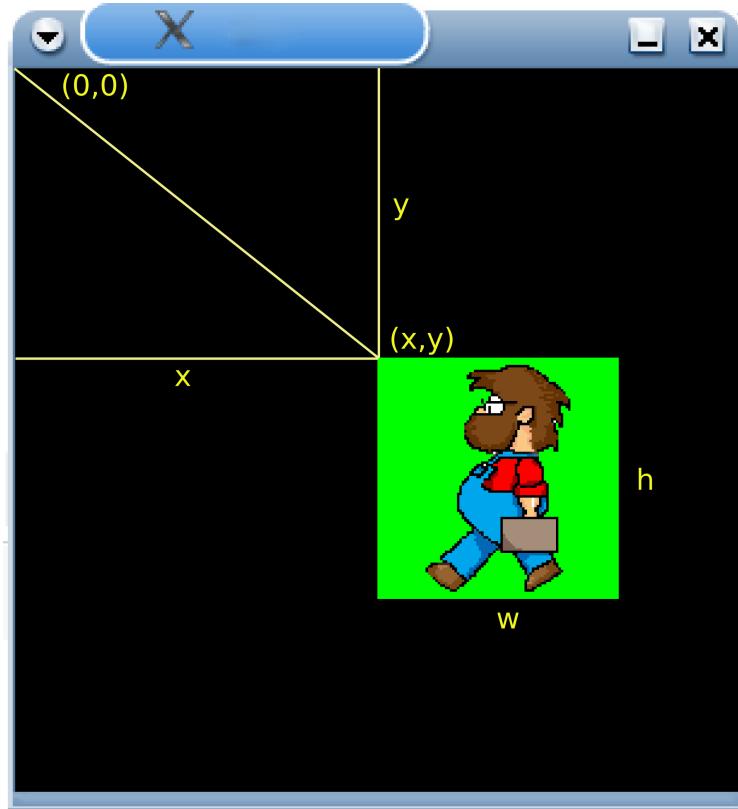


Figura 5.8: Representación de las variables de SDL_Rect

```

2 ;
3 ;SDL_Rect rect;
4 ;
5 ;int x1, y1;
6 ;
7 ;
8 ;if(x1 >= rect.x &&
9 ;    y1 >= rect.y &&
10 ;   x1 < (rect.x + rect.w) &&
11 ;   y1 < (rect.y + rect.h)) {
12 ;
13 ;    // En este caso el punto (x1, y1) estÁ; dentro del rectÁ;ngulo
14 ;
15 ;}
16 ; else {
17 ;
18 ;    // En este caso el punto (x1, y1) estÁ; fuera del rectÁ;ngulo
19 ;}
```

Como ves el resultado es un simple ejercicio de contención jugando con la posición de ambos elementos y de la anchura y altura del rectángulo con el que estamos trabajando. Esta función nos será útil cuando tratemos el tema de las colisiones.

5. El Subsistema de Video

SDL no ofrece la posibilidad de representar rectángulos rotados en sí mismos por lo que necesitamos una superficie mayor que la del propio rectángulo para poder representarlos en pantalla. Para poder manejarlos se especifica la superficie rectangular de componentes paralelas a la vertical y la horizontal que envuelve dicho rectángulo rotado. Puedes observar en la figura 5.9 el área extra que necesitamos para manejar dicho rectángulo. Si el color amarillo que se muestra en la figura fuese establecido como *color key* para la transparencia se mostraría en pantalla un rectángulo de color gris y a efectos visuales sería lo mismo que poder representar un rectángulo rotado nativamente en SDL. A efectos prácticos esto afecta a otro tipo de acciones como las colisiones que, dependiendo del método utilizado, no responderían con lógica a la estructura rotada ya que para el sistema las comprobaciones serían realizadas con el rectángulo *SDL_Rect* completo.

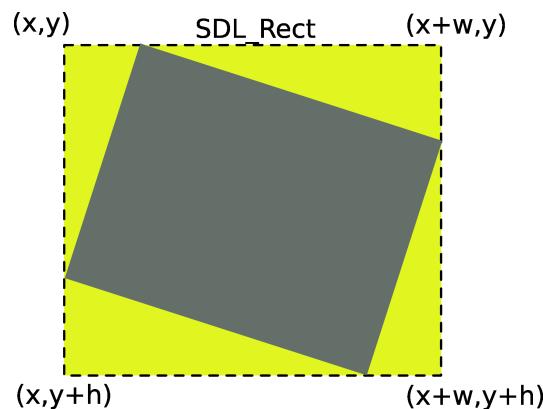


Figura 5.9: Rectángulo rotado en SDL

Este tipo de estructura, *SDL_Rect*, es un tipo de los llamados fundamentales en SDL y es muy usado en las aplicaciones que se desarrollan con esta librería. SDL no proporciona funciones como unir o intersecciar rectángulos que sin embargo sí son soportadas por otras librerías. Si queremos dotar de cierta potencia a esta estructura deberemos de implementar nuestra propia librería. Estas funciones no son de uso común en el desarrollo de videojuegos por lo que debemos de medir bien si nos merece la pena implementarlas.

Como apunte final mencionar que el rectángulo vacío tendrá como componentes h y w iguales a 0. Por convención los rectángulos vacíos definirán su posición en el origen de coordenadas.

5.5.2. SDL_Color. El color en SDL.

Esta es otra de las estructuras fundamentales en el subsistema de vídeo de SDL. SDL_Color nos hará la vida más fácil a la hora de trabajar con la librería SDL y las componentes de color. Hay una gran cantidad de funciones SDL que necesitan información de color. SDL_Color es una estructura que nos permite tener unicidad a la hora de representar la información de color, por ello se dice que este tipo de dato representa el color de forma independiente.

El tipo *SDL_Color* se define de la siguiente forma:

```

1 ;  

2 ;typedef struct {  

3 ;    Uint8 r;  

4 ;    Uint8 g;  

5 ;    Uint8 b;  

6 ;    Uint8 unused;  

7 ;} SDL_Color;  

;

```

El significado de cada uno de los campos es:

- *r*: Indica la cantidad del componente rojo del color que queremos obtener.
- *g*: Indica la cantidad del componente verde del color que queremos obtener.
- *b*: Indica la cantidad del componente azul del color que queremos obtener.
- *unused*: Este es un campo sin uso.

El rango de valores de *r*, *g* y *b* comprende desde 0 hasta 255, donde 0 es la menor intensidad de color y 255 la mayor posible. Si quisieramos, por ejemplo, mostrar el color verde puro deberíamos inicializar estos valores con *(0, 255, 0)* con lo que estaríamos indicando que la intensidad de color para el rojo sería 0, 255 para el verde y 0 para el azul. El miembro que no se usa es libre para almacenar la información que queramos, como puede ser el canal alpha, o bien cualquier otra cosa que consideremos oportuna.

Para que consigas habituarte a manejar los colores RGB es aconsejable que dediques algún tiempo al estudio de este espacio de colores, ya que es fundamental para la representación de colores en pantalla. Como puedes observar en la figura 5.10 este espacio de colores es tridimensional. Las dimensiones del mismo se definen como rojo (*r*), verde (*g*) y azul (*b*). Cada una de estas dimensiones tiene un rango limitado entre 0,0 y 1,0 en cada uno de sus ejes. En SDL 0,0 es representado con 0 y el mayor valor 1,0, como vimos antes, se representa con el valor 255 ofreciendo un total de 256 valores para cada una de las componentes de color. Esto provoca que tengamos $256 \times 256 \times 256 = 16,777,216$ valores distintos que es el número de colores diferentes que podemos representar mediante este espacio de colores. Deben de ser

5. El Subsistema de Video

más que suficientes para cualquier tipo de aplicación que queramos desarrollar.

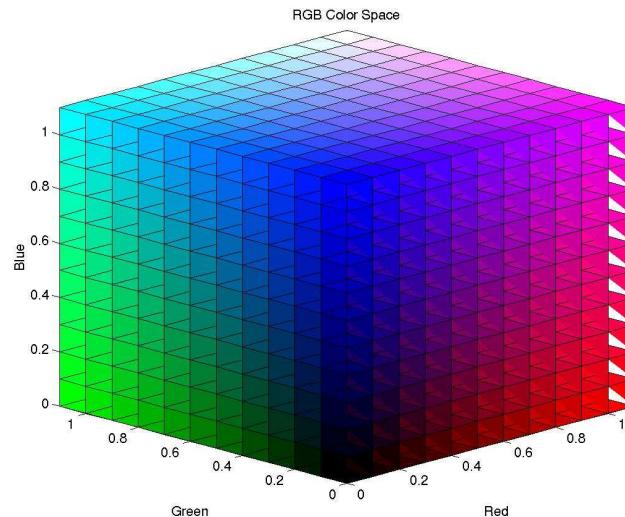


Figura 5.10: Espacio de colores RGB

Puedes intuir que no todos los modos de vídeo ni todos los sistemas pueden representar esta cantidad de colores. La gran cantidad de colores que nos permite representar *SDL_Color* sólo estará disponible cuando establezcamos un modo de 24 o más bits de color. En el modo de 16 bits, normalmente, se trunca la cantidad de bits de los colores rojo y azules a 5 y los 5 o 6 restantes se destinan al verde. ¿Por qué el verde? Percibimos más variedad de verdes que de los otros colores.

En el caso de que el verde tomase 6 bits el formato sería el conocido como R5G6B5 y en el caso de dejar 4 bits para cada componente y los cuatro restantes para el canal alfa el formato correspondería con el R4G4B4A4. El truncamiento se realiza en la zona de los valores más bajos de los ejes que suelen ser demasiado oscuros para diferenciarlos por lo que podemos omitirlos. Puedes realizar una prueba que te ayudará a entender porque se truncan los valores más oscuros. Modifica la cantidad de colores con la que trabajas en tu escritorio y limitala a 16 bits. Bien, ¿observas alguna diferencia? Seguramente serán inapreciables a primera vista ya que no es habitual tener una configuración con muchos tonos oscuros y pequeñas diferencias entre ellos. Si abres una fotografía, o si la tenías puesta de fondo de escritorio, podrás notar que las diferencias son mayores al necesitar un color real pero que no se obtiene una mala calidad de imagen.

En el caso de establecer un modo de 8 bits el truncamiento es mucho mayor. Sólo dispondremos de 256 colores para mostrar elementos en nuestra

aplicación. Si queremos mostrar alguna imagen tendrá una calidad muy pobre. Podemos mejorar el comportamiento de esta configuración de color creando una paleta tomando 256 colores de los disponibles a 24 bits y almacenarlos en una estructura *SDL_Palette*. Esto nos permitirá utilizar, al menos, colores adecuados aunque 256 no sean suficientes para nuestra aplicación. Esta técnica es conocida como uso de “Colores Indirectos” y puede ser útil para crear efectos interesantes que no se pueden conseguir con los modos RGB habituales.

Aunque *SDL_Color* es una estructura importante no es usada directamente por el subsistema de vídeo de *SDL*, teniendo como excepción a las paletas de color, existiendo lugares donde puede ser utilizada sin ser obligatorio.

5.5.3. *SDL_Palette*, La Paleta de Colores

Una paleta de color no es más que un subconjunto o agrupación de colores de los que podemos mostrar por pantalla que van a ser de uso común en el desarrollo de la aplicación. Por ejemplo si disponemos de 256 colores y vamos a desarrollar un videojuego o una aplicación en el que vamos a usar sólo 16 podemos crear una paleta con estos 16 colores y utilizarla para no tener que hacer referencia cada vez a la cantidad de colores en RGB. Es decir, cada vez que quiera utilizar el color azul no tendremos que especificar el (0,0,255) por componentes RGB si no acceder a, por ejemplo, la primera posición de nuestra paleta donde lo hemos almacenado previamente. En la figura 5.11 puedes ver un ejemplo de paleta de color segura para la web.

El concepto de paleta es próximo al de la paleta de un pintor. Un pintor selecciona un grupo de colores de los que tiene disponibles y los “almacena” en una paleta de madera que llevará consigo para trabajar sobre un lienzo y no tener que recurrir a los botes de pintura.

Hay muchos motivos para usar este tipo de técnicas. Uno de ellos es que se tarda mucho menos en copiar superficies de 8 bits que superficies de 24 o 32 bits, la razón es obvia, el tamaño. Las superficies de 8 bits son más pequeñas y pueden ser transferidas en menor tiempo que las de 32 bits, y las paletas nos ofrecen estas transferencias de 8 bits. Este razonamiento era válido hasta hace no mucho tiempo. Dependiendo de la arquitectura el hardware actual es capaz de transmitir 32 bits de información más rápido que 8 debido a los avances en la tecnología por lo que el uso de paletas está quedando anticuada.

Otro motivo importante para utilizar paletas de color es que mucho de los dispositivos móviles actuales tienen restringida la cantidad de colores a 256. Si queremos producir software de videojuegos para la gran masa no podemos restringirnos al mercado por usar una cantidad de colores mayor.

5. El Subsistema de Video

Modificando la paleta que utilizamos sobre una superficie podemos crear efectos, como movimiento en imágenes estáticas ciclando los colores de la paleta, lo que nos abre un otro campo de posibilidades. Actualmente no es común el uso de estas técnicas ya que se trabaja con una cantidad mayor de 256 colores.

El utilizar paletas supone el uso de colores indirectos para superficies de 8 bits, lo que nos limita a sólo 256 colores. Este es el mayor problema del uso de las paletas de color. Nosotros habitualmente trabajaremos con más de 16 bits, y en el caso de que sea factible, a color real, por lo que no utilizaremos normalmente este tipo de técnica, excepto en casos excepcionales.

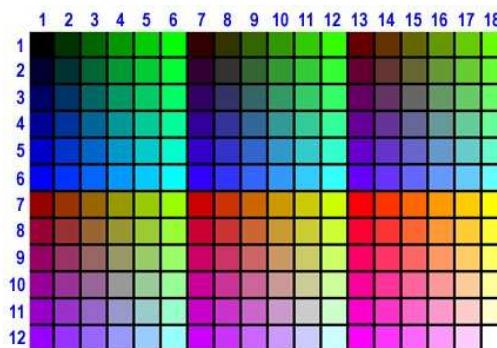


Figura 5.11: Paleta de colores segura para la web

La estructura que almacena este tipo de información en SDL es `SDL_Palette` y se define como sigue:

```
1 ;typedef struct {
2 ;    int ncolors;
3 ;    SDL_Color *colors;
4 ;} SDL_Palette;
```

La definición de este tipo de datos es bastante lógica ya que vamos a guardar una colección de colores. Pasamos a describir cada uno de los campos de la estructura:

- *ncolors*: Indica el número de colores almacenados en la paleta.
- *colors*: Puntero a la estructura que almacena los colores de la paleta, normalmente un vector.

La variable definida con el tipo `SDL_Color*` es un puntero o vector con los colores de que componen la paleta. La paleta sólo puede ser usada en el modo de 8 bits por pixel que equivale a 256 colores, como hemos visto.

5.5. Conceptos y Estructuras Fundamentales

Es importante especificar los 8 bpp en la creación de la superficie donde vamos a aplicar la paleta. Si vas a trabajar en modo a pantalla completa es una buena idea usar la bandera `SDL_HWPALLETTE` porque proporcionará un mayor control sobre el acceso a la paleta. Esto es debido a que normalmente los sistemas operativos, sobre todo los que trabajan con ventanas, se reservan una paleta de colores para sus propósitos limitando, más si cabe, el número de colores que podemos elegir en la paleta.

La función que permite asignar una paleta a una superficie es:

```
int SDL_SetPalette(SDL_Surface *surface, int flags, SDL_Color *colors,  
                   int firstcolor, int ncolors);
```

La función devuelve 1 si todo ha ido bien. Si no se pueden establecer los colores devolverá 0 y es que habrá ocurrido un error. Si has pasado como parámetro la bandera `SDL_HWPALLETTE` en el parámetro `flag` esta función siempre devolverá 1 ya que tendremos un control total sobre la paleta. La función recibe como parámetros la superficie en la que vamos a establecer la paleta. El parámetro `flags` nos permite establecer una paleta lógica mediante `SDL_LOGPAL` o una física mediante `SDL_PHYSPAL`. Estas banderas son exclusivas y no se pueden combinar entre sí. Las paletas lógicas son las utilizadas en los *blits*, concepto que explicaremos más adelante.

El cuarto valor es un puntero a un vector de colores (la paleta) y en `firstcolor` definimos cual será el primero de los colores. Para terminar indicaremos el número total de colores a utilizar. Solo podrás configurar una paleta para una superficie en el caso de que se utilicen 8 bpp en dicha superficie.

Otra función que nos permite establecer una paleta para una superficie es:

```
int SDL_SetColors(SDL_Surface *surface, SDL_Color *colors, int  
                  firstcolor, int ncolors);
```

Como puedes observar es muy parecida a la anterior. La única diferencia es que no podemos pasar banderas como parámetro ya que asume la combinación de los dos posibles valores expuestos anteriormente por omisión. Cuando trabajes a pantalla completa utiliza esta función ya que el sistema te permitirá crear cualquier tipo de paleta de color. En el caso de hacerlo sobre una ventana mejor especifica que tipo de paleta quieras utilizar por si es posible crearla en el sistema.

5. El Subsistema de Video

5.5.4. SDL_PixelFormat. Formato de Píxel

Cuando hablamos del formato del pixel, nos referimos a la manera en que un pixel guarda la información del color. Un píxel en memoria es una secuencia de posiciones de memoria. Según el formato de píxel que estemos utilizando necesitaremos más o menos de estas posiciones. Evidentemente no necesitamos las mismas posiciones de memoria para guardar un píxel a 4 bits de color que a 8 bits.

Desde que la gran variedad de tarjetas de vídeo se hizo presente surgió la necesidad de tener una estructura que nos permitiese guardar la información sobre un píxel que fuera independiente a un hardware concreto. En esta estructura almacenamos todo lo necesario para que, después de un proceso, nuestro hardware represente el color que queremos mostrar.

Esta información es almacenada en la estructura *SDL_PixelFormat* que incluye mucha información relevante como veremos a continuación. Esta información describe exactamente que tipo de píxeles son almacenados en una superficie. Nos permite establecer y tomar los valores de dichos píxeles.

Durante la exposición del subsistema de vídeo se hace referencia numerosas veces a este formato de píxel cuya estructura es:

```
1 ;  
2 ;typedef struct {  
3 ;    SDL_Palette *palette;  
4 ;    Uint8 BitsPerPixel;  
5 ;    Uint8 BytesPerPixel;  
6 ;    Uint32 Rmask, Gmask, Bmask, Amask;  
7 ;    Uint8 Rshif, Gshift, Bshift, Ashift;  
8 ;    Uint8 Rloss, Gloss, Bloss, Aloss;  
9 ;    Uint32 colorkey;  
10 ;} SDL_PixelFormat;  
;
```

Como puedes observar esta estructura incluye información sobre la profundidad de color, la cantidad de bits reservados por cada canal de color, la paleta que se utiliza... El significado de cada campo es el siguiente:

- *palette*: Es un puntero a la paleta de colores. Si este campo apunta a **NULL** la profundidad de color (bpp) es superior a 8 bits, ya que esos modos no usan paletas.
- *BitsPerPixel*: Es el número de bits usado para representar cada pixel en la superficie. Suele tomar los valores 8, 16, 24 o 32 bits. Cuanto mayor es el número, mayor es la cantidad de colores que se pueden representar. Si el valor de este campo es 8 entonces existirá una paleta apuntada por el miembro *palette*.

5.5. Conceptos y Estructuras Fundamentales

- *BytesPerPixel*: Número de bytes utilizado para representar cada pixel en la superficie. Es equivalente a BitsPerPixel pero almacenado en bytes. Los valores que puede tomar son análogos a los de *BitsPerPixel* pero en bytes que son 1, 2, 3 o 4.
- *xmask*: Máscara binaria utilizada para obtener el valor del componente de color *x* que es R para rojo, G para verde, B para azul, A para alpha.
- *xshift*: Es el desplazamiento binario hacia la izquierda para el componente de color *x* en el valor del píxel. Mismas equivalencias, naturalmente, que el parámetro anterior con respecto a *x*.
- *xloss*: Es la pérdida de precisión de cada componente de color ($2^{RGBALoss}$). Mismas equivalencias con respecto a *x* que el apartado anterior.
- *colorkey*: Color clave es el color que se establece como transparente en la superficie. El color almacenado en este parámetro no se mostrará por pantalla. Estudiaremos este aspecto con más detenimiento.
- *alpha*: Parámetro para el *alpha blending*. Entre 0 - 255. Básicamente es la cantidad de transparencia aplicada. Este concepto será presentado en secciones posteriores a la actual con más detalle.

Los valores *mask*, *shift* y *loss* se utilizan para convertir a o desde los valores de un color de 24 bits. Si quisieramos representar el color rojo en esta estructura deberíamos de inicializar los valores *Bmask* y *Gmask* a 0 mientras que la componente *Rmask* tendría que ser inicializada a 255 que es el máximo valor posible. *Rshift* introduciría tantos 1 en el bit 0 del color como se le especificase en su campo. Por ejemplo si *Rshift* valiese 10, desplazaría 10 veces el color rojo.

Rloss es la diferencia entre una representación de 8 bits y la representación que se haya establecido en un momento dado en el formato de color. Si utilizamos una representación de 16 bits de color, 5 serían destinados al color rojo, con lo que perderíamos 3 bits de precisión.

Vamos a explicar estos conceptos más detenidamente ya que no son fáciles de digerir. Por defecto SDL toma los valores 0xFF0000, 0x00FF00 y 0x0000FF para las máscaras *Rmask*, *Gmask* y *Bmask* respectivamente. Los valores de *Rshift*, *Gshift* y *Bshift* se traducen en el número de ceros que hay a la derecha del bit a 1 con menor valor. Los valores *Rloss*, *Gloss* y *Bloss* podemos traducirlo también a ocho menos el número de unos que haya a la izquierda del bit a 1 con menor valor, contando en el mismo. Esta cifra puede ser negativa. Y esto, ¿para qué sirve?

A partir de los valores anteriores, dado un píxel deseado, el color se calcula de la siguiente manera:

5. El Subsistema de Video

ComponenteXtemp = color_deseado AND Xmask;

ComponenteXtemp >>= Xshift;

ComponenteXtemp <<= Xloss;

Donde tenemos que sustituir X por R, G o B según la componente de color que queramos tratar. Si realizas estos cálculos con las máscaras por omisión obtendrás el mismo color RGB original. Sin embargo si cambiamos las máscaras obtendremos como resultado otro color diferente, dependiente de dicha máscara. Veamos un ejemplo:

```
1 ;/* - Ejemplo de trabajo con máscaras - */
2 ;
3 ;Tenemos los valores:
4 ;
5 ;      R = 0xFFFF000
6 ;      G = 0x0000FF
7 ;      B = 0x000F00
8 ;
9 ;Por lo que los valores de las máscaras serán:
10 ;
11 ;      Rmask = 0xFFFF000
12 ;      En binario: 111111111110000000000000
13 ;
14 ;      Rshift = 12 (Hasta el primer 1 hay doce ceros)
15 ;
16 ;      Rloss = 252 (Desde el primer 1 hay doce unos, contando este)
17 ;      Por lo que 8 - 12 = -4, que en Uint8 es 252
18 ;
19 ;Siguiendo el mismo razonamiento:
20 ;
21 ;      Gmask = 0x0000FF
22 ;      Gshift = 0
23 ;      Gloss = 0
24 ;
25 ;      Bmask = 0x000F00
26 ;      Bshift = 8
27 ;      Bloss = 4
28 ;
29 ;Con estas máscaras y la forma de calcular el color presentada en
30 ;el tutorial vamos a obtener el color resultante si el color original
31 ;fuese el color rosa (0xFF00FF)
32 ;
33 ;El proceso sería:
34 ;
35 ;Uint32 temporal;
36 ;Uint8 Red, Green, Blue;
37 ;Uint8 Rosa = 0xFF00FF;
38 ;
39 ;/* ROJO */
40 ;
41 ;temporal = Rosa & Rmask; (0xFF0000 = 0xFF00FF AND 0xFFFF000)
```

5.5. Conceptos y Estructuras Fundamentales

```
42 ;temporal >>= Rshift; (Rshift = 12 => temporal == 0x000FF0)
43 ;temporal <= Rloss; (Rloss = 252 => temporal == 0x000000)
44 ;Red = temporal; (Red = 0)
45 ;
46 ;/* VERDE */
47 ;
48 ;temporal = Rosa & Gmask; (0x0000FF = 0xFF00FF AND 0x0000FF)
49 ;temporal >>= Gshift; (Rshift = 0 => temporal == 0x0000FF)
50 ;temporal <= Gloss; (Rloss = 0 => temporal == 0x0000FF)
51 ;Green = temporal; (Green = 0xFF)
52 ;
53 ;/* AZUL */
54 ;
55 ;temporal = Rosa & Bmask; (0x0000F0 = 0xFF00FF AND 0x0000F0)
56 ;temporal >>= Bshift; (Rshift = 12 => temporal == 0x000000)
57 ;temporal <= Bloss; (Rloss = 252 => temporal == 0x000000)
58 ;Blue = temporal; (Red = 0)
59 ;
60 ;Por lo que el color resultante sería 0x00FF00 que corresponde con el color
61 ;verde.
;
```

¿Qué podemos sacar como conclusión de este ejemplo? Podemos tener miles de píxeles en una superficie, y haciendo uso de las máscaras, podemos conseguir obtener diferentes imágenes. El manejo del color en hexadecimal puede ser engorroso. Es conveniente utilizar la función `SDL_MapRGB` que nos ayudará a manejar los colores. Esta función tiene el siguiente prototipo:

```
Uint32 SDL_MapRGB(SDL_PixelFormat *fmt, Uint8 r, Uint8 g, Uint8 b);
```

Según el formato del píxel (bpp) la representación de un color puede variar. Esta función se encarga que a partir del formato de píxel y la cantidad de cada color RGB que queramos establecer devuelva el color en dicho formato de píxel.

La función necesita un puntero al formato del pixel, por lo que utilizaremos normalmente el campo `format` de `SDL_Surface` además de la intensidad de color rojo, verde y azul. La función devuelve un número que contiene el color demandado en un formato manejable por SDL. Si el bpp es de 16 bits el valor retornado será del tipo `Uint16` y si es de 8 bits el tipo devuelto será de tipo `Uint8`.

Si tenemos la máscara definida como en el ejemplo anterior en una superficie, al pasar como parámetros los valores correspondientes para obtener el color rosa, la función devolverá el color verde. Si están establecidas las máscaras por omisión al pasar como parámetros los valores correspondientes para obtener el color rosa, la función devolverá el color rosa en formato `Uint32`.

Existe una variante de esta función con un parámetro más añadido que nos permite trabajar con canales alpha. El prototipo de dicha función es:

5. El Subsistema de Video

```
Uint32 SDL_MapRGB(SDL_PixelFormat *fmt, Uint8 r, Uint8 g, Uint8 b,  
                  Uint8 a);
```

Como puedes observar es idéntica a la anterior pero con canal alpha. Puede darse el caso de que tengamos un color transformado y queramos conocer los valores de las componentes del mismo. Para realizar esta tarea SDL proporciona dos funciones que realizan la acción inversa de las dos últimas funciones. Los prototipos de estas funciones son:

```
void SDL_GetRGB(Uint32 pixel, SDL_PixelFormat *fmt, Uint8 *r, Uint8  
                 *g, Uint8 *b); void SDL_GetRGBA(Uint32 pixel, SDL_PixelFormat *fmt,  
                                         Uint8 *r, Uint8 *g, Uint8 *b, Uint8 *a);
```

Estas funciones reciben como parámetro de entrada el color y el formato del píxel y devuelve por referencia en *r*, *g* y *b* las componentes del píxel en cuestión. En la segunda función además devuelve la componente del canal alpha.

5.5.5. **SDL_Surface, La Superficie**

La estructura principal de manejo y almacenamiento en SDL es la superficie. Una superficie en SDL es, fundamentalmente, la abstracción de un rectángulo compuesto por un bloque de píxeles que posee información gráfica. El concepto de superficie es fundamental en SDL. Una superficie es un área de memoria, ya sea en memoria principal o en memoria de vídeo, donde podemos dibujar, almacenar imágenes y realizar otras tareas. Toda la maquetación de la aplicación, movimiento de personajes, efectos... los implementaremos mediante el manejo de superficies.

Este tipo de dato viene soportado por la estructura *SDL_Surface* y está definida de la siguiente forma:

```
;  
1 ;  
2 ;  
3 ;  
4 ;  
5 ;  
6 ;  
7 ;  
8 ;  
9 ;  
10 ;} SDL_Surface;  
;
```

El significado de cada campo es el siguiente:

- *flags*: Son unas banderas que indican las características de la superficie en cuestión. Estas banderas están definidas mediante constantes y definen aspectos como donde crear la superficie o si habilitamos el doble búffer.

5.5. Conceptos y Estructuras Fundamentales

- *format*: Puntero al formato de píxel de la superficie, es decir, en que formato estará la superficie en cuestión.
- *w*: Ancho de la superficie en píxeles. Es una variable sin signo ya que, junto a *h* no pueden tomar valores negativos. Una superficie en SDL es rectángular. No tiene sentido que tenga un valor negativo para indicar su anchura.
- *h*: Altura de la superficie en píxeles. Es del mismo tipo que *w*.
- *pitch*: Corresponde a la longitud de la scanline (línea de escaneo) de la superficie en bytes. Representa el ancho real de la superficie en memoria en bytes, mientras que *w* representa el ancho visible de la superficie. El valor del *pitch* siempre es mayor que el valor del ancho visible. Esto provoca tener un excedente de bits para representar una superficie. La razón de tener esta zona perdida de memoria es para tener los datos alineados. Esto provoca que cada línea de nuestra superficie esté alineada en memoria con lo que conseguimos un acceso más rápido. Puedes observar lo que ocurre en la figura 5.12. Para poder acceder directamente a un pixel (*x, y*) accederemos por el cálculo de posición $(y \times \text{pitch}) + (x \times \text{bytesPorPixel})$.
- *pixels*: Puntero al comienzo de los datos de la superficie. Es el vector de píxeles que constituye la superficie. La razón de que sea un puntero *void* es que no existe la manera estándar de representar los datos de pixel.
- *clip_rect*: Estructura que indica el área rectangular y sólo rectangular de clipping de la superficie, que es el área dónde podemos realizar blit en dicha superficie. Las técnicas de clipping y de blitting serán explicadas con posterioridad.
- *refcount*: Usado cuando se libera la superficie, es un contador de referencia. Cuando la superficie se crea toma el valor 1, y cuando es liberada se decrementa dicho valor en una unidad. Si existe un número mayor de 1 de superficies que depende de ésta podemos incrementar este valor tantas veces como dependencias existan lo que provoca que no se libere hasta que no quede ninguna dependencia pendiente.

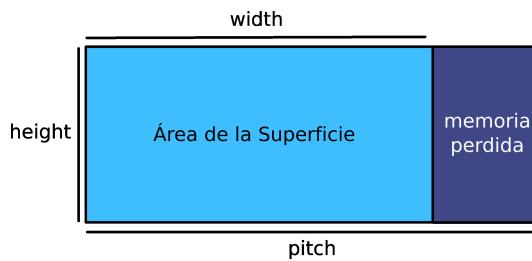


Figura 5.12: Pitch de la superficie

5. El Subsistema de Video

Todos los campos comentados son de sólo lectura para el usuario de manera general. El único de ellos modificable es el puntero a los píxeles ya que mediante éste colocaremos los píxeles en una superficie. El campo *flags* es modificado por la función *SetVideoMode()* al establecer el modo de vídeo con los parámetros de entrada que se le pasan para definir el tipo de superficie que se quiere crear. Esta es la única forma de inicializar el parámetro *flags* en SDL, que es fundamental ya que define propiedades como dónde crear la superficie o la utilización o no de doble búffer. Podemos manipular los datos apuntados por *pixels* y realizar una modificación *píxel-por-píxel* así como incrementar el número de dependencias existentes.

Cuando inicializamos la superficie principal única con *SetVideoMode()* establecemos el tamaño de pantalla a utilizar en la aplicación. Aunque esta definición es libre existen unos tamaños básicos que se adaptan a las proporciones de las pantallas actuales. La mayoría de los monitores mantienen la proporción 4:3 aunque proliferan en la actualidad las pantallas de formato panorámico de proporción 16:9.

Dimensiones de superficie actuales más comunes:

Estándar	Aspecto o Razón	Width(Ancho)	Height(Alto)	Comentarios
VGA	4:3	640	480	Base en sistemas actuales
SVGA	4:3	800	600	Habitual 15 "
XGA	4:3	1024	768	Habitual 17 "
SXGA	4:3	1280	1024	Habitual 19 "
WXGA	16:9	1200	800	Panorámico 15.4"
WSXGA	16:9	1440	900	Panorámico 15.4"

Cuadro 5.1: Superficies más comunes.

La función *SDL_SetVideoMode()* devuelve la única superficie que será visible. Sólo una superficie representa el área de visualización en la pantalla en un momento dado. Todas las demás superficies hacen la función de búfferes para almacenar datos hasta el momento que deban ser mostrados en la pantalla principal. Es posible que necesitemos la creación de otras superficies para almacenar elementos auxiliares con el objetivo de construir escenas, almacenar gráficos, mostrar efectos... Estas otras superficies, al no ser la devuelta por *SDL_SetVideoMode()*, no podrán ser mostradas directamente en pantalla pero sí copiadas en la superficie visible. Este proceso es conocido como bit blitting y es una técnica muy utilizada desde el comienzo de la creación de los videojuegos.

La manera de crear una nueva superficie es utilizando la función *SDL_CreateRGBSurface()*. Esta función nos permite crear una nueva super-

5.5. Conceptos y Estructuras Fundamentales

ficie RGB vacía lista para ser usada. El prototipo de la función es:

```
SDL_Surface *SDL_CreateRGBSurface(Uint32 flags, int width, int height, int depth, Uint32 Rmask, Uint32 Gmask, Uint32 Bmask, Uint32 Amask);
```

Puedes observar como la función devuelve un puntero a una superficie que es, al fin y al cabo, lo que queremos obtener. Los parámetros que recibe la función están íntimamente relacionados con los de la estructura *SDL* y tienen prácticamente la misma semántica que estos. Es lógico ya que van a crear una estructura de este tipo. Pasamos a estudiar estos parámetros:

- *flags*: Son unas banderas que indican las características de la superficie en cuestión, como donde se crea o si tiene habilitado el doble búffer.
- *int width*: Anchura de la superficie a crear.
- *int height*: Altura de la superficie a crear.
- *int depth*: Profundidad de color. Igual que bpp en SetVideoMode.
- *Uint32 Rmask*: Entero de 32 bits sin signo para la máscara roja.
- *Uint32 Gmask*: Entero de 32 bits sin signo para la máscara verde.
- *Uint32 Bmask*: Entero de 32 bits sin signo para la máscara azul.
- *Uint32 Amask*: Entero de 32 bits sin signo para la máscara alfa o de transparencia.

Para comprender los parámetros Rmask, Gmask, Bmask y Amask debemos estudiar las estructuras internas de *SDL*. Estos parámetros pertenecen a la estructura *SDL_PixelFormat*. Cada parámetro de los comentados representa a la cantidad de color correspondiente que queremos añadir como si de una mezcla RGB se tratase. Esta estructura es explicada con detalle en una sección anterior. Los valores que puede tomar el campo *flags* de esta estructura son los siguientes:

- **SDL_SWSURFACE**: Crea la superficie de vídeo en memoria principal. Esta es la mejor opción si vamos a modificar una superficie “a mano”. El procesador tiene acceso directo a esta memoria por lo que puede escribir y leer directamente sobre ella.
- **SDL_HWSURFACE**: Crea la superficie en la memoria de vídeo. El procesador no tiene acceso directo a esta memoria por lo que no podremos modificar o leer los píxeles de una superficie creada en esta memoria.

5. El Subsistema de Video

- **SDL_SRCCOLORKEY:** Permite el uso de transparencias de color de base mediante la definición de un color clave (Color Key).
- **SDL_SRCALPHA:** Activa el alpha blending.

A la hora de elegir en que memoria crear la superficie debemos de tener en cuenta los siguientes aspectos. En la memoria del sistema tenemos un acceso más rápido a los píxeles pero conseguimos una menor rapidez en el blitting. En la memoria de vídeo ocurre lo contrario, tenemos un acceso lento a los píxeles pero un mayor rendimiento a la hora de realizar blitting. Si tenemos que modificar píxeles y realizar blitting y creamos las superficies en zonas de memoria diferentes se puede producir un cuello de botella en el bus que intercomunica ambos componentes al tener que estar volcando los datos de una memoria a otra continuamente.

No es habitual la creación de superficies vacías como las que devuelve esta función. Son utilizadas para volcar datos o hacer modificaciones de otras superficies. Normalmente trabajaremos con superficies creadas a partir de la carga de una imagen de un fichero a una superficie de este tipo.

5.5.6. SDL_Overlay

El concepto de este tipo de estructura y el de *SDL_Surface* es muy similar. Lo único que varía es el uso de este tipo de superficie. El overlay es usado para mostrar vídeo YUV en streaming como puede ser un mpg o un tipo similar de vídeo. Puedes usar este tipo de estructura siempre que quieras introducir un vídeo en tu aplicación, como presentación o algo similar. Esta estructura está definida de la siguiente forma:

```
1 ;typedef struct{
2 ;    Uint32 format;
3 ;    int w, h;
4 ;    int planes;
5 ;    Uint16 *pitches;
6 ;    Uint8 **pixels;
7 ;    Uint32 hw_overlay:1;
8 ;} SDL_Overlay;
```

- *format:* Indica el formato del overlay. Este parámetro tiene que ser definido de una forma específica que presentaremos a continuación.
- *w, h:* Como es habitual en SDL estas variables indican el ancho y alto de la superficie de overlay.
- *planes:* Especifica el número de planos que existen para el overlay. Normalmente este valor se encuentra entre 1 y 3. Los planos son imágenes individuales que juntas forman una imagen completa.

- *pitches*: Lista que contiene el número de pitches por cada plano. Existe al menos un pitch por plano, lo que demuestra que cada plano es una imagen individual.
- *pixels*: Vector de punteros para los datos de cada plano. Hay un puntero para cada plano y un pitch correspondiente para cada puntero de píxel.
- *hw_overlay*: Este campo será 1 si el overlay está acelerado por hardware y 0 si no lo está. Esto afectará al rendimiento del overlay pero no cambiará nada con respecto a nuestro código.

Todos los campos son de sólo lectura excepto el campo *pixels* que debe ser definido antes de usar la superficie. El campo *format* puede tener los siguientes valores:

```

;_________________________________
1 ;#define SDL_YV12_OVERLAY 0x32315659 /* Planar mode: Y + V + U */
2 ;#define SDL_IYUV_OVERLAY 0x56555949 /* Planar mode: Y + U + V */
3 ;#define SDL_YUY2_OVERLAY 0x32595559 /* Packed mode: Y0+U0+Y1+V0 */
4 ;#define SDL_UYVY_OVERLAY 0x59565955 /* Packed mode: U0+Y0+V0+Y1 */
5 ;#define SDL_YVYU_OVERLAY 0x55595659 /* Packed mode: Y0+V0+Y1+U0 */
;_________________________________

```

Para comprender estos formatos debes hacer un estudio profundo sobre los formatos YUV. Las técnicas de overlay no son muy comunes en el desarrollo de los videojuegos de SDL. Nosotros vamos a estudiar como podemos manejar estas superficies.

5.5.6.1. Creando un Overlay

Crear una superficie de overlay es muy simple en SDL. Para realizar esta tarea SDL proporciona la función:

```
SDL_Overlay *SDL_CreateYUVOverlay(int width, int height, Uint32
format, SDL_Surface *display);
```

Esta función devuelve un puntero a una estructura *SDL_Overlay*. Si recibes de la función el valor NULL es que ha ocurrido un error. Como puedes ver existen cuatro parámetros de entrada. Los dos primero, *width* y *height*, son el ancho y la altura del overlay. El tercer parámetro, *format*, debe ser una de las constantes que acabamos de ver hace unas líneas. El último parámetro, *display*, es una puntero a una superficie donde el overlay será renderizado, es decir, mostrado.

5.5.6.2. Destruyendo el Overlay

Una vez utilizado el overlay podremos destruirlo, o lo que es lo mismo liberarlo, para que no siga consumiendo recursos del sistema. Puedes utilizar la siguiente función con este fin:

5. El Subsistema de Video

```
void SDL_FreeYUVOverlay(SDL_Overlay *overlay);
```

Esta función no devuelve ningún valor y recibe como parámetro de entrada la superficie a liberar creada previamente con *SDL_CreateYUVOverlay()*.

5.5.6.3. Bloqueando y Desbloqueando un Overlay

Como puedes ver el tipo overlay tiene un comportamiento muy parecido al de superficie que hemos estudiado pero con un objetivo diferente y SDL nos proporciona funciones parecidas para ambos tipos de datos. Por ejemplo, podemos acceder directamente los datos de un overlay como hacíamos en las superficies.

Antes de acceder los datos de un píxel (para escribir o leer) debes primero bloquear el overlay. Una vez hayas terminado de utilizar la superficie deberás de desbloquearla para que esté disponible para el sistema de nuevo. Las funciones que se encargan de bloquear y desbloquear el sistema son

```
int SDL_LockYUVOverlay(SDL_Overlay *overlay); void  
SDL_UnlockYUVOverlay(SDL_Overlay *overlay);
```

Ambas funciones necesitan sólo un parámetro, un puntero a la capa que queremos bloquear o desbloquear. En el caso de *SDL_LockYUVOverlay()* la función devuelve 0 si todo fue bien y -1 en el caso de existir algún error.

Entre las llamadas a *SDL_LockYUVOverlay()* y *SDL_UnlockYUVOverlay()* puedes manipular los *planos* del *overlay* y si quieres acceder a los punteros almacenados en el miembro *píxels* de la estructura *SDL_Overlay*. Como con las superficies, debes prestar atención al pitch del plano, almacenado en el vector de *pitches* de la estructura *SDL_Overlay*.

5.5.6.4. Dibujando un Overlay

Para terminar con este tipo de datos querrás saber como dibujar el overlay en la superficie que indicamos cuando llamamos a la función *SDL_CreateYUVOverlay*. Para esto utilizaremos la función:

```
int SDL_DisplayYUVOverlay(SDL_Overlay *overlay, SDL_Rect *dstrect);
```

Esta función devuelve 0 si todo fue bien y puedes tomar un valor distinto de cero como indicador de que existie algún problema. En la documentación de SDL no se especifica cual es el valor de error.

5.5.7. Blitting

Blitting es el proceso de hacer blit. Blit proviene de las palabras “block transfer” o en castellano, transferencia de bloques. Antiguamente la transferencia de bloques era abreviada como BLT. Como BLT no era pronunciable heredó una ‘i’ y de ahí surgió el término blit. Conceptualmente hacer blit puede ser equivalente al conocidísimo copy-paste aunque realmente hace más cosas que simplemente mover datos de una superficie a otra. Llamaremos blitter a la unidad de hardware o software que se encarga del blit y blitting a la acción de transferir bloques.

Para realizar un blit necesitas dos superficies, una de origen y una de destino, y dos rectángulos, también uno de origen y otro de destino. Una vez tengas esta información puedes realizar la llamada a la función de blitting entre superficies de SDL:

```
int SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surfcae  
*dst, SDL_Rect *dsrect);
```

Como puedes observar esta función recibe como parámetros toda la información que demandabamos antes, pasando todas las estructuras como punteros a las originales. Si los parámetros que definen los rectángulos de origen y de destino se inicializan con el valor *NULL* indicará que las superficies serán utilizadas enteras, ya que no se especifica rectángulo alguno. Debes saber que los rectángulos de origen y destino no tienen que coincidir ni en altura ni anchura, es más, el valor que tenga el rectángulo destino no influye en absoluto en la tarea de blitting, sólo importa las coordenadas donde queremos copiar la región. Esto es debido a que que no es posible redimensionar una superficie al hacer blitting con la librería básica de SDL.

Esta función devuelve un entero. Si el valor de este entero es 0 todo habrá ido bien mientras que si el valor devuelto es -1 indica que existe un error. Esta función devuelve un valor especial para describir una situación específica. Si la función devuelve el valor -2 indica que al menos una de las superficies están almacenadas en la memoria de vídeo por lo que necesitaremos refrescar para mostrar la imagen.

En el caso de realizar blitting entre dos superficies con formato de píxel diferentes la unidad de blitter de SDL se encarga de “traducir” un formato de píxel en otro. Este proceso es transparente al usuario pero supone una sobrecarga del sistema. Hay que tener en cuenta que ciertas operaciones de este estilo pueden afectar al rendimiento. La regla a tener en cuenta es bastante sencilla, a cuánto más complejidad, más lentitud. Cuantas más cosas obliguemos al blitter a realizar más tardará en realizarlas, así de simple. Una buena práctica es tener todas nuestras superficies en

5. El Subsistema de Video

el mismo formato de píxel por lo que ahorraremos al blitter hacer traducciones.

Existen dos tipos de implementación del blitting. Una se realiza en la tarjeta gráfica, implementado por hardware y, como no podía ser de otra manera, una implementación software. El blitter hardware es mucho más rápido, pero como suele pasar con las implementaciones en hardware, es mucho más limitado. En el caso de que una operación no pueda ser realizada por el blitter de hardware, bien porque no la soporte o bien porque las superficies no estén en memoria de vídeo, será el blitter de software el que se encargue de realizar las operaciones de blitting.

En el apartado dedicado al estudio de las transparencias en SDL encontrarás varios ejemplos de blitting. El blitting nos permite “maquetar” una imagen para mostrarla por pantalla con varios componentes independientes. En el ejemplo de la figura 5.13 puedes ver un ejemplo del proceso de blitting.

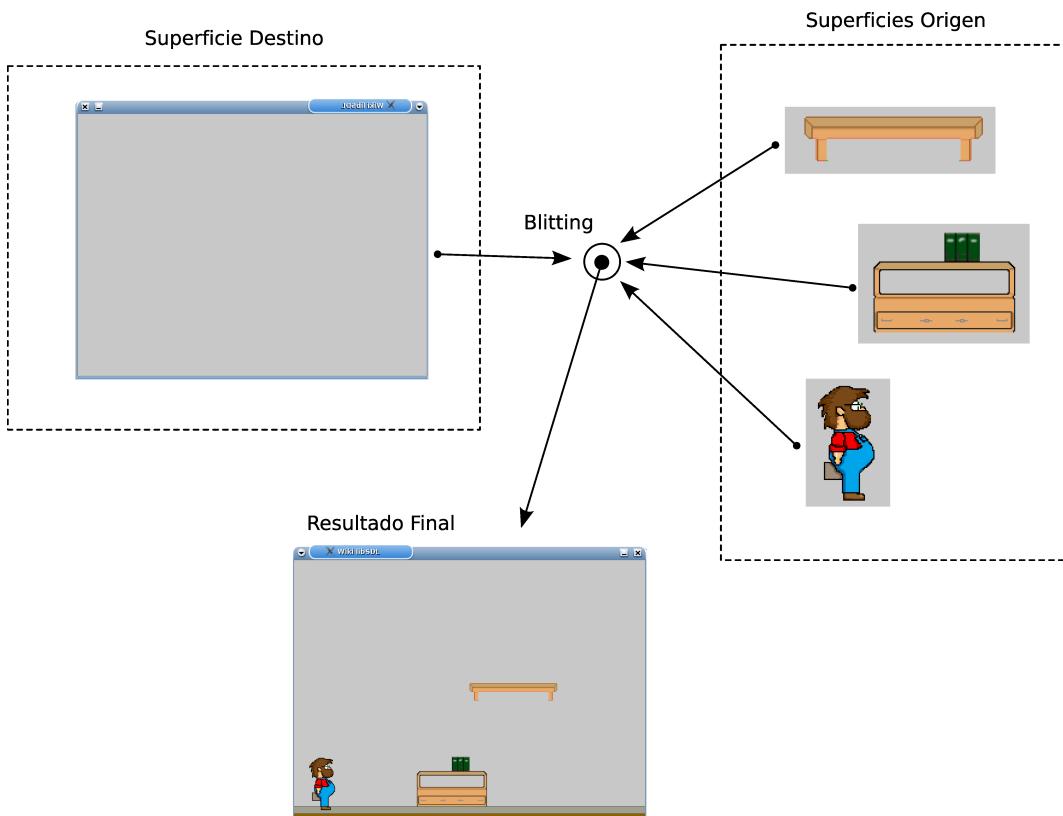


Figura 5.13: Resultado del blitting

5.5.8. El Flipping y el Doble Búffer

Bien, ya podemos montar un maravilloso paisaje con nuestros personajes “pegados” en él mediante blitting, pero ahora ¿cómo los mostramos? Ya

5.5. Conceptos y Estructuras Fundamentales

sabemos que para mostrarlo esta superficie debe ser la principal o pantalla que será la única que se muestre en el monitor, ahora, necesitamos actualizarla con nuestro nuevo montaje.

Para esto podemos utilizar la función *SDL_UpdateRect()*. Provoca que la tarjeta gráfica envíe lo que haya en un momento dado en el rectángulo del parámetro a que sea mostrado por pantalla. Esto conlleva un proceso hardware. Al realizar esta tarea se nos presenta un grave problema. El monitor tiene una determinada frecuencia de actualización y si llamamos a esta función en el momento que el monitor está dibujando algo se producirá la aparición de parpadeos y guiños muy molestos a la hora de visualizar un videojuego. Este defecto es conocido como flicking.

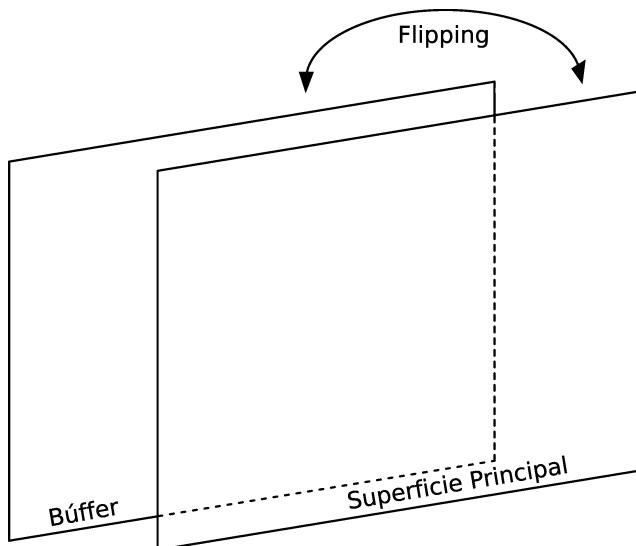


Figura 5.14: Doble Búffer

Lo ideal es que el refresco de nuestra superficie esté sincronizado justo con el momento en el que la pantalla dibuja un fotograma y así evitar efectos no deseados. Esta técnica es conocida como sincronización de refresco vertical o VSYNC.

Para conseguir esta sincronía el sistema utiliza un doble búffer. Se trata de tener disponibles, como puedes suponer, dos búfferes. Compondremos todo lo necesario en el búffer secundario con el objetivo de no dibujar nada directamente en pantalla. Cuando tengamos todo listo intercambiaremos este búffer secundario con el primario dibujando todos los elementos de una vez sincronizado con el refresco del monitor. Prácticamente es disponer de una pantalla virtual donde trabajamos las imágenes que luego serán mostradas mediante un intercambio de búfferes o flipping.

5. El Subsistema de Video

Puedes ver que implementar este comportamiento puede ser una ardua tarea para un programador medio, pero estamos de enhorabuena. SDL se encarga de todo por nuestra cuenta. Nosotros simplemente compondremos las imágenes y solicitaremos que se actualice la pantalla. El intercambio entre estos dos búfferes es conocido como flip o flipping.

Como ya vimos en ejemplos anteriores, y a nivel práctico, inicializaremos la librería SDL con el uso del doble búffer activado y a la hora de realizar la actualización de la pantalla llamaremos a *SDL_Flip()* en vez de a *SDL_UpdateRect()*.

5.6. Manejando Superficies

5.6.1. Introducción

La operación más básica que podemos realizar a la hora de desarrollar un programa gráfico es colocar píxeles en la pantalla. En este apartado mostraremos como podemos realizar esta acción usando la librería SDL. El proceso de colocar un píxel en pantalla en SDL no es de las tareas más sencillas que se pueden realizar con esta librería. Normalmente trabajaremos con mapas de bits cargados de memoria secundaria que nos permitirán una mayor versatilidad y una manera más comoda de trabajo.

5.6.2. Modificando la superficie

Para colocar un píxel en pantalla lo primero debemos verificar es si la superficie donde queremos colocar el píxel debe ser bloqueada. El concepto es sencillo. Para modificar una superficie debemos de usarla en exclusividad y para lograr esto la bloqueamos. El tener que bloquear una superficie o no es dependiente del sistema. Como regla general realizaremos el bloqueo lo que permitirá que nuestro código sea portable. En cualquier caso podremos bloquear la superficie en la que vamos a dibujar, y una vez que hayamos dibujado, deberemos de desbloquear la superficie.

Para comprobar si la superficie dónde vamos a colocar el píxel debe ser bloqueada SDL proporciona la macro *SDL_MUSTLOCK(surface)*. Esta macro devuelve 1 si la superficie en cuestión debe ser bloqueada o cero en caso contrario. Se le debe pasar un puntero a una superficie del tipo *SDL_Surface*.

Si la superficie debe ser bloqueada SDL nos ofrece la siguiente función para realizar esta tarea:

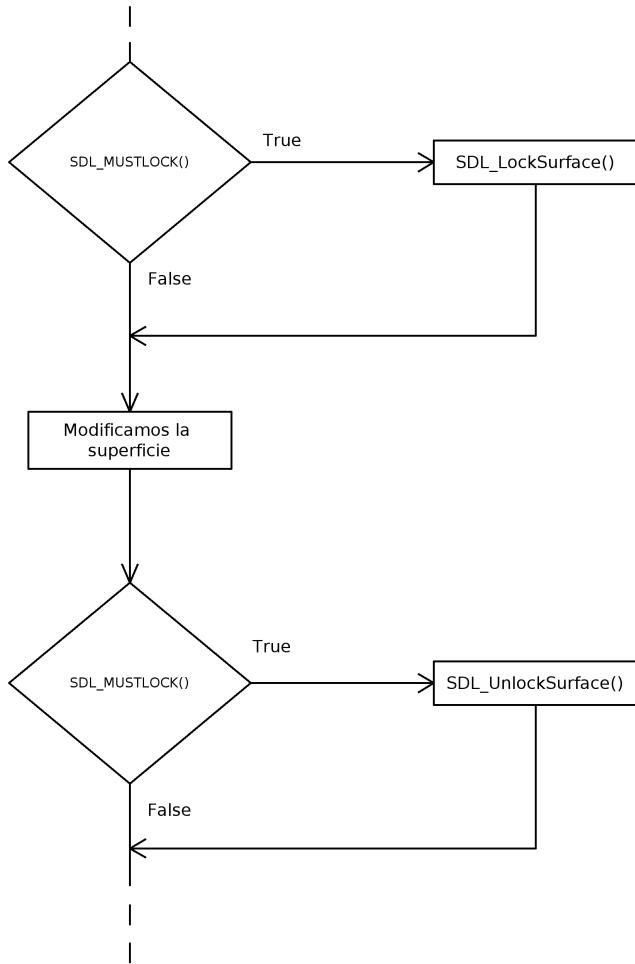


Figura 5.15: Diagrama de flujo. Modificando una superficie.

```
int SDL_LockSurface(SDL_Surface *surface);
```

Esta función recibe como parámetro la superficie que queremos bloquear. Esta función devuelve 0 si se bloquea la superficie con éxito y -1 si no se consigue.

Ahora que tenemos la superficie bloqueada por lo que podemos modificarla a nuestro antojo. En el siguiente ejemplo vamos a colocar un píxel en una superficie y así tendrás una referencia de como modificar superficies píxel a píxel. Recuerda las recomendaciones sobre el tipo de superficie a utilizar para este tipo de tareas del apartado donde estudiamos el tipo *SDL_Surface*.

Lo primero que necesitamos son dos funciones que nos permitan colocar un píxel de un color en una posición concreta y otra que nos permita consultar que la operación ha sido realizada con éxito. Con este fin implementamos las dos funciones que exponemos a continuación:

5. El Subsistema de Video

```
;  
1 ;// Listado: pixels.h  
2 ;// Fichero de Cabecera  
3 ;// Funciones para el manejo de píxeles dentro de superficies  
4 ;  
5 ;#ifndef _PIXELS_H_  
6 ;#define _PIXELS_H_  
7 ;  
8 ;  
9 ;enum colores {R, G, B};  
10 ;  
11 ;// Esta función sustituye el color del píxel (x, y) de la superficie  
12 ;// surface por el color que recibe en el parámetro pixel  
13 ;  
14 ;void PutPixel(SDL_Surface *superficie, int x, int y, Uint32 pixel);  
15 ;  
16 ;  
17 ;// Esta función devuelve el color del píxel de la  
18 ;// posición (x, y) de la superficie  
19 ;  
20 ;Uint32 GetPixel(SDL_Surface *superficie, int x, int y);  
21 ;  
22 ;#endif  
;
```

La implementación de estas funciones es:

```
;  
1 ;// Listado: pixels.c  
2 ;// Funciones para el manejo de píxeles dentro de superficies  
3 ;  
4 ;  
5 ;#include <SDL/SDL.h>  
6 ;#include "pixels.h"  
7 ;  
8 ;  
9 ;void PutPixel(SDL_Surface *superficie, int x, int y, Uint32 pixel) {  
10 ;  
11 ;    // Obtenemos la profundida de color  
12 ;  
13 ;    int bpp = superficie->format->BytesPerPixel;  
14 ;  
15 ;  
16 ;    // Obtenemos la posición del píxel a sustituir  
17 ;  
18 ;    Uint8 *p = (Uint8 *)superficie->pixels + y * superficie->pitch + x*bpp;  
19 ;  
20 ;  
21 ;    // Según sea la profundidad de color  
22 ;  
23 ;    switch (bpp) {  
24 ;  
25 ;        case 1: // 8 bits (256 colores)  
26 ;  
27 ;            *p = pixel;
```

5.6. Manejando Superficies

```
28 ;         break;
29 ;
30 ;     case 2: // 16 bits (65536 colores o HigColor)
31 ;         *(Uint16 *)p = pixel;
32 ;         break;
33 ;
34 ;     case 3: // 24 bits (True Color)
35 ;
36 ;         // Depende de la naturaleza del sistema
37 ;         // Puede ser Big Endian o Little Endian
38 ;
39 ;         if (SDL_BYTEORDER == SDL_BIG_ENDIAN) {
40 ;
41 ;             // Calculamos cada una de las componentes de color
42 ;             // 3 bytes, 3 posiciones
43 ;
44 ;             p[R]=(pixel >> 16) & 0xFF;
45 ;             p[G]=(pixel >> 8) & 0xFF;
46 ;             p[B]=pixel & 0xFF;
47 ;         }
48 ;         else {
49 ;
50 ;             // Calculamos cada una de las componentes de color
51 ;             // 3 bytes, 3 posiciones
52 ;
53 ;             p[R]=pixel & 0xFF;
54 ;             p[G]=(pixel >> 8) & 0xFF;
55 ;             p[B]=(pixel >> 16) & 0xFF;
56 ;
57 ;         }
58 ;         break;
59 ;
60 ;     case 4: // 32 bits (True Color + Alpha)
61 ;
62 ;         *(Uint32 *) p = pixel;
63 ;         break;
64 ;     }
65 ;}
66 ;
67 ;
68 ;
69 ;
70 ;Uint32 GetPixel(SDL_Surface *superficie, int x, int y) {
71 ;
72 ;    // Obtenemos la profundidad de color
73 ;
74 ;    int bpp = superficie->format->BytesPerPixel;
75 ;
76 ;
77 ;    // Obtenemos la posición del pixel a consultar
78 ;
79 ;    Uint8 *p = (Uint8 *)superficie->pixels + \
80 ;                y * superficie->pitch + x * bpp;
```

5. El Subsistema de Video

```
81 ;
82 ;
83 ;    // Según sea la profundidad de color
84 ;
85 ;    switch (bpp) {
86 ;
87 ;        case 1: // 256 colores
88 ;
89 ;            return *p;
90 ;
91 ;        case 2: // 65536 colores
92 ;
93 ;            return *(Uint16 *)p;
94 ;
95 ;        case 3: // True Color
96 ;
97 ;            // Según la naturaleza del sistema
98 ;
99 ;            if (SDL_BYTEORDER == SDL_BIG_ENDIAN)
100 ;
101 ;                // OR entre los distintos componentes del color
102 ;
103 ;                return p[R] << 16 | p[G] << 8 | p[B];
104 ;
105 ;            else
106 ;
107 ;                // OR entre los distintos componentes del color
108 ;
109 ;                return p[R] | p[G] << 8 | p[B] << 16;
110 ;
111 ;        case 4: // True Color + Alpha
112 ;
113 ;            return *(Uint32 *)p;
114 ;
115 ;        default:
116 ;
117 ;            return 0;
118 ;    }
119 ;}
```

La primera de las funciones es *PutPixel()*. La lógica de esta función es bastante simple. Recibe como parámetros una superficie *surface*, una posición que viene dada por las coordenadas (x, y) y un color expresado como un entero de 32 bits sin signo pasado en el último parámetro.

Lo primero que hacemos es obtener la profundida de color y almacenarla en la variable *bpp*. Con este fin consultamos el formato en la estructura de la superficie del parámetro de entrada. El siguiente paso es calcular la posición real en memoria de las coordenadas (x, y) . En *surface* tenemos un puntero a la primera posición de datos de la superficie, le sumamos la posición en *x* por el número de bytes por cada píxel (ya que el *bpp* provocará un mayor o

5.6. Manejando Superficies

menor desplazamiento “horizontal”) y a todo esto le sumamos la posición y por el tamaño de una fila real en memoria (de ahí utilizar el valor de *pitch*).

Si no sabes porqué realizamos esta operación puede que necesites un repaso al concepto de pitch. Sigue la ecuación mirando la figura 5.12 y lo verás todo más claro.

Con esta operación conseguimos tener un puntero al píxel que queremos modificar. La modificación del color de este píxel dependerá principalmente de la profundidad de color con la que esté definida la superficie. En el caso de que la superficie tenga un byte de color nos posibilitará asignar el color directamente al píxel en cuestión. En el caso de que sean dos los colores destinados a la profundidad de color nos bastará un casting para que todo funcione correctamente.

Si dedicamos tres bytes a la profundidad de color es que para representar cada byte en la superficie utilizaremos tres bytes. Según sea la configuración de nuestro sistema tendremos en el primer byte la información para el color rojo, en el segundo para el verde y en el tercero para el azul, o a la inversa según la configuración de nuestra máquina, como puedes ver en la estructura selectiva que selecciona el orden de byte del sistema. Si nos encontramos en este caso tendremos que actualizar cada uno de los valores de las componentes de color. Para lograr esto realizamos desplazamientos sobre el valor de *pixel* en 8 o 16 posiciones con el fin de realizar una **AND** con ocho bits a uno y así “aislar” el valor para dicha componente de color. Las constantes R, G y B pertenecen a un enumerado donde R es igual a 0, G a uno y B a dos. En la figura 5.16 puedes apreciar el proceso por el cual obtenemos la componente de color roja para poder trabajar con ella.

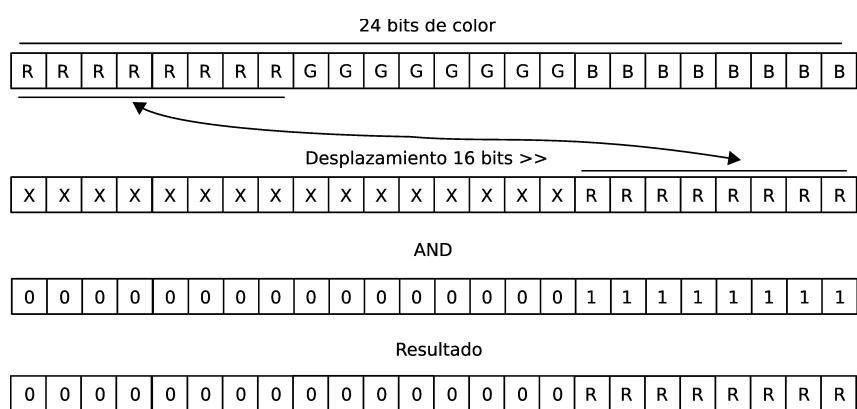


Figura 5.16: Obteniendo una componente de color de 8 bits.

En el caso de disponer de cuatro bytes para la profundidad de color la asignación será automática.

5. El Subsistema de Video

Hemos incluido una función que nos permite conocer el color de un determinado píxel. Esta función nos servirá para comprobar si hemos modificado el píxel correctamente. Se trata de la función *GetPixel()* y recibe como parámetros la superficie que queremos consultar y la posición (*x, y*) dentro de la misma. Devuelve el color de dicho píxel.

La lógica es parecida a la de la función anterior pero esta vez en vez de modificar el valor de la posición lo devolvemos mediante la sentencia *return*. En el caso de representar cada píxel con tres bytes tenemos también un caso en especial. La solución a este caso pasa esta vez por construir un valor a devolver que sea válido. Para ello realizamos desplazamientos sobre cada componente de color, según sea la naturaleza del sistema (big endian o little endian), y una OR entre ellas con el fin de construir el valor a devolver. Como puedes ver no son operaciones complicadas.

Para terminar aquí tienes el programa utilizado para probar estas funciones:

```
1 ;_____
2 ;// Listado: main.c
3 ;// Programa de prueba,
4 ;
5 ;#include <stdio.h>
6 ;#include <SDL/SDL.h>
7 ;
8 ;
9 ;
10 ;int main()
11 ;{
12 ;
13 ;    // Vamos a dibujar píxeles en pantalla
14 ;    SDL_Surface *pantalla;
15 ;
16 ;    // Variables auxiliares
17 ;
18 ;    int x, y; // Para la posición
19 ;    Uint32 color, comprobacion;
20 ;    Uint8 *buffer;
21 ;
22 ;    SDL_Event evento;
23 ;
24 ;
25 ;    // Iniciamos SDL
26 ;
27 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0){
28 ;
29 ;        fprintf(stderr, " No se pudo iniciar SDL: %s\n", SDL_GetError());
30 ;        exit(1);
31 ;
```

5.6. Manejando Superficies

```
32 ;    }
33 ;
34 ;    // Llama a SDL_Quit() al salir
35 ;
36 ;    atexit(SDL_Quit);
37 ;
38 ;    // Es compatible el modo de video?
39 ;
40 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_SWSURFACE) == 0) {
41 ;
42 ;        fprintf(stderr, "Modo no soportado: %s\n", SDL_GetError());
43 ;        exit(1);
44 ;
45 ;    }
46 ;
47 ;
48 ;    // Una vez comprobado establecemos el modo de video
49 ;
50 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_SWSURFACE);
51 ;    if(pantalla == NULL) {
52 ;
53 ;        printf("SDL_SWSURFACE 640 x 480 x 24 no compatible.\n");
54 ;        printf("Error: %s\n", SDL_GetError());
55 ;
56 ;    }
57 ;
58 ;
59 ;    // Si hay que bloquear la superficie se bloquea
60 ;
61 ;    if(SDL_MUSTLOCK(pantalla))
62 ;        SDL_LockSurface(pantalla);
63 ;
64 ;    // Procedemos a dibujar los pixeles en pantalla
65 ;
66 ;    // Posicion, por ejemplo (100, 100)
67 ;
68 ;    x = 100;
69 ;    y = 100;
70 ;
71 ;    // Vamos a dibujar un pixel verde
72 ;
73 ;    color = SDL_MapRGB(pantalla->format, 0, 255, 0);
74 ;
75 ;    // Modificamos el pixel (x, y)
76 ;
77 ;    PutPixel(pantalla, x, y, color);
78 ;
79 ;    // Comprobamos que el cambio se ha realizado correctamente
80 ;
81 ;    comprobacion = GetPixel(pantalla, x, y);
82 ;
83 ;    if(comprobacion != color)
84 ;        fprintf(stderr, "\nCambio de color KO\n");
```

5. El Subsistema de Video

```
85 ;     else
86 ;         fprintf(stdout, "\nCambio de color OK\n");
87 ;
88 ;     // Actualizamos la pantalla para mostrar el cambio
89 ;
90 ;     SDL_Flip(pantalla);
91 ;
92 ;     // Una vez dibujado procedemos a desbloquear la superficie
93 ;     // Siempre y cuando hubiese sido bloqueada
94 ;
95 ;     if(SDL_MUSTLOCK(pantalla))
96 ;         SDL_UnlockSurface(pantalla);
97 ;
98 ;     // Ahora mantenemos el resultado en pantalla
99 ;     // Hasta pulsar escape
100 ;
101 ;    for(;;) {
102 ;
103 ;        // Consultamos los eventos
104 ;
105 ;        while(SDL_PollEvent(&evento)) {
106 ;
107 ;            if(evento.type == SDL_QUIT) // Si es de salida
108 ;                return 0;
109 ;        }
110 ;    }
111 ;
112 ;}
```

Como puedes ver ponemos en el ejemplo todo lo explicado hasta ahora y un poco más. Aparecen dos aspectos nuevos. El primero es el manejo de eventos. Aunque como puedes ver no se complica el código demasiado te invito a esperar al capítulo donde vamos a estudiar los eventos para entrar en profundidad con esta materia. Por ahora es suficiente con saber que el evento reflejado en el ejemplo hará que cuando pulsemos sobre [x] de nuestra ventana la aplicación terminará. La otra función nueva que aparece es *SDL_MapRGB()*. Para obtener el color del píxel en el formato adecuado utilizamos la función con el prototipo:

*Uint32 SDL_MapRGB(SDL_PixelFormat *fmt, Uint8 r, Uint8 g, Uint8 b);*

Según el formato del píxel (bpp) la representación de un color puede variar. Esta función se encarga de, a partir del formato de píxel y la cantidad de cada color RGB que queramos establecer, devolver el color en el formato de píxel adecuado para el formato indicado.

La función necesita un puntero al formato del pixel, por lo que utilizaremos el campo format de *SDL_Surface* además de la intensidad de color rojo, verde y azul. La función devuelve un entero sin signo de 32 bits que contiene el color pedido. Si el bpp es de 16 bits el valor retornado será del tipo *Uint16*

y si es de 8 bits el tipo devuelto será de tipo `Uint8`.

Como ya vimos en el apartado donde estudiábamos las superficies, para acceder directamente a el píxel (x, y) de dicha superficie debíamos de conocer la siguiente proporción. Si `surface` es el vector de píxeles que contiene una superficie para modificar alguna propiedad de la posición (x, y) debemos de acceder a la posición `surface[(y × pitch) + (x × bytesPerPixel)]`. Ya sabemos que el `pitch` es el tamaño real que ocupa la superficie de ancho y que los `bytesPerPixel` indican la cantidad de bytes que necesitamos para representar un píxel por todo esto hay que aplicar la relación anterior para acceder directamente a un píxel. De ahí el cálculo que aparece en nuestras funciones.

Una vez modificada la superficie deberemos de volver a desbloquear la superficie. Para ello tenemos otra función SDL con el siguiente prototipo:

```
void SDL_UnlockSurface(SDL_Surface *surface);
```

Como en la función de bloqueo sólo necesitamos pasar el puntero a la superficie para que sea desbloqueada. Una vez desbloqueada la superficie deberemos de actualizar la información de pantalla para que sea visible. Con esto hemos terminado. Este es el proceso para modificar una superficie.

5.6.3. Primitivas Gráficas

Bien, ya sabemos como modificar superficies. Ahora vamos a realizar unos ejercicios que nos va a permitir tener un primer contacto con las superficies en SDL. A partir de las funciones del apartado anterior vamos a crear funciones que nos permitan realizar algunas dibujos básicos como una línea, un cuadrado o un rectángulo.

5.6.4. Ejercicio 1. La recta

Crea una función que te permita dibujar una recta en una superficie SDL. Para no complicar el ejercicio, en esta primera versión, nos vamos a limitar dibujar rectas horizontales y verticales.

Aquí tienes la solución del ejercicio. Puedes encontrar todos los listados en el material proporcionado con el curso.

```
1 ;// Listado: recta.h
2 ;//
3 ;// Fichero de cabecera
4 ;
```

5. El Subsistema de Video

```
5 ;#ifndef _RECTA_H_
6 ;#define _RECTA_H_
7 ;
8 ;// Dibuja una recta entre dos puntos p0 y p1 definidos por (x0, y0) y
9 ;// (x1, y1) del color definido por el parámetro color.
10 ;
11 ;void Recta(SDL_Surface *superficie,\n
12 ;            int x0, int y0, int x1, int y1, Uint32 color);\n
13 ;
14 ;#endif
;
```

La implementación de la función que dibuja las rectas es:

```
;_____
1 ;// Listado: recta.c
2 ;//
3 ;// Implementación de las funciones
4 ;
5 ;#include <SDL/SDL.h>
6 ;#include <math.h>
7 ;
8 ;#include "pixels.h"
9 ;#include "recta.h"
10 ;
11 ;void Recta(SDL_Surface *superficie,\n
12 ;            int x0, int y0, int x1, int y1, Uint32 color) {
13 ;
14 ;    // Calculamos la longitud de la recta
15 ;
16 ;    int longitud = sqrt(pow((x1 - x0), 2) + pow((y1 - y0), 2));
17 ;
18 ;
19 ;    // La longitud no debe ser negativa
20 ;
21 ;    if(longitud < 0)
22 ;        longitud = -1 * longitud;
23 ;
24 ;    int i = 0;
25 ;
26 ;    for(i = 0; i < longitud; i++) {
27 ;
28 ;        // Si es vertical aumento y
29 ;
30 ;        if(x0 == x1) {
31 ;
32 ;            y0++;
33 ;
34 ;            PutPixel(superficie, x0, y0, color);
35 ;
36 ;        }
37 ;
38 ;        // Si es horizontal aumento x
39 ;
40 ;        if(y0 == y1) {
```

```

41 ;
42 ;           x0++;
43 ;
44 ;           PutPixel(superficie, x0, y0, color);
45 ;
46 ;
47 ;
48 ;       }
49 ;}
;
```

5.6.5. Ejercicio 2. El cuadrado

Crea una función que dados la longitud del lado y la posición de la esquina superior izquierda dibuje un cuadrado de un determinado color en pantalla.

```

;_____
1 ;// Listado: cuadrado.h
2 ;//
3 ;// Fichero de cabecera
4 ;
5 ;#ifndef _RECTA_H_
6 ;#define _RECTA_H_
7 ;
8 ;// Dibuja un cuadrado en la superficie, del color especificado
9 ;// en el parámetro color y con su esquina superior izquierda
10;// en (x, y)
11;
12;void Cuadrado(SDL_Surface *superficie,\n
13;               int x, int y, int lado, Uint32 color);
14;
15#endif
;_____
;
1 ;// Listado: recta.c
2 ;
3 ;#include <SDL/SDL.h>
4 ;
5 ;#include "recta.h"
6 ;#include "cuadrado.h"
7 ;
8 ;void Cuadrado(SDL_Surface *superficie, \
9 ;               int x, int y, int lado, Uint32 color) {
10;
11;    // Dibujamos las verticales paralelas
12;
13;    Recta(superficie, x, y, x, y + lado, color);
14;    Recta(superficie, x + lado, y, x + lado, y + lado, color);
15;
16;    // Dibujamos las horizontales paralelas
17;
18;    Recta(superficie, x, y, x + lado, y, color);
19;    Recta(superficie, x, y + lado, x + lado, y + lado, color);
20;}
;
```

5. El Subsistema de Video

5.6.6. Ejercicio 3. La circunferencia

Crea una función que te dibuje una circunferencia en pantalla de al menos 256 puntos. Realiza un programa de prueba que te dibuje veinte circunferencias aleatorias en una superficie.

```
1 ;// Listado: circunferencia.h
2 ;// Función que crea un círculo en una superficie
3 ;
4 ;#ifndef _CIRCUNFERENCIA_H_
5 ;#define _CIRCUNFERENCIA_H_
6 ;
7 ;void Circunferencia(SDL_Surface *superficie, int x, int y,\n
8 ;                      int radio, Uint32 color);
9 ;
10 ;#endif
;
;

1 ;// Listado: circunferencia.c
2 ;// Creación de un círculo en una superficie de SDL
3 ;
4 ;#include <SDL/SDL.h>
5 ;#include <math.h>
6 ;
7 ;#include "pixels.h"
8 ;
9 ;void Circunferencia(SDL_Surface *superficie, int x, int y,\n
10 ;                      int radio, Uint32 color) {
11 ;
12     int puntos = 0;
13     float angulo_exacto;
14 ;
15     int x0 = x;
16     int y0 = y;
17 ;
18     int x_aux, y_aux;
19 ;
20     while(puntos < 256) {
21 ;
22         angulo_exacto = puntos * M_PI / 128;
23 ;
24         x = radio * cos(angulo_exacto);
25         y = radio * sin(angulo_exacto);
26 ;
27         x_aux = x + radio + x0;
28         y_aux = y + radio + y0;
29 ;
30 ;
31         // Evitamos dibujar fuera de la superficie
32 ;
33 ;
34         if(!(x_aux < 0 || y_aux < 0      \
35             || x_aux > superficie->w || y_aux > superficie->h))
36 ;
```

5.6. Manejando Superficies

```
37 ;           PutPixel(superficie, x_aux, y_aux, color);
38 ;
39 ;           puntos++;
40 ;
41 ;
42 ;}
;_____
;
1 ;// Listado: main.c
2 ;// Programa de prueba,
3 ;
4 ;#include <stdio.h>
5 ;#include <SDL/SDL.h>
6 ;
7 ;#include "circunferencia.h"
8 ;
9 ;
10 ;int main()
11 ;{
12 ;
13 ;    // Vamos a dibujar pixeles en pantalla
14 ;    SDL_Surface *pantalla;
15 ;
16 ;    // Variables auxiliares
17 ;
18 ;    Uint32 color, comprobacion;
19 ;    Uint8 *buffer;
20 ;
21 ;    SDL_Event evento;
22 ;    int i;
23 ;
24 ;    int pos_x, pos_y, radio;
25 ;
26 ;    // Llama a SDL_Quit() al salir
27 ;
28 ;    atexit(SDL_Quit);
29 ;
30 ;    // Iniciamos SDL
31 ;
32 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0){
33 ;
34 ;        fprintf(stderr, " No se pudo iniciar SDL: %s\n", SDL_GetError());
35 ;        exit(1);
36 ;
37 ;    }
38 ;
39 ;    // Es compatible el modo de video?
40 ;
41 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_SWSURFACE) == 0) {
42 ;
43 ;        fprintf(stderr, "Modo no soportado: %s\n", SDL_GetError());
44 ;        exit(1);
45 ;
46 ;    }
```

5. El Subsistema de Video

```
47 ;
48 ;
49 ;    // Una vez comprobado establecemos el modo de video
50 ;
51 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_SWSURFACE);
52 ;    if(pantalla == NULL) {
53 ;
54 ;        printf("SDL_SWSURFACE 640 x 480 x 24 no compatible.\n");
55 ;        printf("Error: %s\n", SDL_GetError());
56 ;
57 ;    }
58 ;
59 ;
60 ;    // Si hay que bloquear la superficie se bloquea
61 ;
62 ;    if(SDL_MUSTLOCK(pantalla))
63 ;        SDL_LockSurface(pantalla);
64 ;
65 ;
66 ;    // Vamos a dibujar pixeles rosa
67 ;
68 ;    color = SDL_MapRGB(pantalla->format, 128, 128, 255);
69 ;
70 ;    // Dibujamos 20 circunferencias aleatorias
71 ;
72 ;    srand(time(NULL)); // Establecemos la semilla
73 ;
74 ;    for(i = 0; i < 20; i++) {
75 ;
76 ;        // Número aleatorio dentro del rango
77 ;
78 ;        pos_x = rand() % 640;
79 ;        pos_y = rand() % 480;
80 ;        radio = rand() % 100;
81 ;
82 ;        Circunferencia(pantalla, pos_x, pos_y, radio, color);
83 ;
84 ;    }
85 ;
86 ;
87 ;    // Actualizamos la pantalla para mostrar el cambio
88 ;
89 ;    SDL_Flip(pantalla);
90 ;
91 ;    // Una vez dibujado procedemos a desbloquear la superficie
92 ;    // Siempre y cuando hubiese sido bloqueada
93 ;
94 ;    if(SDL_MUSTLOCK(pantalla))
95 ;        SDL_UnlockSurface(pantalla);
96 ;
97 ;    // Ahora mantenemos el resultado en pantalla
98 ;    // Hasta pulsar escape
99 ;
```

```
100 ;     for(;;) {
101 ;
102 ;         // Consultamos los eventos
103 ;
104 ;         while(SDL_PollEvent(&evento)) {
105 ;
106 ;             if(evento.type == SDL_QUIT) // Si es de salida
107 ;                 return 0;
108 ;         }
109 ;     }
110 ;
111 ;}
```

5.7. Cargando Mapas de Bits

5.7.1. Introducción

Hay un modo más simple de trabajar para mostrar píxeles en pantalla. Aunque una vez implementada las funciones *GetPixel()* y *PutPixel()* podemos crear figuras sin mucha dificultad imagínate tener que crear un personaje mediante este método. Tendríamos que estar compilando nuestro proyecto continuamente revisando este o aquel detalle además de tener una gran habilidad para dibujar personajes pixelizados.

Todavía no sabemos como animar un nuestro personaje aunque puedes suponer que si tenemos que ir desplazando por pantalla los dibujos que hemos realizado o simplemente modificar parte de lo dibujado mediante píxeles para mostrar que el personaje realiza una acción puede ser una tarea muy muy engorrosa.

Por todo esto SDL nos permite cargar en una superficie una imagen bitmap para que sea mostrada en pantalla. Podremos dibujar nuestros personajes en nuestro programa favorito y, guardándolo en formato BMP, podremos cargarlo directamente a una superficie de SDL. Ya conocemos la estructura de las superficies. Tienes que tener claro que la superficie es la unidad básica de SDL y estará presente en gran parte del desarrollo de nuestra aplicación.

5.7.2. ¿Cómo cargar un mapa de bits?

SDL nos ofrece la posibilidad de cargar un mapa de bits en una superficie. Para explicar el proceso que tenemos que realizar para cargar dicho mapa de bits vamos a estudiar un ejemplo. El ejemplo va a consistir en mostrar una imagen en la superficie principal de nuestra aplicación y así poder mostrarla

5. El Subsistema de Video

por pantalla. Si no conoces alguna de las funciones que vamos a utilizar, no te impacientes, van a ser todas cuidadosamente explicadas.

```
1 ;_____
2 ;// Listado: main.c
3 ;// Programa de prueba,
4 ;// Crear superficies como mostrarlas por pantalla
5 ;
6 ;#include <stdio.h>
7 ;#include <SDL/SDL.h>
8 ;
9 ;// Vamos a cargar una imagen en una superficie
10 ;// y vamos a mostrarla por pantalla
11 ;
12 ;int main()
13 ;{
14 ;
15 ;    // Declaramos los punteros para la pantalla y la imagen a cargar
16 ;
17 ;    SDL_Surface *pantalla, *imagen;
18 ;
19 ;    SDL_Event evento;
20 ;
21 ;    // Variable auxiliar donde almacenaremos la posición donde colocaremos
22 ;    // la imagen dentro de la superficie principal
23 ;
24 ;    SDL_Rect destino;
25 ;
26 ;
27 ;    // Iniciamos el subsistema de video de SDL
28 ;
29 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
30 ;        fprintf(stderr,"No se pudo iniciar SDL: %s\n", SDL_GetError());
31 ;        exit(1);
32 ;    }
33 ;
34 ;    atexit(SDL_Quit);
35 ;
36 ;
37 ;    // Comprobamos que sea compatible el modo de video
38 ;
39 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_SWSURFACE) == 0) {
40 ;
41 ;        fprintf(stderr, "Modo no soportado: %s\n", SDL_GetError());
42 ;        exit(1);
43 ;
44 ;    }
45 ;
46 ;    // Establecemos el modo de video y asignamos la superficie
47 ;    // principal a la variable pantalla
48 ;
49 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE);
50 ;}
```

5.7. Cargando Mapas de Bits

```
51 ;     if(pantalla == NULL) {
52 ;         fprintf(stderr, "No se puede inicializar el modo gráfico: %s", \
53 ;                 SDL_GetError());
54 ;     }
55 ;
56 ;     /***** En este momento tenemos la librería SDL inicializada con el subsistema *****
57 ;         de video en marcha. Además hemos comprobado y establecido el modo
58 ;         video. En este punto se nos mostrará una ventana con una superficie
59 ;         negra. Ahora vamos a cargar la imagen en una superficie
60 ;
61 ;     *****/
62 ;
63 ; // Cargamos un bmp en una nueva superficie para realizar las pruebas
64 ;
65 ;
66 ;     imagen = SDL_LoadBMP("Imagenes/ajuste.bmp");
67 ;
68 ;     if(imagen == NULL) {
69 ;         fprintf(stderr, "No se puede cargar la imagen: %s\n",
70 ;                 SDL_GetError());
71 ;         exit(1);
72 ;     }
73 ;
74 ; // Guardamos el BMP con otro nombre
75 ;
76 ;     if(SDL_SaveBMP(imagen, "./ajuste_sdl.bmp") == -1)
77 ;         fprintf(stderr, "No se puede guardar la imagen\n");
78 ;
79 ;
80 ;
81 ; // Inicializamos la variable de posición y tamaño de destino
82 ; destino.x = 150; // Posición horizontal con respecto a
83 ;                 // la esquina sup right
84 ; destino.y = 120; // Posición vertical con respecto a la esq sup right
85 ; destino.w = imagen->w; // Longitud del ancho de la imagen
86 ; destino.h = imagen->h; // Longitud del alto de la imagen
87 ;
88 ;
89 ; // Copiamos la superficie creada rectángulo en la anterior pantalla
90 ;     SDL_BlitSurface(imagen, NULL, pantalla, &destino);
91 ;
92 ; // Mostramos la pantalla
93 ;     SDL_Flip(pantalla);
94 ;
95 ; // Ya podemos liberar el rectángulo, una vez copiado y mostrado
96 ;     SDL_FreeSurface(imagen);
97 ;
98 ; // Ahora mantenemos el resultado en pantalla
99 ; // hasta cerrar la ventana
100 ;
101 ; for(;;) {
102 ;
103 ;     // Consultamos los eventos
```

5. El Subsistema de Video

```
104 ;
105 ;     while(SDL_PollEvent(&evento)) {
106 ;
107 ;         if(evento.type == SDL_QUIT) // Si es de salida
108 ;             return 0;
109 ;     }
110 ;
111 ;
112 ;
113 ;
114 ;     return 0;
115 ;}
```

Vamos a estudiar el código. *SDL_Surface *pantalla* y **imagen* son punteros a superficies que vamos a utilizar en el desarrollo de esta aplicación. *pantalla* será la superficie principal que será mostrada por pantalla mientras que en *imagen* almacenaremos el mapa de bits.

Puedes observar como definimos una variable del tipo *SDL_Event*. Esta variable nos permitirá controlar que la aplicación termine sólo cuando pulsemos [x] o cuando reciba un evento de terminación del sistema.

En la estructura *SDL_Rect* estableceremos, dentro de unas líneas, el tamaño de la imagen a mostrar, así como la posición donde queremos que se muestre dentro de la superficie principal, de ahí que hayamos decidido darle el nombre de *destino*.

La función *atexit(SDL_Quit)* nos permite indicar al sistema que al finalizar la aplicación la librería SDL debe ser cerrada mediante la función *SDL_Quit()*, por eso recibe como parámetro *SDL_Quit*. Nos libera de tener que realizar llamadas a *SDL_Quit()* en distintos puntos del código. Esta función se encarga de liberar los recursos ocupados por los componentes de SDL y de liberar la memoria asignada a las variables propias de SDL.

A continuación iniciamos el subsistema de vídeo y establecemos el modo de vídeo con las opciones antes estudiadas. La siguiente función que nos encontramos es:

*SDL_Surface *SDL_LoadBMP(const char *file);*

Esta función se encarga de cargar una imagen mapa de bits, y sólo de este tipo, de cualquier profundidad de color en memoria. Recibe como parámetro la ruta dónde se encuentra almacenada la imagen que queremos asignar a la superficie que devuelve la función. En este caso la imagen debe estar almacenada en formato *bmp*. si la carga de la imagen tiene éxito devuelve una superficie de tamaño el de la imagen, en caso contrario devuelve **NULL**.

5.7. Cargando Mapas de Bits

Una función que puede resultarnos útil, complemento de la anterior, es la que guarda un mapa de bits de memoria secundaria en formato BMP. El prototipo se le corresponde con el siguiente:

```
int SDL_SaveBMP(SDL_Surface *surface, const char *file);
```

Esta función recibe como parámetros la superficie que queremos guardar y en segundo lugar el nombre que le queremos dar al nuevo fichero. Como las demás funciones de SDL devuelve 0 en caso de éxito y -1 si hubo algún error. Para ejercitarse en el uso de esta función puedes verla incluida en el fichero que tomamos como ejemplo. Puedes comprobar como se guarda la imagen correctamente.

La siguiente función aparece en el listado es:

```
void SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect, SDL_Surface *dst, SDL_Rect *dstrect);
```

Esta es una de las funciones más importantes de SDL como ya vimos en la sección en la que estudiamos el *blit*. Se encarga de copiar zonas rectangulares de una superficie a otra, o como se suele decir habitualmente, vuelca el contenido de una superficie en otra. En nuestro listado utilizamos para copiar la superficie de la imagen cargada en la superficie principal que será la que se muestra en pantalla. Es decir, mediante una copia de superficies mostraremos nuestra imagen en pantalla.

Entre los parámetros de la función hay dos *SDL_Surface* *. *src* viene de source y es la superficie origen a copiar mientras que *dst* es la superficie destino de la copia, por lo que con esta función volcaremos el contenido de *src* a *dst*.

En cuanto al tipo de datos *SDL_Rect* * también observamos que hay dos parámetros de este tipo. *srcrect* hace referencia a el rectángulo a copiar de la superficie origen mientras que *dstrect* especifica los parámetros para realizar la copia en el destino. Estos parámetros son los conocidos de *SDL_Rect* como la posición (x, y), anchura y profundidad de la superficie que queremos copiar.

En el listado hemos utilizado la estructura *SDL_Rect* para copiar la imagen en la posición (150, 120). Así mismo el tamaño debe ser el de la imagen, por ello hemos asignado los valores de anchura y altura de la imagen a la variable correspondiente dentro de *destino*.

Al parámetro *SDL_Rect* * *srcrect* de la función *SDL_BlitSurface()* le hemos dado el valor NULL. Esto indica que queremos copiar toda la superficie origen. En el caso de que *srcrect* y *dstrect* no coincidan no ocurriría nada ya

5. El Subsistema de Video

que los parámetros *w* y *h* de *dsrrect* se ignoran.

Si la copia se realiza con éxito la función devuelve el valor 0. En el caso de existir algún problema la función devolverá el valor -1.

Mediante la función:

```
int SDL_FillRect(SDL_Surface *dst, SDL_Rect *dstrect, Uint32 color);
```

Podemos dibujar rectángulos en la pantalla. Podemos utilizar la capacidad de esta función y de *SDL_Surface* para borrar la pantalla, o lo que es lo mismo, dibujar un rectángulo negro en la pantalla principal. Los parámetros de esta función son la superficie de destino *dst*, el rectángulo que queremos llenar *dstrect*. y el color que podemos obtener mediante la función *SDL_MapRGB()*. Esta función ya nos es familiar ya que en la mayoría de las funciones que se trabaja con color necesitamos hacer uso de la misma para obtener dicho color en el formato de píxel con el que estemos trabajando. Si le pasamos en el parámetro *dstrect* el valor **NULL** la función entenderá que queremos aplicar el relleno a toda la pantalla. En este ejemplo no vamos a hacer uso de ella pero cuando empecemos a realizar nuestras primeras animaciones la tendremos muy presente.

La siguiente función que aparece en el código es:

```
int SDL_Flip(SDL_Surface *screen);
```

Esta función intercambia los búfferes existentes. Recuerda que para evitar efectos indeseados cuando trabajamos con animaciones o mostramos muchas superficies en una pantalla el sistema puede trabajar con un doble búffer. Uno oculto donde se dibujarán, copiarán y preparán la superficie a mostrar y uno principal que será el mostrado por pantalla. Un vez que tengamos listo el búffer oculto tendremos que llamar a esta función para que lo pase a primer plano y así mostrarlo en pantalla. Para que se realice esta acción debemos estar utilizando doble búffer si no fuese así esta función “sólo” refrescaría la pantalla. Como en muchas de las funciones SDL si el intercambio es satisfactorio es que todo ha ido como se esperaba por lo que la función devolvería el valor 0. En caso de existir algún error devuelve el valor -1.

Como ya hemos visto, la utilización y compatibilidad con el doble búffer permite que no aparezcan parpadeos y guiños (flicking) en la pantalla. Esto es debido a que se dibuja todo lo necesario en una pantalla virtual y cuando está todo pintado se hace visible (flipping). Imagina que no existiese la pantalla virtual. Tendríamos que cargar consecutivamente cada una de las imágenes directamente en la pantalla principal, si son muchas y algo pesadas puede provocar que apreciemos guiños en la misma ya que puede transcurrir un tiempo considerable entre que cargamos la primera imagen y la, por ejemplo,

veinte. De esta forma cargamos todas las imágenes en la pantalla oculta o virtual descrita anteriormente y se procede a su carga una vez completadas las tareas sobre ella. Este tema será tratado con más detenimiento en el desarrollo del curso. Este proceso se hace de forma sincronizada con el retardo vertical del monitor consiguiendo que no aparezcan, los antes comentados, parpadeos.

En el caso de que el sistema no fuese compatible con el doble búffer nos veríamos obligados a usar la función:

```
void SDL_UpdateRect(SDL_Surface *screen, Sint32 x, Sint32 y, Sint32 w,  
                    Sint32 h);
```

Podemos llamarla explícitamente si sabemos que no vamos a usar el doble búffer o lo hará *SDL_Flip()* implícitamente al comprobar que no puede realizar el intercambio de búfferes. Dicha función se hace cargo de que el área que especificamos en sus parámetros esté actualizada. Los parámetros como puedes comprobar ya nos son familiares. Especificamos la superficie a actualizar en el parámetro *screen*, la posición (*x,y*) así como el tamaño de la superficie ancho (*w*) y alto (*h*). Si inicializamos estos valores a 0 SDL actualizará toda la superficie, que en este caso deber ser la pantalla.

Existe otra función que nos permite actualizar superficies rectangulares, pero agrupadas. Se trata de:

```
void SDL_UpdateRects(SDL_Surface *screen, int numrects, SDL_Rect  
                     *rects);
```

El formato es parecido al de la función anterior. Puedes observar que la llamada difiere en un 's', por lo que debemos de ser cuidadosos al realizar las llamadas a estas funciones. En este caso la función recibe como parámetro la superficie principal de la aplicación así como el número de rectángulos que queremos actualizar. Como tercer parámetro recibe un puntero a un vector que contiene las estructuras que definen dicho recuadro. Esta es una buena opción si sólo queremos actualizar ciertos recuadros en la pantalla.

La última función SDL que utilizamos en este listado es:

```
void SDL_FreeSurface(SDL_Surface *surface);
```

Utilizamos un lenguaje que no posee recolección de basura por lo que necesitamos liberar manualmente aquellos elementos que ya no necesitamos para optimizar el consumo de recursos. De esto se encarga este procedimiento. Libera los recursos consumidos por la superficie que recibe como parámetro. Cada vez que acabemos de usar una superficie deberemos de realizar una llamada a esta función por el bien del rendimiento de nuestro sistema y de la aplicación.

5. El Subsistema de Video

Las últimas líneas de código es la espera de un evento de salida, lo que mantendrá la pantalla creada por la SDL abierta hasta que reciba dicho evento. Esto nos permitirá observar el resultado de nuestro listado.

5.7.3. Ejercicio 4

Como puedes ver es muy sencillo cargar superficies en SDL. De todas formas vamos a realizar un pequeño ejercicio de maquetación. Sigue la plantilla de la figura 5.17 y compón esta superficie para que sea mostrada por pantalla.

La superficie tiene un tamaño de 640 x 480 píxeles y el recuadro negro unas dimensiones de 100 x 100.

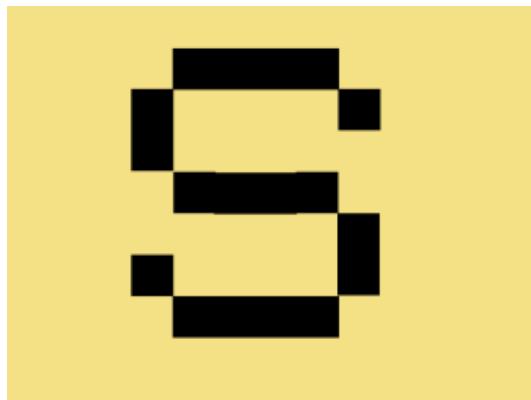


Figura 5.17: Plantilla Ejercicio 4

El objetivo de este ejercicio es simular la creación de imágenes mediante la modificación de píxeles como hacíamos en la sección anterior. Creamos una 'S' gigante con píxeles gigantes de la misma manera que tendríamos que modificar los píxeles para crear una pequeña "s" en pantalla mediante el acceso a los píxeles de una superficie.

Aquí tienes el resultado:

```
1 ;// Listado: main.c
2 ;//
3 ;// Programa de prueba,
4 ;
5 ;#include <stdio.h>
6 ;#include <SDL/SDL.h>
7 ;
8 ;int main()
```

5.7. Cargando Mapas de Bits

```
9 ;{
10 ;
11 ;    // Vamos a dibujar pixeles en pantalla
12 ;    SDL_Surface *pantalla, *recuadro;
13 ;
14 ;    // Variables auxiliares
15 ;
16 ;    SDL_Rect destino;
17 ;    Uint32 color;
18 ;    SDL_Event evento;
19 ;    int i;
20 ;
21 ;    // Llama a SDL_Quit() al salir
22 ;
23 ;    atexit(SDL_Quit);
24 ;
25 ;    // Iniciamos SDL
26 ;
27 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0){
28 ;
29 ;        fprintf(stderr, " No se pudo iniciar SDL: %s\n", SDL_GetError());
30 ;        exit(1);
31 ;
32 ;    }
33 ;
34 ;    // Es compatible el modo de video?
35 ;
36 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_SWSURFACE) == 0) {
37 ;
38 ;        fprintf(stderr, "Modo no soportado: %s\n", SDL_GetError());
39 ;        exit(1);
40 ;
41 ;    }
42 ;
43 ;
44 ;    // Una vez comprobado establecemos el modo de video
45 ;
46 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_SWSURFACE);
47 ;    if(pantalla == NULL) {
48 ;
49 ;        printf("SDL_SWSURFACE 640 x 480 x 24 no compatible.\n");
50 ;        printf("Error: %s\n", SDL_GetError());
51 ;
52 ;    }
53 ;
54 ;
55 ;    // Obtenemos el color deseado
56 ;
57 ;    color = SDL_MapRGB(pantalla->format, 57, 12, 101);
58 ;
59 ;    // Establecemos el fondo de la pantalla
60 ;
61 ;    SDL_FillRect(pantalla, NULL, color);
```

5. El Subsistema de Video

```
62 ;
63 ;    // Cargamos el recuadro negro
64 ;
65 ;    recuadro = SDL_LoadBMP("Imagenes/recuadro.bmp");
66 ;
67 ;    // Creamos la imagen en forma de S
68 ;
69 ;    // Primera linea
70 ;
71 ;    destino.y = 50;
72 ;
73 ;    for(i = 0; i < 4; i++) {
74 ;
75 ;        destino.x = 200 + i * 50;
76 ;        SDL_BlitSurface(recuadro, NULL, pantalla, &destino);
77 ;
78 ;    }
79 ;
80 ;    SDL_Flip(pantalla);
81 ;    sleep(1);
82 ;
83 ;    // Segunda linea
84 ;
85 ;    destino.x = 150;
86 ;    destino.y = 100;
87 ;
88 ;    SDL_BlitSurface(recuadro, NULL, pantalla, &destino);
89 ;
90 ;    destino.x = 400;
91 ;    destino.y = 100;
92 ;
93 ;    SDL_BlitSurface(recuadro, NULL, pantalla, &destino);
94 ;
95 ;    SDL_Flip(pantalla);
96 ;    sleep(1);
97 ;
98 ;    // Tercera linea
99 ;
100 ;    destino.x = 150;
101 ;    destino.y = 150;
102 ;
103 ;    SDL_BlitSurface(recuadro, NULL, pantalla, &destino);
104 ;
105 ;    SDL_Flip(pantalla);
106 ;    sleep(1);
107 ;
108 ;    // Cuarta linea
109 ;
110 ;    destino.y = 200;
111 ;
112 ;    for(i = 0; i < 4; i++) {
113 ;
114 ;        destino.x = 200 + i * 50;
```

5.7. Cargando Mapas de Bits

```
115 ;         SDL_BlitSurface(recuadro, NULL, pantalla, &destino);
116 ;
117 ;     }
118 ;
119 ;     SDL_Flip(pantalla);
120 ;     sleep(1);
121 ;
122 ;
123 ;     // Quinta linea
124 ;
125 ;     destino.x = 400;
126 ;     destino.y = 250;
127 ;
128 ;     SDL_BlitSurface(recuadro, NULL, pantalla, &destino);
129 ;
130 ;
131 ;     // Sexta linea
132 ;
133 ;     destino.x = 150;
134 ;     destino.y = 300;
135 ;
136 ;     SDL_BlitSurface(recuadro, NULL, pantalla, &destino);
137 ;
138 ;     destino.x = 400;
139 ;     destino.y = 300;
140 ;
141 ;     SDL_BlitSurface(recuadro, NULL, pantalla, &destino);
142 ;
143 ;     SDL_Flip(pantalla);
144 ;     sleep(1);
145 ;
146 ;     // Séptima linea
147 ;
148 ;     destino.y = 350;
149 ;
150 ;     for(i = 0; i < 4; i++) {
151 ;
152 ;         destino.x = 200 + i * 50;
153 ;         SDL_BlitSurface(recuadro, NULL, pantalla, &destino);
154 ;
155 ;     }
156 ;
157 ;     // Actualizamos la pantalla para mostrar el cambio
158 ;
159 ;     SDL_Flip(pantalla);
160 ;
161 ;
162 ;     // Ahora mantenemos el resultado en pantalla
163 ;     // Hasta pulsar escape
164 ;
165 ;     for(;;) {
166 ;
167 ;         // Consultamos los eventos
```

5. El Subsistema de Video

```
168 ;
169 ;     while(SDL_PollEvent(&evento)) {
170 ;
171 ;         if(evento.type == SDL_QUIT) // Si es de salida
172 ;             return 0;
173 ;     }
174 ; }
175 ;
176 ;}
```

A lo largo del curso, y en la creación de nuestro videojuego, la tarea de cargar imágenes en superficies la realizaremos con mucha asiduidad así que no te preocupes que practicaremos mucho más con este tipo de dato.

5.8. Transparencias

5.8.1. Introducción

El poder establecer un color transparente en nuestras superficies es fundamental para realizar cualquiera aplicación en SDL. A primera vista no parece algo realmente importante. Vamos a intentar cambiar este punto de vista.

Las transparencias son un perfecto compañero de viaje del blitting. Recuerda que el blit es una copia de un bloque de píxeles de una superficie a otra. Esto no suele ser suficiente. Normalmente necesitaremos que parte de la imagen no sea transferida durante el blit. Esto es debido a que tenemos que crear nuestros personajes en un fichero de forma rectangular sobre un color puro que no querremos que se muestre en pantalla.

Al trabajar con mapas de bits no se nos permite trabajar con transparencias implícitamente ni con formatos de fichero diferentes al rectángulo. Normalmente nuestros personajes no se adaptan a un rectángulo exceptuando casos muy particulares. Para simular el efecto de la transparencia utilizaremos un color de fondo que no sea común para que durante el blitting no se transfiera a la superficie principal, por lo que no será mostrado por pantalla.

El objetivo de que este color no se muestre en pantalla es, principalmente, conseguir un mejor acabado. No sería lógico que todos los personajes de nuestro videojuego tuviesen que ser cuadrados, o al menos estar envueltos en un cuadro visible al jugador.

Como acabamos de exponer diseñaremos a nuestro personaje con un color de fondo puro que nos permita “eliminarlo” a la hora de realizar el blitting creando un efecto de transparencia. Este color de fondo es conocido como

color key o color clave. Un buen color para esta tarea puede ser el verde puro, o el magenta compuesto de 255 de rojo y 255 de azul, ya que son colores poco habituales en el desarrollo de videojuegos. Si habías pensado utilizar en tu aplicación esta combinación de colores sólo tienes que elegir otro color key y problema resuelto.

5.8.2. ¿Para qué un color clave?

Para explicar este apartado primero vamos a realizar un pequeño ejercicio. Con lo ya visto debemos de ser capaces de iniciar la librería, el subsistema de vídeo y de cargar una imagen en pantalla. La prueba es la siguiente:

Iniciaremos el subsistema de vídeo de SDL y establecemos el modo de vídeo en 640 x 480 x 24 e introduciremos la imagen que se nos proporciona del personaje principal (pprincipal.bmp) en la posición (140, 180) de la pantalla. Deberás activar el doble búffer así como almacenar todo lo necesario en la memoria de vídeo.

```
1 ;// Listado: main.c
2 ;//
3 ;// Programa de prueba,
4 ;// Carga una imagen de un personaje en pantalla
5 ;
6 ;#include <stdio.h>
7 ;#include <SDL/SDL.h>
8 ;
9 ;int main()
10 ;{
11 ;    // Declaramos los punteros para la pantalla y la imagen a cargar
12 ;
13 ;    SDL_Surface *pantalla, *personaje;
14 ;
15 ;    // Variables auxiliares
16 ;
17 ;    SDL_Rect posicion;
18 ;    SDL_Event evento;
19 ;
20 ;
21 ;    // Iniciamos el subsistema de video SDL
22 ;    if( SDL_Init(SDL_INIT_VIDEO) < 0) {
23 ;        fprintf(stderr, "No se pudo iniciar SDL: %s\n", SDL_GetError());
24 ;        exit(1);
25 ;    }
26 ;
27 ;    atexit(SDL_Quit);
28 ;
29 ;    // Comprobamos que sea compatible el modo de video
30 ;
```

5. El Subsistema de Video

```
31 ;     if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
32 ;
33 ;         fprintf(stderr, "Modo no soportado: %s\n", SDL_GetError());
34 ;         exit(1);
35 ;
36 ;     }
37 ;
38 ;
39 ;     // Establecemos el modo de video y asignamos la superficie
40 ;     // principal a la variable pantalla
41 ;
42 ;     pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
43 ;
44 ;     if(pantalla == NULL) {
45 ;         fprintf(stderr, "No se pudo establecer el modo de video: %s\n", \
46 ;                 SDL_GetError());
47 ;         exit(1);
48 ;     }
49 ;
50 ;     // Cargamos la imagen del personaje principal
51 ;
52 ;     personaje = SDL_LoadBMP("Imagenes/pprincipal.bmp");
53 ;
54 ;     if(personaje == NULL) {
55 ;         fprintf(stderr, "No se pudo cargar la imagen: %s\n", SDL_GetError());
56 ;         exit(1);
57 ;     }
58 ;
59 ;     // Vamos a inicializar los valores de la posicion y tamaño de la imagen
60 ;
61 ;     posicion.x = 140;
62 ;     posicion.y = 180;
63 ;     posicion.w = personaje->w;
64 ;     posicion.h = personaje->h;
65 ;
66 ;     // Copiamos la imagen en la superficie principal en posición
67 ;
68 ;     SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
69 ;
70 ;     // Intercambiamos las pantallas "oculta" del búffer
71 ;
72 ;     SDL_Flip(pantalla);
73 ;
74 ;     // Liberamos los recursos que no necesitamos
75 ;
76 ;     SDL_FreeSurface(personaje);
77 ;
78 ;     // Ahora mantenemos el resultado en pantalla
79 ;     // hasta cerrar la ventana
80 ;
81 ;     for(;;) {
82 ;
83 ;         // Consultamos los eventos
```

```

84 ;
85 ;     while(SDL_PollEvent(&evento)) {
86 ;
87 ;         if(evento.type == SDL_QUIT) // Si es de salida
88 ;             return 0;
89 ;     }
90 ;
91 ;
92 ;}
;
```

Bien, no ha resultado complicado cargar la imagen ¿no?. Te habrá llamado la atención el color verde que tiene la imagen de fondo y lo artificial que queda sobre el fondo negro, salta a la vista. Una posible solución sería abrir nuestro editor gráfico favorito y cambiar el color de fondo al personaje.

Esto nos salvaría de este apuro, pero y si el fondo fuese una pared de ladrillos con ventanas, ¿cómo harías para que el personaje tuviese un fondo adecuado en cada momento? Lo has visto claro, utilizando transparencias para el color de fondo del personaje. Esta es la solución que nos ofrece SDL. Durante el blitting no se transferirán los píxeles de color verde a la superficie destino, queremos que esos píxeles sean transparentes.

5.8.3. Color Key

La técnica del *color clave* consiste en decidir que color queremos que sea transparente durante el juego y así conseguir un acabado más adecuado, más profesional. En el caso de nuestro personaje principal, y para nuestro tutorial, hemos decidido que este color sea el verde puro (0, 255, 0). Habrás visto muchos decorados en televisión o cine que son totalmente azules o verdes. Estos decorados sirven precisamente para aplicar unas técnicas parecidas a las que vamos a utilizar nosotros. Elegimos este color porque es bastante singular y es raro encontrarlo como parte de una imagen. En la figura 5.18 puedes ver un ejemplo de la aplicación de esta técnica.



Figura 5.18: Ejemplo de personaje sin y con colorkey activado

SDL nos proporciona esta función:

```
int SDL_SetColorKey(SDL_Surface *surface, Uint32 flag, Uint32 key);
```

5. El Subsistema de Video

Esta función nos permite establecer el color “clave”, o lo que es lo mismo, establecer el color transparente en una determinada superficie para que los píxeles de este color no se transfieran al realizar un blit. Los píxeles de este color no serán traspasados a la superficie destino cuando realizemos el blitting. Como es habitual si la función comete su tarea con éxito devuelve el valor 0, y -1 en caso de que no.

Los parámetros que recibe la función son similares a los estudiados para otras funciones. *surface* es la superficie donde establecer el color de la transparencia de blit. El parámetro *flag* puede tomar los siguientes valores:

- 0 para desactivar una transparencia previamente activada.
- **SDL_SRCCOLORKEY** para indicar que el tercer parámetro de la función corresponde al color que queremos que sea transparente.
- **SDL_RLEACCEL** esta opción nos permite usar codificación RLE (*Run Length Encoded*) en la superficie para acelerar el blitting. Evita redundancia de almacenamiento y envío de datos ya que en lugar de almacenar los datos que no van a existir visiblemente (como las transparencias) almacena una cifra que indica cuantos píxeles no han de dibujarse. Para usar esta bandera es estrictamente necesario combinarla con la opción **SDL_SRCCOLORKEY**.

En el tercer parámetro definimos que color queremos que sea transparente. Este color debe ser expresado en el mismo formato de color de la imagen, es decir en el mismo *pixel format*, o formato de píxel, que la superficie. Para conseguir el valor del color que buscamos en este formato usamos la función *SDL_MapRGB()* a la que especificamos el formato y la cantidad de color rojo, verde y azul necesaria para definir el color que queremos obtener. Recordamos el uso de esta función es:

```
Uint32 SDL_MapRGB(SDL_PixelFormat *fmt, Uint8 r, Uint8 g, Uint8 b);
```

Esta función nos devuelve el color deseado en una variable de 32 bits entera sin signo que precisamente es la que necesitamos como parámetro de la función expuesta anteriormente.

SDL_Surface tiene un campo en su estructura donde define el formato de píxel de la superficie. Dependiendo de este formato (bpp) la representación de un color puede variar. Esta función se encarga que a partir del formato de píxel y la cantidad de cada color RGB que queramos establecer devuelva el color en dicho formato de píxel. Para el verde, que es nuestro caso, utilizaremos (0, 255, 0).

5.8.4. Ejemplo 5. Aplicando el color clave

Como no podía ser de otra forma ahora vamos a completar el ejercicio anterior haciendo que el color de fondo del personaje sea transparente.

```
1 ;// Listado: main.c
2 ;// Programa de prueba,
3 ;// Cargar una imagen de un personaje en pantalla, con color key
4 ;
5 ;#include <stdio.h>
6 ;#include <SDL/SDL.h>
7 ;
8 ;int main()
9 ;{
10 ;
11 ;    SDL_Surface *pantalla, *personaje;
12 ;    SDL_Rect posicion;
13 ;    SDL_Event evento;
14 ;
15 ;    // Iniciamos el subsistema de video SDL
16 ;
17 ;    if( SDL_Init(SDL_INIT_VIDEO) < 0) {
18 ;        fprintf(stderr, "No se pudo iniciar SDL: %s\n", SDL_GetError());
19 ;        exit(1);
20 ;    }
21 ;
22 ;    atexit(SDL_Quit);
23 ;
24 ;
25 ;    // Establecemos el modo de video
26 ;
27 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
28 ;
29 ;    if(pantalla == NULL) {
30 ;        fprintf(stderr, "No se pudo establecer el modo de video: %s\n", \
31 ;                SDL_GetError());
32 ;        exit(1);
33 ;    }
34 ;
35 ;    // Cargamos la imagen del personaje principal
36 ;
37 ;    personaje = SDL_LoadBMP("Imagenes/pprincipal.bmp");
38 ;
39 ;    if(personaje == NULL) {
40 ;        fprintf(stderr, "No se pudo cargar la imagen: %s\n", SDL_GetError());
41 ;        exit(1);
42 ;    }
43 ;
44 ;    // Establecemos el color de la transparencia
45 ;    // No será mostrado al realizar el blitting
46 ;
47 ;    SDL_SetColorKey(personaje, SDL_SRCCOLORKEY|SDL_RLEACCEL,\
```

5. El Subsistema de Video

```
48 ;           SDL_MapRGB(personaje->format, 0, 255, 0));
49 ;
50 ;   // Vamos a inicializar los valores de la posicion y tamaño de la imagen
51 ;
52 ;   posicion.x = 140;
53 ;   posicion.y = 180;
54 ;   posicion.w = personaje->w;
55 ;   posicion.h = personaje->h;
56 ;
57 ;   // Copiamos la imagen en la superficie principal
58 ;
59 ;   SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
60 ;
61 ;   // Mostramos la pantalla "oculta" del buffer
62 ;
63 ;   SDL_Flip(pantalla);
64 ;
65 ;   // Liberamos los recursos que no necesitamos
66 ;
67 ;   SDL_FreeSurface(personaje);
68 ;
69 ;   // Ahora mantenemos el resultado en pantalla
70 ;   // hasta cerrar la ventana
71 ;
72 ;   for(;;) {
73 ;
74 ;       // Consultamos los eventos
75 ;
76 ;       while(SDL_PollEvent(&evento)) {
77 ;
78 ;           if(evento.type == SDL_QUIT) // Si es de salida
79 ;               return 0;
80 ;       }
81 ;   }
82 ;}
```

El color clave o *color key* es una propiedad de cada una de las superficies, así que podemos hacer que cada superficie tenga un color clave diferente. El color clave que se considera en el blitting es el de la superficie origen. No tendría sentido poder establecer colores claves diferentes si se especificase en la superficie destino ya que sería común para todas aquellas que volcasemos a ella.

5.8.5. Ejercicio 5

Vamos a construir un semáforo inverso. En vez de encender una de las luces cuando quiere indicar algo la apaga. Dibuja un semáforo y haz que apague un color cada cierto tiempo. La idea es que utilices los colores clave para cambiar de color en vez de superponer superficies. Recuerda que para que temporices el cambio de color puedes usar la función *sleep()*.

Aquí tienes la solucion:

5.8. Transparencias

```
1 ;  
2 ;// Listado: main.c  
3 ;// Programa de prueba,  
4 ;// Cargar una imagen de un personaje en pantalla, con color key  
5 ;  
6 ;// Colores: ( R, G, B )  
7 ;// Rojo: ( 255, 0, 0 )  
8 ;// Naranja: ( 255, 168, 0 )  
9 ;// Verde: ( 0, 255, 0 )  
10;  
11 ;#include <stdio.h>  
12 ;#include <SDL/SDL.h>  
13;  
14 ;int main()  
15 ;{  
16 ;    SDL_Surface *pantalla, *semaforo;  
17 ;    SDL_Rect posicion;  
18 ;    SDL_Event evento;  
19 ;  
20 ;    int R, G, B, i;  
21 ;    int colores[3][3];  
22 ;    Uint32 negro;  
23 ;  
24 ;    // Inicializamos la matriz con los posibles valores  
25 ;  
26 ;    // Semáforo en rojo  
27 ;  
28 ;    colores[0][0] = 255;  
29 ;    colores[0][1] = 0;  
30 ;    colores[0][2] = 0;  
31 ;  
32 ;    // Semáforo en naranja  
33 ;  
34 ;    colores[1][0] = 255;  
35 ;    colores[1][1] = 168;  
36 ;    colores[1][2] = 0;  
37 ;  
38 ;    // Semáforo en verde  
39 ;  
40 ;    colores[2][0] = 0;  
41 ;    colores[2][1] = 255;  
42 ;    colores[2][2] = 0;  
43 ;  
44 ;    // Iniciamos el subsistema de video SDL  
45 ;  
46 ;    if( SDL_Init(SDL_INIT_VIDEO) < 0 ) {  
47 ;        fprintf(stderr, "No se pudo iniciar SDL: %s\n", SDL_GetError());  
48 ;        exit(1);  
49 ;    }  
50 ;  
51 ;    atexit(SDL_Quit);  
52 ;  
53 ;
```

5. El Subsistema de Video

```
54 ; // Establecemos el modo de video
55 ;
56 ; pantalla = SDL_SetVideoMode(160, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
57 ;
58 ; if(pantalla == NULL) {
59 ;     fprintf(stderr, "No se pudo establecer el modo de video: %s\n", \
60 ;             SDL_GetError());
61 ;     exit(1);
62 ; }
63 ;
64 ; negro = SDL_MapRGB(pantalla->format, 0, 0, 0);
65 ;
66 ; // Cargamos la imagen del semáforo
67 ;
68 ; semaforo = SDL_LoadBMP("Imagenes/semaforo.bmp");
69 ;
70 ; if(semaforo == NULL) {
71 ;     fprintf(stderr, "No se pudo cargar la imagen: %s\n", SDL_GetError());
72 ;     exit(1);
73 ; }
74 ;
75 ; // Vamos a inicializar los valores de la posición y tamaño de la imagen
76 ;
77 ; posicion.x = 0;
78 ; posicion.y = 0;
79 ; posicion.w = semaforo->w;
80 ; posicion.h = semaforo->h;
81 ;
82 ; i = 0;
83 ;
84 ; for(;;) {
85 ;
86 ;     // Establecemos el color de la transparencia
87 ;     // No será mostrado al realizar el blitting
88 ;     if(i > 2)
89 ;         i = 0;
90 ;
91 ;     R = colores[i][0];
92 ;     G = colores[i][1];
93 ;     B = colores[i][2];
94 ;
95 ;     i++;
96 ;
97 ;     // Establecemos el color de la transparencia
98 ;
99 ;     SDL_SetColorKey(semaforo, SDL_SRCCOLORKEY | SDL_RLEACCEL,\n
100 ;                     SDL_MapRGB(semaforo->format, R, G, B));
101 ;
102 ;     SDL_FillRect(pantalla, NULL, negro);
103 ;
104 ;     // Copiamos la imagen en la superficie principal
105 ;
106 ;     SDL_BlitSurface(semaforo, NULL, pantalla, &posicion);
```

```

107 ;
108 ;      // Mostramos la pantalla "oculta" del búffer
109 ;
110 ;      SDL_Flip(pantalla);
111 ;
112 ;      // Esperamos 2 segundos
113 ;
114 ;      sleep(2);
115 ;
116 ;
117 ;      // Consultamos los eventos
118 ;
119 ;      while(SDL_PollEvent(&evento)) {
120 ;
121 ;          if(evento.type == SDL_QUIT) {
122 ;
123 ;              // Liberamos los recursos que no necesitamos
124 ;              SDL_FreeSurface(semaforo);
125 ;
126 ;              return 0;
127 ;          }
128 ;      }
129 ;  }
130 ;}
;
```

5.8.6. El Canal Alpha

El canal alfa es un canal que va de la mano de los canales de color RGB en el formato RGBA. Ya sabemos como usar colores transparentes cuando se realiza un blitting excluyendo un sólo color. SDL nos permite aplicar la transparencia en distintos grados a una superficie, es decir, definir el nivel de opacidad de un píxel. Esto es conocido como *Alpha-Blending* o simplemente *Alpha*. Este nivel de transparencia se define en un rango de 0 a 255 donde 0 denota que la superficie es totalmente transparente mientras que el valor 255 indica que es totalmente opaco.

El resultado de una operación de este tipo es que la imagen con la que realicemos el blit quedará transparente en un determinado grado, dejando entrever lo que hay detrás de ella, dependiendo del valor de transparencia que hayamos indicado.

La función que realiza esta tarea en SDL tiene el prototipo:

```
int SDL_SetAlpha(SDL_Surface *surface, Uint32 flag, Unit8 alpha);
```

Puedes observar que la función es muy parecida a *SDL_SetColorKey()* ya que no deja de tener un cometido parecido. En este caso el parámetro *flag* puede tomar los valores 0, para desactivar el alpha o *SDL_SRCALPHA* para

5. El Subsistema de Video

indicar que el tercer parámetro de la función es el alpha o transparencia que queremos aplicar a la superficie *surface*. Junto a `SDL_SRCALPHA` podemos activar `SDL_RLEACCEL` para la aceleración comentada en el subaparatado anterior. Los niveles de transparencia son 0 para totalmente opaco y 255 para totalmente transparente. Como las demás funciones devuelve 0 si se realizó la tarea con éxito y -1 en caso de error.

5.8.6.1. Ejemplo 6

Para entender mejor esta funcionalidad vamos a realizar un ejercicio. A partir de un rectángulo sólido vamos a crear un degradado sobre negro.

```
;  
1 ;// Listado: main.c  
2 ;// Programa de prueba,  
3 ;// Creación de un degradado  
4 ;  
5 ;#include <stdio.h>  
6 ;#include <SDL/SDL.h>  
7 ;  
8 ;int main()  
9 ;{  
10 ;  
11 ;    SDL_Surface *pantalla, *color_base;  
12 ;    SDL_Rect posicion;  
13 ;    SDL_Event evento;  
14 ;    int i;  
15 ;  
16 ;    // Iniciamos el subsistema de video SDL  
17 ;  
18 ;    if( SDL_Init(SDL_INIT_VIDEO) < 0) {  
19 ;        fprintf(stderr, "No se pudo iniciar SDL: %s\n", SDL_GetError());  
20 ;        exit(1);  
21 ;    }  
22 ;  
23 ;    // Establecemos el modo de video  
24 ;  
25 ;    pantalla = SDL_SetVideoMode(640, 500, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);  
26 ;  
27 ;    if(pantalla == NULL) {  
28 ;        fprintf(stderr, "No se pudo establecer el modo de video: %s\n", \  
29 ;                SDL_GetError());  
30 ;        exit(1);  
31 ;    }  
32 ;  
33 ;    atexit(SDL_Quit);  
34 ;  
35 ;    // Cargamos la imagen del personaje principal  
36 ;    color_base = SDL_LoadBMP("Imagenes/color_base.bmp");  
37 ;    if(color_base == NULL) {  
38 ;        fprintf(stderr, "No se pudo cargar la imagen: %s\n", SDL_GetError());
```

5.8. Transparencias

```
39 ;         exit(1);
40 ;
41 ;
42 ;     // Este bucle es que el consigue el efecto del degradado
43 ;     // Por cada iteración disminuye el grado de transparencia
44 ;     // consiguiendo un color más sólido
45 ;
46 ;     for(i = 0; i <= 50; i++) {
47 ;
48 ;         // Ajustamos las propiedades del canal alpha para las transparencias
49 ;         SDL_SetAlpha(color_base, SDL_SRCALPHA|SDL_RLEACCEL, i * 5.1);
50 ;
51 ;         // Establecemos donde vamos a dibujar las tiras
52 ;         posicion.x = 0;
53 ;         posicion.y = 10 * (i - 1);
54 ;         posicion.w = color_base->w;
55 ;         posicion.h = color_base->h;
56 ;
57 ;         // Copiamos la imagen en la superficie principal
58 ;         SDL_BlitSurface(color_base, NULL, pantalla, &posicion);
59 ;
60 ;
61 ;
62 ;         // Mostramos la pantalla "oculta" del buffer
63 ;
64 ;         SDL_Flip(pantalla);
65 ;
66 ;         // Liberamos los recursos que no necesitamos
67 ;
68 ;         SDL_FreeSurface(color_base);
69 ;
70 ;         // Ahora mantenemos el resultado en pantalla
71 ;         // hasta cerrar la ventana
72 ;
73 ;     for(;;) {
74 ;
75 ;         while(SDL_PollEvent(&evento)) {
76 ;
77 ;             if(evento.type == SDL_QUIT) // Si es de salida
78 ;                 return 0;
79 ;         }
80 ;     }
81 ;
82 ;
83 ;     return 0;
84 ;}
```

Como puedes ver el concepto de color clave es diferente al de transparencia. Podemos definir zonas transparentes mediante esta función con mucha libertad en SDL. La razón de no usar transparencias para ocultar detalles de nuestras imágenes que no queremos que se muestren es que, por norma general, la unidad blit de hardware no suelen implementar el blitting con factor alpha,

5. El Subsistema de Video

debido a que la ecuación que mezcla los colores provoca que haya que leer desde la superficie destino lo que ralentiza el proceso. Si decidimos utilizar esta técnica nos condenamos a utilizar el blit por software que es bastante más lento que el de hardware y notando un importante descenso del rendimiento.

El tratamiento de los canales alpha está más refinado en librerías dedicadas a las tres dimensiones, como OpenGL, por lo que si tienes necesidad de realizar transparencias con asiduidad lo más aconsejable es usar este tipo de librerías para desarrollar tu videojuego. En el caso de que utilices SDL con este propósito procura tener todas las imágenes almacenadas en la memoria principal del sistema, así ahorrarás tráfico en el bus, aprovechando al máximo las capacidades del mismo.

5.9. Clipping

El clipping es un añadido al blitting que consiste en establecer una zona dentro de la superficie que sea la que se redibuje o actualice. Es decir, definimos el área en la que se va a realizar blitting o volcado de superficies.

¿Para qué vale esto? Imaginemos que tenemos un juego de carreras de coches de vista interna, es decir desde dentro del coche. En la zona de la luna del vehículo se llevará acabo la acción del juego pero gran parte del interior del coche no variará en ningún momento. Esta técnica nos permite dibujar en cierta zona sin salirnos de ella y sin tener que redibujar zonas que no se van a modificar.

El clipper de una superficie solo afecta al blitting que tenga como destino a dicha superficie. El origen quedará inalterado. En el caso de no establecer ningún área de clipping el sistema tomará como área de clipping la compleitud de la superficie. Esto es debido a que SDL utiliza esta técnica para evitar que modifiquemos zonas de memoria que no correspondan a la superficie, ya que el rectángulo de clipping nunca puede extenderse más allá del límite de la superficie.

La técnica consiste en establecer la zona que se va a redibujar, así no tendremos que ser cuidadosos sobre el repintado en cada momento el salpicadero del coche, por ejemplo, con lo que nos evitamos escribir en píxeles que no necesitamos modificar.

Para establecer este área de clipping SDL proporciona la función cuyo prototipo es:

```
void SDL_SetClipRect(SDL_Surface *surface, SDL_Rect *rect);
```

Ya conocemos las estructuras *SDL_Surface* y *SDL_Rect*. En este caso el primera parámetro es la superficie dónde queremos efectuar el clipping y el segundo el rectángulo dentro del cual queremos que se efectúe el volcado gráfico. Si la función recibe como parámetro *rect* el valor NULL establecerá como área de clipping toda la superficie del primer parámetro. En el caso que el área de clipping sea superior al de la superficie quedará limitada al área de referencia al que queremos aplicar la técnica.

Si deseamos saber cuál es el área de clipping SDL proporciona esta función:

```
void SDL_GetClipRect(SDL_Surface *surface, SDL_Rect *rect);
```

El parámetro *rect* será modificado en esta función y en él se almacenarán los datos del área de clipping. El resto de la llamada es exactamente igual a la de la función anterior.

5.9.1. Ejemplo 7

Para comprobar como funciona el área de clipping vamos a modificar el último listado indicándole cual va a ser el área de la superficie pantalla que queremos que se actualice, en definitiva, en la que podemos dibujar. Vamos a establecer un área de clipping de 100 x 500 en la parte izquierda de la ventana en pantalla.

```
1 ;// Listado: main.c
2 ;// Programa de prueba,
3 ;// Creación de un degradado parcial, estableciendo un área de clipping
4 ;
5 ;#include <stdio.h>
6 ;#include <SDL/SDL.h>
7 ;
8 ;int main()
9 ;{
10 ;    SDL_Surface *pantalla, *color_base;
11 ;    SDL_Rect posicion, clipping;
12 ;    SDL_Event evento;
13 ;    int i;
14 ;
15 ;    // Iniciamos el subsistema de video SDL
16 ;    if( SDL_Init(SDL_INIT_VIDEO) < 0) {
17 ;        fprintf(stderr, "No se pudo iniciar SDL: %s\n", SDL_GetError());
18 ;        exit(1);
19 ;    }
20 ;
21 ;    // Establecemos el modo de video
22 ;    pantalla = SDL_SetVideoMode(640, 500, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
23 ;    if(pantalla == NULL) {
```

5. El Subsistema de Video

```
24 ;         fprintf(stderr, "No se pudo establecer el modo de video: %s\n", \
25 ;                         SDL_GetError());
26 ;         exit(1);
27 ;
28 ;
29 ;     atexit(SDL_Quit);
30 ;
31 ;     // Cargamos la imagen del personaje principal
32 ;
33 ;     color_base = SDL_LoadBMP("Imagenes/color_base.bmp");
34 ;
35 ;     if(color_base == NULL) {
36 ;         fprintf(stderr, "No se pudo cargar la imagen: %s\n", SDL_GetError());
37 ;         exit(1);
38 ;
39 ;
40 ;     // Establecemos el área de clipping
41 ;     clipping.x = 0;
42 ;     clipping.y = 0;
43 ;     clipping.h = 500;
44 ;     clipping.w = 100;
45 ;
46 ;     SDL_SetClipRect(pantalla, &clipping);
47 ;
48 ;     // Vamos a dibujar 10 tiras
49 ;     for(i = 1; i <= 50; i++) {
50 ;         // Ajustamos las propiedades del canal alpha para las transparencias
51 ;         SDL_SetAlpha(color_base, SDL_SRCALPHA|SDL_RLEACCEL, i * 5.1);
52 ;
53 ;         // Establecemos donde vamos a dibujar las tiras
54 ;         posicion.x = 0;
55 ;         posicion.y = 10 * (i - 1);
56 ;         posicion.w = color_base->w;
57 ;         posicion.h = color_base->h;
58 ;
59 ;         // Copiamos la imagen en la superficie principal
60 ;         SDL_BlitSurface(color_base, NULL, pantalla, &posicion);
61 ;
62 ;
63 ;         // Mostramos la pantalla "oculta" del búffer
64 ;         SDL_Flip(pantalla);
65 ;
66 ;         // Liberamos los recursos que no necesitamos
67 ;         SDL_FreeSurface(color_base);
68 ;
69 ;         // Ahora mantenemos el resultado en pantalla
70 ;         // hasta cerrar la ventana
71 ;
72 ;         for(;;) {
73 ;
74 ;             while(SDL_PollEvent(&evento)) {
75 ;
76 ;                 if(evento.type == SDL_QUIT) // Si es de salida
```

5.10. Conversión de Formatos de Píxel

```
77 ;           return 0;
78 ;       }
79 ;   }
80 ;
81 ;}
```

Como puedes ver esta es una herramienta que te ofrece SDL, que es muy simple de utilizar, y puede ahorrar grandes cantidades de tiempo de proceso al sistema.

5.10. Conversión de Formatos de Píxel

En una misma aplicación podemos cargar distintos formatos de imágenes de píxel. Cuando avancemos un poco más en el tutorial comprobarás que existen extensiones para SDL que nos permiten trabajar con un número bastante importante de tipos de imágenes. Seguramente si se da el caso de que trabajes con distintos tipos de formatos será porque no se ha unificado la tarea de diseño gráfico de dicha aplicación o simplemente que hemos usado esta o aquella imagen que nos gustaba sin prestar más atención.

Esto conlleva que en ocasiones podamos tener superficies en nuestra aplicación con diferente formato de píxel. Si tenemos que copiar una superficie en otra habrá que realizar por cada uno de los píxeles una unificación del formato produciéndose conversión al formato de la superficie destino. Aunque esta conversión no es particularmente compleja si que consume tiempo de computación. Esto provoca que, si este tipo de conversión tiene que realizarse con frecuencia en nuestra aplicación, provoca una importante ralentización de la misma.

Para minimizar este problema debemos de tener todas las superficies en el mismo formato, concretamente en el formato de la superficie principal que será la que se muestre por pantalla. Para facilitar esta tarea SDL ofrece la posibilidad de convertir una superficie en el formato de píxel que otra. La función es la siguiente:

```
SDL_Surface *SDL_ConvertSurface(SDL_Surface *src, SDL_PixelFormat
*fmt, Uint32 flags);
```

Esta función permite acelerar las operaciones de blitting ya que, si dos superficies están en distinto formato de píxel, SDL tiene que convertirlas durante la operación de blitting, relentizándola. Si esta conversión ya está realizada es trabajo que ahorramos al sistema durante esta operación.

Como la conversión al formato de la superficie que se va a mostrar es el más común SDL ofrece otra función que nos permite realizar la conversión entre superficies es la función:

5. El Subsistema de Video

```
SDL_Surface *SDL_DisplayFormat(SDL_Surface *surface);
```

Esta función realiza una conversión al formato de pantalla. Recibe como parámetro superficie que queremos convertir y nos devuelve otra superficie convertida con el formato que tiene el framebuffer de vídeo. Si la conversión falla o no existe memoria suficiente la función devolvería el valor `NULL`.

La habitual y lo correcto es tener todas las superficies de nuestra aplicación en el mismo formato de píxel. ¿Qué superficie tomar como referencia para elegir el formato de píxel? Lo más lógico es elegir el formato de la superficie principal de la aplicación, es decir, de la superficie que devolvía `SetVideoMode()` que al fin y al cabo va a ser superficie de destino para todas las demás.

Es importante que realices esta conversión. Es un tema parecido al de la gestión de memoria. Puedes pasar sin gestionar correctamente la memoria, sin liberar lo que ya no te hace falta y seguramente tu aplicación puede hasta que funcione pero no habrás creado una aplicación de calidad que vaya utilizando óptimamente los recursos. Con las transformaciones entre formatos de píxel pasa algo parecido. Puedes pasar sin realizar estas conversiones explícitamente y dejar que se haga implícitamente cada vez que vuelques en contenido de una superficie en otra. Los principales problemas de descuidar este aspecto son dos.

El primero es que mediante la función que hemos presentado sólo necesitarás hacerlo una vez por ejecución, mientras que la conversión implícita se hace una vez por cada blit. Imagínate que estamos animando un personaje sustituyendo unas imágenes por otra y mostrándolas por pantalla, el consumo de tiempo de procesador será enorme.

El segundo es que, normalmente, esta conversión explícita se realiza justo después de la carga de la imagen de un fichero por lo que se hace una vez y además en el proceso de inicialización de la aplicación. En este proceso de inicialización se suelen realizar unas tareas determinadas que se van a realizar única y solamente ahí bien porque sea necesario o bien porque el realizar esta tarea después se puede encontrar con un momento crítico de la aplicación y que ésta no responda como debería.

Por ejemplo imagínate que estamos implementando un juego de acción y no realizamos esta conversión entre formatos de píxel. Ejecutamos nuestra aplicación en nuestro maravilloso ordenador nuevo con la última configuración disponible. Todo va estupéndamente hasta que llegamos al final del nivel y aparecen diez rivales a los que batir. Sólo diez rivales y el proceso de su animación, con la correspondiente conversión de formato de píxel, hace que el sistema no sea capaz de mostrarnos con fluidez el movimiento de estos personajes. Resumiendo, una tarea simple, como es convertir los formatos de píxel, puede determinar que nuestra aplicación sea un éxito o un fracaso.

Aplicaremos esta técnica con asiduidad a la hora de realizar nuestro juego final por lo que postponemos la práctica a dicho momento.

5.11. Recopilando

Este es uno de los capítulos más importantes del tutorial. Hemos aprendido a inicializar el subsistema de video así como todas las características y conceptos asociados a él.

Para mostrar imágenes en pantalla lo haremos a través de superficies. Existirá una superficie principal donde tendremos que volcar todo aquello que queramos mostrar por pantalla. El volcado es conocido como *blitting* y se le pueden aplicar ciertas técnicas que nos permiten obtener un mejor rendimiento y efecto visual. Una de las más importantes es el uso de un doble búffer que nos permitirá realizar acciones sobre la superficie principal sin que el usuario pueda ver el efecto de volcar información en ella.

En las superficies podemos tanto trabajar a nivel de píxel como de mapa de bits, siendo más cómoda esta última alternativa ya que conlleva una menor carga de trabajo.

Asociado a las superficies y al subsistema de video existen unos conceptos soportados mediante estructuras de datos por SDL como son el color, los rectángulos, la paleta de color, el formato de píxel...

Para terminar el capítulo hemos presentado las herramientas necesarias que nos permitirán manejar las superficies con total libertad para que el límite los pongamos nosotros y no la librería SDL.

5. El Subsistema de Video

Capítulo 6

Captura y Gestión de Eventos

6.1. Introducción

Aunque el capítulo sobre el subsistema de video es considerado por muchos programadores como el más importante para el desarrollo de videojuegos una aplicación de este tipo no sería viable si no existiese forma de interactuar con ella.

En un juego para ordenador esta interacción toma el significado de entrada de usuario y es realizada sobre alguno de los dispositivos de los que puede disponer dicho usuario como un teclado, un ratón o un joystick. En SDL tenemos dos formas de trabajar con los dispositivos, la primera manejando los eventos que producen y la segunda conociendo el estado del dispositivo en un momento dado.

A la hora de desarrollar un videojuego el manejo de eventos y de los dispositivos de entrada es tan fundamental como manejar el subsistema de video con cierta habilidad. La respuesta que consigamos dar a dichos eventos ayudarán al éxito de nuestra aplicación. Un videojuego sin una buena respuesta a los eventos es como la historia de una muerte anunciada.

Una de las grandes virtudes de SDL es la comodidad y facilidad que ofrece a la hora de gestionar eventos. El sistema de gestión de eventos es un subsistema dentro de SDL y como tal debe ser inicializado. En este caso el subsistema se inicia automáticamente junto al subsistema de video, por lo que si vamos a realizar una aplicación gráfica, no necesitamos activarlo expresamente.

El subsistema de eventos se inicializa automáticamente junto al subsistema de video por lo que no tendremos que preocuparnos de dicha inicialización.

6. Captura y Gestión de Eventos

6.2. Objetivos

Los objetivos de este capítulo son:

1. Comprender y manejar los eventos de distintos dispositivos.
2. Aprender a utilizar varios métodos para la gestión de los dispositivos de entrada.
3. Crear nuevos eventos y filtros de eventos en SDL.

6.3. Conocimientos previos

Para afrontar este capítulo no son necesarios conocimientos más que aquellos sobre programación que ya hemos utilizado en el tutorial. Concéntrate en dominar el manejo de la entrada ya que será fundamental a la hora de desarrollar tu aplicación.

6.4. Eventos

El evento no es más que una acción que sucede mientras se ejecuta un programa, es decir un suceso que ocurre en el sistema durante la ejecución de un programa. La definición de evento es muy amplia debido a que existen numerosos tipos de eventos. Un evento es lo que ocurre cuando se acciona un dispositivo de entrada. Un simple desplazamiento de ratón produce un evento, así como pulsar el teclado o el reloj del sistema también producen eventos. Hablando de forma más general es una forma de permitir que la aplicación que hemos desarrollado reciba la entrada del usuario (o del sistema operativo) para que pueda así gestionarla.

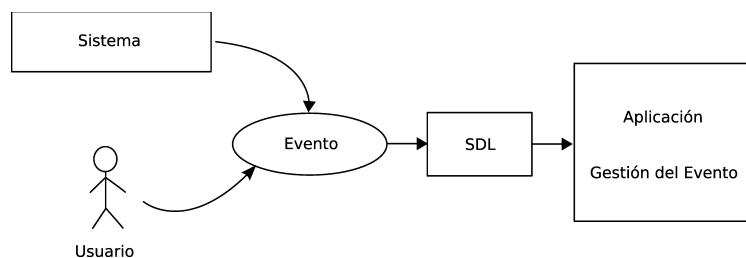


Figura 6.1: Evento

Dependiendo del tipo de evento que se accione podemos conocer, no sólo el evento que se produce, si no además mucha información adicional. Por ejemplo cuando se pulsa una tecla se produce un evento de tecla presionada. Además la variable que nos indica que se ha producido el evento nos permite

conocer exactamente que tecla ha sido la presionada. Como puedes observar la información adicional que podamos conseguir estará intimamente ligada al tipo de evento que se ejecute y será fundamental a la hora de gestionar estos eventos. Enumeraremos tipos de eventos en el siguiente apartado.

La idea fundamental es que todo evento tiene asociada una acción. Acción a la que podremos dar respuesta con la información que nos proporciona sobre los eventos SDL.

SDL trata a los eventos a través de una unión *SDL_Event* definida de la siguiente forma:

```

1 ;_____
2 ;typedef union {
3 ;    Uint8 type;
4 ;    SDL_ActiveEvent active;
5 ;    SDL_KeyboardEvent key;
6 ;    SDL_MouseMotionEvent motion;
7 ;    SDL_MouseButtonEvent button;
8 ;    SDL_JoyAxisEvent jaxis;
9 ;    SDL_JoyBallEvent jball;
10 ;   SDL_JoyHatEvent jhat;
11 ;   SDL_JoyButtonEvent jbutton;
12 ;   SDL_ResizeEvent resize;
13 ;   SDL_QuitEvent quit;
14 ;   SDL_UserEvent user;
15 ;   SDL_SywWMEvent syswm;
15 ;} SDL_Event;
;
```

Vamos a ver que significa cada uno de estos campos:

- *type*: Indica el tipo de evento que se ha producido. Existen varios tipos de eventos, desde los producidos por una entrada directa del usuario por teclado hasta los eventos del sistema. A lo largo del tema iremos estudiando cada uno de los tipos de eventos gestionables por SDL.
- *active*: Evento de activación. Este evento informa acerca de la situación del foco sobre nuestra ventana. Es considerado un evento de sistema y será estudiado en dicho apartado.
- *key*: Es un evento producido por una acción en el teclado.
- *motion*: Evento producido por el movimiento del ratón.
- *button*: Evento producido por una acción sobre el botón del ratón.
- *jaxis*: Evento de movimiento del eje del joystick.
- *jball*: Evento de movimiento del trackball del joystick.

6. Captura y Gestión de Eventos

- *jhat*: Evento producido por el movimiento del minijoystick o hat del dispositivo de juego.
- *jbutton*: Evento activado por una acción en algún botón del joystick.
- *resize*: Evento provocado por el redimensionado de nuestra ventana.
- *quit*: Evento producido al cerrar la aplicación.
- *user*: Evento definido por el usuario.
- *syswm*: Evento indefinido producido por el gestor de ventanas.

Esta unión abarca todas las estructuras de eventos de SDL. Será fundamental conocer el significado de cada uno de estos campos para el uso de estos eventos. Es habitual que no se usen todas las estructuras que abarcan los eventos limitándonos a las más comunes. Normalmente gestionaremos el uso del teclado y del ratón, y en nuestro caso, es fundamental conocer la manera de gestionar el joystick.

6.5. Captura de eventos

La captura de eventos es un aspecto fundamental en la programación de videojuegos. Para sacar más provecho a este apartado es conveniente un estudio previo de las estructuras que sostienen el sistema de eventos en SDL. Para entender muchos de sus campos tienes que conocer las técnicas que nos permiten gestionar estos eventos. En SDL pasa como en muchos aspectos de la informática. El conocimiento es circular y hay que empezar por algún lado. En este caso empezamos exponiendo las técnicas para capturar los eventos y así poder comprender mejor la implementación de SDL. En concreto tres que pasamos a detallar a continuación.

6.5.1. Waiting

La primera técnica que vamos a estudiar consiste en esperar a que ocurra cierto evento, ya sea por haber realizado una acción, ya sea por un suceso en el sistema operativo. Un ejemplo puede ser el de esperar que el usuario pulse un botón de una determinada ventana. Esta técnica es muy utilizada en aplicaciones de gestión donde no hay ninguna tarea en segundo plano a parte de la de esperar el evento para darle respuesta, estando el programa latente, hasta que no ocurre un evento. Implementar esta técnica nos conduce a realizar una aplicación guiada por eventos lo que no nos sirve para la mayoría de los casos en el mundo de los videojuegos.

Esta técnica no es frecuente en el mundo de los videojuegos ya que normalmente estarán ocurriendo cosas en el videojuego aunque no se produzcan

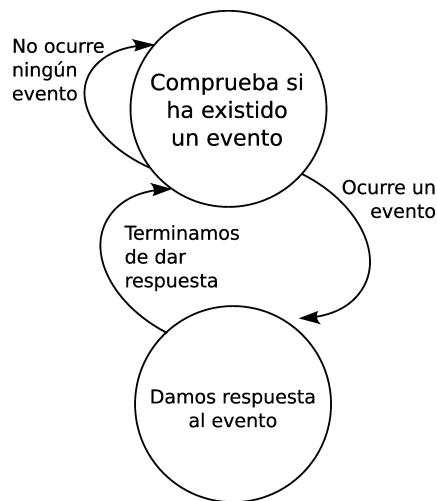


Figura 6.2: Esperando eventos (Waiting)

eventos. Si que es habitual encontrar esta técnica en aplicaciones auxiliares que nos facilitan la creación de este tipo de programas como los editores de niveles. En la figura 6.2 puedes ver un diagrama que representa la esencia de esta técnica. SDL proporciona una función para utilizar este tipo de técnica, cuyo prototipo es:

```
int SDL_WaitEvent(SDL_Event *event);
```

Para usar esta función debemos de sustituir el puntero que recibe como parámetro por una estructura del tipo *SDL_Event*. Cuando ocurra un evento nuestra estructura será rellenada con los datos correspondientes. La función devolverá 1 si la información ha sido copiada correctamente y se ha realizado el borrado del evento de la cola de eventos. Si hubo algún problema la función devolverá 0.

Si por alguna razón pasamos a la función como parámetro el valor NULL SDL esperará un evento y devolverá que se ha producido pero sin variar la cola de eventos. Esto nos puede ser útil para aplicaciones como los salvapantallas con el que volvemos a poner nuestra máquina en activo sea cual sea el tipo de evento de usuario.

Un ejemplo de estructura típica para utilizar esta técnica, en C o C++, es la siguiente:

```

1 ;
2 ;      // Declaramos una variable evento
3 ;      SDL_Event evento;
4 ;
5 ;      // Bucle infinito
  
```

6. Captura y Gestión de Eventos

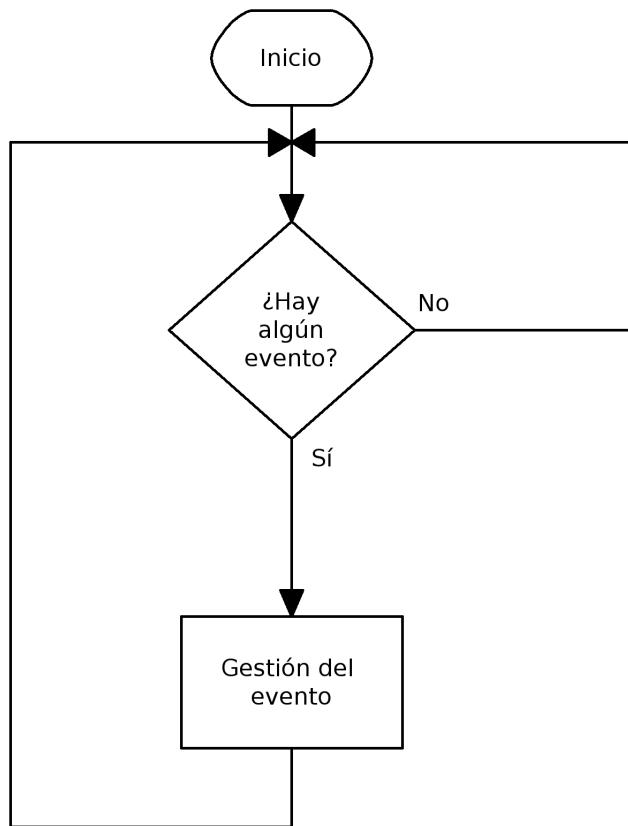


Figura 6.3: Diagrama de flujo. Waiting

```
6 ;     for( ; ; ) {  
7 ;  
8 ;         // Esperamos a que ocurra un evento  
9 ;         SDL_WaitEvent(&evento);  
10 ;  
11 ;         // Gestionamos el evento  
12 ;         if(evento.type == SDL_QUIT) {  
13 ;             // Terminaremos el bucle infinito  
14 ;             // para salir de la aplicación  
15 ;         }  
16 ;  
17 ;         if(evento.type == SDL_KEYDOWN) {  
18 ;             // Gestionamos los eventos asociados  
19 ;             // a presionar un elemento del teclado  
20 ;         }  
21 ;  
22 ;         // Otros manejadores de eventos  
23 ;     }  
24 ;  
25 ;  
26 ;  
27 ;  
28 ;         // Otros manejadores de eventos  
29 ;     }
```

;

El uso de esta estructura es libre, sólo es una guía de como implementar esta técnica en el lenguaje de programación concreto que estamos utilizando.

Con este método la mayoría de tiempo de programa pasa esperando a que ocurra algo. Una vez ocurra algo el programa reacciona según el evento que se produzca. Una vez le haya dado respuesta la aplicación volverá a modo en espera hasta el infinito como puedes ver en la figura 6.3. Con este método, si no se producen muchos eventos por instante, se desaprovecha mucho tiempo de computación.

6.5.2. Acceso Directo al Estado de Eventos del Dispositivo

La segunda técnica que vamos a presentar es la del acceso directo al estado de eventos del dispositivo. Podemos acceder simbólicamente al hardware y leer el estado de un dispositivo discriminando la información con la idea de seleccionar sólo la que es relevante para nosotros. En realidad accedemos al sistema de eventos que es una capa superior a la de hardware que nos presenta la información para que podamos manejarla de forma más cómoda.

Cuando nos refiramos en el tutorial que vamos a acceder directamente a la información del dispositivo haremos referencia en realidad a que tomaremos la información del estado de los eventos de dicho dispositivo.

Esta forma de actuar suele ser más compleja proporcionando unos resultados similares que las otras alternativas, aunque es habitual encontrarla en el desarrollo de videojuegos cuando se quiere un determinado tipo y tiempo de respuesta.

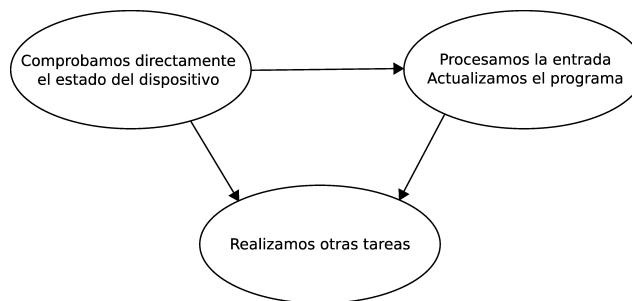


Figura 6.4: Acceso directo al estado del dispositivo.

Con este método podemos perder eventos. Nosotros consultamos el estado del dispositivo en un instante, un momento determinado, con un estado determinado. Si justo en el momento anterior estaba ocurriendo otra cosa, es decir el dispositivo estaba en otro estado, y no hemos realizado la consulta

6. Captura y Gestión de Eventos

perderemos dicha información. Depende de la velocidad del videojuego que estemos diseñando y de los recursos que dipongamos puede ser interesante no procesar toda la información de entrada. Por ejemplo si no nos interesa procesar una determinada acción con este método ni siquiera capturaremos la entrada del teclado por lo que no tendremos ni que descartar eventos. Como ves SDL dispone y nosotros decidimos.

Al ser el más complejo de programar, será el método que utilicemos en nuestro videojuego final para que tengas un ejemplo de como implementar la gestión de entrada con esta técnica. Junto a ésta utilizaremos la técnica de polling para completar la respuesta de la aplicación.

Para esta técnica necesitaremos también una función de polling o sondeo, pero en vez de realizarlo a la pila de eventos tendremos que consultar el dispositivo en cuestión directamente. En cada apartado de la sección de “Tipos de Eventos” explicaremos como consultar dichos eventos mediante sondeo y esta técnica de acceso directo al dispositivo.

Cada vez que queramos consultar el estado de un dispositivo tendremos que actualizar la información que disponemos de ella mediante la función:

```
void SDL_PumpEvents(void);
```

En otras técnicas, en SDL, la llamada a esta función se hace implícitamente pero en este caso al consultar directamente el estado de los dispositivos deberemos de realizar la llamada manualmente.

6.5.3. Polling

La última técnica que vamos a presentar es el uso de polling o sondeo. Esta es una técnica muy común en el desarrollo de aplicaciones de cualquier tipo. El sondeo consiste en ir almacenando eventos en una cola que será consultada en el momento que estimemos oportuno. En ese momento es cuando realizamos dicho sondeo, o lo que es lo mismo, es el momento en el que consultamos los eventos.

En SDL proporciona dicha cola de eventos a la que le realizaremos una consulta o polling periodicamente por lo que no tendremos que realizar tareas de mantenimiento de dicha estructura. En la figura 6.5 tienes un diagrama que explica en que consiste esta técnica.

Cuando se estime oportuno se da respuesta a los eventos en un momento determinado. El programa consulta si existen más eventos a procesar. Si no existiesen la aplicación irá realizando tareas como la de mover enemigos, realizar animaciones, en definitiva, procesos en segundo plano... hasta que se

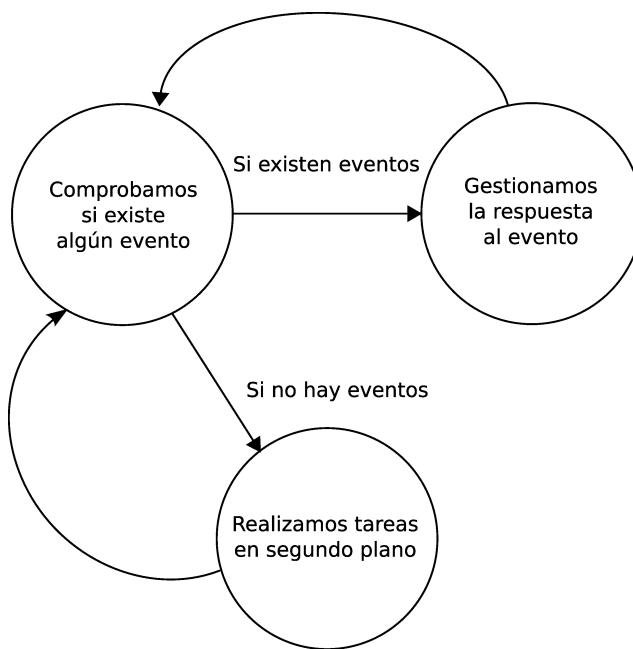


Figura 6.5: Polling o Sondeo.

produjese un evento al que queramos dar respuesta. En resumen, mientras que no existan eventos que queramos procesar la aplicación irá realizando tareas en segundo plano hasta que aparezca algún evento al que queramos dar respuesta. Una vez dada la respuesta volveremos al punto inicial realizando tareas hasta que exista un evento que sea relevante para nosotros.

Este método es muy adecuado para la programación de videojuegos debido a la estructura del *game loop* que presentábamos en capítulos anteriores. Leemos la entrada del medio que se configure para ello, se procesa y vuelta a empezar. Mediante esta técnica conseguimos no perder datos de los eventos de entrada teniendo que ser nosotros los que descartemos los eventos que no necesitamos ignorándolos, es decir, sin darles respuesta.

SDL almacena todos los eventos sin procesar en una cola de eventos. Para poder utilizar esta técnica SDL nos proporciona una función que consulta dicha cola y saca de la misma al evento que lleve más tiempo en ella. Dicha función tiene el prototipo:

```
int SDL_PollEvent(SDL_Event *event);
```

Esta función “busca” nuevos elementos pendientes en la cola. Devuelve 0 si no hay eventos pendientes. En este caso la semántica del 0 no es la de expresar un error si no que indica que no existen eventos pendientes de procesar. Si los hubiese, el valor que devuelve es 1. El parámetro *event* es un paso por

6. Captura y Gestión de Eventos

referencia que almacena el evento disponible que vamos a capturar. Todos los eventos tienen un campo en común que nos permite consultar el tipo de evento del que se trata definido como un entero sin signo de 8 bits llamado *type*.

El evento obtenido de la cola mediante el parámetro *event* es eliminado de la misma excepto si el valor que recibe la función es **NULL** lo que provoca que el evento no se elimine como pasaba en la técnica de *waiting*.

La estructura habitual en un bucle de lectura de eventos utilizando *polling* es la siguiente:

1. Comprobamos si existe algún evento pendiente.
2. Si existe lo procesamos y volvemos al paso 1.
3. Si no existe hacemos algo durante un pequeño periodo de tiempo y volvemos al paso 1.

Si queremos evitar la inanición por un exceso de eventos en la cola podemos modificar y seguir el siguiente esquema:

1. Comprobamos si existe algún evento pendiente.
2. Si existe lo procesamos.
3. Hacemos algo durante un pequeño periodo de tiempo.
4. Volvemos al paso 1.

6.5.4. ¿Cuál es la mejor opción?

Esta es una pregunta a la que tendrás que responder tú mismo. No hay una mejor técnica. Depende del problema que tengamos que resolver utilizaremos un método u otro. Por ejemplo, si estamos esperando una pulsación de una tecla la mejor solución es esperar un evento hasta que se produzca utilizando la técnica de *waiting*. Sin embargo si hemos creado una aplicación educativa donde, por ejemplo, a cada click de ratón sobre un animal se produce un sonido seguramente la mejor opción será utilizar la técnica de *polling* o sondeo.

Si nos encontramos el caso de que tenemos que ofrecer una reacción rápida a una interacción con el teclado, y no nos importa perder algún evento ya que la respuesta tiene que ser crítica, seguramente la mejor opción sea consultar el estado de los eventos mediante un acceso directo. Esto lo podremos aplicar al manejo de un personaje en un videojuego. En el videojuego final verás que hemos combinado varias alternativas para obtener así la mejor solución.

Para SDL todas las opciones son prácticamente la misma pero se ha mostrado un interfaz diferente. En realidad aplicar las dos primeras técnicas

Tipo de evento	Estructura que maneja el evento
SDL_ACTIVEEVENT	SDL_ActiveEvent
SDL_KEYDOWN	SDL_KeyboardEvent
SDL_KEYUP	SDL_KeyboardEvent
SDL_MOUSEMOTION	SDL_MouseMotionEvent
SDL_JOYAXISMOTION	SDL_JoyAxisEvent
SDL_JOYBALLMOTION	SDL_JoyBallEvent
SDL_JOYHATMOTION	SDL_JoyHatEvent
SDL_JOYBUTTONDOWN	SDL_JoyButtonEvent
SDL_JOYBUTTONUP	SDL_JoyButtonEvent
SDL_QUIT	SDL_QuitEvent
SDL_SYSWMEVENT	SDL_SysWMEvent
SDL_VIDEORESIZE	SDL_ResizeEvent
SDL_USEREVENT	SDL_UserEvent

Cuadro 6.1: Eventos soportados por SDL y las estructuras que los manejan.

no es más que pedirle a SDL que mire el estado de los eventos del dispositivo por nosotros y nos prepare una cola con la secuencialidad en que se han producido los eventos.

Veremos varios ejemplos de como utilizar cada una de estas técnicas.

6.6. SDL_Event

La estructura fundamental para el manejo de eventos en SDL es *SDL_Event*. Esta estructura se utiliza para:

- Leer eventos de la cola de eventos.
- Insertar eventos en dicha cola.

Los eventos soportados por SDL y las estructuras que nos permiten menajarlos puedes consultarlos en el cuadro 6.1.

En toda aplicación que desarrollemos vamos a necesitar manejar los eventos pendientes. Lo más habitual es usar la técnica de *polling*, dentro de un bucle que va recogiendo y procesando los eventos según se vayan produciendo y la necesidad de procesarlos o no, además de permitirnos realizar otras tareas menos prioritarias, aunque como ya hemos visto la técnica a utilizar dependerá del problema que queramos resolver.

6. Captura y Gestión de Eventos

6.6.1. Ejemplo 1

En el siguiente ejemplo vamos a estudiar minuciosamente que hemos implementado en los ejemplos de los temas anteriores para capturar el evento de salida y vamos a permitir que se cierre la ventana de la aplicación pulsando cualquier tecla. Aquí tienes el listado de ejemplo:

```
;_____
1 ;// Ejemplo 1
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de prueba,
5 ;// Gestiona eventos de para salir de la aplicación
6 ;
7 ;#include <iostream>
8 ;#include <SDL/SDL.h>
9 ;
10 ;using namespace std;
11 ;
12 ;// Este programa carga una imagen y la muestra por pantalla
13 ;// Gestiona mediante eventos la salida del programa principal
14 ;
15 ;int main()
16 ;{
17 ;
18 ;    // Declaramos los punteros para la pantalla y la imagen a cargar
19 ;
20 ;    SDL_Surface *pantalla, *imagen;
21 ;
22 ;
23 ;    // Variable para la gestión del evento de salida
24 ;
25 ;    SDL_Event evento;
26 ;
27 ;
28 ;    // Variable auxiliar donde almacenaremos la posición donde colocaremos
29 ;    // la imagen cargada dentro de la superficie principal
30 ;
31 ;    SDL_Rect destino;
32 ;
33 ;    // Iniciamos el subsistema de video de SDL
34 ;
35 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
36 ;
37 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
38 ;        exit(1);
39 ;    }
40 ;
41 ;    atexit(SDL_Quit);
42 ;
43 ;    // Comprobamos que sea compatible el modo de video
44 ;
45 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
```

6.6. SDL _ Event

```
46 ;
47 ;         cerr << "Modo no soportado: " << SDL_GetError();
48 ;         exit(1);
49 ;
50 ;     }
51 ;
52 ;     // Establecemos el modo de video y asignamos la superficie
53 ;     // principal a la variable pantalla
54 ;
55 ;     pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
56 ;
57 ;     if(pantalla == NULL) {
58 ;         cerr << "No se pudo establecer el modo de video: \n"
59 ;             << SDL_GetError() << endl;
60 ;
61 ;         exit(1);
62 ;     }
63 ;
64 ;
65 ;     // Cargamos ajuste.bmp en la superficie imagen
66 ;
67 ;     imagen = SDL_LoadBMP("Imagenes/ajuste.bmp");
68 ;
69 ;     if(imagen == NULL) {
70 ;
71 ;         cerr << "No se puede cargar la imagen: " << endl;
72 ;         exit(1);
73 ;     }
74 ;
75 ;
76 ;     // Inicializamos la variable de posición y tamaño de destino
77 ;
78 ;     destino.x = 150; // Posición horizontal con respecto
79 ;                         // a la esquina superior derecha
80 ;
81 ;     destino.y = 150; // Posición vertical con respecto a la
82 ;                         // esquina superior derecha
83 ;
84 ;     destino.w = imagen->w; // Longitud del ancho de la imagen
85 ;     destino.h = imagen->h; // Longitud del alto de la imagen
86 ;
87 ;     // Copiamos la superficie en la pantalla principal
88 ;
89 ;     SDL_BlitSurface(imagen, NULL, pantalla, &destino);
90 ;
91 ;     // Mostramos la pantalla
92 ;
93 ;     SDL_Flip(pantalla);
94 ;
95 ;     // Ya podemos liberar el rectángulo, una vez copiado y mostrado
96 ;
97 ;     SDL_FreeSurface(imagen);
98 ;
```

6. Captura y Gestión de Eventos

```
99 ; // Gestionamos el evento pulsar una tecla
100; // Esperamos que se pulsa una tecla para salir
101;
102; while(true) { // o bien for( ; ; )
103;
104;     while(SDL_PollEvent(&evento)) {
105;
106;         // ¿Se ha pulsado una tecla o es un evento de salida?
107;
108;         if(evento.type == SDL_KEYDOWN || evento.type == SDL_QUIT)
109;             return 0;
110;
111;     }
112;
113;
114; }
```

Ya te deben ser familiares la mayoría de las estructuras y variables que utilizamos en el ejemplo. Vamos a ir introduciendo un poco más de código en C++, pero no te preocupes, algo muy básico. La primera novedad que nos encontramos con respecto a los ejemplos y ejercicios del capítulo anterior es la directiva `#include <iostream>`. Esta es equivalente a la `stdio.h` de C aunque mucho más potente. En ella se incluye la entrada-salida estándar de C++.

Seguidamente entramos en la función principal del ejemplo. Definimos dos superficies, una la principal y otra para cargar una imagen en ella. A continuación creamos una variable de tipo `SDL_Event` donde almacenaremos el evento a gestionar para saber con qué tipo de evento estamos tratando.

El resto del código es idéntico a los explicados en el capítulo donde tratábamos el subsistema de video de SDL. Vamos a centrarnos en el bucle donde damos respuesta al evento. El primer bucle que nos encontramos es un bucle infinito. Lo implementamos mediante un `while` con una condición que se cumpla siempre, del estilo de una tautología, o bien mediante un bucle `for` en el que no utilizamos ninguna variable de control.

Con esto ya tenemos la primera parte de la implementación del *game loop*. Un bucle que itere siempre hasta que se quiera salir de la aplicación que nos permita realizar distintas acciones dentro del mismo. En nuestro caso vamos a consultar en cada vuelta del bucle si se ha producido algún evento. Para esto utilizaremos otro bucle `while`. ¿Por qué un bucle y no un estructura selectiva?

Cuando nuestra aplicación sea más compleja este será el único punto para procesar los eventos por cada vuelta del bucle. Hasta que llegue a este punto puede haber pasado un tiempo suficiente para haberse producido varios eventos. Con este bucle podremos tratarlos todos, y una vez dentro, descartar

aquellos que no nos interese procesar.

En este ejemplo concreto consultaremos la cola de eventos y sólo procesaremos el evento de pulsación del teclado (`SDL_KEYDOWN`) o si se ha producido un evento de salida (`SDL_QUIT`). Si es así saldremos terminaremos con la aplicación.

Como podemos ver lo único novedoso, porque no se había tratado todavía, en el listado es el control por medio de eventos y la disposición de la estructura de control de la aplicación. La variable *evento*, del tipo `SDL_Event`, es fundamental ya que es la que nos permite obtener los datos del evento, tanto el tipo como la información adicional que sea necesaria.

Como podrás ver gestionar este evento no es un proceso traumático aunque no hayas trabajado manejando eventos. Seguiremos viendo otros ejemplos durante el curso.

6.7. Tipos de Eventos

Para explicar lo concerniente a un evento hay que entrar en detalle sobre el tipo de evento que queremos tratar. La información que ofrece un evento de teclado no es parecida a la que ofrece, por ejemplo, un evento de joystick o de ratón, dependerá de la naturaleza del dispositivo. Por ello vamos a ahondar lo necesario en cada tipo de evento y en como podemos manejarlo con ayuda de la SDL.

En cada tipo de eventos que sea factible explicaremos como realizar la captura de eventos mediante *polling* y mediante la técnica de acceso directo al dispositivo brindando así la posibilidad de tener las herramientas necesarias para elegir la mejor opción a la hora de desarrollar nuestra aplicación.

Vamos a presentar los diferentes tipos de eventos asociados a cada tipo de dispositivo.

6.8. Teclado

6.8.1. Introducción

El teclado desde los inicios de la informática moderna es el dispositivo de entrada por excelencia. Actualmente existe una gran variedad de teclados en

6. Captura y Gestión de Eventos

el mercado pero todos tienen unas características comunes.

La disposición de las teclas que utilizamos en el llamado mundo occidental se remonta a las primeras máquinas de escribir. Existe varias versiones sobre el origen de la disposición. Es una historia muy curiosa que te animo que investigues. A nosotros nos es suficiente con conocer que vamos a trabajar con una disposición de tipo QWERTY. En el teclado español se le agrega la tecla ñ y cambia la posición de algunos caracteres especiales para que sean más fácilmente utilizables.

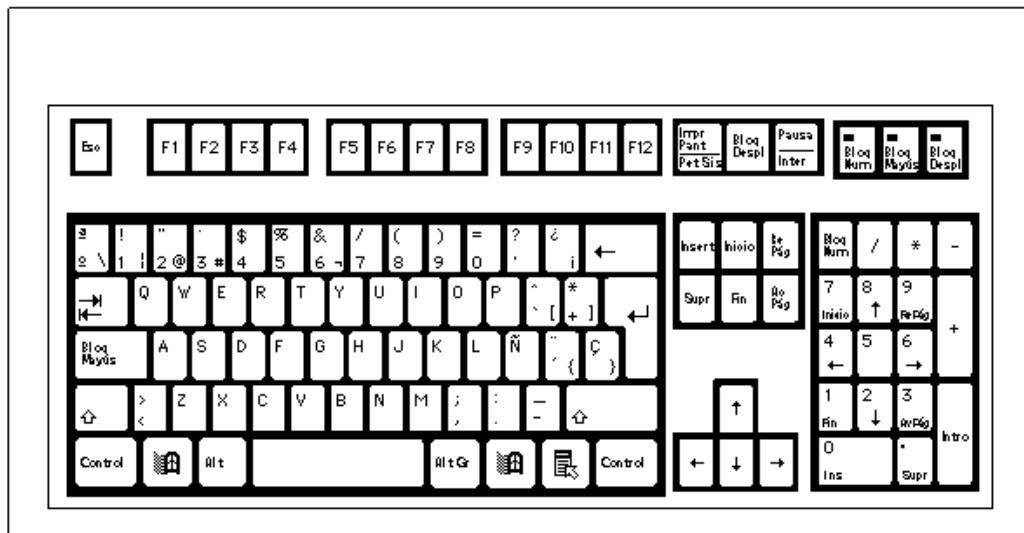


Figura 6.6: Distribución habitual de un teclado español.

Este dispositivo lo podemos controlar mediante manejo de eventos o conociendo su estado. Dependiendo del tipo de implementación que queramos realizar optaremos por una u otra. Vamos a comenzar presentando la manera de manejar el teclado mediante eventos.

6.8.2. Manejo del teclado mediante eventos

Cada vez que pulsamos una tecla SDL crea un evento. Este evento nos informa de la pulsación de una tecla y contiene información acerca de qué tecla fue pulsada. En el momento de soltar la tecla SDL también crea un evento con lo que tenemos dos eventos por cada pulsación de tecla. Depende de la respuesta que le queramos dar a la acción de usuario utilizaremos un evento u otro.

A parte de ofrecernos información sobre que se ha pulsado o soltado una tecla los eventos de teclado contienen más información relevante como la de conocer que tecla ha sido presionada o liberada.

Además SDL nos informa de si hay alguna tecla especial como **Ctrl**, **Alt** o **Shift** pulsadas. Esto nos puede ser muy útil a la hora de realizar varias tareas con una misma tecla, ya que la combinación de una tecla y la pulsación de una tecla especial nos puede permitir que respondamos a la pulsación de dicha tecla de una manera totalmente diferente. Si utilizas entornos gráficos y te gusta utilizar el teclado estarás muy acostumbrado a usar estas combinaciones de teclas.

En resumen, disponemos de toda la información necesaria para reaccionar ante los diferentes eventos de teclado.

Cada vez que se produce un evento de teclado, ya sea pulsar o soltar una tecla, este queda almacenado en la estructura *SDL_KeyboardEvent*. Esta estructura está definida de la siguiente manera:

```

1 ;typedef struct{
2 ;    Uint8 type;
3 ;    Uint8 state;
4 ;    SDL_keysym keysym;
5 ;} SDL_KeyboardEvent;
;
```

Procedemos a describir los distintos campos:

- *type*: Este campo está definido sobre un entero sin signo de 8 bits. Para el caso del teclado puede tomar los valores **SDL_KEYDOWN** si el tipo de evento es la pulsación de una tecla o bien **SDL_KEYUP** para el evento asociado con la acción de soltar una tecla. Todas las estructuras de SDL sobre eventos que vamos a estudiar tienen este campo. Especifica únicamente que tipo de evento ha ocurrido.

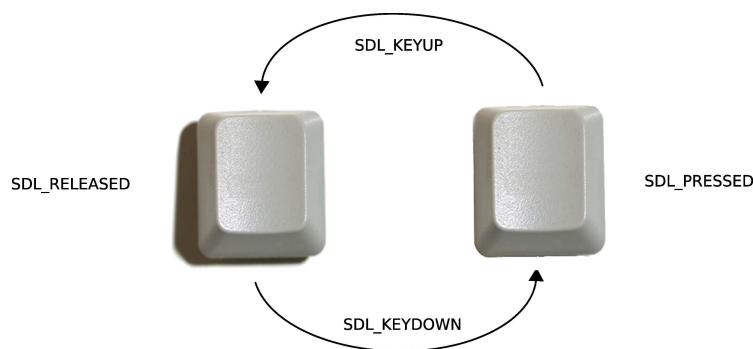


Figura 6.7: Eventos y estados en el teclado.

- *state*: Este campo posee información que podemos obtener con los datos que nos ofrece el primer campo. La información que nos proporciona po-

6. Captura y Gestión de Eventos

dríamos considerarla redundante por lo que podemos no usar este campo si lo consideramos oportuno. Nos facilita el capturar este tipo de estado no teniendo que implementar funciones que nos calculasen estos estados a partir del campo *type* de los eventos. Eso sí, las macros que definen el estado del botón son diferentes a las anteriores, claro está. **SDL_PRESSED** indica que la tecla está presionada y **SDL_RELEASED** nos indica que la tecla en cuestión no está pulsada sino “libre”.

Para obtener esta información a partir del primer campo bastaría con saber si la tecla ha sido presionada y no se ha soltado todavía para saber que está presionada. Podemos saber que una tecla no está pulsada si no se ha producido el evento **SDL_KEYDOWN** en la tecla en cuestión.

En la figura 6.7 puedes ver un diagrama sobre los eventos y estados del teclado.

- **keysym:** Este campo posee información sobre que tecla es la pulsada o liberada. Como puedes observar está definido sobre una estructura **SDL_keysym** que tiene el siguiente formato:

```
1 ;  
2 ;typedef struct {  
3 ;    Uint8 scancode;  
4 ;    SDLKey sym;  
5 ;    SDLMod mod;  
6 ;    Uint16 unicode;  
7 ;} SDL_keysym;  
;
```

- **scancode:** Es el código que genera el teclado a nivel de hardware. Puede cambiar dependiendo del sistema que utilicemos y del tipo de teclado que tengamos conectado. Utilizar este código en el desarrollo de la aplicación puede provocar que perdamos la compatibilidad con otras plataformas y otros sistemas. Por esta razón desaconsejamos el uso de este campo. Como regla general ignoraremos este campo por el hecho de que no nos permite asegurar la portabilidad. Usar un tipo de dato dependiente de un hardware en cuestión es una mala idea con SDL, a no ser que no tengamos otra alternativa, ya que perderemos potencial.
- **sym:** Este es el campo más importante de esta estructura ya que contiene un código identificativo que produce SDL según la tecla que haya sido pulsada o liberada. Esto supone que sea igual a todas las plataformas donde usemos SDL ya que es independiente de SDL y no del hardware que estemos utilizando. Es decir que si pulsamos, por ejemplo, la tecla ‘a’ SDL almacenará en este campo un código único para esta tecla, exactamente **SDLK_a**, que podremos consultar y nos dará la información de que la tecla pulsada es dicha ‘a’. Esto nos va a permitir que nuestro código sea portable entre diferentes

sistemas. Puedes consultar la tabla todos los códigos que ofrece SDL que está al final del capítulo.

Puedes ver en la tabla de códigos que, seguramente, existen más constantes que teclas tiene tu teclado. Esto es porque no todos los teclados de todos los sistemas son iguales y SDL tiene que garantizar la respuesta a todas las teclas de todas las plataformas en las que asgura poder trabajar sin problemas. Es una buena práctica no usar teclas especiales de un sistema, como puede ser la '*tecla windows*' ya que si queremos portar nuestro código a otro sistema operativo puede que perdamos funcionalidad. Este campo siempre está disponible. Otros, como el de *unicode*, puede que no podamos utilizarlo en un sistema determinado por lo que perderemos funcionalidad. Este es otro punto a favor para elegir a este campo como candidato a utilizar cuando gestionemos los eventos de teclado.

- *mod*: Este campo nos proporciona información sobre las teclas especiales **Ctrl**, **Alt**, **Shift** que se encuentran en el teclado y que se utilizan para dotar a una mayor funcionalidad a éste. Junto a la tabla de constantes **SDLKey** al final del capítulo puedes consultar la tabla de constantes de teclas modificadoras de SDL.
- *unicode*: En este campo se almacena el carácter ASCII o Unicode de la tecla pulsada. Por defecto este campo no es rellenado ya que supone realizar una conversión previa que produce una sobrecarga por lo que, en la mayoría de los casos, no es interesante que se cumpliera este campo.

Para activar el campo se utiliza la siguiente función:

```
int SDL_EnableUNICODE(int enable);
```

La función devuelve el estado anterior al cambio. El parámetro que recibe la función puede tomar tres valores:

- 0 : Desactiva la traducción Unicode.
- 1 : Activa la traducción Unicode.
- -1: No varía el estado de la traducción. Se utiliza para consultar el estado en un momento dado.

Si los 9 bit más altos del código están a 0 tendremos un carácter ASCII, es decir si el valor unicode es menor que 128 (0x80), entonces es un carácter ASCII. Si es mayor o igual a 128 el carácter será Unicode. Este dato es importante a la hora de internacionalizar nuestra aplicación. No corresponde a este tutorial ahondar en temas de codificación de caracteres, pero es interesante si vamos a desarrollar una aplicación que queramos distribuir a gran escala.

6. Captura y Gestión de Eventos

6.8.2.1. Ejemplo 2

Bien, hemos visto una buena parte de las variables que nos proporciona SDL para el manejo del teclado. Vamos a practicar un poco con un ejemplo. La siguiente aplicación nos permite gestionar la pulsación de las teclas cursoras (las flechas) del teclado mostrando un mensaje en consola por cada tecla pulsada.

```
1 ;// Ejemplo 2
2 ;//
3 ;// Listado: main.cpp
4 ;//
5 ;// Programa de pruebas. Eventos de teclado
6 ;// Este programa comprueba si se ha realizado un evento de teclado
7 ;// en las teclas cursoras
8 ;
9 ;#include <iostream>
10 ;#include <iomanip>
11 ;
12 ;#include <SDL/SDL.h>
13 ;
14 ;using namespace std;
15 ;
16 ;int main()
17 ;{
18 ;
19 ;    // Iniciamos el subsistema de video
20 ;
21 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
22 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
23 ;        exit(1);
24 ;    }
25 ;
26 ;    atexit(SDL_Quit);
27 ;
28 ;    // Comprobamos que sea compatible el modo de video
29 ;
30 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
31 ;
32 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
33 ;        exit(1);
34 ;
35 ;    }
36 ;
37 ;
38 ;    // Establecemos el modo de video
39 ;
40 ;    SDL_Surface *pantalla;
41 ;
42 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
43 ;
44 ;    if(pantalla == NULL) {
```

6.8. Teclado

```
45 ;         cerr << "No se pudo establecer el modo de video: "
46 ;             << SDL_GetError() << endl;
47 ;         exit(1);
48 ;
49 ;
50 ;     // Gestionamos la pulsacion de las teclas cursoras
51 ;     // Si se pulsa ESC salimos de la aplicacin
52 ;
53 ;     SDL_Event evento;
54 ;
55 ;     int x = 0;
56 ;     int y = 0;
57 ;
58 ;     for( ; ; ) {
59 ;
60 ;         while(SDL_PollEvent(&evento)) {
61 ;
62 ;             if(evento.type == SDL_KEYDOWN) {
63 ;
64 ;                 switch(evento.key.keysym.sym) {
65 ;                     case SDLK_UP:
66 ;                         ++y;
67 ;                         break;
68 ;
69 ;                     case SDLK_DOWN:
70 ;                         --y;
71 ;                         break;
72 ;
73 ;                     case SDLK_RIGHT:
74 ;                         ++x;
75 ;                         break;
76 ;
77 ;                     case SDLK_LEFT:
78 ;                         --x;
79 ;                         break;
80 ;                     case SDLK_ESCAPE:
81 ;                         return 0;
82 ;
83 ;                     default:
84 ;                         cout << "Ha pulsado otra tecla" << endl;
85 ;                 }
86 ;                 cout << "Valor x: " << setw(2) << x << " Valor y: "
87 ;                     << setw(2) << y << endl;
88 ;             }
89 ;
90 ;             if(evento.type == SDL_QUIT)
91 ;                 return 0;
92 ;
93 ;         }
94 ;
95 ;     return 0;
96 ;};
```

6. Captura y Gestión de Eventos

Lo único novedoso en este listado es la estructura *switch* que discrimina según sea la tecla pulsada. Vamos a dar respuesta cada vez que se pulse una flecha del teclado, y en caso de pulsar alguna otra tecla mostraremos un mensaje por consola indicando que se ha sido pulsada una tecla a la que no se le dará más respuesta que el mismo mensaje.

6.8.2.2. Repetición de tecla

Como puedes comprobar en el ejemplo anterior si mantienes una de las flechas pulsadas no se repetirá el evento de tecla presionada ya que se pulsó solo una vez. SDL te permite controlar la repetición del evento de tecla presionada.

Podemos controlar la habilitación o no de la repetición de tecla. En algunas aplicaciones con las que trabajaremos es interesante que, si pulsamos cierta tecla durante un tiempo determinado, el sistema entienda que queremos que dicha letra o acción se repita varias veces. Para tener activada esta opción debemos de utilizar la función:

```
int SDL_EnableKeyRepeat(int delay, int interval);
```

Como parámetros la función recibe el retardo (*delay*). En este parámetro indicamos que tiempo debe de esperar la aplicación antes de hacer efectiva la repetición de la tecla. En el parámetro *interval* especificaremos cada cuanto tiempo se va a realizar la repetición de la tecla en cuestión para provocar el efecto deseado.

La función devuelve 0 si todo ha ido bien y -1 indicando que ha existido error al establecer el modo de repetición.

La librería SDL proporciona dos constantes que propone como *ideales* para establecer el intervalo y el tiempo de retardo. Estas constantes son `SDL_DEFAULT_REPEAT_DELAY` y `SDL_DEFAULT_REPEAT_INTERVAL`.

6.8.3. Ejercicio 1

Modifica el listado del segundo ejemplo para que cuando pulsemos una tecla se produzca la repetición del evento.

Aquí tienes la solución:

```
1 ;// Ejercicio 1
2 ;//
3 ;// Listado: main.cpp
4 ;//
5 ;// Programa de pruebas. Eventos de teclado
```

```
6 ;// Este programa comprueba si se ha realizado un evento de teclado
7 ;// en las teclas cursoras
8 ;// Activamos la repetición en el teclado
9 ;
10 ;#include <iostream>
11 ;#include <iomanip>
12 ;
13 ;#include <SDL/SDL.h>
14 ;
15 ;using namespace std;
16 ;
17 ;int main()
18 ;{
19 ;    // Iremos definiendo las variables según las necesitemos
20 ;
21 ;    // Iniciamos el subsistema de video
22 ;
23 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
24 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;;
25 ;        exit(1);
26 ;    }
27 ;
28 ;    atexit(SDL_Quit);
29 ;
30 ;    // Comprobamos que sea compatible el modo de video
31 ;
32 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
33 ;
34 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
35 ;        exit(1);
36 ;
37 ;    }
38 ;
39 ;
40 ;    // Establecemos el modo de video
41 ;
42 ;    SDL_Surface *pantalla;
43 ;
44 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
45 ;
46 ;    if(pantalla == NULL) {
47 ;        cerr << "No se pudo establecer el modo de video: "
48 ;            << SDL_GetError() << endl;
49 ;        exit(1);
50 ;    }
51 ;
52 ;    // Activamos la repetición de las teclas
53 ;
54 ;
55 ;    int repeat = SDL_EnableKeyRepeat(SDL_DEFAULT_REPEAT_DELAY,
56 ;                                    SDL_DEFAULT_REPEAT_INTERVAL);
57 ;
58 ;    if(repeat < 0) {
```

6. Captura y Gestión de Eventos

```
59 ;
60 ;         cerr << "No se pudo establecer el modo repetición "
61 ;             << SDL_GetError() << endl;
62 ;         exit(1);
63 ;
64 ;     else {
65 ;
66 ;         cout << "Modo repetición activado:\n "
67 ;             << " Retardo: " << SDL_DEFAULT_REPEAT_DELAY
68 ;                 << "\nIntervalo: " <<SDL_DEFAULT_REPEAT_INTERVAL << endl;
69 ;
70 ;
71 ;
72 ;     // Gestionamos la pulsacion de las teclas cursoras
73 ;     // Si se pulsa ESC salimos de la aplicación
74 ;
75 ;     int x = 0;
76 ;     int y = 0;
77 ;
78 ;     SDL_Event evento;
79 ;
80 ;     for( ; ; ) {
81 ;
82 ;         while(SDL_PollEvent(&evento)) {
83 ;
84 ;             if(evento.type == SDL_KEYDOWN) {
85 ;
86 ;                 switch(evento.key.keysym.sym) {
87 ;                     case SDLK_UP:
88 ;                         ++y;
89 ;                         break;
90 ;
91 ;                     case SDLK_DOWN:
92 ;                         --y;
93 ;                         break;
94 ;
95 ;                     case SDLK_RIGHT:
96 ;                         ++x;
97 ;                         break;
98 ;
99 ;                     case SDLK_LEFT:
100 ;                         --x;
101 ;                         break;
102 ;                     case SDLK_ESCAPE:
103 ;                         return 0;
104 ;
105 ;                     default:
106 ;                         cout << "Ha pulsado otra tecla" << endl;
107 ;                 }
108 ;                 cout << "Valor x: " << setw(2) << x << " Valor y: "
109 ;                     << setw(2) << y << endl;
110 ;
111 ;             }
```

```

112 ;           if(evento.type == SDL_QUIT)
113 ;               return 0;
114 ;
115 ;       }
116 ;
117 ;   return 0;
118 ;}
;
```

6.8.4. Ejercicio 2

Vamos a seguir practicando con los eventos de teclado. Construye un programa que cada vez que pulsemos una tecla nos diga si el carácter asociado a dicha tecla es ASCII o UNICODE.

La solución propuesta es la siguiente:

```

;-----;
1 ;// Ejercicio 2
2 ;//
3 ;// Listado: main.cpp
4 ;//
5 ;// Programa de pruebas. Eventos de teclado
6 ;// Este programa indica si la tecla presionada
7 ;// se asocia a un carácter UNICODE o ASCII
8 ;
9 ;#include <iostream>
10 ;#include <iomanip>
11 ;
12 ;#include <SDL/SDL.h>
13 ;
14 ;using namespace std;
15 ;
16 ;int main()
17 ;{
18 ;    // Iremos definiendo las variables según las necesitemos
19 ;
20 ;    // Iniciamos el subsistema de video
21 ;
22 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
23 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;;
24 ;        exit(1);
25 ;    }
26 ;
27 ;    atexit(SDL_Quit);
28 ;
29 ;    // Comprobamos que sea compatible el modo de video
30 ;
31 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
32 ;
33 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
34 ;        exit(1);
35 ;
36 ;    }
;
```

6. Captura y Gestión de Eventos

```
37 ;
38 ;
39 ;    // Establecemos el modo de video
40 ;
41 ;    SDL_Surface *pantalla;
42 ;
43 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
44 ;
45 ;    if(pantalla == NULL) {
46 ;        cerr << "No se pudo establecer el modo de video: "
47 ;            << SDL_GetError() << endl;
48 ;        exit(1);
49 ;
50 ;
51 ;    // Activamos los caracteres UNICODE
52 ;
53 ;    SDL_EnableUNICODE(1);
54 ;
55 ;    // Si se pulsa ESC salimos de la aplicación
56 ;
57 ;    cout << "Pulsa ESC para terminar" << endl;
58 ;
59 ;    SDL_Event evento;
60 ;
61 ;    for( ; ; ) {
62 ;
63 ;        while(SDL_PollEvent(&evento)) {
64 ;
65 ;            if(evento.type == SDL_KEYDOWN) {
66 ;
67 ;                if(evento.key.keysym.unicode < 128 &&
68 ;                    evento.key.keysym.unicode != 0) {
69 ;
70 ;                    cout << "Carácter ASCII nº "
71 ;                        << setw(3) << evento.key.keysym.unicode << endl;
72 ;
73 ;                }
74 ;
75 ;                if(evento.key.keysym.unicode > 127) {
76 ;
77 ;                    cout << "Carácter UNICODE nº "
78 ;                        << setw(3) << evento.key.keysym.unicode << endl;
79 ;
80 ;                }
81 ;
82 ;                // Tecla ESC
83 ;
84 ;                if(evento.key.keysym.unicode == 27)
85 ;                    return 0;
86 ;
87 ;            if(evento.type == SDL_QUIT)
88 ;                return 0;
89 ;        }
```

```

90 ;    }
91 ;
92 ;    return 0;
93 ;}
;
```

6.8.5. Ejercicio 3

Como puedes ver es bastante simple gestionar el teclado mediante el uso de eventos. Vamos a realizar un ejercicio aprovechando lo que ya hemos visto en el tutorial. Carga una imagen en una pantalla y consigue que se mueva en ella. ¡Ten cuidado que no se te salga de la pantalla! Aprovecha para habilitar la opción de mostrar la ventana SDL a pantalla completa, por ejemplo, cuando se pulse la tecla 'f'.

```

;_____
1 ;// Ejercicio 3
2 ;//
3 ;// Listado: main.cpp
4 ;//
5 ;// Programa de pruebas. Eventos de teclado
6 ;// Esta aplicación carga una imagen y nos permite moverla
7 ;
8 ;#include <iostream>
9 ;#include <iomanip>
10 ;
11 ;#include <SDL/SDL.h>
12 ;
13 ;using namespace std;
14 ;
15 ;int main()
16 ;{
17 ;    // iremos definiendo las variables según las necesitemos
18 ;
19 ;    // Iniciamos el subsistema de video
20 ;
21 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
22 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;;
23 ;        exit(1);
24 ;    }
25 ;
26 ;    atexit(SDL_Quit);
27 ;
28 ;    // Comprobamos que sea compatible el modo de video
29 ;
30 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
31 ;
32 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
33 ;        exit(1);
34 ;
35 ;    }
36 ;
```

6. Captura y Gestión de Eventos

```
37 ;
38 ;    // Establecemos el modo de video
39 ;
40 ;    SDL_Surface *pantalla;
41 ;
42 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
43 ;
44 ;    if(pantalla == NULL) {
45 ;
46 ;        cerr << "No se pudo establecer el modo de video: "
47 ;            << SDL_GetError() << endl;
48 ;        exit(1);
49 ;
50 ;    }
51 ;
52 ;    // Cargamos una imagen en una superficie
53 ;
54 ;    SDL_Surface *personaje = SDL_LoadBMP("Imagenes/personaje.bmp");
55 ;
56 ;    if(personaje == NULL) {
57 ;
58 ;        cerr << "No se pudo cargar la imagen: "
59 ;            << SDL_GetError() << endl;
60 ;
61 ;        exit(1);
62 ;    }
63 ;
64 ;    // Establecemos el color de la transparencia
65 ;    // No será mostrado al realizar el blitting
66 ;
67 ;    SDL_SetColorKey(personaje, SDL_SRCCOLORKEY|SDL_RLEACCEL,\n
68 ;                    SDL_MapRGB(personaje->format, 0, 255, 0));
69 ;
70 ;    // Posición inicial del personaje
71 ;
72 ;    SDL_Rect posicion;
73 ;
74 ;    posicion.x = 300;
75 ;    posicion.y = 220;
76 ;    posicion.w = personaje->w;
77 ;    posicion.h = personaje->h;
78 ;
79 ;
80 ;    // Activamos la repetición de las teclas
81 ;
82 ;
83 ;    int repeat = SDL_EnableKeyRepeat(1,
84 ;                                    SDL_DEFAULT_REPEAT_INTERVAL);
85 ;
86 ;    if(repeat < 0) {
87 ;
88 ;        cerr << "No se pudo establecer el modo repetición "
89 ;            << SDL_GetError() << endl;
```

6.8. Teclado

```
90 ;         exit(1);
91 ;
92 ;
93 ;
94 ;     // Copiamos la imagen en la superficie principal
95 ;
96 ;     SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
97 ;
98 ;     // Mostramos la pantalla "oculta" del buffer
99 ;
100 ;    SDL_Flip(pantalla);
101 ;
102 ;
103 ;    // Gestionamos la pulsacion de las teclas cursoras
104 ;    // Si se pulsa ESC salimos de la aplicacin
105 ;
106 ;    SDL_Event evento;
107 ;
108 ;    for( ; ; ) {
109 ;
110 ;        while(SDL_PollEvent(&evento)) {
111 ;
112 ;            if(evento.type == SDL_KEYDOWN) {
113 ;
114 ;                switch(evento.key.keysym.sym) {
115 ;
116 ;                    case SDLK_UP:
117 ;
118 ;                        posicion.y -= 4;
119 ;
120 ;                        if(posicion.y < 0)
121 ;                            posicion.y = 0;
122 ;
123 ;                        break;
124 ;
125 ;                    case SDLK_DOWN:
126 ;
127 ;                        posicion.y += 4;
128 ;
129 ;                        if(posicion.y > 380)
130 ;                            posicion.y = 380;
131 ;
132 ;                        break;
133 ;
134 ;                    case SDLK_RIGHT:
135 ;
136 ;                        posicion.x += 4;
137 ;
138 ;                        if(posicion.x > 560)
139 ;                            posicion.x = 560;
140 ;
141 ;                        break;
142 ;
```

6. Captura y Gestión de Eventos

```
143 ;         case SDLK_LEFT:
144 ;
145 ;             posicion.x -= 4;
146 ;
147 ;             if(posicion.x < 0)
148 ;                 posicion.x = 0;
149 ;
150 ;             break;
151 ;
152 ;         case SDLK_ESCAPE:
153 ;
154 ;             SDL_FreeSurface(personaje);
155 ;             return 0;
156 ;
157 ;         case SDLK_f:
158 ;
159 ;             SDL_WM_ToggleFullScreen(pantalla);
160 ;             break;
161 ;
162 ;         default:
163 ;             cout << "Ha pulsado otra tecla" << endl;
164 ;
165 ;
166 ;             // Limpiamos la pantalla
167 ;
168 ;             SDL_FillRect(pantalla, NULL, 0);
169 ;
170 ;             // Cambiamos la posición del personaje
171 ;
172 ;             SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
173 ;
174 ;             // Actualizamos la pantalla principal
175 ;
176 ;             SDL_Flip(pantalla);
177 ;
178 ;             cout << " Valor x: " << setw(3) << posicion.x
179 ;                 << " Valor y: " << setw(3) << posicion.y << endl;
180 ;
181 ;
182 ;         if(evento.type == SDL_QUIT) {
183 ;
184 ;             SDL_FreeSurface(personaje);
185 ;             return 0;
186 ;
187 ;         }
188 ;
189 ;
190 ;     return 0;
191 ;};
```

Este ejercicio podríamos verlo como una fusión entre los conocimientos adquiridos en el tema anterior y los de este. No hay ninguna estructura de datos ni función nueva. Solamente hemos ido actualizando la pantalla con los

nuevos valores de la posición de nuestro personaje, redibujando al personaje en la nueva posición.

Hemos decidido que por cada pulsación (o repetición) de tecla se desplace el personaje en cuatro posiciones para no hacer los movimientos demasiado lentos para el intervalo de repetición que hemos establecido.

Controlamos que el personaje no pueda salir de la ventana mediante unas estructuras selectivas en cada uno de los casos que se actualiza la posición del personaje y así asegurarnos que no vamos a perderlo en el infinito.

En este ejemplo se ve claramente la estructura del *game loop*. En un primer lugar actualizamos la lógica del juego para en un segundo momento mostrar en pantalla los cambios realizados.

6.8.6. Ejercicio 4

Realiza un programa que nos permita configurar las teclas que vamos a utilizar para manejar el personaje del ejercicio anterior. Realiza el programa en modo consola ya que retomaremos este ejercicio para dotarlo de una interfaz gráfica más adelante.

Realiza una primera versión utilizando vectores de bajo nivel. Te será más sencillo si todavía no dominas C++.

Vamos a estudiar la solución:

```

1 ;_____
1 ;// Listado: teclado.h
2 ;//
3 ;// Funciones para la configuración del teclado
4 ;
5 ;
6 ;#ifndef _TECLADO_H_
7 ;#define _TECLADO_H_
8 ;
9 ;#include <SDL/SDL.h>
10 ;
11 ;enum Teclas {
12 ;
13 ;    UP,
14 ;    DOWN,
15 ;    LEFT,
16 ;    RIGHT,
17 ;    QUIT,
18 ;    FS, // FullScreen
19 ;};
20 ;

```

6. Captura y Gestión de Eventos

```
21 ;#define NUM_TECLAS 6
22 ;
23 ;// Nos permite personalizar las teclas que queremos utilizar
24 ;// en nuestra aplicación.
25 ;
26 ;// Recibe el número de teclas a configurar y devuelve
27 ;// en teclas un vector con las teclas personalizadas
28 ;
29 ;// Para un correcto funcionamiento no debe estar activada
30 ;// la repetición de teclas en SDL
31 ;
32 ;
33 ;int configura_teclado(SDLKey *teclas);
34 ;
35 ;#endif
```

Este es el fichero de cabecera de la función que va a configurar las teclas que vamos a utilizar en nuestra aplicación. En él definimos un enumerado con el nombre de las teclas que vamos a utilizar para evitar trabajar con índices en el vector que pueden llevar a confusión.

La función *configura_teclado* devuelve 0 en caso de éxito. Devuelve en el parámetro *teclas* un vector de seis posiciones que son las teclas que se van a configurar.

Vamos a ver la implementación de esta función:

```
;_____
1 ;// Listado: teclado.cpp
2 ;
3 ;// Implementación de las funciones
4 ;
5 ;
6 ;#include <iostream>
7 ;
8 ;#include "teclado.h"
9 ;
10 ;using namespace std;
11 ;
12 ;
13 ;
14 ;int configura_teclado(SDLKey *teclas) {
15 ;
16 ;    cout << " == Configurador de teclado == \n Pulse ARRIBA"
17 ;        << endl;
18 ;
19 ;
20 ;    // Evento auxiliar para guardar la tecla presionada
21 ;
22 ;    SDL_Event evento;
```

```
25 ; // Configuramos las teclas principales
26 ;
27 ; // ARRIBA
28 ;
29 ; cout << "ARRIBA: ";
30 ;
31 ; do {
32 ;
33 ;     SDL_WaitEvent(&evento); // Esperamos un evento
34 ;
35 ; } while(evento.type != SDL_KEYDOWN); // Pero sólo de tecla presionada
36 ;
37 ; teclas[UP] = evento.key.keysym.sym; // Almacenamos el símbolo
38 ;
39 ; cout << "OK\n Pulse ABAJO:" << endl;
40 ;
41 ;
42 ; // ABAJO
43 ;
44 ; do {
45 ;
46 ;     SDL_WaitEvent(&evento);
47 ;
48 ; } while(evento.type != SDL_KEYDOWN);
49 ;
50 ; teclas[DOWN] = evento.key.keysym.sym;
51 ;
52 ; cout << "OK \n Pulse IZQUIERDA" << endl;
53 ;
54 ;
55 ; // IZQUIERDA
56 ;
57 ; do {
58 ;
59 ;     SDL_WaitEvent(&evento);
60 ;
61 ; } while(evento.type != SDL_KEYDOWN);
62 ;
63 ; teclas[LEFT] = evento.key.keysym.sym;
64 ;
65 ; cout << "OK \n Pulse DERECHA" << endl;
66 ;
67 ;
68 ; // DERECHA
69 ;
70 ; do {
71 ;
72 ;     SDL_WaitEvent(&evento);
73 ;
74 ; } while(evento.type != SDL_KEYDOWN);
75 ;
76 ; teclas[RIGHT] = evento.key.keysym.sym;
77 ;
```

6. Captura y Gestión de Eventos

```
78 ;     cout << "OK \n Pulse SALIR" << endl;
79 ;
80 ;     // SALIR
81 ;
82 ;     do {
83 ;
84 ;         SDL_WaitEvent(&evento);
85 ;
86 ;     } while(evento.type != SDL_KEYDOWN);
87 ;
88 ;     teclas[QUIT] = evento.key.keysym.sym;
89 ;
90 ;     cout << "OK \n Pulse PANTALLA COMPLETA" << endl;
91 ;
92 ;     // PANTALLA COMPLETA
93 ;
94 ;     do {
95 ;
96 ;         SDL_WaitEvent(&evento);
97 ;
98 ;     } while(evento.type != SDL_KEYDOWN);
99 ;
100 ;    teclas[FS] = evento.key.keysym.sym;
101 ;
102 ;    cout << "OK" << endl;
103 ;
104 ;    return 0;
105 ;}
```

Como puedes ver se repite para cada una de las teclas una estructura bien definida. El objetivo de cada una de estas estructuras es personalizar cada una de las teclas a utilizar en el programa. Dicha estructura consiste en un bucle *do while* que se repite hasta que se produce un evento de teclado. Una vez se ha producido se introduce el símbolo de la tecla pulsada en el evento en una posición del vector *teclas* que es el utilizado para guardar la configuración que estamos inicializando.

Esta estructura se repite para cada una de las teclas a configurar. Sería mucho más elegante haber creado un bucle que itere seis veces para configurar las seis teclas. No es muy complejo de programar en C++, te animo a que lo implementes y así pongas en práctica tu manejo de este lenguaje.

Para terminar vamos a revisar los cambios que hemos realizado en el programa principal para que acepte nuestra función de configuración:

```
1 ;// Ejercicio 4
2 ;//
3 ;// Listado: main.cpp
4 ;//
5 ;// Programa de pruebas. Eventos de teclado
```

```
6 ;// Nos permite configurar que teclas queremos utilizar
7 ;// y nos pedirá confirmación
8 ;// Esta aplicación carga una imagen y nos permite moverla
9 ;
10 ;#include <iostream>
11 ;#include <iomanip>
12 ;
13 ;#include <SDL/SDL.h>
14 ;#include "teclado.h"
15 ;
16 ;using namespace std;
17 ;
18 ;int main()
19 ;{
20 ;    // iremos definiendo las variables según las necesitemos
21 ;
22 ;    // Iniciamos el subsistema de video
23 ;
24 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
25 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
26 ;        exit(1);
27 ;    }
28 ;
29 ;    atexit(SDL_Quit);
30 ;
31 ;
32 ;    // Comprobamos que sea compatible el modo de video
33 ;
34 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
35 ;
36 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
37 ;        exit(1);
38 ;
39 ;    }
40 ;
41 ;    // Establecemos el modo de video
42 ;
43 ;    SDL_Surface *pantalla;
44 ;
45 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
46 ;
47 ;    if(pantalla == NULL) {
48 ;
49 ;        cerr << "No se pudo establecer el modo de video: "
50 ;            << SDL_GetError() << endl;
51 ;        exit(1);
52 ;
53 ;    }
54 ;
55 ;    // Cargamos una imagen en una superficie
56 ;
57 ;    SDL_Surface *personaje = SDL_LoadBMP("Imagenes/personaje.bmp");
58 ;
```

6. Captura y Gestión de Eventos

```
59 ;     if(personaje == NULL) {
60 ;
61 ;         cerr << "No se pudo cargar la imagen: "
62 ;             << SDL_GetError() << endl;
63 ;
64 ;         exit(1);
65 ;
66 ;
67 ;     // Establecemos el color de la transparencia
68 ;     // No será mostrado al realizar el blitting
69 ;
70 ;     SDL_SetColorKey(personaje, SDL_SRCCOLORKEY|SDL_RLEACCEL,\n
71 ;                     SDL_MapRGB(personaje->format, 0, 255, 0));
72 ;
73 ;     // Posición inicial del personaje
74 ;
75 ;     SDL_Rect posicion;
76 ;
77 ;     posicion.x = 300;
78 ;     posicion.y = 220;
79 ;     posicion.w = personaje->w;
80 ;     posicion.h = personaje->h;
81 ;
82 ;
83 ;     // Activamos la repetición de las teclas
84 ;
85 ;
86 ;     // Configuramos las teclas (versión consola)
87 ;
88 ;     SDLKey teclas[6];
89 ;
90 ;     configura_teclado(teclas);
91 ;
92 ;
93 ;     // Activamos la repetición de las teclas
94 ;
95 ;     int repeat = SDL_EnableKeyRepeat(1,
96 ;                                     SDL_DEFAULT_REPEAT_INTERVAL);
97 ;
98 ;     if(repeat < 0) {
99 ;
100 ;         cerr << "No se pudo establecer el modo repetición "
101 ;             << SDL_GetError() << endl;
102 ;         exit(1);
103 ;     }
104 ;
105 ;     // Copiamos la imagen en la superficie principal
106 ;
107 ;     SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
108 ;
109 ;     // Mostramos la pantalla "oculta" del buffer
110 ;
111 ;     SDL_Flip(pantalla);
```

```
112 ;
113 ;
114 ;    // Gestionamos la pulsacion de las teclas cursoras
115 ;    // Si se pulsa ESC salimos de la aplicacin
116 ;
117 ;    SDL_Event evento;
118 ;
119 ;    for( ; ; ) {
120 ;
121 ;        while(SDL_PollEvent(&evento)) {
122 ;
123 ;            if(evento.type == SDL_KEYDOWN) {
124 ;
125 ;                if(teclas[UP] == evento.key.keysym.sym) {
126 ;
127 ;                    posicion.y -= 4;
128 ;
129 ;                    if(posicion.y < 0)
130 ;                        posicion.y = 0;
131 ;
132 ;                } else if(teclas[DOWN] == evento.key.keysym.sym) {
133 ;
134 ;                    posicion.y += 4;
135 ;
136 ;                    if(posicion.y > 380)
137 ;                        posicion.y = 380;
138 ;
139 ;                } else if(teclas[RIGHT] == evento.key.keysym.sym) {
140 ;
141 ;                    posicion.x += 4;
142 ;
143 ;                    if(posicion.x > 560)
144 ;                        posicion.x = 560;
145 ;
146 ;                } else if(teclas[LEFT] == evento.key.keysym.sym) {
147 ;
148 ;                    posicion.x -= 4;
149 ;
150 ;                    if(posicion.x < 0)
151 ;                        posicion.x = 0;
152 ;
153 ;                } else if(teclas[QUIT] == evento.key.keysym.sym) {
154 ;
155 ;                    SDL_FreeSurface(personaje);
156 ;
157 ;                    return 0;
158 ;
159 ;                } else if(teclas[FS] == evento.key.keysym.sym) {
160 ;
161 ;                    SDL_WM_ToggleFullScreen(pantalla);
162 ;
163 ;                } else {
164 ;
```

6. Captura y Gestión de Eventos

```
165 ;           cout << "Tecla desconocida" << endl;
166 ;
167 ;
168 ;
169 ;           // Limpiamos la pantalla
170 ;
171 ;           SDL_FillRect(pantalla, NULL, 0);
172 ;
173 ;           // Cambiamos la posición del personaje
174 ;
175 ;           SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
176 ;
177 ;           // Actualizamos la pantalla principal
178 ;
179 ;           SDL_Flip(pantalla);
180 ;
181 ;           cout << " Valor x: " << setw(3) << posicion.x
182 ;                           << " Valor y: " << setw(3) << posicion.y << endl;
183 ;
184 ;
185 ;           if(evento.type == SDL_QUIT) {
186 ;
187 ;               SDL_FreeSurface(personaje);
188 ;               return 0;
189 ;
190 ;           }
191 ;
192 ;
193 ;       return 0;
194 ;
```

Lo primero que hemos añadido en el listado es una variable *teclas* que vamos a utilizar para guardar la configuración de teclas personalizada. Seguidamente hacemos una llamada a *configura_teclado()* para completar esta variable. Es importante que no tengamos activada la repetición de teclas ya que podría darse el caso de que almacenásemos la misma tecla para realizar dos acciones.

El otro cambio que hemos tenido que realizar ha sido en la estructura selectiva que respondía a los eventos producidos por los usuarios. La variable de control sigue siendo la misma, *evento*, pero ahora para saber qué debemos hacer en vez de utilizar las constantes que define SDL para cada tecla utilizamos el vector que hemos utilizado para guardar la configuración personalizada. En los casos de un *switch* no podemos poner una variable de ahí que hayamos cambiado esta estructura selectiva por sucesivos *if*.

6.8.7. Manejo del teclado consultando su estado

SDL proporciona una función que nos permite conocer el estado del teclado en un momento dado. Esta función es la que nos ofrece el poder utilizar dos

técnicas de trabajo, como vimos anteriormente, una para manejar el teclado mediante eventos y otra accediendo al estado del teclado. El prototipo de la función SDL que nos informa del estado del teclado es:

```
Uint8 *SDL_GetKeyState(int * numkeys);
```

Esta función nos devuelve un puntero a un vector con el estado de cada una de las teclas que componen el teclado. El parámetro que recibe especifica el tamaño del vector que queremos obtener. Es habitual pasar `NULL` como parámetro lo que produce que la función nos devuelva el valor del estado del evento para toda y cada una teclas.

Para consultar los datos del vector utilizamos las mismas constantes de teclado que proporciona SDL en el campo `sym` que puedes consultar en el anexo del capítulo. Si en el momento de llamar a la función la tecla está pulsada el valor almacenado en la posición de dicha tecla será 1. Si no lo está, como puedes suponer, será 0.

La forma de trabajar con este método sería la siguiente:

```
;  
1 ;// Definimos el vector donde vamos a guardar  
2 ;// el estado del teclado  
3 ;  
4 ;Uint8* teclado;  
5 ;  
6 ;// Capturamos el estado del teclado  
7 ;  
8 ;teclado = SDL_GetKeyState(NULL);  
9 ;  
10 ;// Ahora la estructura selectiva para  
11 ;// reaccionar ante las teclas pulsadas en el teclado  
12 ;  
13 ;if(teclado[SDLK_] == 1) {  
14 ;  
15 ;    // La tecla SDLK_ está pulsada  
16 ;  
17 ;    // ...  
18 ;  
19 ;    // Aquí van las acciones a realizar  
20 ;  
21 ;}
```

Lo primero que hacemos es declarar una variable del tipo `Uint8` que necesitamos para almacenar el estado del teclado. Seguidamente llamamos a la función `SDL_GetKeyState()` pasándole el valor `NULL` como parámetro para que nos devuelva el estado de todas las teclas.

El siguiente paso es responder a las teclas que queramos. Sabiendo que un valor del vector a 1 significa que la tecla está pulsada y 0 que está liberada

6. Captura y Gestión de Eventos

podemos actuar como lo hacíamos con los eventos.

6.8.7.1. Ejemplo 3

Vamos a implementar un ejemplo del manejo del teclado conociendo el estado de sus eventos directamente:

```
;_____
1 ;// Listado 3
2 ;//
3 ;// Listado: main.cpp
4 ;//
5 ;// Programa de pruebas. Manejando el teclado conociendo su estado
6 ;
7 ;
8 ;#include <iostream>
9 ;#include <iomanip>
10 ;#include <SDL/SDL.h>
11 ;
12 ;
13 ;using namespace std;
14 ;
15 ;
16 ;int main()
17 ;{
18 ;
19 ;    // Iniciamos el subsistema de video
20 ;
21 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
22 ;
23 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
24 ;        exit(1);
25 ;    }
26 ;
27 ;    atexit(SDL_Quit);
28 ;
29 ;
30 ;    // Comprobamos que sea compatible el modo de video
31 ;
32 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
33 ;
34 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
35 ;        exit(1);
36 ;
37 ;    }
38 ;
39 ;    // Establecemos el modo de video
40 ;
41 ;    SDL_Surface *pantalla;
42 ;
43 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
44 ;
```

```
45 ;     if(pantalla == NULL) {
46 ;
47 ;         cerr << "No se pudo establecer el modo de video: "
48 ;             << SDL_GetError() << endl;
49 ;
50 ;         exit(1);
51 ;
52 ;
53 ; // Si se pulsa ESC salimos de la aplicación
54 ;
55 ;     int x = 0;
56 ;     int y = 0;
57 ;
58 ;     Uint8 *teclado;
59 ;
60 ; // Para que no imprima los valores cuando no haya cambios
61 ;
62 ;     bool cambio = false;
63 ;
64 ;     for( ; ; ) {
65 ;
66 ;         // Actualiza el estado de los dispositivos
67 ;
68 ;         SDL_PumpEvents();
69 ;
70 ;         // Tomamos el estado
71 ;
72 ;         teclado = SDL_GetKeyState(NULL);
73 ;
74 ;         cambio = false;
75 ;
76 ;         if(teclado[SDLK_UP]) {
77 ;             ++y;
78 ;             cambio = true;
79 ;         }
80 ;
81 ;         if(teclado[SDLK_DOWN]) {
82 ;             --y;
83 ;             cambio = true;
84 ;         }
85 ;
86 ;         if(teclado[SDLK_RIGHT]) {
87 ;             ++x;
88 ;             cambio = true;
89 ;         }
90 ;
91 ;         if(teclado[SDLK_LEFT]) {
92 ;             --x;
93 ;             cambio = true;
94 ;         }
95 ;
96 ;         if(teclado[SDLK_ESCAPE] || teclado[SDLK_q]) {
97 ;
```

6. Captura y Gestión de Eventos

```
98 ;           return 0;
99 ;
100 ;
101 ;       if(cambio == true)
102 ;           cout << "Valor x: " << setw(7) << x
103 ;                           << " Valor y:" << setw(7) << y << endl;
104 ;
105 ;   }
106 ;
107 ;}
```

La primera diferencia con los listados anteriores es la aparición de una variable de tipo `Uint8 * teclado` para almacenar el estado del teclado. Una vez en el bucle que controla la aplicación lo primero que hacemos es una llamada a la función:

```
void SDL_PumpEvents(void);
```

Esta llamada la realizamos siempre antes de consultar directamente el estado del teclado y cada vez que vayamos a hacerlo. Debemos de hacer una llamada a esta función siempre para que se actualice el estado de cualquier dispositivo que vayamos a consultar. Esta función se encarga de capturar los eventos e introducirlo en la cola de eventos, es decir actualiza el estado de los eventos de los dispositivos. Bajo circunstancias normales, cuando en vez de consultar el estado del teclado trabajemos con eventos, no tendremos que llamar a esta función porque las funciones `SDL_WaitInput` y `SDL_PollInput` se encargan de llamar a esta función internamente liberándonos de esta tarea.

Es crucial llamar a esta función para leer información directamente de los dispositivos de entrada, por lo que es un desahogo que SDL pueda hacerlo por nosotros. Es fundamental que llamemos a esta función antes de conocer el estado de un dispositivo si vamos consultar el estado del mismo directamente.

En cuanto al resto del listado la lógica es bastante simple. Consultamos las posiciones del vector que sean relevantes en nuestra aplicación. En este caso los cursores (o flechas del teclado) y las tecla ESC para poder salir de la aplicación.

Como ves la conversión del listado de eventos a trabajar directamente con el estado del teclado ha sido bastante simple. Basta con añadir dos funciones y cambiar en las condiciones de la estructura selectiva la estructura del evento por la de la tabla de estado.

Otra función interesante para el manejo del teclado mediante esta técnica es la que nos permite conocer el estado de las teclas modificadoras. La declaración del a función es:

```
SDLMod SDL_GetModState(void);
```

Como puedes observar la función no recibe parámetros y devuelve una estructura del tipo `SDLMod`. Esta estructura está compuesta por una serie de banderas de bits definidas en forma de constantes al final del capítulo en la tabla “Constantes modificadoras de teclado SDL”. Si queremos que se puedan utilizar en nuestro videojuego tendremos que utilizar esta función.

Si queremos podemos establecer el estado de una de estas teclas modificadoras. En un momento dado nos puede interesar “forzar” a que cierta tecla modificadora esté pulsada. Para esto usamos la siguiente función:

```
void SDL_SetModState(SDLMod modstate);
```

Esta función recibe como parámetro una combinación de bits, como la devuelta por la función anterior, del tipo `SDLMod`. El estado de la tecla modificadora será cambiado por el pasado como parámetro.

6.8.8. Otras Funciones

Para terminar el tema de la gestión del teclado vamos a presentar una función que podíamos haber encuadrado en cualquier subsección referente a este dispositivo. Dicha función se encarga de devolver el nombre SDL de para cualquier tecla basada en la constante `SDLK_x` donde *x* hace referencia a la tecla en cuestión como, por ejemplo, `SDLK_a` hace referencia a la tecla ‘a’. El prototipo de la función es:

```
char *SDL_GetKeyName(SDLKey key);
```

Como era esperable la función recibe como parámetro la constante `SDLK_x` en cuestión y devuelve el nombre SDL para dicha tecla.

6.8.9. Ejercicio 5

Completa el ejercicio que nos permitía configurar permitiendo al usuario cambiar la configuración si no es la que deseaba. Para ello hay que mostrar las teclas que ha configurado y seguidamente cuestionar si la configuración es de su agrado.

Aquí tienes la solución:

```
1 ;// Listado: teclado.cpp
2 ;//
3 ;// Implementación de las funciones
4 ;
5 ;
```

6. Captura y Gestión de Eventos

```
6 ;#include <iostream>
7 ;
8 ;#include "teclado.h"
9 ;
10 ;using namespace std;
11 ;
12 ;
13 ;
14 ;int configura_teclado(SDLKey *teclas) {
15 ;
16 ;    bool valida = false;
17 ;
18 ;    do {
19 ;
20 ;        cout << " == Configurador de teclado == \n Pulse ARRIBA"
21 ;            << endl;
22 ;
23 ;
24 ;        // Evento auxiliar para guardar la tecla presionada
25 ;
26 ;        SDL_Event evento;
27 ;
28 ;
29 ;        // Configuramos las teclas principales
30 ;
31 ;        // ARRIBA
32 ;
33 ;        cout << "ARRIBA: ";
34 ;
35 ;        do {
36 ;
37 ;            SDL_WaitEvent(&evento); // Esperamos un evento
38 ;
39 ;            } while(evento.type != SDL_KEYDOWN); // Pero sólo de tecla presionada
40 ;
41 ;        teclas[UP] = evento.key.keysym.sym; // Almacenamos el símbolo
42 ;
43 ;        cout << "OK\n Pulse ABAJO: " << endl;
44 ;
45 ;
46 ;        // ABAJO
47 ;
48 ;        do {
49 ;
50 ;            SDL_WaitEvent(&evento);
51 ;
52 ;            } while(evento.type != SDL_KEYDOWN);
53 ;
54 ;        teclas[DOWN] = evento.key.keysym.sym;
55 ;
56 ;        cout << "OK \n Pulse IZQUIERDA" << endl;
57 ;
58 ;}
```

```
59 ;         // IZQUIERDA
60 ;
61 ;     do {
62 ;
63 ;         SDL_WaitEvent(&evento);
64 ;
65 ;     } while(evento.type != SDL_KEYDOWN);
66 ;
67 ;     teclas[LEFT] = evento.key.keysym.sym;
68 ;
69 ;     cout << "OK \n Pulse DERECHA" << endl;
70 ;
71 ;     // DERECHA
72 ;
73 ;     do {
74 ;
75 ;         SDL_WaitEvent(&evento);
76 ;
77 ;     } while(evento.type != SDL_KEYDOWN);
78 ;
79 ;     teclas[RIGHT] = evento.key.keysym.sym;
80 ;
81 ;     cout << "OK \n Pulse SALIR" << endl;
82 ;
83 ;     // SALIR
84 ;
85 ;     do {
86 ;
87 ;         SDL_WaitEvent(&evento);
88 ;
89 ;     } while(evento.type != SDL_KEYDOWN);
90 ;
91 ;     teclas[QUIT] = evento.key.keysym.sym;
92 ;
93 ;     cout << "OK \n Pulse PANTALLA COMPLETA" << endl;
94 ;
95 ;     // PANTALLA COMPLETA
96 ;
97 ;     do {
98 ;
99 ;         SDL_WaitEvent(&evento);
100 ;
101 ;     } while(evento.type != SDL_KEYDOWN);
102 ;
103 ;     teclas[FS] = evento.key.keysym.sym;
104 ;
105 ;     cout << "OK" << endl;
106 ;
107 ;     cout << "\nLa configuración obtenida es la siguiente: " << endl;
108 ;
109 ;     cout << "ARRIBA : " << SDL_GetKeyName(teclas[UP]) << endl;
110 ;
111 ;     cout << "ABAJO : " << SDL_GetKeyName(teclas[DOWN]) << endl;
```

6. Captura y Gestión de Eventos

```
112 ;         cout << "IZQUIERDA : " << SDL_GetKeyName(teclas[LEFT]) << endl;
113 ;         cout << "DERECHA : " << SDL_GetKeyName(teclas[RIGHT]) << endl;
114 ;         cout << "SALIR : " << SDL_GetKeyName(teclas[QUIT]) << endl;
115 ;         cout << "PANTALLA COMPLETA: " << SDL_GetKeyName(teclas[FS]) << endl;
116 ;
117 ;         cout << "¿Es correcta? (s o n) : " << endl;
118 ;
119 ;     do {
120 ;
121 ;         SDL_WaitEvent(&evento);
122 ;
123 ;         } while(evento.type != SDL_KEYDOWN);
124 ;
125 ;         if(evento.key.keysym.sym == SDLK_s)
126 ;             valida = true;
127 ;
128 ;
129 ;
130 ;     } while(valida == false);
131 ;
132 ;     cout << "Situate en la ventana SDL y utiliza las teclas configuradas"
133 ;         << endl;
134 ;
135 ;     return 0;
136 ;}
```

Como puedes ver es muy parecida a la del ejercicio 4. Hemos añadido dos cosas importantes. La primera es encapsular al proceso en un bucle *do while* para repetir la configuración hasta que el usuario la de por buena. La segunda es que para saber si el usuario está conforme con los valores introducidos los mostraremos por pantalla mediante la función *SDL_GetKeyName()* para mostrar por pantalla símbolos que sean humanamente comprensibles. Capturaremos un evento y si corresponde con la pulsación de la tecla 's' daremos la configuración por buena. En cualquier otro caso repetiremos el proceso.

Ya hemos visto las formas que tenemos de manejar el dispositivo de entrada por excelencia del ordenador. Ahora la decisión entre usar un tipo de técnica de manejo de teclado u otra está en tu mano y en el de el tipo de aplicación que vayas a desarrollar. La mayor complicación que puede aparecer en el manejo del teclado por eventos es la aparición de varias estructuras anidadas y la necesidad de utilizar varias teclas a la vez no especiales. Conociendo en profundidad y claramente éstas no tendremos problemas en manejar este dispositivo en el desarrollo de nuestra aplicación. En los ejemplos utilizaremos ambos tipos de técnicas, para que te familiarices con ellas, y seas capaz de discernir cual te conviene en todo momento.

6.9. Ratón

El ratón es, junto al teclado, el elemento de entrada más común de las computadoras actuales. En ciertos tipos de juego nos puede interesar utilizar el ratón como un medio de entrada más ágil que el teclado para determinadas tareas. Como pasa con los demás dispositivos que estamos estudiando en este curso el ratón puede ser controlado a través de eventos o bien conociendo el estado de éste en un momento dado. Empecemos estudiando el manejo por eventos.

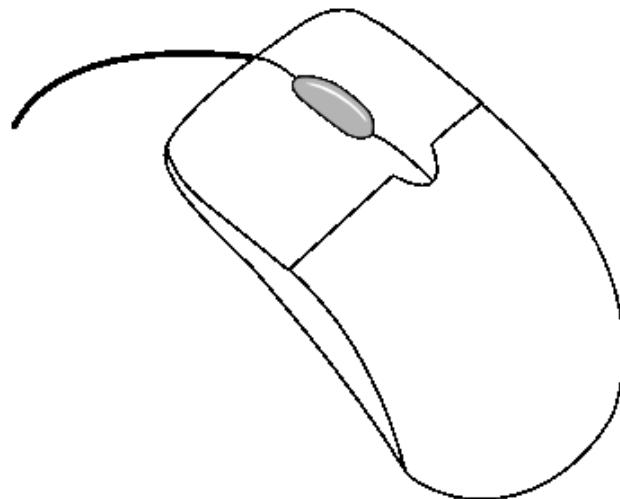


Figura 6.8: Ratón.

6.9.1. Manejo del ratón mediante eventos

Por la naturaleza del dispositivo existen varios tipos de eventos asociados a él, exactamente tres, que son comunes a cualquier tipo de ratón. Como pasaba con el teclado existen numerosos tipos de ratos en el mercado pero tienen unas características comunes que han permitido la creación de unos eventos asociados a este tipo de dispositivos. Vamos a estudiar los tres tipos de eventos del ratón.

El primero se refiere al movimiento del mismo. Cuando movemos el ratón sobre una superficie SDL genera un evento que nos indica la posición del puntero así como la distancia recorrida desde la última posición de reposo. A la misma vez añade el estado de los botones del ratón por si estamos

6. Captura y Gestión de Eventos

realizando una acción combinada, como por ejemplo, desplazar un objeto.

Imagínate que acabas de arrancar la aplicación y mueves el ratón. SDL generará un evento del tipo movimiento de ratón y guardaría tres tipos de información. La primera es donde está el puntero del ratón, la segunda la distancia al punto donde el ratón estuvo en reposo por última vez y la tercera si existe algún botón presionado y cual es. Como puedes ver tenemos en un sólo evento toda la información que podemos necesitar del ratón.

En cuanto a las acciones sobre los botones del ratón SDL genera un evento cuando se pulsa uno de los botones del ratón y otro diferente cuando soltamos dicho botón, con un comportamiento análogo al que se producía cuando pulsabamos una tecla en la gestión de eventos del teclado. Además, como no podía ser de otra manera, nos proporciona información que especifica que botón hemos pulsado y en qué posición lo hicimos.

Cuando movemos el ratón y se produce el evento correspondiente el subsistema nos devuelve una estructura definida de la siguiente forma:

```
1 ;  
2 ;typedef struct{  
3 ;    Uint8 type;  
4 ;    Uint8 state;  
5 ;    Uint16 x, y;  
6 ;    Sint16 xrel, yrel;  
7 ;} SDL_MouseMotionEvent;  
;
```

Pasamos a describir cada uno de los campos de esta estructura:

- *type*: Identifica el tipo de evento, en este caso **SDL_MOUSEMOTION**, es decir movimiento del cursor o puntero del ratón.
- *state*: Es un campo de banderas de bits que devuelve el estado de los botones del ratón. Puede ser consultado con la macro **SDL_BUTTON()**, pasándole como parámetro 1, 2 ó 3 para indicar el botón que queremos consultar. En SDL el número uno se identifica con el botón izquierdo, el dos con el botón central y tres con el botón derecho. Si lo prefieres puedes consultar el valor del botón pulsado haciendo uso de las constantes **SDL_BUTTON_LMASK** para cuando el botón izquierdo está pulsado, **SDL_BUTTON_MMASK** para cuando está pulsado el central y **SDL_BUTTON_RMASK** para cuando se pulsa el botón derecho. Parece más cómodo recordar la secuencia de números que usar estas máscaras.
- *x, y*: Proporcionan las coordenadas (x, y) de la posición del ratón.
- *xrel, yrel*: Son la posición relativa con respecto del puntero del ratón con respecto al último evento del mismo, es decir, desde la última posición de

reposo. Dependiendo del tipo de efecto que queramos poner de manifiesto nos interesará manejar la posición absoluta o la posición relativa del movimiento del ratón.

Se asocia al ratón otro tipo de evento referente a la pulsación de los botones. Cuando se pulsa uno de los botones del dispositivo SDL crea un evento en una estructura definida de la siguiente forma:

```

1 ;  

2 ;typedef struct {  

3 ;    Uint8 type;  

4 ;    Uint8 button;  

5 ;    Uint8 state;  

6 ;    Uint16 x, y;  

7 ;} SDL_MouseButtonEvent;  

;
```

Pasamos a describir los distintos campos de la estructura:

- *type*: Indica el tipo de evento y en este caso puede ser de dos tipos `SDL_MOUSEBUTTONDOWN` para indicar que se pulsó el botón del ratón o `SDL_MOUSEBUTTONUP` para indicar que el botón fue liberado. Lo habitual es responder a la acción de pulsación del botón y usar el evento de liberación del botón para casos más particulares y efectos concretos.
- *button*: Este campo puede tomar tres valores depende del botón del ratón que se pulse. `SDL_BUTTON_LEFT` para el botón izquierdo `SDL_BUTTON_MIDDLE` para el botón central y `SDL_BUTTON_RIGHT` para indicar que se pulsó o se soltó en botón derecho.
- *state*: Este campo, igual que en los eventos de teclado, posee una información redundante ya que la podemos obtener a través de los datos del campo *type*. Sus posibles valores son `SDL_PRESSED` para el botón del ratón pulsado y `SDL_RELEASED` si el botón del ratón está liberado.
- *x, y*: Almacena las coordenadas del ratón de cuando se produjo la pulsación del botón del ratón o su liberación, es decir, en el momento que se produjo el evento de botón de ratón.

6.9.1.1. Ejemplo 4

Ya hemos visto mucha información nueva referente a los eventos del ratón. Vamos a ponerla en práctica para entender un poquito mejor de que va todo esto.

```

1 ;  

2 ;// Listado 4 - Eventos  

3 ;//
```

6. Captura y Gestión de Eventos

```
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Eventos de ratón
5 ;// Este programa comprueba si se ha realizado un evento de ratón
6 ;// y muestra por consola los eventos de ratón que se van produciendo
7 ;
8 ;#include <iostream>
9 ;#include <iomanip>
10 ;
11 ;#include <SDL/SDL.h>
12 ;
13 ;using namespace std;
14 ;
15 ;int main()
16 ;{
17 ;
18 ;    // Iniciamos el subsistema de video
19 ;
20 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
21 ;
22 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
23 ;        exit(1);
24 ;
25 ;    }
26 ;
27 ;
28 ;    atexit(SDL_Quit);
29 ;
30 ;    // Comprobamos que sea compatible el modo de video
31 ;
32 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
33 ;
34 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
35 ;        exit(1);
36 ;
37 ;    }
38 ;
39 ;
40 ;    // Establecemos el modo de video
41 ;
42 ;    SDL_Surface *pantalla;
43 ;
44 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
45 ;
46 ;    if(pantalla == NULL) {
47 ;
48 ;        cerr << "No se pudo establecer el modo de video: "
49 ;            << SDL_GetError() << endl;
50 ;
51 ;        exit(1);
52 ;    }
53 ;
54 ;    SDL_Event evento;
55 ;
```

```

56 ; // Bucle "infinito"
57 ;
58 ; for( ; ; ) {
59 ;
60 ;     while(SDL_PollEvent(&evento)) {
61 ;
62 ;         if(evento.type == SDL_KEYDOWN) {
63 ;
64 ;             if(evento.key.keysym.sym == SDLK_ESCAPE)
65 ;                 return 0;
66 ;
67 ;         }
68 ;
69 ;         if(evento.type == SDL_QUIT)
70 ;             return 0;
71 ;
72 ;         if(evento.type == SDL_MOUSEMOTION){
73 ;
74 ;             cout << "X: " << setw(3) << evento.motion.x
75 ;                         << " - Y: " << setw(3) << evento.motion.y << endl;
76 ;
77 ;         }
78 ;
79 ;         if(evento.type == SDL_MOUSEBUTTONDOWN) {
80 ;
81 ;             if(evento.button.type == SDL_MOUSEBUTTONDOWN) {
82 ;                 cout << "X: " << setw(3) << evento.button.x
83 ;                     << " - Y: " << setw(3) << evento.button.y
84 ;                     << " Botón pulsado " << (int) evento.button.button << endl;
85 ;
86 ;             }
87 ;
88 ;         }
89 ;     }
90 ; }
91 ;
92 ;}
;
```

Como puedes ver lo novedoso de este listado se refiere al control de los eventos del ratón. Todo lo demás es común a ejemplos anteriores. En este listado gestionamos los dos tipos de eventos del ratón. Por una parte el del tipo de movimiento y por otra el del tipo de uso de los botones del ratón.

Cuando movemos el ratón se produce un evento `SDL_MOUSEMOTION` y cada vez que se produzca este tipo de evento mostramos en consola la posición actual del ratón. Cuando pulsamos un botón del ratón se produce un evento del tipo `SDL_MOUSEBUTTONDOWN` y cada vez que pulsemos con el ratón en nuestra ventana SDL añadiremos una marca “Botón pulsado x” a la posición del ratón que se muestra por consola. Así sabremos exactamente en qué puntos hemos clickeado con nuestro ratón. La ‘x’ se refiera a que en el mensaje incluiremos el número de botón que se ha pulsado. Recuerda 1 para el izquierdo, 2 para el

6. Captura y Gestión de Eventos

central y 3 para el derecho.

6.9.2. Acceso directo al estado del ratón

La otra alternativa para controlar el ratón es acceder a su estado en un momento dado, tal como pasaba con el teclado. Las funciones que nos permiten controlar este dispositivo de manera directa son dos:

```
Uint8 SDL_GetMouseState(int *x, int *y);  
Uint8 SDL_GetRelativeMouseState(int *x, int *y);
```

Estas funciones nos devuelven por referencia dos parámetros. En la primera función los parámetros toman el valor de la posición actual (x, y) donde se encuentra la posición absoluta del puntero del ratón. La segunda se utiliza cuando queremos conocer el movimiento relativo ya que devuelve en (x, y) los incrementos-decrementos de posición con respecto a la última vez que estuvo el ratón en reposo.

La función también devuelve el estado de los botones en una máscara de bits que puede ser consultada mediante la macro `SDL_BUTTON(button)` donde `button` puede tomar los valores uno, dos o tres dependiendo del botón al que se haga referencia siendo uno el izquierdo, dos el central y tres el derecho. En el caso de que sólo queramos conocer el estado de los botones podemos pasar como parámetros en `x` e `y` el valor `NULL`.

Como ocurría en el manejo del estado del teclado, debemos de llamar a la función `SDL_PumpEvents()` para que se actualice la información disponible acerca del ratón y así obtener información veraz en las funciones que acceden directamente al estado del ratón.

6.9.2.1. Ejemplo 5

```
;  
1 ;// Listado 5 - Eventos  
2 ;//  
3 ;// Listado: main.cpp  
4 ;// Programa de pruebas. Eventos de ratón  
5 ;// Este programa comprueba si se ha realizado un movimiento de ratón  
6 ;// y muestra por consola los movimientos que el ratón va produciendo  
7 ;// Si pulsas un botón del ratón la aplicación termina  
8 ;  
9 ;#include <iostream>  
10 ;#include <SDL/SDL.h>  
11 ;  
12 ;using namespace std;  
13 ;
```

```
14 ;
15 ;int main()
16 ;{
17 ;
18 ;
19 ;    // Iniciamos el subsistema de video
20 ;
21 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
22 ;
23 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
24 ;        exit(1);
25 ;
26 ;
27 ;        atexit(SDL_Quit);
28 ;
29 ;    // Comprobamos que sea compatible el modo de video
30 ;
31 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
32 ;
33 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
34 ;        exit(1);
35 ;
36 ;
37 ;
38 ;    // Establecemos el modo de video
39 ;
40 ;    SDL_Surface *pantalla;
41 ;
42 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
43 ;
44 ;    if(pantalla == NULL) {
45 ;
46 ;        cerr << "No se pudo establecer el modo de video: "
47 ;            << SDL_GetError() << endl;
48 ;
49 ;        exit(1);
50 ;
51 ;
52 ;    // Variables auxiliares
53 ;
54 ;    int x, y;
55 ;
56 ;    Uint8 botones = 0;
57 ;
58 ;    cout << "Pulsa un botón del ratón para salir" << endl;
59 ;
60 ;    for( ; ; ) {
61 ;
62 ;        // Actualiza el estado de los dispositivos
63 ;
64 ;        SDL_PumpEvents();
65 ;
66 ;        // Referencia para no pintar siempre la posición
```

6. Captura y Gestión de Eventos

```
67 ;
68 ;     int x0 = x;
69 ;     int y0 = y;
70 ;
71 ;     // Tomo el estado del dispositivo
72 ;
73 ;     botones = SDL_GetMouseState(&x, &y);
74 ;
75 ;     // Si existen cambios los muestro por consola
76 ;
77 ;     if(x0 != x || y0 != y)
78 ;         cout << "x: " << x << " y: " << y << endl;
79 ;
80 ;
81 ;     // Si pulso un botón salgo
82 ;
83 ;     if(botones != 0)
84 ;         return 0;
85 ;
86 ;
87 ; }
88 ;}
```

La única novedad que presentamos en este listado es la llamada a la función *SDL_GetMouseState()*. Como hemos visto esta función nos devuelve la posición del puntero del ratón en la aplicación, en este caso en las variables *x* e *y*. En la variable *botones* almacenamos si se ha pulsado algún botón. De ser así saldremos de la aplicación.

Cada vez que haya un cambio de posición del ratón en la pantalla mostraremos un mensaje en consola con la nueva posición del mismo.

6.9.3. Creando cursores

Vamos a crear un puntero o cursor personalizado que nos permita saber gráficamente en qué posición de la pantalla está el ratón en un momento dado. Seguramente una de las formas de saber que nuestra aplicación está corriendo y la tenemos en primer plano es que el cursor ha cambiado y se muestra nuestro puntero personalizado.

SDL nos permite cambiar la apariencia del cursor. Para esto nos proporciona la estructura **SDL_Cursor**. Siempre trabajaremos con un puntero a esta estructura sin modificar sus campos internos por lo que el estudio de su composición es algo que no es fundamental para trabajar con ella, no es necesario. Presentamos ahora dicha estructura con una pequeña descripción de cada uno de sus campos :

```

1 ;struct {
2 ;
3 ;    SDL_Rect area; // Rectángulo que define el área del cursor
4 ;    Sint16 hot_x, hot_y; // El punto de acción del cursor
5 ;    Uint8 *data; // Datos de píxel del cursor en blanco y negro
6 ;    Uint8 *mask; // Máscara del cursor en blanco y negro
7 ;    Uint8 *save[2]; // Indica el lugar donde se almacenará el cursor
8 ;    WMcursor *wm_cursor; // Puntero al cursor del gestor de ventanas
9 ;
10;} SDL_Cursor;
;
```

Una vez vista esta estructura pasamos a describir las funciones que nos permiten manejar los cursores. Para crear un cursor SDL nos proporciona siguiente función:

```
SDL_Cursor *SDL_CreateCursor(Uint8 *data, Uint8 *mask, int w, int h,
                             int hot_x, int hoy_y);
```

Como puedes ver esta función devuelve un puntero a `SDL_Cursor`. Recibe varios parámetros de entrada. Los parámetros `data` y `mask` contienen los datos de píxel del cursor, es decir, los datos necesarios para pintar dicho cursor. En los parámetros `w` y `h` indicaremos el ancho y alto del cursor, con la condición que el ancho debe ser múltiplo de ocho. Los dos últimos parámetros son los más importantes en cuanto a la funcionalidad del puntero. En ellos especificamos la parte del cursor que está apuntando realmente, es decir el punto de acción. A veces será el centro, otras será una de las esquinas (como pasa en las flechas), todo dependiendo del tipo de cursor que diseñemos.

Retomando los dos primeros parámetros, `mask` y `data` son las representaciones en blanco y negro de la imagen del cursor y su máscara. Un bit representa un píxel del cursor. `data` y `mask` son dos imágenes monocromáticas que son combinadas para obtener el cursor personalizado.

Lo que ocupa un píxel dentro de un bloque de datos es suficiente para almacenar toda la información necesaria para guardar un cursor. El tamaño del cursor en bytes es igual al ancho dividido entre ocho y multiplicado por la altura del cursor. Así por ejemplo para almacenar un cursor de 32 x 32 necesitamos 32 x 4 bytes. En caso de duda siempre puedes usar esta fórmula:

$$\text{vector size} = h \times w / 8$$

Para saber que efecto produce ciertos datos combinados con cierta máscara observa la siguiente tabla:

Una vez creado el cursor tendremos que establecerlo como activo. Para esto tenemos otra función SDL

6. Captura y Gestión de Eventos

Data	Mask	Resultado del píxel en Pantalla
0	0	Transparente
0	1	Blanco
1	0	Color invertido, si es posible, sino negro
1	1	Negro

Cuadro 6.2: Efecto de date y mask sobre la superficie.

```
void SDL_SetCursor(SDL_Cursor *cursor);
```

Esta función no devuelve valor alguno y recibe el cursor que queremos establecer como apuntador de nuestro ratón. El cursor será establecido inmediatamente. Si en un momento dado tenemos que guardar el cursor que está establecido para reponerlo más adelante SDL proporciona la función:

```
SDL_Cursor *SDL_GetCursor(void);
```

Esta función no recibe ningún parámetro y devuelve el cursor que está establecido actualmente. Una vez hayamos terminado de trabajar con el cursor deberemos de liberar los recursos que consume. Para ello SDL proporciona la función:

```
void SDL_FreeCursor(SDL_Cursor *cursor);
```

Esta función tampoco devuelve ningún valor y recibe el cursor que queremos que sea liberado. Esta función es muy simple, no necesita de más explicación.

Existe una función que nos puede ser útil durante el desarrollo de nuestra aplicación. Esta función proporciona la capacidad de colocar el puntero del ratón en una posición determinada, su prototipo es:

```
void SDL_WarpMouse(Uint16 x, Uint16 y);
```

La función recibe como parámetros la posición (x, y) donde queremos colocar el puntero del dispositivo. Igualmente útil nos puede resultar la función:

```
int SDL_ShowCursor(int toggle);
```

Esta función sirve para activar-desactivar la visualización del cursor en pantalla. Si pasamos como parámetro 0 el cursor no será visualizado, mientras que si pasamos 1 se nos mostrará el puntero del ratón en la pantalla. La función devuelve 1 si el cursor se veía antes de la llamada y 0 si no se visualizaba.

6.9.3.1. Ejemplo 6

Llegados a este punto tenemos muchas cosas que practicar. Vamos a realizar un ejemplo con todas estas nuevas funciones así como vamos a personalizar el cursor de nuestra aplicación. Para calcular la máscara y los datos del cursor vamos a utilizar una función de la documentación de SDL que nos permite automatizar el proceso.

El ejemplo consiste en implementar una aplicación que nos permita limitar el recorrido del ratón por la aplicación y que según sea el botón que pulsemos establezca un cursor nuevo, vuelva al original u oculte el cursor.

Vamos a estudiar los listados:

```

;_____
1 ;// Listado: cursor.h
2 ;//
3 ;// Funciones para personalizar el cursor mediante una imagen XPM
4 ;//
5 ;
6 ;#ifndef _CURSOR_H_
7 ;#define _CURSOR_H_
8 ;
9 ;#include <SDL/SDL.h>
10 ;
11 ;// Tamaño del apuntador o cursor
12 ;
13 ;const int TAMANO = 32;
14 ;
15 ;
16 ;// Pasamos una matriz con una imagen XPM y nos devuelve
17 ;// un cursor que utilizar con SDL
18 ;
19 ;SDL_Cursor *Cursor_personalizado_XPM(const char *matriz[]);
20 ;
21 ;#endif
;_____
```

Este es el fichero de cabecera el módulo que hemos creado para introducir aquellas funciones que creemos que nos van a permitan trabajar con cursores en SDL. En este fichero definimos el tamaño del cursor, en nuestro caso de 32 píxeles. La implementación de la función que declaramos en este fichero la tenemos en el siguiente fichero:

```

;_____
1 ;// Listado: cursor.cpp
2 ;//
3 ;// Implementación
4 ;
5 ;#include <iostream>
6 ;#include "cursor.h"
7 ;_____
```

6. Captura y Gestión de Eventos

```
8 ;
9 ;SDL_Cursor *Cursor_personalizado_XPM(const char *matriz[])
10 ;{
11 ;    // Variables auxiliares
12 ;    int i, fila, col;
13 ;
14 ;    Uint8 data[4 * TAMANO]; // (h * w / 8)
15 ;    Uint8 mask[4 * TAMANO];
16 ;
17 ;    //int hot_x, hot_y;
18 ;
19 ;    i = -1;
20 ;
21 ;    // Recorremos toda la matriz
22 ;
23 ;    for ( fila = 0; fila < TAMANO; ++fila ) {
24 ;
25 ;        for ( col = 0; col < TAMANO; ++col ) {
26 ;
27 ;            // Si no es múltiplo de 8, desplazamos
28 ;
29 ;            if ( col % 8 ) {
30 ;
31 ;                data[i] <<= 1;
32 ;                mask[i] <<= 1;
33 ;
34 ;            } else {
35 ;
36 ;                ++i;
37 ;                data[i] = mask[i] = 0;
38 ;
39 ;            }
40 ;
41 ;            switch(matriz[4 + fila][col]) {
42 ;
43 ;                case 'X':
44 ;                    data[i] |= 0x01;
45 ;                    mask[i] |= 0x01;
46 ;                    break;
47 ;
48 ;                case '.':
49 ;                    mask[i] |= 0x01;
50 ;                    break;
51 ;
52 ;                case ' ':
53 ;                    break;
54 ;                }
55 ;            }
56 ;        }
57 ;
58 ;    return SDL_CreateCursor(data, mask, TAMANO, TAMANO, 15, 15);
59 ;};
```

La función aquí definida recibe una imagen en formato XPM almacenada en una matriz de char. Se utiliza el tipo *char* por que está definido sobre 8 bits que son los necesarios para este tipo de imagen, lo mismo podríamos haber escogido el tipo *Uint8*, cualquiera de estas alternativas es válida.

Las imágenes XPM es un formato de imagen basado en texto ASCII usado en sistemas X-Window. Fue creado a finales de los 80 y tiene una estructura bien definida que veremos en la explicación del fichero principal dónde se define una variable que contiene una imagen de este tipo.

En cuanto a la función parte de la imagen XPM y construye automáticamente los datos de data y mask necesarios para la creación del cursor. Al final de la función hace una llamada a *SDL_CreateCursor()* con los datos calculados y el resultado de ella es lo que devolvemos como nuevo cursor. Vamos a ver el fichero principal y la formación de la imagen XPM.

```

1 ;// Listado 6 - Eventos
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Eventos de ratón. Personalización del cursor
5 ;// Este programa prueba diferentes funciones
6 ;// referentes al manejo del ratón
7 ;
8 ;#include <iostream>
9 ;#include <iomanip>
10 ;
11 ;#include <SDL/SDL.h>
12 ;#include "cursor.h"
13 ;
14 ;
15 ;
16 ;using namespace std;
17 ;
18 ;int main()
19 ;{
20 ;
21 ;    // Iniciamos el subsistema de video
22 ;
23 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
24 ;
25 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
26 ;        exit(1);
27 ;
28 ;    }
29 ;
30 ;
31 ;    atexit(SDL_Quit);
32 ;
33 ;    // Comprobamos que sea compatible el modo de video

```

6. Captura y Gestión de Eventos

```
34 ;
35 ;     if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
36 ;
37 ;         cerr << "Modo no soportado: " << SDL_GetError() << endl;
38 ;         exit(1);
39 ;
40 ;     }
41 ;
42 ;
43 ;     // Establecemos el modo de video
44 ;
45 ;     SDL_Surface *pantalla;
46 ;
47 ;     pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
48 ;
49 ;     if(pantalla == NULL) {
50 ;
51 ;         cerr << "No se pudo establecer el modo de video: "
52 ;             << SDL_GetError() << endl;
53 ;
54 ;         exit(1);
55 ;     }
56 ;
57 ;     // Imagen en XPM para personalizar el cursor
58 ;
59 ;     const char *punto_mira[] = {
60 ;
61 ;         // ancho alto num_colors bytes_per_pixel */
62 ;         " 32 32 3 1",
63 ;
64 ;         // definición de los colores
65 ;         ". c #000000",
66 ;         "X c #ffffff",
67 ;         " c None",
68 ;
69 ;         // pixels
70 ;         " ", //1
71 ;         " ",
72 ;         " ",
73 ;         " ",
74 ;         " ", //5
75 ;         " ",
76 ;         " ",
77 ;         " XXXXXX ",
78 ;         " XXXX X XXXX ",
79 ;         " XXX XXX ", //10
80 ;         " XX XX ",
81 ;         " X X X ",
82 ;         " X X X ",
83 ;         " X X X ",
84 ;         " XX XXXXXXXXXXXXX XX ", //15
85 ;         " X X X ",
86 ;         " X X X ",
```

```
87 ;         " X X X ",  
88 ;         " X X X ",  
89 ;         " XX XX ", //20  
90 ;         " XXX XXX ",  
91 ;         " XXXX X XXXX ",  
92 ;         " XXXXXX ",  
93 ;         " ",  
94 ;         " ", //25  
95 ;         " ",  
96 ;         " ",  
97 ;         " ",  
98 ;         " ",  
99 ;         " ", // 30  
100 ;        " ",  
101 ;        " ", // 32  
102 ;        "0,0"  
103 ;    };  
104 ;  
105 ;    // Rellenamos la pantalla de un color diferente al negro  
106 ;  
107 ;    Uint32 color = SDL_MapRGB(pantalla->format, 25, 100, 155);  
108 ;  
109 ;    // Sólo en el rango que vamos a permitir para el ratón  
110 ;  
111 ;    SDL_Rect delimitador;  
112 ;  
113 ;    delimitador.x = 100;  
114 ;    delimitador.y = 100;  
115 ;    delimitador.w = 440;  
116 ;    delimitador.h = 280;  
117 ;  
118 ;    SDL_FillRect(pantalla, &delimitador, color);  
119 ;  
120 ;    // Actualizamos la pantalla  
121 ;  
122 ;    SDL_Flip(pantalla);  
123 ;  
124 ;    // Guardamos el cursor original  
125 ;  
126 ;    SDL_Cursor *original = SDL_GetCursor();  
127 ;  
128 ;  
129 ;    SDL_Event evento;  
130 ;  
131 ;    // Bucle "infinito"  
132 ;  
133 ;    for( ; ; ) {  
134 ;  
135 ;        while(SDL_PollEvent(&evento)) {  
136 ;  
137 ;            if(evento.type == SDL_KEYDOWN) {  
138 ;  
139 ;                if(evento.key.keysym.sym == SDLK_ESCAPE) {
```

6. Captura y Gestión de Eventos

```
140 ;
141 ;           SDL_FreeCursor(original);
142 ;
143 ;           return 0;
144 ;
145 ;
146 ;
147 ;
148 ;           if(evento.type == SDL_QUIT)
149 ;               return 0;
150 ;
151 ;           // Movemos el ratón
152 ;
153 ;           if(evento.type == SDL_MOUSEMOTION){
154 ;
155 ;               if(evento.motion.x > 540 || evento.motion.x < 100 ||
156 ;                  evento.motion.y > 380 || evento.motion.y < 100 ) {
157 ;
158 ;                   // Si se sale de este rango
159 ;                   // Vuelve dentro de él
160 ;
161 ;                   cout << "Te has salido del rectángulo azul" << endl;
162 ;
163 ;                   SDL_WarpMouse(250, 200);
164 ;
165 ;
166 ;                   cout << "X: " << setw(3) << evento.motion.x
167 ;                     << " - Y: " << setw(3) << evento.motion.y << endl;
168 ;
169 ;
170 ;
171 ;           // Pulsamos un botón del ratón
172 ;
173 ;           if(evento.type == SDL_MOUSEBUTTONDOWN) {
174 ;
175 ;               if(evento.button.button == 1) {
176 ;                   cout << "X: " << setw(3) << evento.button.x
177 ;                     << " - Y: " << setw(3) << evento.button.y
178 ;                     << " Botón izquierdo pulsado, cursor personalizado" <<
179 ;                     endl;
180 ;
181 ;               // Personalizamos el cursor
182 ;
183 ;               SDL_Cursor *apuntador;
184 ;
185 ;               apuntador = Cursor_personalizado_XPM(punto_mira);
186 ;
187 ;               // Lo establecemos
188 ;
189 ;               SDL_SetCursor(apuntador);
190 ;
191 ;               // Nos aseguramos de que se muestra
```

```

192 ;           SDL_ShowCursor(1);
193 ;
194 ;       } else if(evento.button.button == 2) {
195 ;
196 ;           // Mostramos el cursor
197 ;
198 ;           SDL_ShowCursor(1);
199 ;
200 ;           // Reestablecemos el original
201 ;
202 ;           SDL_SetCursor(original);
203 ;
204 ;           cout << "Botón central, vuelta al cursor original" << endl;
205 ;
206 ;       } else {
207 ;
208 ;           // Ocultamos el cursor
209 ;
210 ;           SDL_ShowCursor(0);
211 ;
212 ;           cout << "Botón derecho pulsado, se oculta el cursor" << endl;
213 ;
214 ;       }
215 ;   }
216 ; }
217 ;
218 ; return 0;
219 ;}

```

La primera novedad que encontramos en este listado es la variable que contiene la imagen XPM. El formato de este tipo de imagen no es muy complejo. En la primera línea de la matriz (que debe ser de un tipo de datos de 8 bits) ponemos el ancho, alto, número de colores y los bytes que vamos a dedicar a cada píxel. En una segunda tanda definimos los colores a utilizar y un símbolo que lo represente para que sea más sencillo crear la imagen.

La tercera parte es la imagen propiamente definida. Los espacios en blancos serán transparentes y cada una de los puntos marcados con un símbolo será sustituido por el color correspondiente. De esta manera creamos nuestro nuevo cursor.

Las siguientes novedades las encontramos ya en el código que forma parte del bucle del juego. La primera función que nos encontramos es la de liberar el cursor creado (*SDL_FreeCursor*) en el caso que queramos salir de la aplicación. Cuando el cursor sale de un rango definido volveremos a colocar el puntero en el centro de la pantalla mediante *SDL_WarpMouse()*.

Si pulsamos el botón izquierdo la aplicación cambiará el cursor por el que hemos creado mediante XPM. Si pulsamos el botón central se reestablecerá

6. Captura y Gestión de Eventos

el cursor original mediante *SDL_SetCursor()* y una copia previa que hicimos antes de sustituirlo. Para acabar, si pulsamos el botón derecho, se ocultará el cursor por lo que no lo podremos ver aunque en consola podremos comprobar que se sigue moviendo. Para volver a mostrarlo basta con pulsar uno de los otros dos botones.

Ya sabes todo lo necesario para hacer uso del ratón con SDL. No pierdas la oportunidad y practica todo lo que puedas.

6.10. Dispositivos de Juegos. El Joystick

6.10.1. Introducción

Hay multitud de tipos de dispositivos de juego en el mercado y cada vez más la gama se va ampliando. Lo que está claro es que son el medio de entrada preferido para los más asiduos al videojuego. Los diseños actuales guardan una ergonomía casi perfecta y están especialmente pensados para pasar horas y horas siendo utilizados.

Para este tipo de dispositivo no existe un estándar, debido a esto SDL posee un subsistema completo dedicado a los joysticks y gamepads. Dentro de SDL no es el subsistema más completo por la misma razón de antes, no existe un estándar. Existen numerosos tipos de joystick, desde los típicos gamepads hasta los más avanzados volantes para juegos de carreras, pasando por los añejos controladores de vuelo. En la figura 6.9 tienes un ejemplo de los modelos que estamos comentando.



Figura 6.9: Tipos de joysticks

Los eventos que se generan dependen del joystick en particular con el que se trabaje. Pueden generar hasta cinco tipo de eventos diferentes. Entre ellos se encuentran el movimiento de los ejes, la pulsación de los botones, eventos de trackball... como se podía esperar si nuestro joystick no tiene trackball no generará este tipo de eventos. Los ejes responden dentro de un rango de

6.10. Dispositivos de Juegos. El Joystick

valores bien determinados y los joystick suelen tener, al menos, dos ejes. Uno dedicado movimiento vertical y otro para el horizontal ya que suelen usarse para el control de la posición (o la velocidad) en algún lugar de la pantalla.

Los botones del joystick suelen tener dos posiciones, como los botones del ratón y del teclado, que son pulsado y liberado (en inglés up y down). El hat o minijoystick suele tener nueve posiciones para seleccionar la mejor vista posible y si incluye trackball este suele ser usado para elegir entre diferentes opciones o controlar la velocidad.

Cada uno de estos tipos de eventos tiene su propia estructura. Los principales eventos de los dispositivos de juegos son el movimiento de los ejes y la pulsación de los botones.

6.10.2. Gestión del joystick mediante eventos

Vamos a estudiar como se comunica el subsistema de eventos con el joystick y los dispositivos de juegos. Luego dedicaremos una sección a manejar el joystick accediendo directamente al estado del dispositivo. El movimiento de los ejes del joystick produce un evento SDL del tipo *SDL_JoyAxisEvent* que posee la siguiente estructura:

```
1 ;  
2 ;typedef struct {  
3 ;     Uint8 type;  
4 ;     Uint8 which;  
5 ;     Uint8 axis;  
6 ;     Sint16 value;  
7 ;} SDL_JoyAxisEvent;  
;
```

Pasamos a describir cada uno de los campos que componen esta estructura:

- *type*: Como en todas las demás estructuras utilizadas en los eventos indica el tipo de evento que se acciona. En este caso al mover el eje del joystick se produce un evento del tipo *SDL_JOYAXISMOTION*.
- *which*: Este campo indica que joystick produjo el evento. Es útil cuando tenemos más de un joystick conectado al ordenador y tenemos que diferenciar entre los eventos de los diferentes joysticks.
- *axis*: Nos indica cuál de los ejes del joystick fue el movido. Igual que en el parámetro anterior un joystick puede tener varios ejes por lo que se hace fundamental conocer cuál de los ejes fue accionado para ofrecer una respuesta adecuada.
- *value*: Proporciona el valor del movimiento. Este valor está en un rango que comprende desde -32768 hasta 32767. Tendremos que calcular la

6. Captura y Gestión de Eventos

proporción de este valor que nos interese según el tipo o suavidad de movimiento que queramos establecer. Existen varios tipos de ejes los graduales o analógicos y los digitales o deterministas. En el primer tipo el eje puede tomar cualquier valor en el rango que se define para este parámetro. El tipo digital sólo admite tres valores concretos, el valor 0 o el valor máximo o mínimo del rango, para este tipo de ejes no existen los demás valores intermedios.

Como ocurría con el ratón otros eventos se asocian al mismo dispositivo. En este caso el evento se produce al pulsar o soltar uno de los botones del joystick y viene dado por la estructura *SDL_JoyButtonEvent* que está definida de la siguiente forma:

```
1 ;-----  
2 ;typedef struct {  
3 ;    Uint8 type;  
4 ;    Uint8 which;  
5 ;    Uint8 button;  
6 ;    Uint8 state;  
7 ;} SDL_JoyButtonEvent;  
;
```

Vamos a estudiar cada uno de los campos de esta estructura:

- *type*: Indica el tipo de evento, en este caso puede tomar dos valores que son o bien *SDL_JOYBUTTONDOWN* cuando se presiona el botón o *SDL_JOYBUTTONUP* cuando se suelta dicho botón. Como podrás observar los tipos son análogos a los presentados en el estudio del ratón y el teclado.
- *which*: Este campo indica que joystick produjo el evento. Es fundamental cuando tenemos más de un joystick conectado al ordenador.
- *button*: Nos indica cuál de los botones del dispositivo de juegos fue el pulsado.
- *state*: Este campo posee información que podemos obtener a través de los datos proporcionados por el tipo de evento. Es información redundante que nos ahorra realizar cálculos para obtener este estado. En este caso los valores que puede tomar el campo son *SDL_PRESSED* para cuando el botón del joystick está pulsado o bien *SDL_RELEASED* cuando dicho botón se libera.

6.10.2.1. Ejemplo 7

Ya tenemos una cantidad suficiente de información para generar un ejemplo. El joystick es un tipo de dispositivo de los que hay que abrir antes de utilizarlo, cosa que no es común en SDL. Vamos a implementar un pequeño

6.10. Dispositivos de Juegos. El Joystick

programa que nos permita trabajar con eventos generados por el joystick. Nada complicado.

```
1 ;// Ejemplo 7
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Eventos de dispositivo de juegos
5 ;
6 ;
7 ;#include <iostream>
8 ;#include <SDL/SDL.h>
9 ;
10 ;using namespace std;
11 ;
12 ;int main()
13 ;{
14 ;
15 ;    // Iniciamos el subsistema de video
16 ;
17 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK) < 0) {
18 ;
19 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
20 ;        exit(1);
21 ;    }
22 ;
23 ;    atexit(SDL_Quit);
24 ;
25 ;    // Comprobamos que sea compatible el modo de video
26 ;
27 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
28 ;
29 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
30 ;        exit(1);
31 ;
32 ;    }
33 ;
34 ;
35 ;    // Establecemos el modo de video
36 ;
37 ;    SDL_Surface *pantalla;
38 ;
39 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
40 ;
41 ;    if(pantalla == NULL) {
42 ;
43 ;        cerr << "No se pudo establecer el modo de video: "
44 ;            << SDL_GetError() << endl;
45 ;
46 ;        exit(1);
47 ;    }
48 ;
49 ;    cout << "\n\nPulse ESC para salir." << endl;
```

6. Captura y Gestión de Eventos

```
50 ;
51 ;     SDL_Joystick *joy;
52 ;
53 ;     // Abrimos el joystick
54 ;
55 ;     joy = SDL_JoystickOpen(0);
56 ;
57 ;     // Bucle infinito
58 ;     // Gestionamos los eventos
59 ;
60 ;     SDL_Event evento;
61 ;
62 ;     for( ; ; ) {
63 ;
64 ;         while(SDL_PollEvent(&evento)) {
65 ;
66 ;             if(evento.type == SDL_KEYDOWN) {
67 ;
68 ;                 if(evento.key.keysym.sym == SDLK_ESCAPE)
69 ;                     return 0;
70 ;
71 ;             }
72 ;
73 ;             if(evento.type == SDL_QUIT)
74 ;                 return 0;
75 ;
76 ;             if(evento.type == SDL_JOYAXISMOTION)
77 ;                 cout << "Eje : " << (int) evento.jaxis.axis
78 ;                     << " -> Valor: " << evento.jaxis.value << endl;
79 ;
80 ;             if(evento.type == SDL_JOYBUTTONDOWN)
81 ;                 if(evento.jbutton.type == SDL_JOYBUTTONDOWN)
82 ;                     cout << "Boton: " << (int) evento.jbutton.button << endl;
83 ;
84 ;         }
85 ;
86 ;     }
87 ;
88 ;}
```

Vemos algunas cosas novedosas. Hemos tenido que inicializar un subsistema especial específico del joystick pasándole como parámetro `SDL_INIT_JOYSTICK` a la función `SDL_Init()`. Tendremos que hacer esto cada vez que queramos utilizar el joystick sea cual sea el método que vayamos a utilizar.

La siguiente novedad es la creación de una variable de tipo `SDL_Joystick` para almacenar el resultado de la función `SDL_JoystickOpen()` que se encarga, como veremos en unas líneas, de abrir el dispositivo poniéndolo a nuestra disposición para poder utilizarlo.

Con respecto al resto del listado ninguna otra novedad más que la propia gestión de los eventos del joystick con sus peculiaridades y sus significados. Normalmente no se gestiona el uso del joystick mediante eventos y se accede directamente al estado de sus componentes para dar una determinada respuesta.

6.10.3. Otros eventos del joystick

Existen dos tipos de eventos más que son considerados menos importantes que los estudiados hasta ahora. Hacen referencia al hat o minijoystick del joystick y al trackball del joystick. Precisamente por ser elementos menos comunes en los dispositivos de juego se relega estos eventos a una especie de segunda categoría.

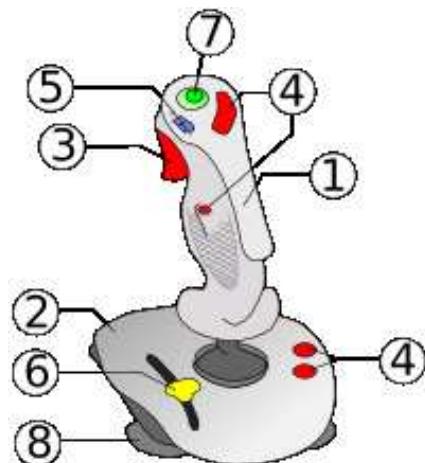


Figura 6.10: Elementos de un Joystick

El hat o minijoystick de un dispositivo de juego se utiliza para establecer la vista del jugador. En la figura 6.10 se trataría del elemento número 7. Si estamos manejando un simulador de vuelo este hat nos permitirá mirar a ambos lados de la posición de la cabina. Normalmente, como puedes ver, va colocado en lo alto del joystick para tener un fácil acceso con el dedo pulgar. Las posiciones de este complemento son nueve que puedes ver en la figura 6.11. Cuando cambia la posición del hat se produce un evento.

La estructura que soporta los eventos de este tipo es:

```

1 ;typedef struct {
2     Uint8 type;
3     Uint8 which;
4     Uint8 hat;
5     Uint8 value;
;
```

6. Captura y Gestión de Eventos

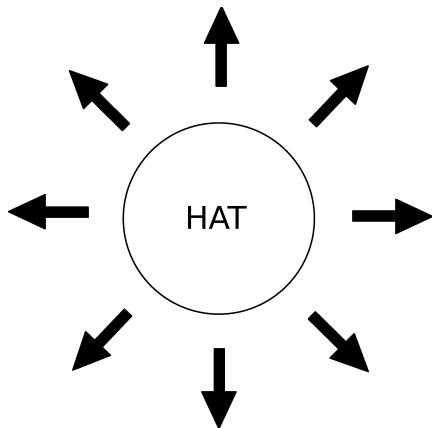


Figura 6.11: Posiciones del hat

```
6 ;} SDL_JoyHatEvent;
```

El campo *type* y *which* tienen el significado habitual. El primero indica el tipo de evento y el segundo nos permite distinguir en qué joystick se produjo el evento. Pueden existir varios hats dentro de un joystick por lo que el campo *hat* nos permite distinguir cuál de ellos produjo el evento. Por último *value* nos indica el valor tomado por el hat. Este minijoystick puede tomar 5 valores “puros”

- **SDL_HAT_CENTERED:** Hat en posición neutra.
- **SDL_HAT_UP:** En posición arriba.
- **SDL_HAT_RIGHT:** A la derecha.
- **SDL_HAT_LEFT:** Indica que el hat está situado a la izquierda.
- **SDL_HAT_DOWN:** Indica que el hat está situado abajo.

Los otros cuatro valores posibles son combinaciones de estos cinco.

- **SDL_HAT_RIGHUP:** Indica que el hat está arriba a la derecha. Equivale a `SDL_HAT_RIGHT | SDL_HAT_UP`
- **SDL_HAT_RIGHTDOWN:** Indica que el hat está abajo a la derecha. Equivale a `SDL_HAT_RIGHT | SDL_HAT_DOWN`
- **SDL_HAT_LEFTUP:** Indica que el hat está arriba a la izquierda. Equivale a `SDL_HAT_LEFT | SDL_HAT_UP`
- **SDL_HAT_LEFTDOWN:** Indica que el hat está situado a la izquierda. Equivale a `SDL_HAT_LEFT | SDL_HAT_DOWN`

6.10. Dispositivos de Juegos. El Joystick

En cuanto al trackball del joystick decir que es el elemento menos común dentro de estos dispositivos de juego. El tipo de eventos que genera son parecidos a los de un ratón donde todos los movimientos son relativos ya que un trackball no puede saltar de una posición a otra sin partir de la anterior. Este tipo de evento sólo es válido para trackballs que no estén integrados en un ratón o en un teclado.

Este tipo de eventos está soportado por una estructura llamada *SDL_JoyBallEvent* que pasamos a definir:

```
1 ;typedef struct{
2 ;    Uint8 type;
3 ;    Uint8 which;
4 ;    Uint8 ball;
5 ;    Sint16 xrel, yrel;
6 ;} SDL_JoyBallEvent;
```

Los elementos de la estructura ya te deben de ser conocidos. Como es habitual *type* indica el tipo de evento que se produce, en este caso *SDL_JOYBALLMOTION*. El miembro *which* nos identifica de nuevo en que joystick se está produciendo el evento. El campo *ball* nos distingue que trackball es el que produce el evento dentro del joystick, ya que SDL supone que un joystick puede tener más de un trackball, y para terminar *xrel* y *yrel* que indican el movimiento relativo del “puntero” que domine el trackball en cuestión.

6.10.4. Acceso directo al estado del Joystick

Acabamos de estudiar como consultar las acciones del joystick por medio de eventos. Esta manera de actuar es extensible a todos los dispositivos de juegos. Como vimos con el teclado, se puede consultar el estado del joystick de forma directa. Además de conocer el estado del joystick podemos conocer otras informaciones como las características del mismo, como por ejemplo, el número de botones o el número de joysticks conectados. La tarea principal que debemos controlar son los ejes del joystick así como la pulsación de sus botones.

Los ejes del joystick producen el efecto de movimiento. Algunos joysticks incorporan más de dos ejes incluyendo uno para la velocidad o para otros objetivos. El efecto que producen los dos ejes principales del joystick puedes verlos en la figura 6.12.

El eje de un joystick es un dispositivo de entrada que seguramente habrás utilizado sin parar a pensar en como funciona. Cuando mueves la palanca de un joystick hacia arriba y abajo, o bien, delante-atrás se produce el movimiento sobre el eje vertical. Si lo movemos de forma lateral lo haremos sobre el eje horizontal.

6. Captura y Gestión de Eventos

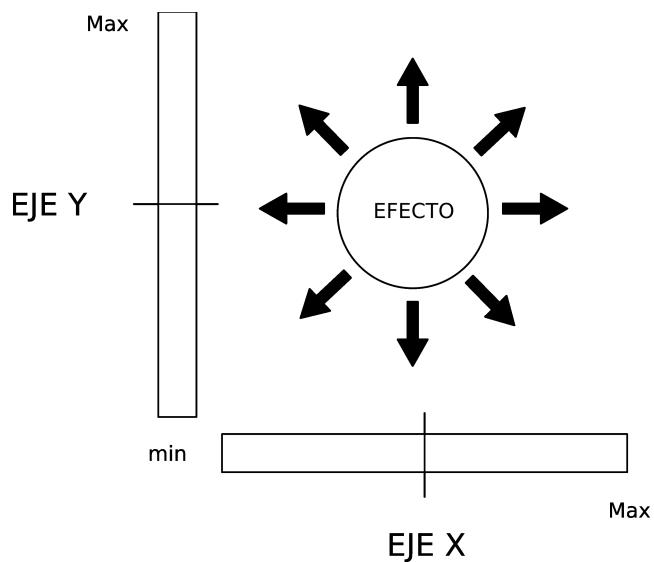


Figura 6.12: Ejes del joystick

La única estructura que nos proporciona información sobre el joystick en SDL es la estructura *SDL_Joystick* cuya definición está oculta para el programador, por lo cual si queremos obtener información acerca del joystick tendremos que hacerlo a través de las funciones que nos proporciona SDL.

Ya sabemos que para manejar las acciones del usuario sobre el joystick desde SDL deberemos inicializarlos con la bandera *SDL_INIT_JOYSTICK* en la función *SDL_Init()*. Además tendremos que abrir aquel dispositivo de juego que queramos utilizar, pero... ¿cómo saber que dispositivo abrir?

Hay tres tipos de funciones en SDL para el manejo de los dispositivos de juego. El primer tipo nos permite cerrar y abrir el uso de un dispositivo en particular para poder trabajar con él en el sistema. Otro tipo es el que nos permite examinar varios tipos de información que podemos obtener del joystick, como cuantos ejes, botones y demás dispositivos posee.

El último tipo de funciones es la que nos permite comprobar el estado de cada uno de los componentes del joystick. Como decíamos al principio este método de acceder al joystick es muy parecido al acceso directo al dispositivo que estudiamos para el ratón y el teclado.

Antes de empezar a usar joystick en nuestra aplicación hay que responder a unas cuantas cuestiones. ¿Cuántos joysticks hay conectados al sistema? ¿Cuál es el nombre de estos dispositivos? ¿Cuántos ejes tienen cada uno de esos joysticks? ¿Cuántos botones? ¿Cuántos hats y cuántos trackballs?

6.10.5. Información sobre el Dispositivo de Juego

Para saber el número de joysticks que tenemos conectados al ordenador SDL proporciona una función que nos ofrece esta información. El prototipo es el siguiente:

```
int SDL_NumJoysticks(void);
```

Como puedes observar no recibe ningún parámetro y devuelve un entero que es precisamente la información que demandamos, el número de dispositivos de juego conectados. Ya tenemos respuesta a una de nuestras preguntas.

Puede ser necesario distinguir varios joysticks conectados. Podemos darle la opción al usuario que decida con cual quiere jugar, o bien si estamos desarrollando un juego para varios jugadores que cada uno elija si es el jugador 1 o 2. Para esto puede ser interesante la siguiente función:

```
const char *SDL_JoystickName(int index);
```

Esta función recibe como parámetro el número de joystick que queremos consultar mientras que devuelve una cadena de caracteres con el nombre del joystick o en su defecto, el nombre de su driver. El rango de números de joystick comienza en 0 y llega hasta el valor devuelto por *SDL_NumJoysticks()* menos 1. Es más interesante mostrar al usuario el nombre del joystick que el número asignado por SDL porque así le será más fácil reconocer que mando tiene “entre manos”.

Para poder utilizar los joysticks mediante este subsistema lo primero que debemos hacer es “abrirlo”. El concepto de abrir el joystick no es coger un destornillador y mirar los circuitos, sino preparar el dispositivo para tener acceso a él para poder consultar las acciones que se van produciendo en los distintos joysticks. El prototipo de la función que nos permite realizar esta acción es:

```
SDL_Joystick *SDL_JoystickOpen(int index);
```

Como ocurría en la función antes expuesta recibe como parámetro el número de joystick que queremos abrir. La función devuelve un puntero a *SDL_Joystick* que es el tipo de estructura que deberemos utilizar para manejar las funciones de este sistema. Recuerda que la definición de esta estructura está oculta al programador-usuario y que debemos de recurrir a estas funciones para obtener la información contenida en ella. La implementación de esta estructura no es importante, lo importante es saber que necesitamos el puntero que devuelve esta función para trabajar correctamente los dispositivos de juegos.

6. Captura y Gestión de Eventos

Como puedes ver para la función que consulta el nombre del dispositivo no hacía falta tener abierto el joystick. Esto es porque perdería toda su utilidad. La función está pensada para ser utilizada en un paso previo al de la inicialización con el objetivo que el usuario pueda elegir entre las opciones existentes abriendo el que más le convenga.

Como es habitual al existir una función que abre un dispositivo debe de existir una función que la cierre, ya que es necesario que lo cerremos antes de terminar la aplicación. El prototipo de esta función es:

```
void SDL_JoystickClose(SDL_Joystick *joystick);
```

Esta función recibe como parámetro el joystick al que, previamente abierto, queramos cerrar el acceso. Si tenemos varios joysticks conectados puede ser útil saber qué joysticks tenemos abiertos. Con este objetivo proporciona SDL la siguiente función:

```
int SDL_JoystickOpened(int index);
```

Como en muchas de las funciones que manejan los dispositivos de juegos *index* la función recibe el número de joystick que queremos consultar para saber si está abierto o no. Esta función devuelve 1 si el joystick en cuestión está abierto y 0 si está cerrado.

Es fundamental conocer las características del joystick en cuestión para poder ofrecer una funcionalidad acorde con el modelo que se esté utilizando. Por esto SDL ofrece dos funciones que nos proporcionan una información vital para este fin. La primera de ellas es:

```
int SDL_JoystickNumAxes(SDL_Joystick *joystick);
```

Esta función nos permite conocer el número de ejes que posee nuestro joystick. Los mínimo es dos aunque cada día es más común encontrar joysticks con aceleradores o frenos incorporados, así como dispositivos de juegos en forma de volante. La función recibe como parámetro el puntero del joystick que devolvía la función *SDL_JoystickOpen()* y devuelve un número entero que es el número de ejes de los que disponemos.

Sigamos con funciones que nos permiten conocer mejor el dispositivo que vamos a manejar. La siguiente función es:

```
int SDL_JoystickNumButtons(SDL_Joystick *joystick);
```

Esta función nos permite conocer el número de botones que posee el joystick. La variedad del número de botones que tienen los dispositivos de juegos que hay en el mercado es mayor que el del número de ejes disponible. Esta función recibe también como parámetro el puntero del joystick previamente abierto y, como antes, devuelve el número de botones que posee el dispositivo.

6.10. Dispositivos de Juegos. El Joystick

6.10.5.1. Ejemplo 8

Para ver como utilizar estas funciones vamos a estudiar el siguiente ejemplo:

```
1 ;// Ejemplo 8
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Información acerca del joystick
5 ;// Abriendo un dispositivo
6 ;
7 ;
8 ;#include <iostream>
9 ;#include <SDL/SDL.h>
10 ;
11 ;using namespace std;
12 ;
13 ;int main()
14 ;{
15 ;    // Iniciamos el subsistema de video
16 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK) < 0) {
17 ;
18 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
19 ;        exit(1);
20 ;    }
21 ;
22 ;    atexit(SDL_Quit);
23 ;
24 ;    // Comprobamos que sea compatible el modo de video
25 ;
26 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
27 ;
28 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
29 ;        exit(1);
30 ;
31 ;    }
32 ;
33 ;
34 ;    // Consultamos el número de joysticks existentes
35 ;
36 ;    int num_joysticks = SDL_NumJoysticks();
37 ;
38 ;    // Si no hay joysticks conectados
39 ;
40 ;    if(num_joysticks < 1) {
41 ;
42 ;        cerr << "Conecte un dispositivo de juego antes de ejecutar" << endl;
43 ;        exit(1);
44 ;
45 ;    } else {
46 ;
47 ;        // Al menos hay uno
48 ;
49 ;        cout << "Hay conectados " << num_joysticks
50 ;            << " joysticks. " << endl;
```

6. Captura y Gestión de Eventos

```
51 ;    }
52 ;
53 ;    // Mostramos información de cada uno de los joysticks
54 ;
55 ;    for(int i = 0; i < num_joysticks; i++) {
56 ;
57 ;        // Obtenemos el nombre del joystick
58 ;
59 ;        const char *nombre = SDL_JoystickName(i);
60 ;
61 ;        // Lo abrimos
62 ;
63 ;        SDL_Joystick *joy = SDL_JoystickOpen(i);
64 ;
65 ;        // Mostramos en pantalla toda la información
66 ;        // disponible del joystick
67 ;
68 ;        cout << "Abrimos el joystick " << nombre
69 ;                << " - " << i << endl;
70 ;        cout << "\tTiene " << SDL_JoystickNumAxes(joy)
71 ;                << " ejes" << endl;
72 ;        cout << "\tTiene " << SDL_JoystickNumButtons(joy)
73 ;                << " botones" << endl;
74 ;
75 ;        // Cerramos el joystick
76 ;
77 ;        SDL_JoystickClose(joy);
78 ;    }
79 ;
80 ;    return 0;
81 ;}
```

Como puedes ver en el listado nos hemos limitado a sacar toda la información disponible de los dispositivos de juegos instalados en nuestro sistema. Este es el objetivo de este listado, ofrecerte un ejemplo de como utilizar cada una de las funciones presentadas en este apartado.

En la siguiente sección vamos a entrar en materia y utilizar el joystick para lo mismo que utilizamos en apartados anteriores el ratón o el teclado.

6.10.6. Obteniendo el información directa de los dispositivos de juegos

Para usar los eventos producidos por el joystick necesitamos habilitarlos. La función que nos permite realizar esta acción en SDL es:

```
int SDL_JoystickEventState(int state);
```

Para el parámetro de entrada *state* existen tres posibles valores. Si pasamos **SDL_QUERY** obtenemos el estado actual del joystick que pueden

6.10. Dispositivos de Juegos. El Joystick

ser uno de estos dos: activado o desactivado. Con `SDL_ENABLE` activamos la lectura de eventos del joystick y con `SDL_IGNORE` desactivaremos la generación de los eventos producidos por el dispositivo. Si pasamos uno de los dos últimos parámetros la función nos devolverá el nuevo estado al que pasa la gestión de eventos del joystick. Por omisión estos eventos están activados.

Cuando el polling de eventos del joystick está activado podemos leer dichos eventos con las funciones `SDL_PollEvent` o `SDL_WaitEvent` automáticamente. Si decidimos no utilizar los eventos del joystick deberemos de, antes de consultar el estado del joystick, actualizar la información que poseemos de él. Para esto SDL nos proporciona una función cuyo cometido es actualizar todos los dispositivos de juegos que tengamos abiertos. El prototipo de dicha función es:

```
void SDL_JoystickUpdate(void);
```

Antes de cada lectura de estado del dispositivo hay que realizar una llamada a esta función, esto debe de quedar muy claro, es un equivalente a la función `SDL_PumpEvents` que utilizábamos para acceder directamente al teclado o al ratón pero para joystick. La función no recibe ni devuelve parámetros. Esta función es llamada automáticamente cuando tenemos activado la generación de eventos del joystick, ya que es la manera que tiene SDL de obtener información de estos dispositivos.

Después de actualizar la información disponible del joystick podemos leer directamente los valores de los diferentes componentes que forman el joystick. Para conocer el estado de los ejes del joystick utilizamos la siguiente función:

```
Sint16 SDL_JoystickGetAxis(SDL_Joystick *joystick, int axis);
```

Observamos que como parámetros recibe un puntero al joystick y el eje que queremos consultar. La función devuelve un entero con signo cuyo valor estará entre -32768 y 32768. Nos tocará a nosotros decidir que significa cada valor dentro del rango de estos valores para proporcionar una respuesta adecuada para nuestro videojuego. Existen ejes progresivos que pueden tomar todos los valores del rango y digitales que sólo toman valores discretos, normalmente el 0, los máximos y los mínimos.

Como esta función existe otra análoga pero para la consulta del estado de los botones. La principal, como no podía ser de otra forma, es la diferencia es el significado del dato que devuelve. Observemos el prototipo:

```
Uint8 SDL_JoystickGetButton(SDL_Joystick *joystick, int button);
```

Como parámetro le indicamos el joystick de que queremos comprobar el estado y especificamos que botón queremos consultar. Esta función devuelve 1 si el botón está pulsado y 0 en caso de no estarlo.

6. Captura y Gestión de Eventos

6.10.6.1. Ejemplo 9

Vamos a utilizar todas estas funciones en un ejemplo para que puedas comprobar su utilidad. En esta aplicación manejaremos el joystick número 0 para manejar a nuestro personaje a través de la pantalla. Accedemos directamente al estado del joystick para obtener la información del mismo.

```
;_____
1 ;// Ejemplo 9
2 ;//
3 ;// Listado: eventos_10.
4 ;// Programa de pruebas. Joystick captura de movimientos
5 ;// Consultando el estado del joystick directamente
6 ;
7 ;
8 ;#include <iostream>
9 ;#include <SDL/SDL.h>
10 ;
11 ;using namespace std;
12 ;
13 ;int main()
14 ;{
15 ;
16 ;    // Iniciamos el subsistema de video
17 ;
18 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK) < 0) {
19 ;
20 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
21 ;        exit(1);
22 ;
23 ;    }
24 ;
25 ;    atexit(SDL_Quit);
26 ;
27 ;
28 ;    // Comprobamos que sea compatible el modo de video
29 ;
30 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
31 ;
32 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
33 ;        exit(1);
34 ;
35 ;    }
36 ;
37 ;    // Establecemos el modo de video
38 ;
39 ;    SDL_Surface *pantalla;
40 ;
41 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
42 ;
43 ;    if(pantalla == NULL) {
44 ;
45 ;        cerr << "No se pudo establecer el modo de video: "
```

6.10. Dispositivos de Juegos. El Joystick

```
46 ;           << SDL_GetError() << endl;
47 ;
48 ;       exit(1);
49 ;
50 ;
51 ;   // Cargamos una imagen en una superficie
52 ;
53 ;   SDL_Surface *personaje = SDL_LoadBMP("Imagenes/personaje.bmp");
54 ;
55 ;   if(personaje == NULL) {
56 ;
57 ;       cerr << "No se pudo cargar la imagen: "
58 ;           << SDL_GetError() << endl;
59 ;
60 ;       exit(1);
61 ;
62 ;   // Establecemos el color de la transparencia
63 ;   // No será mostrado al realizar el blitting
64 ;
65 ;   SDL_SetColorKey(personaje, SDL_SRCCOLORKEY|SDL_RLEACCEL,\n
66 ;                   SDL_MapRGB(personaje->format, 0, 255, 0));
67 ;
68 ;   // Posición inicial del personaje
69 ;
70 ;   SDL_Rect posicion;
71 ;
72 ;   posicion.x = 300;
73 ;   posicion.y = 220;
74 ;   posicion.w = personaje->w;
75 ;   posicion.h = personaje->h;
76 ;
77 ;
78 ;   // Copiamos la imagen en la superficie principal
79 ;
80 ;   SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
81 ;
82 ;   // Mostramos la pantalla "oculta" del buffer
83 ;
84 ;   SDL_Flip(pantalla);
85 ;
86 ;
87 ;   // Si hay un joystick conectado lo abrimos
88 ;
89 ;   SDL_Joystick *joy;
90 ;
91 ;   if(SDL_NumJoysticks() > 0) {
92 ;
93 ;       joy = SDL_JoystickOpen(0);
94 ;       cout << "\nAbrimos el joystick " << SDL_JoystickName(0)
95 ;           << " para la prueba. " << endl;
96 ;
97 ;   } else {
98 ;
```

6. Captura y Gestión de Eventos

```
99 ;         cout << "Para llevar acabo esta prueba debe haber un joystick "
100 ;                 << "conectado. Si es así compruebe su configuración" << endl;
101 ;
102 ;         exit(1);
103 ;
104 ;     }
105 ;
106 ;     // Mostramos información del dispositivo
107 ;
108 ;     int num_ejes = SDL_JoystickNumAxes(joy);
109 ;     int num_botones = SDL_JoystickNumButtons(joy);
110 ;
111 ;     cout << "Este joystick tiene " << num_ejes << " ejes y "
112 ;             << num_botones << " botones." << endl;
113 ;
114 ;     cout << "\nPulse ESC para salir.\n" << endl;
115 ;
116 ;     // Variable auxiliar
117 ;
118 ;     SDL_Event evento;
119 ;
120 ;     for( ; ; ) {
121 ;
122 ;         // Actualizamos el estado del joystick
123 ;
124 ;         SDL_JoystickUpdate();
125 ;
126 ;         // Recorremos todos los ejes en búsqueda de cambios de estado
127 ;
128 ;         for(int i = 0; i < num_ejes; i++) {
129 ;
130 ;             int valor_eje = SDL_JoystickGetAxis(joy, i);
131 ;
132 ;             //cout << "Eje " << i << " -> " << valor_eje << endl;
133 ;
134 ;             if(valor_eje != 0) {
135 ;
136 ;                 if(i == 0) {
137 ;
138 ;                     if(valor_eje > 0)
139 ;                         posicion.x++;
140 ;
141 ;                     if(valor_eje < 0)
142 ;                         posicion.x--;
143 ;
144 ;                 } else {
145 ;
146 ;                     if(valor_eje > 0)
147 ;                         posicion.y++;
148 ;
149 ;                     if(valor_eje < 0)
150 ;                         posicion.y--;
151 ;                 }
152 ;             }
153 ;         }
154 ;     }
155 ; }
```

6.10. Dispositivos de Juegos. El Joystick

```
152 ;        }
153 ;    }
154 ;
155 ;    // Limpiamos la pantalla
156 ;
157 ;    SDL_FillRect(pantalla, NULL, 0);
158 ;
159 ;    // Cambiamos la posición del personaje
160 ;
161 ;    SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
162 ;
163 ;    // Actualizamos la pantalla principal
164 ;
165 ;    SDL_Flip(pantalla);
166 ;
167 ;
168 ;    // Recorremos todos los botones en búsqueda acciones
169 ;
170 ;    for(int i = 0; i < num_botones; i++) {
171 ;
172 ;        int pulsado = SDL_JoystickGetButton(joy, i);
173 ;
174 ;        if(pulsado) {
175 ;
176 ;            cout << "Ha pulsado el botón " << i << endl;
177 ;
178 ;        }
179 ;    }
180 ;
181 ;    // Bucle que controla eventos de salida
182 ;
183 ;    while(SDL_PollEvent(&evento)) {
184 ;
185 ;        if(evento.type == SDL_KEYDOWN) {
186 ;
187 ;            if(evento.key.keysym.sym == SDLK_ESCAPE) {
188 ;
189 ;                SDL_JoystickClose(joy);
190 ;                return 0;
191 ;
192 ;            }
193 ;
194 ;            if(evento.type == SDL_QUIT) {
195 ;
196 ;                SDL_JoystickClose(joy);
197 ;                return 0;
198 ;
199 ;            }
200 ;        }
201 ;    }
202 ;}
```

6. Captura y Gestión de Eventos

Como puedes ver el ejemplo sigue la estructura habitual que estamos siguiendo en el tutorial. Hasta la carga del personaje en una superficie y la puesta en marcha del *game loop* del juego no hay nada novedoso que no se haya visto en algún otro ejemplo. Hasta este punto hemos inicializado SDL, establecido el modo de video, cargado una imagen en una superficie, abierto el dispositivo de juego identificado como 0 y volcado la imagen de nuestro personaje principal en la superficie que será mostrada en pantalla. Además hemos mostrado información acerca del joystick que vamos a utilizar.

Vamos a explicar lo novedoso del listado. Una vez en el bucle lo primero que hacemos es actualizar la información del joystick. Seguidamente buscamos en todos sus ejes cambios de estado. ¿Cómo lo hacemos?. El estado de reposo de los ejes del joystick se indica con que dicho eje esté a 0. En el caso de existir cualquier variación (osea cualquier valor distinto de 0) actuaremos cambiando la lógica del juego moviendo hacia un lado u otro la posición del personaje principal. Así de sencillo.

Seguidamente refrescaremos la pantalla dibujando a nuestro personaje en su nueva posición consiguiendo así un efecto de movimiento a través del joystick. Seguidamente tenemos un bucle que consulta los botones del dispositivo de juego y si existe algún botón presionado lo muestra por pantalla.

Para terminar tenemos un polling sobre la cola de eventos por si se presiona alguna tecla de salida o recibe alguna petición de este estilo el programa liberando los recursos y dando por terminado el programa.

6.10.7. Otras funciones de estado del joystick

Para los minijsicks o los hats de punto de vista la función que nos permite acceder a ellos es:

```
Uint8 SDL_JoystickGetHat(SDL_Joystick *joystick, int hat);
```

Como ocurre en las demás funciones de manejo de partes del dispositivo de juego debemos indicar que hat de que joystick queremos conocer el valor mediante los parámetros de entrada. Esta función devuelve la posición actual del hat. Como cuando consultábamos este tipo de dispositivo mediante eventos el valor devuelto es una combinación de bits de bandera exactamente como la expuesta en dicho apartado anterior para este mismo fin.

Para terminar podemos querer tener información sobre cuánto se ha movido el trackball del joystick desde la última actualización. Esto lo podemos conseguir con la función:

```
int SDL_JoystickGetBall(SDL_Joystick *joystick, int ball, int *dx, int *dy);
```

Los dos primeros parámetros de la función son parámetros de entrada. Tenemos que indicar que “ball” de que joystick queremos consultar. Los dos últimos parámetros son de salida. En estos se almacena el incremento o decremento de posición respecto a x e y que ha sufrido el puntero del trackball desde la última actualización, es decir la posición relativa del trackball. Si al consultar estos valores se produce algún error la función devolverá -1. Devolverá 0 en el caso de que todo haya ido bien.

No consideramos interesante proponer ningún ejercicio sobre el manejo de estas funciones ya que actúan exactamente igual que las demás vistas en SDL. Una vez se aprenda a manejar ciertos eventos y funciones para conocer el estado de un dispositivo en SDL todas las demás se manipulan de una forma semejante. Esto es un punto a favor sobre la unicidad que han tenido los desarrolladores al plantear el manejo de los distintos dispositivos facilitando al programador su uso.

6.11. Eventos del Sistema

6.11.1. Introducción

Como ya has podido comprobar existen multitud de eventos. Muy importantes y a tener en cuenta son los que se generan por sucesos en el sistema operativo. Por ejemplo, un evento muy común, es el de cerrar una ventana que no queremos utilizar más. Cuando se produce esta acción el sistema genera un evento. La aplicación debería estar preparada para recibir dicho evento y así cerrarla de una forma ordenada.

El cambio de tamaño de la ventana es muy común en sistemas que dispongan de gestor de ventanas, ya que cada usuario adapta el escritorio a sus necesidades. Si permitimos este redimensionamiento tenemos que estar preparados para establecer el nuevo modo de video en el videojuego, adaptándolo a nuestras necesidades. Del mismo modo si colocamos una ventana encima de la ventana de nuestra aplicación deberemos de estar preparados para redibujar la aplicación en pantalla cuando sea necesario.

Este tipo de eventos, como habrás podido observar, no son entradas directas del usuario si no que se activan al querer realizar diferentes tareas. La mayoría de estos eventos no ofrecen información adicional más que la de que se ha realizado el propio evento.

6. Captura y Gestión de Eventos

6.11.2. Evento *Quit*

Uno de los eventos mas importantes sucede cuando el sistema operativo quiere cerrar la aplicación, ya sea porque así lo desea el usuario o porque el sistema decide terminar con su ejecución. El tipo de este evento es *SDL_QuitEvent*:

```
1 ;  
2 ;typedef struct {  
3 ;    Uint8 type  
4 ;} SDL_QuitEvent;  
5 ;
```

El campo *type* de esta estructura toma el valor *SDL_QUIT*, que cuando se recibe este evento hay que realizar las operaciones necesarias para terminar con la ejecución del programa como guardar datos, liberar la memoria reservada... todo para una correcta terminación. Este tipo de eventos no almacena información adicional.

Hemos hecho uso de este evento en todos los ejemplos que llevamos realizados hasta el momento. Si es necesario revisa cualquier listado y podrás observar que está ahí. Es fundamental que tu aplicación incorpore la gestión de este evento para un cierre ordenado. No es una buena idea que el usuario haga clic en la famosa X que ofrecen los gestores de ventana en sus marcos para terminar con la aplicación y que no realice dicha acción.

En un entorno de ventanas el evento de salida o *Quit* ocurre cuando el usuario decide cerrar la ventana. En modo a pantalla completa deberemos de proveer al usuario de otro método de cerrar la ventana como pulsar escape o alguna tecla que configuremos a tal efecto. Hasta el momento hemos cumplido en los ejemplos incluyendo todo el código necesario para realizar estas dos acciones. ¡No seas tú menos!

6.11.3. Evento *Video Expose*

Cuando el gestor de ventanas del sistema operativo realiza variaciones en las ventanas nuestra aplicación debe ser redibujada. Estas variaciones puedes venir dadas por el propio sistema operativo o porque el usuario ha decidido, por ejemplo, mover las ventanas. Para ello se lanza un evento *SDL_ExposeEvent* que nos permite controlar cuando deberemos ejecutar dicha acción. *SDL_ExposeEvent* tiene la siguiente estructura:

```
1 ;  
2 ;typedef struct{  
3 ;    Uint8 type;  
4 ;} SDL_ExposeEvent;  
5 ;
```

El campo *type* de la estructura tendrá el valor `SDL_VIDEOEXPOSE` que es el correspondiente al evento comentado. A parte del propio evento esa estructura tampoco aporta información adicional. Su verdadero valor es saber que la aplicación debe de ser redibujada.

Para manejar este evento basta con incluir en el bucle donde tratamos el polling de los eventos un caso que trate este tipo de mensaje. Una vez en el caso debería de bastar con hacer una llamada a la función `SDL_Flip()` para que actualice la información en la superficie principal redibujándola.

6.11.4. Evento de Redimensionamiento

Si el usuario decide cambiar el tamaño de la ventana de la aplicación se lanzará el evento `SDL_ResizeEvent`, siempre y cuando nuestra aplicación se esté ejecutando en una ventana. Este evento tiene la siguiente estructura:

```
1 ;  
2 ;typedef struct {  
3 ;    Uint8 type;  
4 ;    int w, h;  
5 ;} SDL_ResizeEvent;
```

Los campos de esta estructura tienen el siguiente significado:

- *type*: Define el tipo de evento, en este caso `SDL_VIDEORESIZE`.
- *w, h*: Indican la nueva anchura *w* y altura *h* de la ventana.

Cuando sucede este evento deberemos de recalcular la posición de los elementos de la aplicación. Como puedes observar esta estructura si ofrece información adicional, en este caso la altura y anchura de la nueva ventana que nos servirá de referencia si tenemos que realizar alguna modificación en nuestra aplicación.

Lo ideal es que nuestro videojuego se adapte a cualquier situación pero esto no es una tarea fácil. Hasta que no ahondemos más en las bondades de SDL no consideramos oportuno sobrecargar el tutorial con todo lo que tendríamos que realizar para tratar un redimensionamiento ya que es dependiente de la aplicación que estemos diseñando. La mayoría de las veces crearemos aplicaciones que se ejecutarán en un tamaño de ventana fija o a pantalla completa por lo que la acción de redimensionar nuestro programa no es algo que vaya a ser común.

6. Captura y Gestión de Eventos

6.11.5. El Evento *Active*

Un evento propio de los entornos multiventana es *SDL_ActiveEvent*. La mayoría de los seres humanos sólo podemos realizar una tarea la mismo tiempo. Un ordenador es diferente. Podemos tener varias ventanas abiertas pero estar utilizando directamente sólo una. Esta es la conocida como la ventana activa. Este evento nos permite conocer si el usuario tiene activada la ventana donde se ejecuta nuestra aplicación o bien está en un segundo plano. La estructura del evento es la siguiente:

```
1 ;  
2 ;typedef struct {  
3 ;    Uint8 type;  
4 ;    Uint8 gain;  
5 ;    Uint8 state;  
6 ;} SDL_ActiveEvent;  
;
```

Los posibles valores de los campos de la estructura son los siguientes:

- *type*: Establece el tipo de evento, en este caso el valor **SDL_ACTIVEEVENT**.
- *gain*: Este campo tendrá el valor 1 si la aplicación está en primer plano, tiene la atención del usuario. Si pierde el estado de primer plano pasará a valor 0.
- *state*: El valor de este campo nos amplía información con respecto al campo *gain*. Los posibles valores de este campo son:
 - **SDL_APPMOUSEFOCUS**: Este valor indica el ratón entró (si *gain* vale 1) o salió (si *gain* vale 0) de la ventana de nuestra aplicación.
 - **SDL_APPINPUTFOCUS**: Indica si nuestra ventana ganó o perdió el foco de entrada de teclado dependiendo del valor de *gain*.
 - **SDL_APPACTIVE**: Indica si nuestra aplicación SDL fue maximizada o minimizada.

Teniendo el foco activo sobre una determinada aplicación quiere decir que sólo esa recibe la entrada directamente del usuario en un momento dado. El evento *active* ocurre cuando dicha ventana gana o pierde el foco.

Este tipo de evento del sistema también puede ser consultado accediendo directamente al estado del mismo. SDL proporciona una función que nos facilita esta tarea cuyo prototipo es:

```
Uint8 SDL_GetAppState(void);
```

El valor devuelto por esta función coincide con los posibles valores del campo *state* de la estructura que nos proporciona el evento.

6.11.6. Ejercicio 6

Añade al ejemplo 9 la capacidad de quedarse en pause cuando pierda el foco. Aquí tienes el resultado:

```
1 ;// Ejercicio 6
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Joystick captura de movimientos
5 ;// Consultando el estado del joystick directamente
6 ;// La aplicación debe de quedarse en PAUSE cuando pierda el foco
7 ;
8 ;
9 ;#include <iostream>
10 ;#include <SDL/SDL.h>
11 ;
12 ;using namespace std;
13 ;
14 ;int main()
15 ;{
16 ;
17 ;    // Iniciamos el subsistema de video
18 ;
19 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK) < 0) {
20 ;
21 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
22 ;        exit(1);
23 ;
24 ;    }
25 ;
26 ;    atexit(SDL_Quit);
27 ;
28 ;
29 ;    // Comprobamos que sea compatible el modo de video
30 ;
31 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
32 ;
33 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
34 ;        exit(1);
35 ;
36 ;    }
37 ;
38 ;    // Establecemos el modo de video
39 ;
40 ;    SDL_Surface *pantalla;
41 ;
42 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
43 ;
44 ;    if(pantalla == NULL) {
45 ;
46 ;        cerr << "No se pudo establecer el modo de video: "
47 ;            << SDL_GetError() << endl;
```

6. Captura y Gestión de Eventos

```
48 ;
49 ;     exit(1);
50 ;
51 ;
52 ; // Cargamos una imagen por superficie
53 ;
54 ;     SDL_Surface *personaje = SDL_LoadBMP("Imagenes/personaje.bmp");
55 ;     SDL_Surface *pausa = SDL_LoadBMP("Imagenes/pausa.bmp");
56 ;     SDL_Surface *activa = SDL_LoadBMP("Imagenes/activa.bmp");
57 ;
58 ; if(personaje == NULL) {
59 ;
60 ;     cerr << "No se pudo cargar la imagen: "
61 ;         << SDL_GetError() << endl;
62 ;
63 ;     exit(1);
64 ;
65 ; // Establecemos el color de la transparencia
66 ; // No será mostrado al realizar el blitting
67 ;
68 ;     SDL_SetColorKey(personaje, SDL_SRCCOLORKEY|SDL_RLEACCEL,\n
69 ;                     SDL_MapRGB(personaje->format, 0, 255, 0));
70 ;
71 ; // Posición inicial del personaje
72 ;
73 ;     SDL_Rect posicion;
74 ;
75 ;     posicion.x = 300;
76 ;     posicion.y = 220;
77 ;     posicion.w = personaje->w;
78 ;     posicion.h = personaje->h;
79 ;
80 ;
81 ; // Copiamos la imagen en la superficie principal
82 ;
83 ;     SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
84 ;
85 ; // Mostramos la pantalla "oculta" del búffer
86 ;
87 ;     SDL_Flip(pantalla);
88 ;
89 ;
90 ; // Si hay un joystick conectado lo abrimos
91 ;
92 ;     SDL_Joystick *joy;
93 ;
94 ; if(SDL_NumJoysticks() > 0) {
95 ;
96 ;     joy = SDL_JoystickOpen(0);
97 ;     cout << "\nAbrimos el joystick " << SDL_JoystickName(0)
98 ;         << " para la prueba. " << endl;
99 ;
100 ; } else {
```

6.11. Eventos del Sistema

```
101 ;
102 ;     cout << "Para llevar acabo esta prueba debe haber un joystick "
103 ;         << "conectado. Si es así compruebe su configuración" << endl;
104 ;
105 ;     exit(1);
106 ;
107 ;
108 ;
109 ; // Mostramos información del dispositivo
110 ;
111 ; int num_ejes = SDL_JoystickNumAxes(joy);
112 ; int num_botones = SDL_JoystickNumButtons(joy);
113 ;
114 ; cout << "Este joystick tiene " << num_ejes << " ejes y "
115 ;     << num_botones << " botones." << endl;
116 ;
117 ; cout << "\nPulse ESC para salir.\n" << endl;
118 ;
119 ; // Variables auxiliares
120 ;
121 ; SDL_Event evento;
122 ; bool desactiva = false;
123 ;
124 ; for( ; ; ) {
125 ;
126 ;     if(desactiva == false) {
127 ;
128 ;         // Actualizamos el estado del joystick
129 ;
130 ;         SDL_JoystickUpdate();
131 ;
132 ;         // Recorremos todos los ejes en búsqueda de cambios de estado
133 ;
134 ;         for(int i = 0; i < num_ejes; i++) {
135 ;
136 ;             int valor_eje = SDL_JoystickGetAxis(joy, i);
137 ;
138 ;             //cout << "Eje " << i << " -> " << valor_eje << endl;
139 ;
140 ;             if(valor_eje != 0) {
141 ;
142 ;                 if(i == 0) {
143 ;
144 ;                     if(valor_eje > 0)
145 ;                         posicion.x++;
146 ;
147 ;                     if(valor_eje < 0)
148 ;                         posicion.x--;
149 ;
150 ;                 } else {
151 ;
152 ;                     if(valor_eje > 0)
153 ;                         posicion.y++;
```

6. Captura y Gestión de Eventos

```
154 ;
155 ;           if(valor_eje < 0)
156 ;               posicion.y--;
157 ;
158 ;           }
159 ;
160 ;
161 ;
162 ;
163 ;           // Limpiamos la pantalla
164 ;
165 ;           SDL_FillRect(pantalla, NULL, 0);
166 ;
167 ;           // Cambiamos la posición del personaje
168 ;
169 ;           SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
170 ;
171 ;           // Actualizamos la pantalla principal
172 ;
173 ;           SDL_Flip(pantalla);
174 ;
175 ;
176 ;
177 ;           // Recorremos todos los botones en búsqueda acciones
178 ;
179 ;           for(int i = 0; i < num_botones; i++) {
180 ;
181 ;               int pulsado = SDL_JoystickGetButton(joy, i);
182 ;
183 ;               if(pulsado) {
184 ;
185 ;                   cout << "Ha pulsado el botón " << i << endl;
186 ;
187 ;               }
188 ;
189 ;           }
190 ;
191 ;           // Bucle que controla eventos de salida
192 ;
193 ;           while(SDL_PollEvent(&evento)) {
194 ;
195 ;               if(evento.type == SDL_KEYDOWN) {
196 ;
197 ;                   if(evento.key.keysym.sym == SDLK_ESCAPE) {
198 ;
199 ;                       SDL_JoystickClose(joy);
200 ;                       return 0;
201 ;
202 ;                   }
203 ;
204 ;                   if(evento.type == SDL_ACTIVEEVENT) {
205 ;
206 ;                       if(evento.active.gain == 0) {
```

6.11. Eventos del Sistema

```
207 ;
208 ;           SDL_BlitSurface(pausa, NULL, pantalla, NULL);
209 ;           desactiva = true;
210 ;
211 ;       } else {
212 ;
213 ;           SDL_BlitSurface(activa, NULL, pantalla, NULL);
214 ;           desactiva = false;
215 ;
216 ;       }
217 ;
218 ;       SDL_Flip(pantalla);
219 ;       sleep(1);
220 ;
221 ;   }
222 ;
223 ;   if(evento.type == SDL_QUIT) {
224 ;
225 ;       SDL_JoystickClose(joy);
226 ;       return 0;
227 ;
228 ;   }
229 ; }
230 ; }
231 ;}
```

Llegados a este punto el listado no necesita mucha explicación. Como novedad hemos introducido varias cosas. La primera es cargar dos imágenes que colocar en pantalla. La primera cuando la aplicación pierde el foco mostrando una especie de salvapantallas que nos avisa de que hemos pausado la aplicación. La otra imagen tiene un cometido parecido. Nos advierte de que hemos recuperado en foco y nos muestra durante 1 segundo el aviso para luego permitirnos seguir “disfrutando” de la aplicación.

Hemos añadido una variable booleana para que haga las veces de llave. Cuando esta variable tenga el valor *true* desactivará la gestión de todos los eventos que manejamos en el programa así como el refresco de pantalla. Esta variable tomará dicho valor cuando se produzca el evento **SDL_ACTIVEEVENT** del tipo de pérdida de foco. En ese momento mostraremos la imagen de pausa y “cerraremos la llave”.

Una vez recuperado el foco mostraremos la imagen advirtiendo de este hecho y seguidamente volveremos a activar la gestión de eventos y el refresco de la pantalla poniendo la variable booleana a *false* que podemos comparar con el efecto de “abrir la llave” permitiendo que la aplicación vuelve a un nivel activo.

6. Captura y Gestión de Eventos

6.12. Eventos del Gestor de Ventanas

Los eventos del gestor de ventanas no suelen ser manejados desde una aplicación SDL, por lo que por omisión tenemos estos eventos desabilitados. Este tipo de eventos es almacenado en una estructura definida así:

```
1 ;  
2 ;typedef struct {  
3 ;    Uint8 type;  
4 ;} SDL_SysWMEvent;  
5 ;
```

Como puedes observar en la estructura no ofrece información adicional. El campo *type* como en las demás estructuras SDL almacena el tipo de evento que se produce. Existen funciones que nos permiten recibir y extender la información acerca del evento actual del gestor de ventanas.

En el capítulo en el que tratamos el subsistema gestor de ventanas trataremos este tema con mayor profundidad.

6.13. Filtrando Eventos

Ya hemos visto un gran número de eventos. En un momento dado nos puede interesar desactivar cierto tipo de eventos. En el ejercicio 6 para que la aplicación no gestionaría evento alguno lo que hicimos fue desactivar en sondeo y el acceso al estado de los eventos de los dispositivos pero no los eventos en sí. Cuando no queramos procesar algún tipo de evento la mejor opción es indicarle a SDL que queremos ignorar dicho evento. Para esto SDL proporciona una función que nos permite establecer el estado de los eventos. Se trata de:

```
Uint8 SDL_EventState(Uint8 type, int state);
```

En el campo *type* pasaremos el tipo de evento con al que vamos a aplicarle la función. El campo *state* puede tomar tres valores:

SDL_IGNORE Ignorará el tipo de eventos pasado como parámetro.

SDL_ENABLE Activará el tipo de eventos que recibió la función como parámetro.

SDL_QUERY Hará que la función devuelva el estado del evento en cuestión.

Por ejemplo si pasamos como valor en *type* **SDL_KEYDOWN** gestionaremos el estado de los eventos de pulsación del teclado. Si en el otro parámetro introducimos **SDL_IGNORE** SDL no nos informará de ninguna manera de eventos de este tipo.

6.13. Filtrando Eventos

Ya sabemos como esperar eventos, realizar un polling para obtener eventos y acceder directamente al estado de eventos de los dispositivos así como desactivar, activar o consultar el estado de ciertos eventos. Ahora vamos a ver la manera de crear un filtro de eventos con SDL. El filtro es un mecanismo muy potente por lo que es interesante saber manejarlo.

La idea consiste en crearnos una función propia que maneje los eventos y así poder olvidarnos totalmente de llamar a las funciones *SDL_PollEvent()* o *SDL_WaitEvent()* ya que cada vez que se produzca un evento de los que incluyamos en el filtro la respuesta vendrá dada por dicho filtro. Esta función tiene que ser prototipada de una cierta manera ya que luego necesitaremos mandar un puntero a función como parámetro de otra función. El prototipo será parecido al siguiente:

```
typedef int (*SDL_EventFilter)(const SDL_Event *event);
```

Tu expresión me dice que no te ha quedado muy claro cuál debe ser el prototipo de tu función. Puedes implementar una función que tenga el siguiente aspecto:

```
int MiFiltrodeEventos(const SDL_Event* event);
```

Cuando creas esta función el parámetro de entrada *event* es un puntero constante a *SDL_Event* que contendrá la información acerca del evento que está en primera posición en la cola de eventos. Puedes manejar lo que necesites. Si decides, por ejemplo, devolver 1 si el evento está disponible en la cola o 0, como tu veas, es tu decisión. Puedes manejar en esta función más de un evento, lo que es una buena idea para tipos de eventos como el *SDL_QUIT* que deben ser manejados por la aplicación.

Para que se la función filtradora que hemos implementado sea tenida en cuenta por SDL tenemos que llamar a la función:

```
void SDL_SetEventFilter(SDL_EventFilter filter);
```

El parámetro *filter* es un puntero a una función que será la encargada de capturar los eventos. Si esta función recibe como parámetro el valor *NULL* el filtro de eventos será desactivado. Podemos consultar que filtro de eventos está activo mediante la función:

```
SDL_EventFilter SDL_GetEventFilter(void);
```

Esta función no recibe ningún parámetro y devuelve un puntero al filtro de eventos actual. En el caso de no tener configurado ningún filtro de eventos la función devolverá un puntero a *NULL*.

6. Captura y Gestión de Eventos

6.13.1. Ejemplo 10

Vamos a retocar el ejercicio número 6. Vamos a desactivar los eventos de ratón y vamos a crear un filtro que maneje todos aquellos eventos que produzcan la salida de la aplicación y muestre un mensaje cada vez que gestione un evento. Aquí tienes el listado:

```
1 ;// Ejemplo 10
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Joystick captura de movimientos
5 ;// Consultando el estado del joystick directamente
6 ;// Ignoramos los eventos de ratón y creamos un filtro
7 ;// Para los eventos que producen la salida de la aplicación
8 ;
9 ;
10 ;#include <iostream>
11 ;#include <SDL/SDL.h>
12 ;
13 ;using namespace std;
14 ;
15 ;// Filtra los eventos de salida para no tener que
16 ;// gestionarlos en el game loop
17 ;
18 ;int FiltroSalida(const SDL_Event* event);
19 ;
20 ;
21 ;// Programa principal
22 ;//
23 ;
24 ;int main()
25 ;{
26 ;
27 ;    // Iniciamos el subsistema de video
28 ;
29 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK) < 0) {
30 ;
31 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
32 ;        exit(1);
33 ;
34 ;    }
35 ;
36 ;    atexit(SDL_Quit);
37 ;
38 ;
39 ;    // Comprobamos que sea compatible el modo de video
40 ;
41 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
42 ;
43 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
44 ;        exit(1);
45 ;
46 ;    }
47 ;}
```

6.13. Filtrando Eventos

```
47 ;
48 ;    // Establecemos el modo de video
49 ;
50 ;    SDL_Surface *pantalla;
51 ;
52 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
53 ;
54 ;    if(pantalla == NULL) {
55 ;
56 ;        cerr << "No se pudo establecer el modo de video: "
57 ;            << SDL_GetError() << endl;
58 ;
59 ;        exit(1);
60 ;
61 ;
62 ;    // Cargamos una imagen por superficie
63 ;
64 ;    SDL_Surface *personaje = SDL_LoadBMP("Imagenes/personaje.bmp");
65 ;    SDL_Surface *pausa = SDL_LoadBMP("Imagenes/pausa.bmp");
66 ;    SDL_Surface *activa = SDL_LoadBMP("Imagenes/activa.bmp");
67 ;
68 ;    if(personaje == NULL) {
69 ;
70 ;        cerr << "No se pudo cargar la imagen: "
71 ;            << SDL_GetError() << endl;
72 ;
73 ;        exit(1);
74 ;
75 ;    // Establecemos el color de la transparencia
76 ;    // No será mostrado al realizar el blitting
77 ;
78 ;    SDL_SetColorKey(personaje, SDL_SRCCOLORKEY|SDL_RLEACCEL,\n
79 ;                    SDL_MapRGB(personaje->format, 0, 255, 0));
80 ;
81 ;    // Posición inicial del personaje
82 ;
83 ;    SDL_Rect posicion;
84 ;
85 ;    posicion.x = 300;
86 ;    posicion.y = 220;
87 ;    posicion.w = personaje->w;
88 ;    posicion.h = personaje->h;
89 ;
90 ;
91 ;    // Copiamos la imagen en la superficie principal
92 ;
93 ;    SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
94 ;
95 ;    // Mostramos la pantalla "oculta" del buffer
96 ;
97 ;    SDL_Flip(pantalla);
98 ;
99 ;
```

6. Captura y Gestión de Eventos

```
100 ; // Si hay un joystick conectado lo abrimos
101 ;
102 ;     SDL_Joystick *joy;
103 ;
104 ;     if(SDL_NumJoysticks() > 0) {
105 ;
106 ;         joy = SDL_JoystickOpen(0);
107 ;         cout << "\nAbrimos el joystick " << SDL_JoystickName(0)
108 ;             << " para la prueba. " << endl;
109 ;
110 ;     } else {
111 ;
112 ;         cout << "Para llevar acabo esta prueba debe haber un joystick "
113 ;             << "conectado. Si es así compruebe su configuración" << endl;
114 ;
115 ;         exit(1);
116 ;
117 ;     }
118 ;
119 ; // Mostramos información del dispositivo
120 ;
121 ;     int num_ejes = SDL_JoystickNumAxes(joy);
122 ;     int num_botones = SDL_JoystickNumButtons(joy);
123 ;
124 ;     cout << "Este joystick tiene " << num_ejes << " ejes y "
125 ;         << num_botones << " botones." << endl;
126 ;
127 ;     cout << "\nPulse ESC para salir.\n" << endl;
128 ;
129 ;
130 ; // Desactivamos los eventos de ratón
131 ;
132 ;     cout << "Estado evento de Ratón -> Movimiento : "
133 ;         << (SDL_EventState(SDL_MOUSEMOTION, SDL_QUERY) ? "activado" : "desactivado")
134 ;         << endl;
135 ;
136 ;     cout << "Estado evento de Ratón -> Pulsación : "
137 ;         << (SDL_EventState(SDL_MOUSEBUTTONDOWN, SDL_QUERY) ? "activado" : "desactivado")
138 ;         << endl;
139 ;
140 ;     cout << "Estado evento de Ratón -> Botón liberado : "
141 ;         << (SDL_EventState(SDL_MOUSEBUTTONUP, SDL_QUERY) ? "activado" : "desactivado")
142 ;         << endl;
143 ;     cout << "\n == Desactivando los eventos de ratón == \n" << endl;
144 ;
145 ;     SDL_EventState(SDL_MOUSEBUTTONDOWN, SDL_IGNORE);
146 ;     SDL_EventState(SDL_MOUSEBUTTONUP, SDL_IGNORE);
147 ;     SDL_EventState(SDL_MOUSEMOTION, SDL_IGNORE);
148 ;
149 ;     cout << "Estado evento de Ratón -> Movimiento : "
150 ;         << (SDL_EventState(SDL_MOUSEMOTION, SDL_QUERY) ? "activado" : "desactivado")
151 ;         << endl;
152 ;
```

6.13. Filtrando Eventos

```
153 ;     cout << "Estado evento de Ratón -> Pulsación : "
154 ;             << (SDL_EventState(SDL_MOUSEBUTTONDOWN, SDL_QUERY) ? "activado" : "desactivado")
155 ;             << endl;
156 ;
157 ;     cout << "Estado evento de Ratón -> Botón liberado : "
158 ;             << (SDL_EventState(SDL_MOUSEBUTTONUP, SDL_QUERY) ? "activado" : "desactivado")
159 ;             << endl;
160 ;
161 ;
162 ;     // Establecemos el filtro creado
163 ;
164 ;     SDL_SetEventFilter(FiltroSalida);
165 ;
166 ;
167 ;     // Variables auxiliares
168 ;
169 ;     SDL_Event evento;
170 ;
171 ;     // Bucle Infinito
172 ;
173 ;     for( ; ; ) {
174 ;
175 ;
176 ;         // Actualizamos el estado del joystick
177 ;
178 ;         SDL_JoystickUpdate();
179 ;
180 ;         // Recorremos todos los ejes en búsqueda de cambios de estado
181 ;
182 ;         for(int i = 0; i < num_ejes; i++) {
183 ;
184 ;             int valor_eje = SDL_JoystickGetAxis(joy, i);
185 ;
186 ;             //cout << "Eje " << i << " -> " << valor_eje << endl;
187 ;
188 ;             if(valor_eje != 0) {
189 ;
190 ;                 if(i == 0) {
191 ;
192 ;                     if(valor_eje > 0)
193 ;                         posicion.x++;
194 ;
195 ;                     if(valor_eje < 0)
196 ;                         posicion.x--;
197 ;
198 ;                 } else {
199 ;
200 ;                     if(valor_eje > 0)
201 ;                         posicion.y++;
202 ;
203 ;                     if(valor_eje < 0)
204 ;                         posicion.y--;
205 ;             }
```

6. Captura y Gestión de Eventos

```
206 ;        }
207 ;    }
208 ;
209 ;
210 ;
211 ;    // Limpiamos la pantalla
212 ;
213 ;    SDL_FillRect(pantalla, NULL, 0);
214 ;
215 ;    // Cambiamos la posición del personaje
216 ;
217 ;    SDL_BlitSurface(personaje, NULL, pantalla, &posicion);
218 ;
219 ;    // Actualizamos la pantalla principal
220 ;
221 ;    SDL_Flip(pantalla);
222 ;
223 ;
224 ;
225 ;    // Recorremos todos los botones en búsqueda acciones
226 ;
227 ;    for(int i = 0; i < num_botones; i++) {
228 ;
229 ;        int pulsado = SDL_JoystickGetButton(joy, i);
230 ;
231 ;        if(pulsado) {
232 ;
233 ;            cout << "Ha pulsado el botón " << i << endl;
234 ;
235 ;        }
236 ;    }
237 ;
238 ;
239 ;    // Bucle que controla eventos de salida
240 ;
241 ;    while(SDL_PollEvent(&evento)) {
242 ;
243 ;        if(evento.type == SDL_KEYDOWN) {
244 ;
245 ;            if(evento.key.keysym.sym == SDLK_ESCAPE) {
246 ;
247 ;                SDL_JoystickClose(joy);
248 ;                return 0;
249 ;            }
250 ;        }
251 ;
252 ;        if(evento.type == SDL_ACTIVEEVENT) {
253 ;
254 ;            if(evento.active.gain == 0) {
255 ;
256 ;                SDL_BlitSurface(pausa, NULL, pantalla, NULL);
257 ;
258 ;            } else {
```

6.13. Filtrando Eventos

```
259 ;
260 ;           SDL_BlitSurface(activa, NULL, pantalla, NULL);
261 ;
262 ;       }
263 ;
264 ;       SDL_Flip(pantalla);
265 ;       sleep(1);
266 ;
267 ;   }
268 ;
269 ;   if(evento.type == SDL_MOUSEBUTTONDOWN ||
270 ;       evento.type == SDL_MOUSEBUTTONUP ||
271 ;       evento.type == SDL_MOUSEMOTION) {
272 ;
273 ;       cerr << "Este evento debería ser ignorado" << endl;
274 ;
275 ;   }
276 ;
277 ;   if(evento.type == SDL_QUIT) {
278 ;
279 ;       SDL_JoystickClose(joy);
280 ;       return 0;
281 ;
282 ;   }
283 ; }
284 ; }
285 ;
286 ;   SDL_JoystickClose(joy);
287 ;   return 0;
288 ;}
289 ;
290 ;
291 ;// Implementación de Filtro Salida
292 ;
293 ;int FiltroSalida(const SDL_Event* event) {
294 ;
295 ;   if(event->type == SDL_QUIT) {
296 ;
297 ;       cout << "Cerrando aplicación" << endl;
298 ;       exit(0);
299 ;
300 ;   }
301 ;
302 ;   if(event->type == SDL_KEYDOWN) {
303 ;
304 ;       if(event->key.keysym.sym == SDLK_ESCAPE) {
305 ;
306 ;           cout << "Cerrando aplicación" << endl;
307 ;           exit(0);
308 ;
309 ;       } else {
310 ;
311 ;           cout << "Evento gestionado por el filtro" << endl;
```

6. Captura y Gestión de Eventos

```
312 ;      }
313 ;      }
314 ;
315 ;      return 1;
316 ;}
```

Veamos las novedades del listado. Lo primero nuevo en los ejemplos que nos encontramos es con la desactivación de los eventos de ratón. Para esto mostramos un mensaje por cada uno de los tipos de eventos con el estado actual, los desactivamos, y seguidamente mostramos otro mensaje con el nuevo estado de dichos eventos. Para realizar esta tarea hemos utilizado la función *SDL_EventState()* que hemos presentado en líneas anteriores.

Justamente después de esto hemos establecido un filtro de eventos mediante la función *SDL_SetEventFilter()*. El filtro lo hemos implementado en la función *FiltroSalida()*. Esta función se encarga de gestionar los eventos que producen la terminación del programa así como de mostrar un mensaje cada vez que se presione el teclado para que podamos comprobar que el filtro funciona. La estructura para manejar los eventos es la misma que utilizábamos en el *game loop* para gestionar dichos eventos con la variación que ahora la variable *event* es un puntero a *SDL_Event* teniendo que tener cuidado en la forma de acceder a la estructura de la variable.

El resto del listado es común a otros ejemplos y ejercicios por lo que no debe suponerte mayor problema. Hemos realizado muchos ejemplos y ejercicios manejando el joystick ya que en la aplicación que vamos a desarrollar vamos a prestar una especial atención al manejo del teclado para que tengas ejemplos de la implementación del manejo de ambos tipos de dispositivos.

6.14. Enviando Eventos

SDL nos permite poner eventos en cola. Los eventos de usuario no ocurrirán hasta que no los envíemos a dicha cola, por lo que esta funcionalidad es de obligado ofrecimiento ya que SDL permite añadir estos eventos de usuario.

La función que soporta esta acción es:

```
int SDL_PushEvent(SDL_Event *event);
```

Esta función recibe como parámetro de entrada un puntero a un *SDL_Event* que contiene información acerca del evento que queremos añadir a la cola de eventos. Como puedes observar esta función devuelve un entero. Si el valor devuelto es 0 es que todo ha ido como se esperaba y el evento se ha añadido a la cola de eventos. Si el valor devuelto es -1 significa que el evento no puede ser añadido a la cola de eventos.

Puedes agregar muchos tipos de eventos a la cola de eventos simplemente rellenando la parte apropiada de la estructura *SDL_Event* y realizando la correspondiente llamada a *SDL_PushEvent()*. Sin embargo, si usamos esta función para poner el evento en la respectiva cola, puede ocurrir que nuestro manejador de eventos no capture el evento porque estemos accediendo directamente al estado de los eventos y tengamos que manejar la cola de eventos con *SDL_WaitEvent* o con *SDL_PollEvent*.

6.15. Recopilando

En este capítulo hemos aprendido a manejar las tres técnicas existentes en SDL para manejar eventos. El *waiting* o espera, *polling* o sondeo y el acceso directo al estado de eventos de SDL.

Hemos presentado todos los tipos de eventos que maneja SDL así como su tratamiento según las tres técnicas anteriores. Se han realizado numerosos ejercicios y ejemplos sobre estos eventos con el fin de proporcionar la habilidad necesaria para manejar este tipo de datos.

Para terminar el capítulo hemos aprendido a realizar otras tareas como crear filtros de eventos, desactivar ciertos tipos de eventos o poner en cola nuevos eventos. Con el contenido de este capítulo debes de ser capaz de gestionar la entrada de usuario en tu aplicación.

6.16. Anexo. Tabla de constantes SDL para el manejo del teclado.

Rango de teclas	Constantes	Comentarios
Desde A hasta la Z	Desde <code>SDLK_a</code> hasta <code>SDLK_z</code>	
Desde 0 al 9	Desde <code>SDLK_0</code> hasta <code>SDLK_9</code>	
Desde F1 hasta F15	Desde <code>SDLK_F1</code> hasta <code>SDLK_F15</code>	
Desde Keypad 0 hasta 9	Desde <code>SDLK_KP0</code> hasta <code>SDLK_KP9</code>	

Cuadro 6.3: Constantes SDLKey comunes.

6. Captura y Gestión de Eventos

Constante	Tecla	Carácter equivalente
SDLK_BACKSPACE	Borrar	
SDLK_TAB	Tabulador	
SDLK_CLEAR		
SDLK_RETURN	Return	
SDLK_PAUSE	Pausa	
SDLK_ESCAPE	Esc	
SDLK_SPACE	Barra espaciadora	
SDLK_EXCLAIM	Exclamación	!
SDLK_QUOTEDBL	Dobles comillas	"
SDLK_HASH	Hash	
SDLK_DOLLAR	Dólar	\$
SDLK_AMPERSAND	Ampersand	&
SDLK_QUOTE	Comilla	,
SDLK_LEFTPAREN	Abre paréntesis	(
SDLK_RIGHTPAREN	Cierra paréntesis)
SDLKASTERISK	Asterisco	*
SDLK_PLUS	Signo más	+
SDLK_COMMA	Coma	,
SDLK_MINUS	Signo menos	-
SDLK_PERIOD	Punto	.
SDLK_SLASH	Barra invertida	/
SDLK_COLON	Dos puntos	:
SDLK_SEMICOLON	Punto y coma	;
SDLK_LESS	Símbolo menor que	<
SDLK_EQUALS	Signo igual	=
SDLK_GREATER	Símbolo mayor que	>
SDLK_QUESTION	Interrogación	?
SDLK_AT	Arroba	@
SDLK_LEFTBRACKET	Abre corchete	[
SDLK_BACKSLASH	Barra inclinada	
SDLK_RIGHTBRACKET	Cierra corchete]
SDLK_CARET	Sombrero	
SDLK_UNDERSCORE	Guión bajo	—
SDLK_BACKQUOTE		
SDLK_DELETE	Suprimir	
SDLK_KP_PERIOD	Punto del keypad	.
SDLK_KP_DIVIDE	Barra del keypad	/
SDLK_KP_MULTIPLY	Asterisco del keypad	*
SDLK_KP_MINUS	Signo menos del keypad	-
SDLK_KP_PLUS	Signo más del keypad	+
SDLK_KP_ENTER	Tecla enter del keypad	
SDLK_KP_EQUALS	Signo igual del keypad	=

6.16. Anexo. Tabla de constantes SDL para el manejo del teclado.

Constante	Tecla	Carácter equivalente
SDLK_UP	Flecha arriba	
SDLK_DOWN	Flecha abajo	
SDLK_RIGHT	Flecha a la derecha	
SDLK_LEFT	Flecha a la izquierda	
SDLK_INSERT	Insertar	
SDLK_HOME	Inicio	
SDLK_END	Fin	
SDLK_PAGEUP	Retrocede página	
SDLK_PAGEDOWN	Avanza página	
SDLK_NUMLOCK	Bloquea número	
SDLK_CAPSLOCK	Bloquea mayúsculas	
SDLK_SCROLLLOCK	Bloquea desplazamiento	
SDLK_RSHIFT	Mayúsculas de la derecha	
SDLK_LSHIFT	Mayúsculas de la izquierda	
SDLK_RCTRL	Control de la derecha	
SDLK_LCTRL	Control de la izquierda	
SDLK_RALT	Alt de la derecha	
SDLK_LALT	Alt de la izquierda	
SDLK_RMETA	Meta de la derecha	
SDLK_LMETA	Meta de la izquierda	
SDLK_LSUPER	Tecla Windows de la izquierda	
SDLK_RSUPER	Tecla Windows de la derecha	
SDLK_MODE	Modo de cambio	
SDLK_HELP	Tecla de ayuda	
SDLK_PRINT	Imprimir pantalla	
SDLK_SYSREQ	Petición al sistema	
SDLK_BREAK	Interrupción	
SDLK_MENU	Tecla menú	
SDLK_POWER	Tecla encendido	
SDLK_EURO	Tecla de euro	

6. Captura y Gestión de Eventos

Constante	Significado
KMOD_NONE	Modificadores no aplicados
KMOD_NUM	Bloqueo numérico activado
KMOD_CAPS	Bloqueo de mayúsculas activado
KMOD_LCTRL	Tecla Control izquierdo pulsado
KMOD_RCTRL	Tecla Control derecho pulsado
KMOD_RSHIFT	Mayúsculas derecha pulsada
KMOD_LSHIFT	Mayúsculas izquierda pulsada
KMOD_RALT	Alt derecha pulsada
KMOD_LALT	Alt izquierda pulsada
KMOD_CTRL	Alguna tecla control pulsada
KMOD_SHIFT	Una mayúsculas pulsada
KMOD_ALT	Una alt pulsada

Cuadro 6.4: Constantes modificadores de teclado SDL.

Capítulo 7

Subsistema de Audio

7.1. Introducción

Está claro que una vez que desarrollemos una aplicación querremos dotarla de sonido para conseguir un mayor realismo o una mayor involucración del jugador en la misma. El audio no es un aspecto esencial de un videojuego pero es un complemento que puede convertir a nuestro programa en un producto de calidad.

En una aplicación, sobre todo en un videojuego, existen varios tipos de sonido o audio. Está el referente a eventos de menú que se les puede asociar algún sonido para crear un efecto concreto. Por ejemplo si al hacer click en un menú este se expande y se contrae podemos asociarle un sonido que nos permita conseguir un efecto de velocidad. Otro tipo de sonidos se asocia con la propia interacción de los personajes del juego. Por ejemplo cuando se acciona el botón de disparo y nuestro personaje hace un movimiento con la espada no está demás asociarle un sonido que simule el efecto de una espada cortando el viento.

El tercer tipo de sonido es la banda sonora del videojuego. Esta nos aportará un estado a la aplicación. Con una buena banda sonora podemos producir en el jugador estados de tranquilidad o nerviosismo según nos interese y según se desarrolle el videojuego. En los videojuegos actuales este es un aspecto muy cuidados ya que todo esfuerzo es poco por hacer que la historia de un videojuego sea interesante.

7.2. Objetivos

Los objetivos de esta unidad son:

1. Conocer el subsistema de audio de SDL.
2. Aprender a inicializar el subsistema para poder utilizarlo en nuestras aplicaciones.

7. Subsistema de Audio

7.3. Conocimientos previos

El sonido tiene unas características físicas que debemos de conocer mínimamente antes de trabajar con el subsistema de audio o de sonido. Estas cualidades vienen dadas por la propia naturaleza del sonido y es necesaria las conozcamos para saber que estamos haciendo en todo momento. El sonido se transmite mediante ondas que utilizan el aire como medio de transmisión.

Las magnitudes físicas del sonido, o lo que es lo mismo, las formas en que podemos medir el sonido son seis dependiendo de la característica del mismo que nos interese. Estas magnitudes están intimamente relacionadas con el concepto de onda. Vamos a detallar las más importantes:

Longitud de onda Es el tamaño de la onda, es decir, la distancia que se comprende entre el incio y el final de la onda.

Frecuencia Es el número de ondas o ciclos que se producen por unidad de tiempo. En nuestro caso la unidad de tiempo será el segundo por lo que la frecuencia será medida en Hertzios.

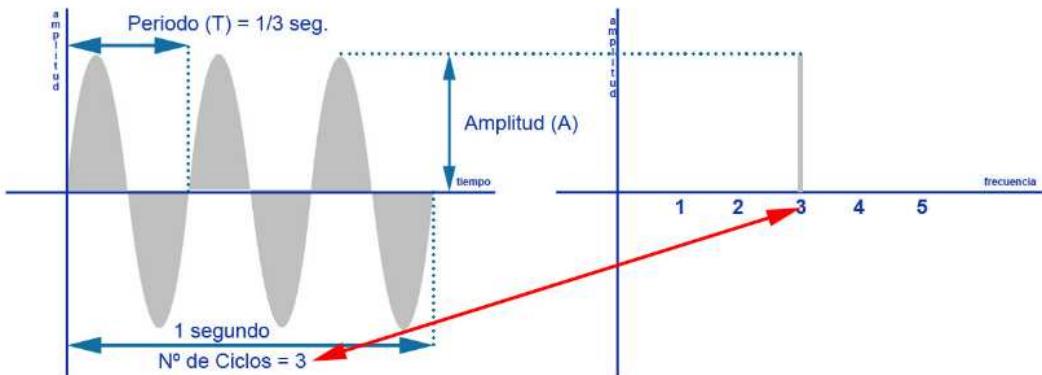


Figura 7.1: Frecuencia

Periodo Es el tiempo que tarda un ciclo en producirse. Es inversamente proporcional a la frecuencia.

Amplitud Es la cantidad de energía que contiene la onda, en nuestro caso la señal sonora.

En las figuras 7.1 y 7.2 puedes ver la representación de estos conceptos en función de distintas variables.

Uno de las características más usadas en el manejo del sonido es la referente a la frecuencia de muestreo. El oído humano es capaz de oír frecuencias que están en el rango que comprende desde los 20 Hz hasta

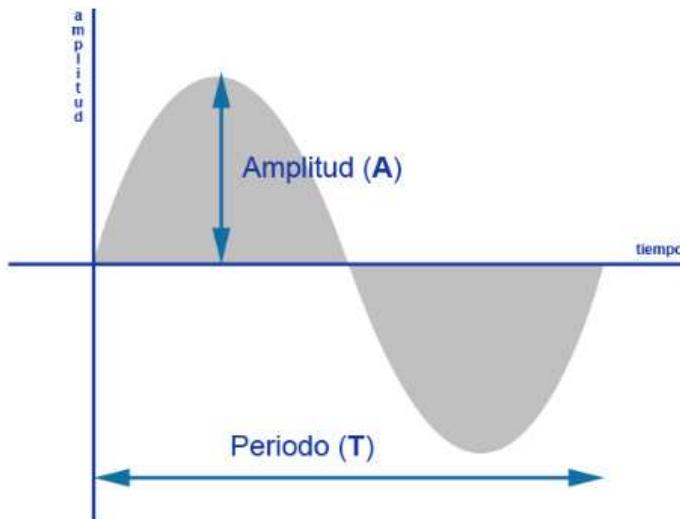


Figura 7.2: Amplitud y Periodo

los 20.000 Hz. Para no perder información la frecuencia de muestreo se establece, como mínimo y valor recomendable, al doble de la frecuencia máxima que se quiere muestrear. Por esto mismo los sonidos que se almacenan en un CD de audio o en cualquier formato digital se muestrean a 44100 Hz.

Otro concepto importante es cuantos bits vamos a dedicar a cada una de las muestras. Los CDs de audio ofrecen una calidad de 16 bits asociados a la comentada frecuencia de muestreo son más que suficientes para almacenar cualquier tipo de sonido.

Esta ha sido una breve introducción a los conceptos asociados a la naturaleza de los sonidos. Una explicación básica que puede servirnos para seguir este capítulo sabiendo en todo momento de que estamos hablando.

7.4. El Subsistema de Audio

El subsistema de audio de SDL nos permite crear y reproducir sonidos en nuestra aplicación. Este subsistema es uno de los aspectos menos versátiles de los que proporciona esta librería. Después de revisar el subsistema de video y el manejo y captura de eventos en SDL podrás observar la escasa potencia de el subsistema de audio en comparación con los dos subsistemas anteriores.

Aunque lo parezca, esto no es sorprendente, ya que SDL está diseñada para trabajar en muchas plataformas y el esfuerzo que se ha realizado para estandarizar los sistemas de audio está muy lejos con respecto al esfuerzo que se hizo entorno a los sistemas de video, por lo que los desarrolladores de la

7. Subsistema de Audio

librería no han tenido las herramientas y definiciones suficientes como para crear un sistema de audio que fuese transportable y potente, por lo que se ha optado por mantener esta portabilidad aunque haya que hacer un mayor esfuerzo para introducir audio en nuestra aplicación.

Cuando desarrollamos videojuegos optaremos por no depender exclusivamente de este sistema ya que es común encontrar programas implementados mediante el uso de una librería que potencia el sistema de audio de SDL como puede ser *SDL_mixer*. Esta librería nos facilita mucho el trabajo con el audio y será estudiada en el apartado de librerías auxiliares de SDL.

Como conclusión decir que debido a que este sistema no es lo suficientemente potente para que nos resulte cómodo trabajar con él en nuestra aplicación tendremos que hacer uso de librerías adicionales de terceros para que el desarrollo no se convierta en una pesadilla.



Figura 7.3: Tarjeta de sonido de gama alta.

El hardware de sonido o de audio tiene varios componentes. El primero que debemos de poseer es una tarjeta de sonido. La variedad de estas en el mercado, al contrario que ocurría con las tarjetas gráficas, cada vez es menor. Para las soluciones domésticas se opta por variantes integradas en placas bases de unos cuantos fabricantes. Hace unos años esto era casi impensable. Es decir, actualmente la mayoría de los ordenadores de consumo traen un chip, más o menos potente, que nos permite reproducir sonido en nuestro pc con total fidelidad para un uso normal.

Normalmente el añadir una tarjeta de sonido es porque necesitamos algún tipo de entrada/salida especial o bien porque vamos a dedicarnos al mundo del audio de una manera algo más profesional. Las alternativas a nivel profesional es amplia, mientras que al ámbito doméstico sigue dominando la misma empresa que lo hacía ya hace años.

Resumiendo, actualmente todos los ordenadores de última generación

incluyen un chip de sonido integrado que nos permite reproducir audio de manera más que aceptable incluyendo salidas como las digitales, tanto coaxiales como ópticas, que proporcionan un resultado excepcional.



Figura 7.4: Altavoces para ordenador.

El segundo componente que debemos de tener son unos altavoces. La variedad de estos en el mercado si que es amplia. Existen numerosos tipos de configuraciones desde los comunes dos altavoces, pasando por los integrados en pantalla o portátiles, hasta las configuraciones con subwoofer desde los tres altavoces hasta los ocho. Depende del chip o tarjeta de sonido que tengamos nos interesará más tener una configuración de altavoces u otra.

Para trabajar con el subsistema de audio de SDL debes de conoce algunas ideas básicas de como funcionan los sonidos, ya que si no se nos escaparán detalles de las estructuras que son fundamentales.

7.4.1. Inicializando el subsistema de Audio

La primera tarea que tenemos que realizar es la de inicializar el subsistema de audio. Para esto tendremos que introducir la constante `SDL_INIT_AUDIO` a la hora de realizar la llamada a la función `SDL_Init()`.

En entornos Windows es fundamental iniciar el subsistema de audio que tendrá que estar activo concurrentemente con el sistema Directx al que Windows confía la gestión de este tipo de aspectos del sistema o tendremos serios problemas a la hora de ejecutar nuestra aplicación.

Como ocurría con el subsistema de video, una vez incializada SDL con el subsistema de audio tenemos que establecer un modo, esta vez, de audio. Para manejar la función encargada de esta tarea es necesario conocer la naturaleza de ciertas estructuras en SDL. Por ello vamos a estudiar primero dichas es-

7. Subsistema de Audio

tructuras y seguidamente aprenderemos a establecer el modo de audio en SDL.

7.5. Conceptos y estructuras del subsistema

Sólo existen dos estructuras en el subsistema de audio de SDL: *SDL_AudioSpec* y *SDL_AudioCVT*. Son unas crípticas audio-estructuras para la especificación de audio y la conversión de audio.

La primera de las estructuras del subsistema que vamos a analizar es *SDL_AudioSpec*. Esta estructura contiene información acerca del formato de audio que vamos a utilizar, el búffer de sonido, el número de canales... Se define de la siguiente forma:

```
1 ;  
2 ;typedef struct {  
3 ;    int freq;  
4 ;    Uint16 format;  
5 ;    Uint8 channels;  
6 ;    Uint8 silence;  
7 ;    Uint16 samples;  
8 ;    Uint32 size;  
9 ;    void (*callback) (void *userdata, Uint8 *stream, int len);  
10 ;   void *userdata;  
11 ;} SDL_AudioSpec;
```

El significado de cada uno de sus campos es el siguiente:

■ *freq*: Especifica la frecuencia en hertzios de reproducción del sample o porción de sonido a reproducir. Este campo determina directamente cuantos bytes por segundo son enviados a través del hardware de audio. Existen varios valores habituales para este campo, dependiendo del formato que vayamos a utilizar y su frecuencia de muestreo. En teoría esta frecuencia puede tomar cualquier valor, pero en la práctica los valores a considerar son los siguientes:

- 11025 o 11 kHz: Esta frecuencia de muestreo es equivalente a la calidad del sonido telefónico.
- 22050 o 22 kHz: Esta es equivalente a la calidad transmisión de las emisoras de radio. Es el valor máximo que percibe el oido humano.
- 44100 o 44 kHz: Esta es la máxima calidad recomendable, aunque últimamente la tendencia marca los 48 KHz como la frecuencia de muestreo para conseguir la máxima calidad. A esta frecuencia se muestrean los CDs de audio. La teoría dice que hay que muestrear una onda al doble de la frecuencia máxima de dicha onda. Como en

7.5. Conceptos y estructuras del subsistema

el humano el máximo es 20050 se establecen los 44.1 KHz como la frecuencia de muestreo más alta recomendable.

El sampleado o grabación de audio se realiza como mínimo al doble de lo audible para no perder detalle del sonido.

- *format*: Indica los bits y tipo del formato del sample, es decir, el formato del sample. Los valores de este campo se determina estatableciendo el número de bits, que son 8 o 16 bits, con o sin signo, y en el caso de los 16 bits si la configuración es big-endian o little-endian. Concretando, los posibles valores son:
 - AUDIO_U8: Sample de 8 bits sin signo, cada canal de audio consiste en una transmisión de enteros sin signo de 8 bits.
 - AUDIO_S8: Sample de 8 bits con signo, cada canal de audio consiste en una transmisión de enteros con signo de 8 bits.
 - AUDIO_U16 o AUDIO_U16LSB: Sample de 16 bits sin signo en formato little-endian, como los anteriores, el canal del audio consiste en una transmisión de enteros sin signo de 16 bits, en el segundo caso en little-endian.
 - AUDIO_S16 o AUDIO_S16LSB: Sample de 16 bits con signo en formato little-endian. El canal del audio consiste en una transmisión de enteros con signo de 16 bits, en el segundo caso en little-endian.
 - AUDIO_16MSB: Sample de 16 bits sin signo en formato big-endian. El canal de audio consiste en una transmisión de datos en formato big-endian de Sint16s.
 - AUDIO_U16SYS: Dependiendo del diseño de nuestro sistema será AUDIO_U16LSB si es little-endian o AUDIO_U16MSB si el sistema es big-endian.
 - AUDIO_S16SYS: Dependiendo del diseño de nuestro sistema será AUDIO_S16LSB si es little-endian o AUDIO_S16MSB si el sistema es big-endian.
- *channels*: Indica el número de canales de audio. Las configuraciones habituales son uno, para un sistema mono, y dos para estéreo. Dependiendo del formato y el número de canales obtendremos el tamaño del sample. Un sample o porción de sonido verá condicionado su tamaño por el número de canales. Si, por ejemplo, un sample de un canal ocupa x bytes es lógico pensar que si este mismo sample lo volvemos estéreo (con lo que hay que dedicarle dos canales) el tamaño del mismo se duplique.
- *silence*: Es un valor calculado que representa al valor para el silencio. Cuando queremos reproducir el silencio dicho valor se escribe en el búffer de audio.

7. Subsistema de Audio

- *samples*: Indica el tamaño del búffer de audio en samples.
- *size*: Indica el tamaño del búffer medido en bytes. Es un campo de los llamados calculados así que no tendremos que preocuparnos por él y es de sólo lectura.
- *void (*callback)(void *userdata, Uint *stream, int len)*: Este campo es un puntero a una función de retrollamada definida por el usuario. Es el que se utiliza cuando se quiere reproducir un sonido. El campo *userdata* suele ser el mismo que el último miembro de esta estructura. El usuario-programador es el encargado de diseñar e implementar dicha función cuyo objetivo es llenar el búffer contenido en el puntero *stream* con una cantidad de bytes que debe ser igual a *len* y es utilizada cuando queremos reproducir cierto sonido. Esta tarea no es complicada cuando se quiere reproducir un sonido aislado, lo que ocurre que en un videojuego la mayor parte del tiempo necesitaremos mezclar varios sonidos que produzcan los efectos del mismo. SDL nos proporciona las herramientas para hacerlo, pero el resultado no es de una calidad-usabilidad aceptable.
- *userdata*: Es un puntero a los datos que son pasados a la función callback de esta estructura.

La estructura *SDL_AudioCVT* contiene información para convertir sonido de un formato a otro. No vamos a mostrar y explicar todos los campos de esta estructura porque en su gran mayoría son usados, creados y mantenidos sólo por la librería SDL y la implementación está oculta al usuario. Es suficiente con saber que convertir un sonido de un formato a otro es una tarea bastante complicada y esta estructura está para facilitarnos la tarea..

7.6. Funciones para el manejo del Audio

Vamos a presentar diferentes funciones que nos van a permitir manejar el subsistema de audio puro de SDL. Recuerda que antes de poder utilizar estas funciones es fundamental que inicialices el subsistema de audio.

7.6.1. Abrir, Pausar y Cerrar

Como con el subsistema de video, para poder utilizar el subsistema tenemos que configurar algunos aspectos más aparte de la propia inicialización del subsistema de audio. En este caso deberemos de abrir el dispositivo de audio. SDL proporciona la siguiente función para realizar esta tarea:

```
int SDL_OpenAudio(SDL_AudioSpec *desired, SDL_AudioSpec *obtained);
```

7.6. Funciones para el manejo del Audio

Esta función recibe como parámetros dos punteros a la estructura *SDL_AudioSpec*. El parámetro *desired* sirve para especificarle las condiciones con las que queremos trabajar para que intente abrir el dispositivo de audio con ellas, aunque no siempre lo consiga con éxito, son las deseables. Este parámetro debe de ser totalmente completado por nosotros con todos los datos necesarios.

El parámetro *obtained* nos indica que condiciones hemos conseguido para trabajar con el dispositivo de audio. Tendremos que adaptar nuestro trabajo a estas condiciones. Podemos pasar el valor `NULL` a este segundo parámetro y así SDL hará todo lo posible para realizar la mejor emulación de las características especificadas en el parámetro *desired*. Esto no es recomendable porque normalmente este esfuerzo no es necesario para tener obtener un resultado decente.

En el caso de que *desired* y *obtained* tuvieran la misma información significaría que habríamos conseguido establecer todas nuestras condiciones. La función, como es habitual en SDL, devuelve 0 si todo ha ido correctamente y -1 si existiese algún tipo de error.

Una vez abierto el dispositivo de sonido deberemos de iniciar la reproducción de audio. La función que nos permite realizar este trabajo está definida de la siguiente manera:

```
void SDL_PauseAudio(int pause_on);
```

Como podrás observar en el prototipo de la función esta puede utilizarse también para parar el audio, o lo que es lo mismo, con esta función podemos activar/pausar los sonidos. Si recibe como parámetro el valor 0 activamos la reproducción de sonido o, siguiendo la sintaxis de la función, no pausa la reproducción de audio. Si recibe 1 la función se encarga de pausar el audio.

Una vez hayamos terminado de trabajar con el dispositivo de audio debaremos de proceder a cerrarlo. SDL proporciona la siguiente función para realizar esta tarea:

```
void SDL_CloseAudio(void);
```

No existe un prototipo de función más simple. Esta función no recibe ningún parámetro, ni tampoco devuelve nada por lo que podemos, si lo creemos conveniente, utilizarla con la función *atexit()* y que se cierre el dispositivo automáticamente al terminar la aplicación.

La última función que vamos a estudiar en esta sección es la que nos permite comprobar el estado de reproducción de audio. Con este fin usamos la función:

```
SDL_audiostatus SDL_GetAudioStatus(void);
```

7. Subsistema de Audio

Esta función no recibe ningún parámetro y devuelve uno de estos posibles valores:

- **SDL_AUDIO_STOPPED** Para indicar que la reproducción de audio está parada.
- **SDL_AUDIO_PLAYING** Para indicar que se está reproduciendo sonido.
- **SDL_AUDIO_PAUSED** Para indicar que la reproducción de audio está en pausa.

Ya hemos visto todo los conceptos que tenemos que saber sobre la base de este subsistema. En nuestra aplicación final no trabajaremos con él y haremos uso de una librería auxiliar que es mucho más cómoda y intuitiva de utilizar.

7.6.1.1. Ejemplo 1

Para terminar este apartado veamos un ejemplo de como usar todas las estructuras y funciones que hemos estudiado en este capítulo:

```
1 ;// Ejemplo 1 - Subsistema de Audio
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Generando un sonido aleatorio
5 ;
6 ;
7 ;#include <iostream>
8 ;#include <iomanip>
9 ;
10 ;#include <SDL/SDL.h>
11 ;
12 ;using namespace std;
13 ;
14 ;void funcion_retrollamada(void *userdata, Uint8 *buffer, int len);
15 ;
16 ;int main()
17 ;{
18 ;
19 ;    // Iniciamos el subsistema de video
20 ;
21 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {
22 ;
23 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
24 ;        exit(1);
25 ;
26 ;    }
27 ;
28 ;
29 ;    atexit(SDL_Quit);
```

7.6. Funciones para el manejo del Audio

```
30 ;
31 ;    // Comprobamos que sea compatible el modo de video
32 ;
33 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
34 ;
35 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
36 ;        exit(1);
37 ;
38 ;    }
39 ;
40 ;
41 ;    // Establecemos el modo de video
42 ;
43 ;    SDL_Surface *pantalla;
44 ;
45 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
46 ;
47 ;    if(pantalla == NULL) {
48 ;
49 ;        cerr << "No se pudo establecer el modo de video: "
50 ;            << SDL_GetError() << endl;
51 ;
52 ;        exit(1);
53 ;    }
54 ;
55 ;    // Configuramos el subsistema de audio
56 ;
57 ;    // Especificamos las opciones de audio
58 ;
59 ;    SDL_AudioSpec espec_deseadas;
60 ;    SDL_AudioSpec espec_obtenidas;
61 ;
62 ;    espec_deseadas.freq = 11025;
63 ;    espec_deseadas.format = AUDIO_S16SYS;
64 ;    espec_deseadas.channels = 2;
65 ;    espec_deseadas.samples = 4096;
66 ;    espec_deseadas.callback = funcion_retrollamada;
67 ;    espec_deseadas.userdata = NULL;
68 ;
69 ;    // Abrimos el dispositivo de audio
70 ;
71 ;    if(SDL_OpenAudio(&espec_deseadas, &espec_obtenidas) < 0) {
72 ;
73 ;        cerr << "No se puede abrir el dispositivo de audio" << endl;
74 ;        exit(1);
75 ;    }
76 ;
77 ;    cout << "\n - Configuración de Audio conseguida - \n" << endl;
78 ;
79 ;    cout << "Frecuencia : " << (int) espec_obtenidas.freq << endl;
80 ;    cout << "Canales : " << (int) espec_obtenidas.channels << endl;
81 ;    cout << "Samples : " << (int) espec_obtenidas.samples << endl;
82 ;
```

7. Subsistema de Audio

```
83 ;
84 ;
85 ;
86 ;    // Variables auxiliares
87 ;
88 ;    SDL_Event evento;
89 ;
90 ;    cout << "\nPulsa ESC para salir\n" << endl;
91 ;
92 ;    cout << "Pulsa 'p' para reproducir el sonido aleatorio" << endl;
93 ;
94 ;    // Bucle "infinito"
95 ;
96 ;    for( ; ; ) {
97 ;
98 ;        while(SDL_PollEvent(&evento)) {
99 ;
100 ;            if(evento.type == SDL_KEYDOWN) {
101 ;
102 ;                if(evento.key.keysym.sym == SDLK_ESCAPE)
103 ;                    return 0;
104 ;
105 ;                if(evento.key.keysym.sym == SDLK_p) {
106 ;
107 ;                    if(SDL_GetAudioStatus() == SDL_AUDIO_STOPPED ||
108 ;                        SDL_GetAudioStatus() == SDL_AUDIO_PAUSED) {
109 ;
110 ;                        cout << "Reproducindo" << endl;
111 ;                        SDL_PauseAudio(0);
112 ;
113 ;                } else {
114 ;
115 ;                    cout << "Pausa" << endl;
116 ;                    SDL_PauseAudio(1);
117 ;                }
118 ;
119 ;            }
120 ;        }
121 ;
122 ;        if(evento.type == SDL_QUIT)
123 ;            return 0;
124 ;
125 ;    }
126 ;}
127 ;
128 ;}
129 ;
130 ;// Función de retrollamada para el subsistema de audio
131 ;// Rellenamos el búffer de audio con valores aleatorios entre 0 y 255
132 ;
133 ;void funcion_retrollamada(void *userdata, Uint8 *buffer, int len) {
134 ;
135 ;    for(int i = 0; i < len; i++)
```

```
136 ;         buffer[i] = i % 256;  
137 ;}
```

Vamos a estudiar directamente los aspectos novedosos de este listado. En primer lugar encontramos que en la llamada a *SDL_Init()* hemos introducido la constante que nos permite inicializar el subsistema de audio. La siguiente novedad la encontramos cuando definimos las variables del tipo *SDL_AudioSpec*. Definimos dos variables de este tipo. La primera *espec_deseadas* para inicializarla con los valores de configuración con los que deseamos que se abra el subistema de audio.

En *espec obtenidas* tendremos las especificaciones de la configuración que ha podido establecer SDL. Las mostraremos por pantalla para saber en todo momento en que modo estamos trabajando.

En el bucle del *game loop* introduciremos el manejo del evento de la pulsación de la tecla *p*. Cuando pulsemos dicha tecla si la reproducción del audio está parada o en pausa se pondrá en marcha y en el caso de estar en marcha se pausará. Para conseguir este efecto hemos tenido que hacer uso de dos funciones SDL. La primera *SDL_GetAudioStatus()* que nos indica el estado actual de la reproducción y la segunda *SDL_PauseAudio()* para modificar el estado actual de la reproducción.

7.6.2. Bloqueando y desbloqueando

En teoría la función de callback de la estructura de audio de SDL se ofrece para tener el mayor control posible del búffer de audio. Actualmente realizar esta tarea de este modo es como volver a encender fuego con el golpeo de dos piedras. El puntero *userdata* de la estructura *SDL_AudioSpec* sabemos que es pasado a la función de callback. Este campo es un puntero void por lo que podemos introducir en el buffer de audio cualquier dato que nos permita realizar efectos con el sonido. Si en un momento dado queremos cambiar el tipo de sonido vamos a tener un gran dolor de cabeza.

Si cambiamos los datos apuntados por *usardata* hasta que realmente queramos que se reproduzca el audio el mismo trozo de sonido ha de estar en el búffer repetidamente. Para que nuestro sonido se reproduzca en el momento adecuado primero debemos de usar la función *void SDL_LockAudio()* para parar la llamada continua a la función de callback, y luego llamaremos a la función *void SDL_UnlockAudio()* para que se vuelva a producir la llamada a la función de callback cuando queramos que se reproduzca el sonido. Esto facilita el tener que “vaciar” el buffer para producir este efecto.

Esta forma de trabajar con los sonidos es una de las mayotas creadoras de

7. Subsistema de Audio

problemas en la creación de videojuegos por lo que, en un principio, vamos a evitar esta metodología.

7.6.3. Manejando el formato WAV

Seguramente querrás conseguir reproducir algún sonido que no cree dolores de cabeza a todas las personas que se encuentren cerca tuya como el sonido aleatorio del primer ejemplo. Para esto reproduciremos sonidos que tengan formato wav sin compresión que es uno de los formatos base de audio por excelencia.

WAV es el apócope de *waveform audio format* que es un formato de audio digital que se presenta, normalmente, sin compresión de datos desarrollado por IBM y Microsoft. Este formato permite trabajar con muchas de las configuraciones que acepta el subsistema de audio en SDL.

SDL proporciona facilidades para trabajar con este tipo de formato aunque de una manera un poco cruda. El prototipo de la función que nos permite realizar esta tarea es el siguiente.

```
SDL_AudioSpec *SDL_LoadWAV(const char *file, SDL_AudioSpec *spec,  
                           Uint8 **audio_buf, Uint32 *audio_len);
```

Esta función tiene cuatro parámetros. Su funcionalidad es cargar un archivo de formato wav en memoria. Como puedes observar el primer parámetro sirve para indicar dónde está el fichero wav que queremos abrir. La función coloca en el parámetro *spec* la información acerca del fichero wav en una estructura *SDL_AudioSpec* que contiene información esencial como el formato y el número de canales. En *audio_len* se almacena tamaño en bytes del mismo. El parámetro *audio_buf* es relleno con un puntero a los datos de audio del sonido. Esta función viene preparada para pasar todos sus parámetros a la función de retrollamada de la estructura que maneja el subsistema de audio en SDL. Devuelve un puntero a *SDL_AudioSpec* con la información del sample de audio o el valor *NULL* si se produjo un error.

Cuando terminemos de trabajar con los datos del fichero wav podemos liberar la memoria que éste consume. Para ello SDL proporciona la siguiente función:

```
void SDL_FreeWAV(Uint8 *audio_buf);
```

El único parámetro que necesita la función es el parámetro *audio_buf* devuelto por la función *SDL_LoadWav()*. No vamos a perder mucho más tiempo con este subsistema ya que una vez conozcas *SDL_mixer* no volverás a utilizar ninguna de las funciones aquí presentadas.

7.7. ¿Porqué no utilizar este subsistema?

Comparado con otras partes de SDL, el subsistema de audio es de muy bajo nivel debido a que debe de mantener la compatibilidad con todas las plataformas que soporta. Hay cosas mejores que hacer que conocer en detalle el manejo de búfferes de un sistema de audio existiendo una alternativa totalmente viable que nos permitirá el manejo de este subsistema a un nivel mucho más alto.

En definitiva para manejar sonidos la mejor alternativa es acudir a una librería externa, en concreto a *SDL_mixer*.

7.8. Recopilando

En este capítulo hemos conocido algunas características del subsistema de audio en SDL, su inicialización y su cierre, así como funciones que nos permiten trabajar con él a un bajo nivel. Como no consideramos que sea factible el hecho de trabajar a estos niveles consideramos que no es una buena alternativa crear una aplicación utilizando exclusivamente el subsistema de audio de SDL.

7. Subsistema de Audio

Capítulo 8

Subsistema de CDROM

8.1. Introducción

El CD-ROM ha sido el soporte por excelencia de los videojuegos bastantes muchos años. El formato wav es un formato fácil de utilizar pero es muy muy pesado. El CD-Audio está especialmente diseñado para soportar este formato de sonido y SDL nos ofrece la posibilidad de trabajar fácilmente con la unidad de CD de nuestro sistema para reproducir música directamente desde dicho dispositivo.

Es decir, podemos utilizar la capacidad del CD para almacenar la música de nuestro videojuego ahorrándonos así un espacio considerable en el disco duro. Además podemos permitir al usuario que personalice la música del videojuego introduciendo diferentes CDs de audio en la unidad.



Figura 8.1: Unidad de CD

La música de fondo ha sido una de las principales bazas en la industria de los videojuegos. Seguramente habrás disfrutado de alguna banda sonora de videojuego mientras te absorbía la trama del juego o un cambio en el transcurso del mismo era marcado por un mayor tempo de la música de fondo. Por todo esto SDL proporciona esta API, especializada, con el fin de poder

8. Subsistema de CDROM

utilizar audio CD's en nuestros videojuegos.

8.2. Objetivos

Los objetivos de este capítulo son

1. Comprender los conceptos asociados al manejo del CD de Audio.
2. Conocer las estructuras y funciones que nos proporcionan información de este subsistema.
3. Aprender a manejar la unidad de CD con las funciones que nos proporciona SDL.

8.3. Subsistema de CDROM

Nos encontramos con una API bastante completa que nos permite incluso desarrollar un completo reproductor de CD basado en SDL, aunque existen muy buenos reproductores de CD libres pero cada uno dedica su tiempo libre a lo que quiere. El subsistema de CD de SDL está compuesto por dos estructuras y once funciones. Dividaremos a las funciones en dos grupos, las informativas y las que nos facilitan el manejo del CD.

8.3.1. Inicialización

Como el resto de subsistemas de la SDL debemos de indicar que queremos utilizar el susbsistema de CD-ROM al inicializar SDL. Para inicializar este subsistema debemos de pasar la constante `SDL_INIT_CDROM` cuando realicemos la llamada a la función `SDL_Init()` para iniciar la librería.

A diferencia de otros subsistemas no debemos establecer un modo ya que no es un concepto aplicable al tipo de dispositivo que vamos a manejar con este subsistema pero sí antes de empezar a utilizar el CD-ROM deberemos de abrir el dispositivo, pero no abrir la bandeja físicamente, si no prepararlo para que podamos recibir información y datos de él. La función que realiza este cometido en SDL es:

```
SDL_CD *SDL_CDOpen(int drive);
```

Esta función recibe como parámetro la unidad que queremos abrir y devuelve un puntero a una estructura de datos `SDL_CD` con la información disponible del CD. La unidad 0 es la unidad por defecto del sistema.

8.4. Conceptos y estructuras fundamentales

Como es habitual en SDL, cuando terminemos de utilizar un dispositivo, en este caso el CD-ROM, deberemos de cerrarlo. Recuerda no físicamente. La función encargada de cerrar este dispositivo es:

```
void SDL_CDClose(SDL_CD *cdrom);
```

Esta función recibe como parámetro el valor que nos devolvió la función que se encargaba de la apertura del dispositivo.

8.4. Conceptos y estructuras fundamentales

Dos son las estructuras pertenecientes a este subsistema. Simplemente se encargan de proporcionar información sobre el dispositivo de CD y sobre el medio que tenemos en el mismo. Veamos la primera de ellas:

```
;-----  
1 ;typedef struct {  
2 ;     int id;  
3 ;     CDstatus status;  
4 ;     int numtracks;  
5 ;     int cur_track;  
6 ;     int cur_frame;  
7 ;     SDL_CDtrack track[SDL_MAX_TRACKS + 1];  
8 ;} SDL_CD;  
;
```

Vamos a pasar a describir cada uno de los campos de esta estructura.

- *id*: Este campo es un identificador privado que diferencia únicamente a cada unidad de CD-ROM. Normalmente este campo no tiene ninguna repercusión en nuestra aplicación. Es un campo que tiene un mero valor informativo.
- *status*: Indica el estado de la unidad de CD. Es del tipo *CDstatus*. Este es un tipo enumerado que puede tomar los siguientes valores constantes:
 - **CD_TRAYEMPTY**: Indica que no hay ningún CD en la unidad.
 - **CD_STOPPED**: Nos indica que el CD está detenido.
 - **CD_PLAYING**: Como puedes intuir, indica que el CD está reproduciéndose.
 - **CD_PAUSED**: Hace referencia a que el CD está en pausa.
 - **CD_ERROR**: Este es un estado de error. Indica que algo no funciona bien en la reproducción del CD.

8. Subsistema de CDROM

- *numtracks*: Este campo nos indica cuantas pistas tiene el CD. Si alguna vez has escuchado un CD de audio sabrás que estos CD's están divididos en pistas o *tracks* y que, normalmente, en cada pista se almacena una canción. Es la manera más sencilla de indexar música en un CD.
- *cur_track*: Contiene la pista donde estamos posicionados actualmente, es decir, la que estamos reproduciendo o en pausa actualmente.
- *cur_frame*: Indica cual es el frame donde nos encontramos dentro de la pista que se está reproduciendo. El frame es una medida propia de este soporte de almacenamiento, ya que debido a la versatilidad del mismo, en cuanto a tipo de información almacenable, no es posible establecer un único patrón en tiempo o capacidad para éste, por lo que se establece una medida compatible con todos los tipos de información que puede albergar un CD. Un frame equivale a unos 2 Kilobytes. Podemos considerar que el frame es la unidad fundamental de medida de este tipo de soporte, como el *byte* es la única básica de medición de memoria de un ordenador personal.
- *SDL_CDtrack track[SDL_MAX_TRACKS + 1]*: Este parámetro es un vector que almacena la información sobre cada una de las pistas del medio que tengamos introducido.

Cada elemento del vector de la estructura anterior que guarda información de las pistas es del tipo *SDL_CDtrack* que posee la siguiente estructura:

```
1 ;typedef struct {
2     Uint8 id;
3     Uint8 type;
4     Uint32 length;
5     Uint32 offset;
6 } SDL_CDtrack;
```

El significado de cada uno de sus campos es:

- *id*: Indica el número de la pista del CD insertado en la unidad de CD. El rango de este campo comprende del 0 al 99, siendo 0 el indicador para la primera pista del CD.
- *type*: Este campo puede tomar dos valores. Un CD, como sabes, puede almacenar varios tipos de información. Si la pista contiene audio tomará el valor *SDL_AUDIO_TRACK* mientras que si la pista en cuestión almacena datos toma el valor de *SDL_DATA_TRACK*. Lógicamente no podremos reproducir pistas del tipo *SDL_DATA_TRACK* ya que no contendrá datos de audio.
- *length*: Indica el tamaño de la pista en frames. Este tipo de medida no es usable por seres humanos, está orientado a ser manejable por el

ordenador. SDL proporciona una constante que permite convertir estos frames a segundos, una unidad mucho más amigable para nosotros. Esta constante es `CD_FPS`. Diviendo el número de frames entre esta constante tendremos la longitud de la pista en segundos. El proceso de convertir estos segundos a minutos es un proceso sencillo.

- *offset*: Indica la posición de inicio de la pista medida en frames. Este es el valor que utiliza el sistema para situarse en una determinada pista directamente.

8.5. Funciones del Subsistema de CD

Las funciones que proporciona SDL en el subsistema de CDROM se dividen, de nuevo, en dos grupos. Las que proporcionan información y las que se utilizan para reproducir los CD's de audio. Las funciones informativas son dos frente a las nueve que nos permiten interactuar con la unidad para manejarla.

8.5.1. Funciones Informativas

Para conocer información referente a las unidades de CD de las que dispone el sistema SDL proporciona diferentes funciones dependiendo del dato que queramos conocer. Antes de hacer nada con el subsistema de CD tenemos que saber de cuantas unidades dispone el sistema y sus respectivos nombres. Vamos a estudiar las funciones que nos permiten conocer esta información. Por supuesto, antes de utilizar cualquiera de estas funciones debemos de inicializar el subsistema de CD de SDL.

La primera función que vamos a presentar es:

```
int SDL_CDNumDrives(void);
```

Esta función no recibe nada como parámetro y devuelve el número de unidades de las que dispone el sistema. Con la información que nos proporciona esta función podemos llamar a la siguiente función:

```
const char *SDL_CDName(int drive);
```

A esta función le pasamos el número de unidad que queremos consultar y nos devuelve una cadena de caracteres con un identificador entendible por los humanos del CD-ROM que hemos consultado. El rango del parámetro *drive* va desde 0 hasta un valor menos que el valor devuelto por la función `int SDL_CDNumDrives(void)`.

8. Subsistema de CDROM

8.5.2. Ejemplo 1

Ya hemos visto todas las funciones y estructuras que proporcionan información en el subsistema de CDROM. Vamos a implementar un ejemplo que nos permita conocer la información de las unidades de CD que tenemos instaladas en nuestro ordenador.

```
1 ;// Ejemplo 1 - Subsistema de CDROM
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Información CD Device
5 ;
6 ;
7 ;#include <iostream>
8 ;#include <iomanip>
9 ;
10 ;#include <SDL/SDL.h>
11 ;
12 ;using namespace std;
13 ;
14 ;
15 ;
16 ;int main()
17 ;{
18 ;
19 ;    // Iniciamos el subsistema de video
20 ;
21 ;    if(SDL_Init(SDL_INIT_CDROM) < 0) {
22 ;
23 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
24 ;        exit(1);
25 ;
26 ;    }
27 ;
28 ;
29 ;    atexit(SDL_Quit);
30 ;
31 ;
32 ;    // Comprobamos el número de unidades de CD
33 ;
34 ;    int num_drives = SDL_CDNumDrives();
35 ;
36 ;    cout << "\nNúmero de unidades de CD: "
37 ;        << num_drives << endl;
38 ;
39 ;    // Mostramos el nombre de las unidades de CD
40 ;
41 ;    for(int i = 0; i < num_drives; i++) {
42 ;
43 ;        cout << "Unidad " << i << " nombre "
44 ;            << SDL_CDName(i) << endl;
45 ;
46 ;    }
```

8.5. Funciones del Subsistema de CD

```
47 ;  
48 ;  
49 ;}  
;
```

El ejemplo se limita a aplicar las funciones que hemos presentado antes. Empezamos inicializando el subsistema de CD mediante *SDL_Init()* especificando que queremos arrancar el subsistema de CD mediante *SDL_INIT_CDROM*. Seguidamente llamamos a la función *SDL_CDNumDrives()* que nos proporciona el número de unidades de CD disponibles en el sistema.

Lo siguiente que nos encontramos en el código es un bucle que va mostrando el nombre de las unidades de CD con ayuda de la función *SDL_CDName()*. La información variará dependiendo de la configuración del sistema.

8.5.3. Funciones para la reproducción del CD

Ya sabemos cuántas y cuales unidades de CD tenemos disponible en el sistema. También sabemos como abrir y cerrar las unidades de CD instaladas. Ahora nos toca reproducir un CD audio.

Para conocer el estado del CD-ROM SDL proporciona la siguiente función:

```
CDstatus SDL_CDStatus(SDL_CD *cdrom);
```

Como la función anterior necesita el puntero devuelto por la función *SD_Open*, que abría el dispositivo, como parámetro en ésta. Esta función nos devuelve el estado actual del CD_ROM. Como puedes intuir por el tipo devuelto, los posibles valores que puede devolver esta función son los mismos que puede tomar el campo *status* de la estructura *SDL_CD*.

Ahora, y gracias a esta función, tenemos suficiente información para conocer el contenido del CD que está introducido en nuestra unidad de CD.

8.5.4. Ejemplo 2

En este ejemplo vamos a mostrar toda la información disponible acerca del medio que tenemos insertado en nuestra unidad de CD.

```
;  
1 ;// Ejemplo 1 - Subsistema de CDROM  
2 ;//  
3 ;// Listado: main.cpp  
4 ;// Programa de pruebas. Muestra la información del CD introducido en la unidad.  
5 ;  
6 ;  
7 ;#include <iostream>  
8 ;#include <SDL/SDL.h>
```

8. Subsistema de CDROM

```
9 ;
10 ;using namespace std;
11 ;
12 ;
13 ;
14 ;int main()
15 ;{
16 ;
17 ;    // Iniciamos el subsistema de video
18 ;
19 ;    if(SDL_Init(SDL_INIT_CDROM) < 0) {
20 ;
21 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
22 ;        exit(1);
23 ;
24 ;    }
25 ;
26 ;
27 ;    atexit(SDL_Quit);
28 ;
29 ;    // Comprobamos el número de unidades de CD
30 ;
31 ;    if(SDL_CDNumDrives() < 1) {
32 ;
33 ;        cout << "Debe existir alguna unidad de CD conectada" << endl;
34 ;        exit(1);
35 ;
36 ;    }
37 ;
38 ;    SDL_CD *unidad = SDL_CDOpen(0);
39 ;
40 ;    // Mostramos la información
41 ;
42 ;    if(unidad->status != CD_TRAYEMPTY) {
43 ;
44 ;        cout << "\nID: " << unidad->id << endl;
45 ;        cout << "Número de pistas: " << unidad->numtracks << endl;
46 ;
47 ;        for(int i = 0; i < unidad->numtracks; i++) {
48 ;
49 ;            cout << "\n\tPista: " << unidad->track[i].id << endl;
50 ;            cout << "\tLongitud: " << unidad->track[i].id / CD_FPS << endl;
51 ;        }
52 ;
53 ;    } else {
54 ;
55 ;        cout << "\nDebe introducir un CD de Audio antes"
56 ;            << " de ejecutar esta aplicación. " << endl;
57 ;    }
58 ;
59 ;
60 ;
61 ;    SDL_CDClose(unidad);
```

8.5. Funciones del Subsistema de CD

```
62 ;  
63 ;     return 0;  
64 ;  
65 ;  
66 ;}
```

En este ejemplo sólo inicializamos el subsistema de CDROM. Si existe alguna unidad de CD (cosa que comprobamos con *SDL_CDNumDrives()*) abrimos la primera de ellas. El valor devuelto por la función que abre el dispositivo de CDROM lo utilizamos para mostrar toda la información disponible sobre el CD que tenemos insertado. En el caso de no existir ningún medio dentro de la unidad lo advertimos con un mensaje de consola.

Para terminar cerramos el dispositivo previamente abierto y terminamos con la ejecución del programa.

Veamos ahora las funciones necesarias para reproducir pistas de audio de un CD. La primera y fundamental es hacer que empiece a sonar el CD mediante la activación del “play”. El prototipo de la función que nos permite realizar esta tarea es:

```
int SDL_CDPlay(SDL_CD *cdrom, int start, int length);
```

Esta función recibe tres parámetros. Primero el puntero a *SDL_CD* que devuelve la función de apertura del CD. Comienza la reproducción en el frame indicado por el parámetro *start* durante los frames indicados por el parámetro *length*. Si por algún motivo no se puede realizar la reproducción la función devuelve -1, si se consigue realizar la reproducción la función devuelve el valor 0. Podemos obtener los parámetros *start* y *length* mirando la información de las pistas de audio del CD-ROM.

Si deseamos reproducir una pista o varias concretas nos es más cómodo utilizar la función:

```
int SDL_CDPlayTracks(SDL_CD *cdrom, int start_track, int start_frame,  
                      int ntracks, int nframes);
```

En el parámetro *start_track* indicamos cuál es la pista de inicio, mientras que en *ntracks* indicamos cuántas pistas queremos reproducir. Podemos especificar en qué frame dentro de la pista seleccionada queremos que empiece la reproducción a través del parámetro *start_frame* mientras que el parámetro *nframes* nos permite especificar cuántos frames de la última pista queremos reproducir. Si la función devuelve 0 es que la reproducción se llevó a cabo sin problemas. Si ocurrió algún error tomará el varlo 1.

Una vez reproducido el CD seguramente querramos pararlo. La función que nos ofrece SDL para realizar esta tarea es:

8. Subsistema de CDROM

```
int SDL_CDStop(SDL_CD *cdrom);
```

Esta función recibe como parámetro el puntero que nos proporcionó la función que habría el dispositivo. El valor devuelto por la misma es 0 si todo fue bien y -1 si no se consiguió parar la reproducción con éxito.

Sigamos estudiando las funciones que nos proporciona SDL para un control total de la reproducción de CD's:

```
int SDL_CDPause(SDL_CD *cdrom);
```

Esta función pausa la reproducción desde el CD. Recibe como parámetro el puntero que obteníamos de la función que abría el dispositivo. Como es habitual en esta librería devuelve 0 en caso de éxito y -1 si no se ha podido llevar a cabo el pause. La siguiente función es:

```
int SDL_CDResume(SDL_CD *cdrom);
```

Esta función es complementaria a la anterior. Se encarga de continuar la reproducción una vez pausada. Analogamente devuelve 0 en caso de conseguir continuar la reproducción y -1 si no lo consigue. La última función del subsistema de CD de SDL es:

```
int SDL_CD Eject(SDL_CD *cdrom);
```

Esta función si que realiza la apertura del dispositivo físicamente, excepto si la unidad en cuestión es del tipo conocido como slot-in. Se encarga de expulsar el CD que esté introducido en la unidad o, al menos, abrir la bandeja dónde se coloca este dispositivo. Como en las demás funciones, recibe como parámetro el puntero ue devuelve la función que abre lógicamente el dispositivo y retorna 0 en caso de éxito y -1 en caso de no poder haber realizado la acción.

8.5.5. Ejemplo 3

Vamos a crear un reproductor de CD que capture eventos del teclado para realizar las acciones sobre la unidad de CD. Será un ejemplo básico que puedes complementar con interfaz gráfica y con nuevas funciones. Vamos a ver el código:

```
1 ;// Ejemplo 1 - Subsistema de CD
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Controlando la reproducción
5 ;
6 ;
7 ;#include <iostream>
8 ;
9 ;#include <SDL/SDL.h>
```

8.5. Funciones del Subsistema de CD

```
10 ;
11 ;using namespace std;
12 ;
13 ;
14 ;
15 ;int main()
16 ;{
17 ;
18 ;    // Iniciamos el subsistema de video
19 ;
20 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_CDROM) < 0) {
21 ;
22 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
23 ;        exit(1);
24 ;
25 ;    }
26 ;
27 ;
28 ;    atexit(SDL_Quit);
29 ;
30 ;    // Comprobamos que sea compatible el modo de video
31 ;
32 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
33 ;
34 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
35 ;        exit(1);
36 ;
37 ;    }
38 ;
39 ;
40 ;    // Establecemos el modo de video
41 ;
42 ;    SDL_Surface *pantalla;
43 ;
44 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
45 ;
46 ;    if(pantalla == NULL) {
47 ;
48 ;        cerr << "No se pudo establecer el modo de video: "
49 ;            << SDL_GetError() << endl;
50 ;
51 ;        exit(1);
52 ;    }
53 ;
54 ;    // Comprobamos si existe alguna unidad de CD conectada
55 ;
56 ;    if(SDL_CDNumDrives() < 1) {
57 ;
58 ;        cout << "Debe existir alguna unidad de CD conectada" << endl;
59 ;        exit(1);
60 ;
61 ;    }
62 ;}
```

8. Subsistema de CDROM

```
63 ; // Si es así abrimos por omisión la primera
64 ;
65 ;     SDL_CD *unidad = SDL_CDOpen(0);
66 ;
67 ; // Mostramos la información
68 ;
69 ;     if(unidad->status != CD_TRAYEMPTY) {
70 ;
71 ;         cout << "\nID: " << unidad->id << endl;
72 ;         cout << "Número de pistas: " << unidad->numtracks << endl;
73 ;
74 ;         for(int i = 0; i < unidad->numtracks; i++) {
75 ;
76 ;             cout << "\n\tPista: " << unidad->track[i].id << endl;
77 ;             cout << "\tLongitud: " << unidad->track[i].length / CD_FPS << endl;
78 ;         }
79 ;
80 ;     } else {
81 ;
82 ;         // Si el dispositivo de CDROM está vacío
83 ;
84 ;         cout << "\nDebe introducir un CD de Audio antes"
85 ;             << " de ejecutar esta aplicación. " << endl;
86 ;     }
87 ;
88 ;
89 ; // Variables auxiliares
90 ;
91 ;     SDL_Event evento;
92 ;
93 ;     cout << "\nPulsa ESC para salir\n" << endl;
94 ;     cout << " s: Play \n t: Stop \n p: Pause - Resume \n e: Eject" << endl;
95 ;
96 ; // Bucle "infinito"
97 ;
98 ;     for( ; ; ) {
99 ;
100 ;         while(SDL_PollEvent(&evento)) {
101 ;
102 ;             if(evento.type == SDL_KEYDOWN) {
103 ;
104 ;                 if(evento.key.keysym.sym == SDLK_ESCAPE) {
105 ;
106 ;                     SDL_CDClose(unidad);
107 ;                     return 0;
108 ;
109 ;                 }
110 ;
111 ;                 if(evento.key.keysym.sym == SDLK_s) {
112 ;
113 ;                     SDL_CDPlayTracks(unidad, 0, 0,
114 ;                                     unidad->numtracks,
115 ;                                     unidad->track[unidad->numtracks].offset);
```

8.5. Funciones del Subsistema de CD

```
116 ;                }
117 ;
118 ;        if(evento.key.keysym.sym == SDLK_p) {
119 ;
120 ;            if(unidad->status == CD_PAUSED) {
121 ;
122 ;                SDL_CDRResume(unidad);
123 ;
124 ;            } else {
125 ;
126 ;                SDL_CDPause(unidad);
127 ;
128 ;            }
129 ;
130 ;        }
131 ;
132 ;        if(evento.key.keysym.sym == SDLK_t) {
133 ;
134 ;            SDL_CDStop(unidad);
135 ;        }
136 ;
137 ;        if(evento.key.keysym.sym == SDLK_e) {
138 ;
139 ;            SDL_CDEject(unidad);
140 ;
141 ;        }
142 ;
143 ;
144 ;
145 ;
146 ;
147 ;    }
148 ;
149 ;    if(evento.type == SDL_QUIT) {
150 ;
151 ;        SDL_CDClose(unidad);
152 ;        return 0;
153 ;    }
154 ;
155 ;}
156 ;}
157 ;
158 ;}
```

Como puedes ver en el código utilizamos todas las funciones que hemos presentado en este capítulo. Después de inicializar los subsistemas y establecer un modo de pantalla vemos si es posible abrir un dispositivo de CDROM. De ser así lo abrimos y sacamos por consola toda la información posible sobre el medio que tenemos en el dispositivo.

Una vez en el *game loop* definimos casos para distintas pulsaciones de tecla y una vez que se produzcan alguno de estos eventos haremos accionar alguna

8. Subsistema de CDROM

característica del subsistema de CDROM. Por ejemplo si pulsamos la tecla *e* expulsaremos el CD. Si pulsamos la tecla *s* el sistema comenzará a reproducir el CD... y así con todas las opciones que puedes ver en el código.

8.6. Recopilando

En este capítulo hemos aprendido a utilizar el CDROM como medio para reproducir música de un CD de audio en nuestra aplicación. Desde el comiendo inicializando el subsistema de CD hasta todos los aspectos de la reproducción, pasando por las estructuras y funciones que nos permiten sacar información de estos dispositivos.

Actualmente utilizar este medio para incluir sonido en una aplicación no es algo común ya que con los formatos comprimidos actualmente el espacio en disco ya no es un problema pero siempre es interesante poder barajar todas las alternativas.

Capítulo 9

Control del Tiempo

9.1. Introducción. Conocimientos Previos.

Si algo caracteriza a la época actual en lo que a los ordenadores se refiere es la variedad de máquinas que siguen funcionando actualmente. Ordenadores con 6 o 7 años de antiguedad siguen funcionando perfectamente para realizar tareas cotidianas funcionando a frecuencias mucho inferiores que las computadoras de última generación.

Esto puede provocar que la respuesta de nuestro videojuego no sea la adecuada. Podemos desarrollar nuestro juego en un sistema que no sea de última generación limitándonos a los recursos disponibles y ver como nuestra aplicación va excesivamente rápida en otro ordenador que posee unas características diferentes. Con respecto a los recursos del sistema, como buenos programadores, debemos de ser lo más eficiente que podamos ser con lo que conseguiremos que nuestro videojuego se pueda ejecutar en un mayor número de máquinas con un mejor resultado.

Para que la respuesta de nuestro juego sea parecida en distintos sistemas tenemos que ser capaces de controlar el tiempo y SDL nos ayuda a realizar esta tarea. Todos los ordenadores no funcionan a la misma velocidad y tenemos que establecer mecanismos que hagan que el comportamiento de nuestra aplicación sea cual sea la máquina en la que se ejecuta. Una de las principales características principales de SDL es la portabilidad que perderíamos si no controlamos estos aspectos.

SDL nos proporciona varias funciones para manejar el tiempo, por ejemplo para obtener el tiempo actual, esperar un intervalo de tiempo... todo para que consigamos obtener la respuesta deseada del sistema.

9.2. Objetivos

Los objetivos de este capítulo son:

9. Control del Tiempo

1. Conocer las funciones que nos permiten controlar el tiempo en SDL.
2. Manejar la técnica de espera activa para el control del tiempo.
3. Practicar la ejecución de acciones en intervalos de tiempo determinados.

9.3. Funciones para el manejo del tiempo

9.3.1. Marcas de tiempo

SDL proporciona varias funciones para el manejo y control del tiempo. Un aspecto fundamental es poder tomar una referencia de tiempo con respecto al comienzo de la ejecución del programa. Para realizar esta tarea SDL proporciona la función:

```
Uint32 SDL_GetTicks(void);
```

Esta función devuelve los milisegundos que pasan entre la inicialización de SDL y el momento actual, es decir el número de milisegundos que han transcurrido desde que se inicializó la librería SDL. Nos permite controlar el tiempo transcurrido entre dos instantes dados dentro del programa con lo que podemos controlar la velocidad del juego.

Podemos establecer que ciertas acciones deban ocurrir en ciertos momentos distantes de la iniciación de SDL en un número de segundos determinado.

9.3.1.1. Ejemplo 1

Vamos a ver mediante un ejemplo como utilizar esta función. Este realiza una cuenta de 10 segundos utilizando las funciones que nos ofrece la SDL. Veamos el código de la aplicación:

```
;  
1 ;// Ejemplo 1  
2 ;//  
3 ;// Listado: main.cpp  
4 ;// Programa de pruebas. Control del tiempo  
5 ;  
6 ;  
7 ;#include <iostream>  
8 ;#include <SDL/SDL.h>  
9 ;  
10 ;using namespace std;  
11 ;  
12 ;int main()  
13 ;{  
14 ;    // Iniciamos el subsistema de video  
15 ;
```

9.3. Funciones para el manejo del tiempo

```
16 ;     if(SDL_Init(SDL_INIT_VIDEO) < 0) {
17 ;
18 ;         cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
19 ;         exit(1);
20 ;
21 ;
22 ;     atexit(SDL_Quit);
23 ;
24 ;     // Tomamos una señal de tiempo
25 ;
26 ;     Uint32 tiempo_transcurrido;
27 ;
28 ;     tiempo_transcurrido = SDL_GetTicks();
29 ;
30 ;
31 ;     // Lo mostramos en segundos
32 ;
33 ;     cout << "El tiempo transcurrido es "
34 ;         << tiempo_transcurrido / 1000 << " segundos." << endl;
35 ;
36 ;     // Durante 10 segundos
37 ;
38 ;     unsigned int i = 0;
39 ;
40 ;     while(tiempo_transcurrido < 10000) {
41 ;
42 ;         // Una vez por segundo
43 ;
44 ;         if(tiempo_transcurrido > (i * 1000)) {
45 ;
46 ;             cout << "Han pasado " << i << " segundo(s)" << endl;
47 ;             i++;
48 ;
49 ;
50 ;             tiempo_transcurrido = SDL_GetTicks();
51 ;
52 ;
53 ;             cout << "La aplicación termina a " << SDL_GetTicks()
54 ;                 << " ms de haber empezado" << endl;
55 ;
56 ;         return 0;
57 ;     }
```

La implementación del ejemplo es bastante simple. Se trata de tomar una marca de tiempo real para mostrarla por pantalla. Como la resolución de *SDL_GetTicks()* es de milisegundos si queremos mostrar el resultado en segundos deberemos de dividir entre 1000 el valor devuelto por esta función. Utilizamos una variable de control *i* para, mediante una estructura selectiva, mostrar una vez por segundo el mensaje del instante en el que nos encontramos.

9. Control del Tiempo

9.3.2. Pausando el Tiempo

Hemos hablado muchas veces en este tutorial acerca del control del tiempo. Es importante que nuestra aplicación se ejecute con el mismo *tempo* en cualquier máquina que pueda soportarlo. Para esto tenemos que establecer un tiempo mínimo entre dos sucesos consecutivos. Un caso bastante claro es el de las animaciones. Supón que tenemos una animación de treinta segundos. Esta animación tiene que durar treinta segundos sea cual sea el ordenador en el que se ejecute siempre que tenga suficiente potencia para soportarla. Para conseguir esto tendremos que establecer un *timing*, un tiempo mínimo de espera, para cada acción que se realice.

Si el tiempo entre dos sucesos no es el deseado SDL proporciona una función que nos permite detener la aplicación durante un tiempo determinado expresado en milisegundos para, seguidamente, continuar con la ejecución del programa. El prototipo de esta función es:

```
void SDL_Delay(Uint32 ms);
```

La función recibe como parámetro un entero de 32 bits que indica el tiempo a esperar. No es más que eso, una espera, es indicarle a la aplicación que no haga nada durante un tiempo determinado. El valor del error de esta espera depende del sistema operativo. Como regla general la media del error es de unos 10 ms que es más que suficiente para conseguir un resultado que se aproxime al deseado.

9.3.2.1. Ejemplo 2

Vamos a realizar un ejemplo que permita realizar una determinada acción después de un determinado tiempo entre acciones, sea cual sea la máquina en que se ejecute siempre que cumpla con unos requisitos mínimos que le permita lanzar la aplicación.

```
;  
1 // Ejemplo 2  
2 //  
3 // Listado: main.cpp  
4 // Programa de pruebas. Control del tiempo  
5 ;  
6 ;  
7 #include <iostream>  
8 #include <SDL/SDL.h>  
9 ;  
10 using namespace std;  
11 ;  
12 int main()  
13 {
```

9.3. Funciones para el manejo del tiempo

```
14 ; // Iniciamos el subsistema de video
15 ;
16 ; if(SDL_Init(SDL_INIT_VIDEO) < 0) {
17 ;
18 ;     cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
19 ;     exit(1);
20 ;
21 ;
22 ;     atexit(SDL_Quit);
23 ;
24 ; // Comprobamos que sea compatible el modo de video
25 ;
26 ; if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
27 ;
28 ;     cerr << "Modo no soportado: " << SDL_GetError() << endl;
29 ;     exit(1);
30 ;
31 ;
32 ;
33 ; // Antes de establecer el modo de video
34 ; // Establecemos el nombre de la ventana
35 ;
36 ;     SDL_WM_SetCaption("Ejemplo 2", NULL);
37 ;
38 ;     SDL_Surface *pantalla;
39 ;
40 ;     pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
41 ;
42 ;     if(pantalla == NULL) {
43 ;
44 ;         cerr << "No se pudo establecer el modo de video: "
45 ;             << SDL_GetError();
46 ;
47 ;         exit(1);
48 ;
49 ;
50 ;         cout << "Pulsa ESC para terminar. " << endl;
51 ;         cout << "Pulsa f para cambiar a pantalla completa. " << endl;
52 ;
53 ; // Variables auxiliares
54 ;
55 ;     SDL_Event evento;
56 ;     SDL_Surface *imagen;
57 ;
58 ;     SDL_Rect destino;
59 ;
60 ; // Inicializamos la variable de posición y tamaño de destino
61 ; // Para la imagen que vamos a cargar
62 ;
63 ;     destino.x = 150;
64 ;     destino.y = 150;
65 ;     destino.w = imagen->w;
66 ;     destino.h = imagen->h;
```

9. Control del Tiempo

```
67 ;
68 ;    // Bucle infinito
69 ;
70 ;    for( ; ; ) {
71 ;
72 ;        cout << "Esperamos un segundo" << endl;
73 ;
74 ;        SDL_Delay(1000);
75 ;
76 ;
77 ;        // Cargagamos un bmp en la superficie
78 ;        // para realizar las pruebas
79 ;
80 ;        imagen = SDL_LoadBMP("Imagenes/ajuste.bmp");
81 ;
82 ;        if(imagen == NULL) {
83 ;            cerr << "No se puede cargar la imagen: "
84 ;                << SDL_GetError() << endl;
85 ;                exit(1);
86 ;        }
87 ;
88 ;        // Blit a la superficie principal
89 ;
90 ;        SDL_BlitSurface(imagen, NULL, pantalla, &destino);
91 ;
92 ;        // Actualizamos la pantalla
93 ;
94 ;        SDL_Flip(pantalla);
95 ;
96 ;
97 ;        // Esperamos que pase un segundo
98 ;
99 ;        cout << "Esperamos un segundo" << endl;
100 ;
101 ;        SDL_Delay(1000);
102 ;
103 ;        // Cargamos otra imagen en la misma superficie
104 ;
105 ;        imagen = SDL_LoadBMP("Imagenes/ajuste2.bmp");
106 ;
107 ;        if(imagen == NULL) {
108 ;
109 ;            cerr << "No se puede cargar la imagen: "
110 ;                << SDL_GetError() << endl;
111 ;                exit(1);
112 ;
113 ;        }
114 ;
115 ;        // Blit a la superficie principal
116 ;
117 ;        SDL_BlitSurface(imagen, NULL, pantalla, &destino);
118 ;
119 ;        // Actualizamos la pantalla
```

9.3. Funciones para el manejo del tiempo

```
120 ;
121 ;      SDL_Flip(pantalla);
122 ;
123 ;      while(SDL_PollEvent(&evento)) {
124 ;
125 ;          if(evento.type == SDL_KEYDOWN) {
126 ;
127 ;              if(evento.key.keysym.sym == SDLK_ESCAPE) {
128 ;
129 ;                  SDL_FreeSurface(imagen);
130 ;                  return 0;
131 ;
132 ;              if(evento.key.keysym.sym == SDLK_f) {
133 ;
134 ;                  // Si pulsamos f pasamos a pantalla completa
135 ;
136 ;                  if(!SDL_WM_ToggleFullScreen(pantalla))
137 ;
138 ;                      cerr << "No se puede pasar a pantalla completa."
139 ;                           << endl;
140 ;
141 ;
142 ;              }
143 ;
144 ;          if(evento.type == SDL_QUIT)
145 ;
146 ;              return 0;
147 ;
148 ;      }
149 ;}
```

En este ejemplo vemos como se va intercambiando una imagen. Una vez colocamos una de las imágenes esperamos un segundo y colocamos la siguiente y vuelta a empezar hasta que el usuario pulse la tecla ESC.

9.3.3. Ejercicio 1

Vamos a realizar un pequeño ejercicio conceptual que marcará una forma de actuar con el tiempo en nuestras aplicaciones. Vamos a implementar una función que haciendo uso de *SDL_GetTicks()* que emule el comportamiento de *SDL_Delay()*. Haz un programa de prueba que te permita comprobar el funcionamiento de dicha función. Aquí tienes el resultado para este ejercicio:

```
1 ;// Ejercicio 1
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Control del tiempo
5 ;// Emulación de SDL_Delay()
6 ;
7 ;
```

9. Control del Tiempo

```
8 ;#include <iostream>
9 ;#include <SDL/SDL.h>
10 ;
11 ;using namespace std;
12 ;
13 ;void emuSDL_Delay(Uint32 ms);
14 ;
15 ;int main()
16 ;{
17 ;    // Iniciamos el subsistema de video
18 ;
19 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
20 ;
21 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
22 ;        exit(1);
23 ;    }
24 ;
25 ;    atexit(SDL_Quit);
26 ;
27 ;    int esperar;
28 ;
29 ;    cout << "Introduzca el tiempo que desea esperar (ms): ";
30 ;    cin >> esperar;
31 ;
32 ;    int inicio = SDL_GetTicks();
33 ;
34 ;    cout << "Referencia INICIAL: " << inicio << endl;
35 ;
36 ;    emuSDL_Delay(esperar);
37 ;
38 ;    int final = SDL_GetTicks();
39 ;
40 ;    // Información para comprobar la eficacia
41 ;    // de la función implementada
42 ;
43 ;    cout << "Referencia FIN: " << final << endl;
44 ;    cout << "Ha esperado " << (final - inicio) << " ms" << endl;
45 ;    cout << "Se ha producido un error de "
46 ;        << esperar - (final - inicio) << " ms" << endl;
47 ;
48 ;
49 ;    return 0;
50 ;}
51 ;
52 ;
53 ;// Esta función provoca la pausa de la aplicación
54 ;// durante el tiempo pasado como parámetro
55 ;
56 ;void emuSDL_Delay(Uint32 ms) {
57 ;
58 ;    Uint32 referencia, actual;
59 ;
60 ;    referencia = SDL_GetTicks();
```

```

61 ;
62 ;    do {
63 ;
64 ;        actual = SDL_GetTicks();
65 ;
66 ;    } while(ms > (actual - referencia));
67 ;}
;
```

Para simular el comportamiento de *SDL_Delay()* hemos implementado una función que espera un intervalo de tiempo pasado como parámetro. En dicha función hay un bucle que se ejecuta un número de veces determinado por dos marcas de tiempo. La primera tiene un tiempo de referencia, cuando se llamó a la función. La segunda va variando según pasa el tiempo. Si la diferencia entre la primera y la segunda es mayor que el tiempo que queríamos esperar rompe dicho bucle saliendo de la función.

Como puedes ver no es complicado manejar el tiempo con ayuda de SDL. No es un aspecto que se complique en demasiado. Es fundamental tener un buen diseño que nos permita establecer los tiempos correctamente. El aplicar la técnica es bastante

9.4. Timers

9.4.1. Introducción

Existen numerosas aplicaciones que necesitan programar tareas para que se ejecuten en cierto momento. Otras nos libran de hacer tareas repetitivas programando una determinada acción cada cierto tiempo. Imagina que necesitamos que nuestra aplicación realice alguna acción cada cierto tiempo. SDL nos ofrece la posibilidad de programar un tipo de dato conocido como *timers* con este fin.

9.4.2. Añadiendo un temporizador

SDL nos permite añadir a nuestra aplicación temporizadores que ejecuten una función de tipo *callback* cada cierto tiempo. Estos temporizadores son conocido como *timers*.

Para poder utilizar los timers en SDL tenemos que iniciar este subsistema. Cuando realicemos la llamada a la función *SDL_Init()* con el parámetro *SDL_INIT_TIMER*.

La función que nos permite realizar esta tarea tiene el siguiente prototipo:

```
SDL_TimerID SDL_AddTimer(Uint32 interval, SDL_NewTimerCallback  
                                callback, void *param);
```

9. Control del Tiempo

El primer parámetro indica el intervalo al que se ejecutará nuestra función expresado en milisegundos. El segundo parámetro que debemos de pasar es el referente a la función *callback* que queremos que se ejecute a intervalos regulares de tiempo. En último lugar tenemos los parámetros de la función callback. El parámetro *callback* del tipo *SDL_NewTimerCallback* viene dado por la siguiente definición de tipo:

```
typedef Uint32 (*SDL_NewTimerCallback)(Uint32 interval, void *param);
```

La función *callback* es ejecutada en un hilo o thread diferente al hilo principal de ejecución por lo que no debe llamar desde la función callback del timer a funciones del programa principal en ejecución

La resolución del timer es, como la de todos los elementos temporizadores en SDL, de 10 ms. Si especificamos un intervalo de 23 ms, la función callback tardará en ejecutarse aproximadamente 30 ms después que haya sido llamada. Como puedes ver la función devuelve el identificador del temporizador o *timer* agregado, o bien, **NULL** si ha ocurrido un error durante la creación de dicho *timer*.

9.4.3. Eliminando un temporizador

De la misma manera que podemos necesitar programar una acción para que se ejecute en un intervalo de tiempo puede que necesitemos eliminar dicha temporización una vez que haya cumplido con su misión. SDL nos proporciona las herramientas necesarias para llevar acabo esta operación.

Para eliminar un timer deberemos utilizar la funciones SDL:

```
SDL_bool SDL_RemoveTimer(SDL_TimerID id);
```

Esta función recibe como parámetro el identificador del temporizador a eliminar que previamente fue obtenido mediante *SDL_AddTimer()*. Esta función devuelve un valor booleano que indica si la operación se realizó efectivamente o no.

9.4.4. Modificando el temporizador

Una vez establecido un *timer* podemos necesitar cambiar alguna de sus propiedades. Las propiedades de un temporizador son dos: el intervalo de tiempo y la función de tipo *callback* a ejecutar en dicho intervalo de tiempo. Para realizar esta tarea SDL proporciona la función *SDL_SetTimer()* que es capaz de cambiar el intervalo de tiempo de ejecución y que incluso puede cambiar la función *callback* que ejecuta. El prototipo de la función es el siguiente:

```
int SDL_SetTimer(Uint32 interval, SDL_TimerCallback callback);
```

En el primer parámetro indicaremos el nuevo intervalo de tiempo y el segundo parámetro, como puedes ver, nos permite especificar la nueva función callback siempre y cuando queramos cambiarla.

El tipo *SDL_TimerCallback* está definido como:

```
typedef Uint32 (*SDL_TimerCallback)(Uint32 interval);
```

¿Cómo eliminarías un *timer* de tu aplicación haciendo uso de esta función?

Una forma ocurrente de cancelar un timer en ejecución es realizar la siguiente llamada *SDL_SetTimer(0, NULL)*.

9.4.5. Ejemplo 3

Vamos a practicar un poco con el concepto de los *timers*. Vamos a desarrollar una pequeña aplicación que nos permita repetir activar la temporización de dos temporizadores previamente configurados. Uno de ellos se ejecutará al segundo de pulsarse la tecla “t” mientras que el otro lo hará al segundo y medio.

Para realizar este ejercicio tienes que definir dos funciones de tipo “callback” compatibles con los temporizadores. Este ejemplo te puede ser muy útil cuando quieras retrasar la respuesta a una acción hasta un momento determinado sin que pare la ejecución del programa.

Aquí tienes el código de la aplicación:

```
1 ;// Ejemplo 3
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Control del tiempo: TIMERS
5 ;
6 ;
7 ;#include <iostream>
8 ;#include <SDL/SDL.h>
9 ;
10 ;using namespace std;
11 ;
12 ;Uint32 Funcion_Callback(Uint32 intervalo, void *parametros);
13 ;Uint32 Funcion_Callback2(Uint32 intervalo);
14 ;
15 ;int main()
16 ;{
17 ;    // Iniciamos el subsistema de video
18 ;
19 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER) < 0) {
20 ;
21 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
22 ;        exit(1);
23 ;    }
24 ;
25 ;    atexit(SDL_Quit);
```

9. Control del Tiempo

```
26 ;
27 ;    // Comprobamos que sea compatible el modo de video
28 ;
29 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
30 ;
31 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
32 ;        exit(1);
33 ;
34 ;    }
35 ;
36 ;    // Antes de establecer el modo de video
37 ;    // Establecemos el nombre de la ventana
38 ;
39 ;    SDL_WM_SetCaption("Ejemplo 3", NULL);
40 ;
41 ;    // Establecemos el modo
42 ;
43 ;    SDL_Surface *pantalla;
44 ;
45 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
46 ;
47 ;    if(pantalla == NULL) {
48 ;
49 ;        cerr << "No se pudo establecer el modo de video: "
50 ;            << SDL_GetError();
51 ;
52 ;        exit(1);
53 ;
54 ;    }
55 ;    // Añadimos un Timer
56 ;
57 ;    SDL_TimerID mytimer;
58 ;
59 ;    mytimer = SDL_AddTimer(1000, Funcion_Callback, pantalla);
60 ;
61 ;    if(mytimer == NULL) {
62 ;
63 ;        cerr << "No se pudo establecer el timer: "
64 ;            << SDL_GetError();
65 ;
66 ;        exit(1);
67 ;
68 ;
69 ;    // Información en pantalla
70 ;
71 ;    cout << "Pulsa ESC para terminar. " << endl;
72 ;    cout << "Pulsa f para cambiar a pantalla completa. " << endl;
73 ;    cout << "Pulsa t para establecer un nuevos timer en 1 y 1,5 seg" << endl;
74 ;
75 ;    // Variables auxiliares
76 ;
77 ;    SDL_Event evento;
78 ;
```

```
79 ;
80 ;    // Bucle infinito
81 ;
82 ;    for( ; ; ) {
83 ;
84 ;        while(SDL_PollEvent(&evento)) {
85 ;
86 ;            if(evento.type == SDL_KEYDOWN) {
87 ;
88 ;                if(evento.key.keysym.sym == SDLK_ESCAPE) {
89 ;
90 ;                    // Eliminamos el timer al salir
91 ;
92 ;                    SDL_RemoveTimer(mytimer);
93 ;
94 ;                    cout << "Timer eliminado" << endl;
95 ;
96 ;                    return 0;
97 ;                }
98 ;
99 ;            if(evento.key.keysym.sym == SDLK_f) {
100 ;
101 ;                // Si pulsamos f pasamos a pantalla completa
102 ;
103 ;                if(!SDL_WM_ToggleFullScreen(pantalla))
104 ;
105 ;                    cerr << "No se puede pasar a pantalla completa."
106 ;                                << endl;
107 ;            }
108 ;
109 ;            if(evento.key.keysym.sym == SDLK_t) {
110 ;
111 ;                // Un nuevo timer en 1 segundos
112 ;
113 ;                SDL_SetTimer(1000, Funcion_Callback2);
114 ;
115 ;                // Repetimos el a los 1,5 segundos
116 ;
117 ;                mytimer = SDL_AddTimer(1500, Funcion_Callback, pantalla);
118 ;            }
119 ;
120 ;        }
121 ;    }
122 ;}
123 ;}
124 ;
125 ;
126 ;Uint32 Funcion_Callback(Uint32 intervalo, void *parametros) {
127 ;
128 ;
129 ;    cout << "Entro en el callback" << endl;
130 ;
131 ;    SDL_Rect destino;
```

9. Control del Tiempo

```
132 ;
133 ;     SDL_Surface *imagen;
134 ;
135 ;     // Cargamos un bmp en la superficie
136 ;     // para realizar las pruebas
137 ;
138 ;     imagen = SDL_LoadBMP("Imagenes/ajuste.bmp");
139 ;
140 ;     // Inicializamos la variable de posición y tamaño de destino
141 ;     // Para la imagen que vamos a cargar
142 ;
143 ;     destino.x = 150;
144 ;     destino.y = 150;
145 ;     destino.w = imagen->w;
146 ;     destino.h = imagen->h;
147 ;
148 ;     if(imagen == NULL) {
149 ;         cerr << "No se puede cargar la imagen: "
150 ;             << SDL_GetError() << endl;
151 ;         exit(1);
152 ;     }
153 ;
154 ;     // Blit a la superficie principal
155 ;
156 ;     SDL_BlitSurface(imagen, NULL, (SDL_Surface *) parametros, &destino);
157 ;
158 ;     SDL_Delay(intervalo);
159 ;
160 ;     // Actualizamos la pantalla
161 ;
162 ;     SDL_Flip((SDL_Surface *) parametros);
163 ;
164 ;
165 ;     // Cargamos otra imagen en la misma superficie
166 ;
167 ;     imagen = SDL_LoadBMP("Imagenes/ajuste2.bmp");
168 ;
169 ;     if(imagen == NULL) {
170 ;
171 ;         cerr << "No se puede cargar la imagen: "
172 ;             << SDL_GetError() << endl;
173 ;         exit(1);
174 ;
175 ;     }
176 ;
177 ;     // Blit a la superficie principal
178 ;
179 ;     SDL_BlitSurface(imagen, NULL, (SDL_Surface *) parametros, &destino);
180 ;
181 ;     SDL_Delay(intervalo);
182 ;
183 ;     // Actualizamos la pantalla
184 ;
```

```
185 ;         SDL_Flip((SDL_Surface *) parametros);
186 ;
187 ;
188 ;
189 ;     return 0;
190 ;
191 ;}
192 ;
193 ;Uint32 Funcion_Callback2(Uint32 intervalo) {
194 ;
195 ;    cout << "Entro en función callback 2 (1 seg de demora)" << endl;
196 ;
197 ;    return 0;
198 ;
199 ;}
```

9.5. Recopilando

En este capítulo hemos aprendido las técnicas básicas para controlar el tiempo haciendo uso de SDL como herramienta principal. Es fundamental que domines este aspecto para que tu aplicación tenga un comportamiento bien definido para cualquier máquina que pueda soportar su ejecución.

La programación de tareas para realizar acciones después de un intervalo es muy sencilla con SDL. Hemos presentado la manera de utilizarla para que puedas familiarizarte con ella en tus aplicaciones.

9. Control del Tiempo

Capítulo 10

Gestor de Ventanas

10.1. Introducción

En la actualidad la mayoría de sistemas operativos poseen una interfaz “amigable” para trabajar con tu ordenador. Esta interfaz está soportada por un gestor de ventanas o *windows manager* que permite a los programadores a través de una API desarrollar aplicaciones para un gestor de ventanas concreto.

Este sistema gestor nos permite realizar acciones cotidianas como cambiar el tamaño de las ventanas, minimizarlas, copiar y pegar entre aplicaciones... permite mostrar un pequeño ícono en la esquina superior izquierda de la aplicación y más...

SDL proporciona la capacidad de ser compatible con varios sistemas, tanto poseedores de un windows manager o gestor de ventanas como de aquellos que no lo posean. Por esto proporciona unas funciones que, independientemente del sistema operativo que estemos corriendo y del gestor de ventanas que este utilice, podamos manejar algunas capacidades de este gestor. Existen ciertas incompatibilidades en alguna de las funciones que veremos en el desarrollo del temario. Desde el grupo que sigue el desarrollo de esta librería se trabaja para solucionar estos problemas.

En el caso de (C)Microsoft Windows el sistema gestor de ventanas es único y está integrado en el sistema operativo y varía según la versión de éste. En otros sistemas como Linux podemos optar por varios windows managers. Los más comunes son KDE o GNOME aunque cuando la disponibilidad de recursos escasea se opta por entornos menos pesados como XFCE o MWM.

10.2. Objetivos

Los objetivos de este capítulo son:

1. Conocer las funciones que proporciona SDL para interactuar con el gestor

10. Gestor de Ventanas

de ventanas.

2. Manejar las funciones que nos permiten personalizar la ventana de nuestra aplicación.
3. Gestionar los eventos producidos por el gestor de ventanas.

10.3. Funciones para el manejo de ventanas

SDL proporciona una serie de funciones que nos permiten interactuar con el gestor de ventanas. Varios son los aspectos que podemos configurar de una ventana. Vamos a presentar cuáles son estos elementos y las funciones que nos permiten personalizar dichos elementos en nuestra aplicación.

Veamos las funciones que nos permiten interactuar con el gestor de ventanas.

10.3.1. El Título

Una habitual y sana costumbre, que proporciona un aspecto profesional a nuestro trabajo, es renombrar la ventana donde ejecutamos la aplicación con el nombre final que le vayamos a dar a nuestro programa, independientemente del nombre del archivo ejecutable del programa. Para esto SDL proporciona la siguiente función:

```
void SDL_WM_SetCaption(const char *title, const char *icon);
```

Esta función recibe como parámetros el título que queremos aplicar a nuestra ventana así como la ruta y nombre del fichero que contiene el ícono que queremos colocar en la misma. En (C)Microsoft Windows este segundo parámetro no se utiliza ya que no funciona correctamente. Frecuentemente utilizaremos el valor `NULL` en el segundo parámetro de esta función para indicar que no queremos especificar ícono para esa ventana en dicho momento. Esta función debe de ser llamada después de inicializar SDL, claro está. Este título es muy importante en un entorno de ventanas ya que nos permitirá saber si nuestra ventana es la que está produciendo un error y nos facilitará la búsqueda de la ventana cuando le queramos devolver el foco.

Si necesitamos consultar dichos valores podemos utilizar la siguiente función:

```
void SDL_WM_GetCaption(char **title, char **icon);
```

Que devuelve por referencia en valor del título en el parámetro `title` y la ruta dónde se almacena el ícono que se muestra en la esquina superior izquierda de la pantalla en la variable `icon`.

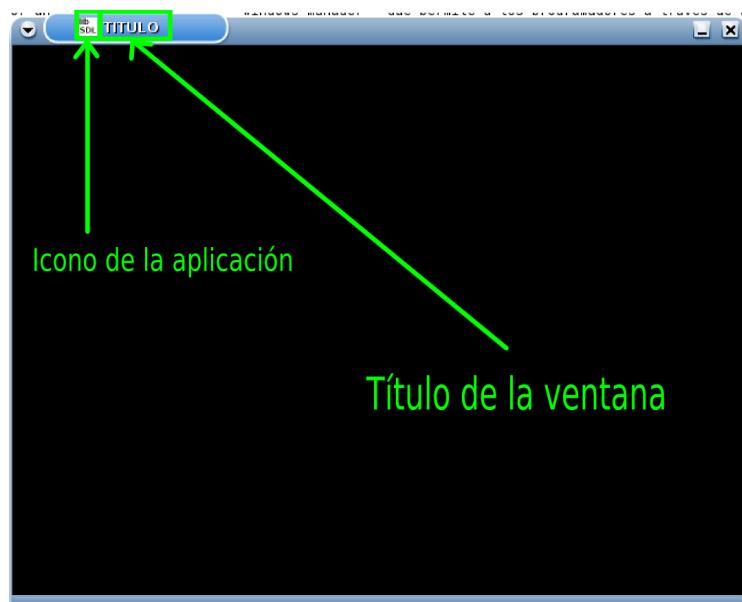


Figura 10.1: Área de título de la ventana

10.3.2. Icono

El ícono de la aplicación tiene que ser un aspecto descriptivo de la misma. Tenemos la posibilidad de incluirlo en la ventana de nuestra aplicación como elemento descriptivo de la misma lo que dotará a la misma de un aspecto más provisional. Es común encontrar aplicaciones generadas con herramientas rápidas para el desarrollo en las que no se ha tenido el cuidado de cambiar el ícono predeterminado que coloca dicha herramienta.

Aunque esta aplicación haya sido muy trabajada seguramente cualquier conocedor de dicha herramienta percibirá una primera impresión de descuido en el desarrollo de la misma lo que no es bueno en el mundo de la programación. La importancia del ícono es importante por el tema de la apariencia pero no es fundamental en el desarrollo de la misma, claro está.

Para establecer el ícono de aplicación en Microsoft Windows(C), o bien en un momento diferente que al establecer el título de la misma para cualquier sistema, utilizaremos la siguiente función:

```
void SDL_WM_SetIcon(SDL_Surface *icon, Uint8 *mask);
```

El ícono debe de tener una resolución de 32x32 píxeles para el sistema operativo Windows. El primer parámetro que recibe la función es la superficie donde hemos cargado la imagen, mientras que el segundo es una máscara que especifica la forma del ícono y sus transparencias. Esta máscara debe estar en formato MSB. Si a *mask* le pasamos el valor NULL las transparencias vendrán

10. Gestor de Ventanas

definidas por el color definido como *color key*.

Hay que tener presente que esta función debe de ser llamada antes de establecer el modo de video con *SDL_SetVideoMode()* si no no se establecerá el modo de video.

10.3.3. Ejemplo 1

Vamos a modificar uno de nuestros ejemplos anteriores añadiéndole un título personalizado a la ventana donde ejecutamos dicho ejemplo. También vamos a añadir un ícono personalizado a dicha ventana. Este ícono, para conseguir un resultado óptimo, debe de ser de 32 x 32 píxel y ser bastante claro.

```
1 ;// Ejemplo 1
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Añadimos a la ventana de nuestra aplicación
5 ;// un título e ícono personalizado
6 ;
7 ;
8 ;#include <iostream>
9 ;#include <SDL/SDL.h>
10 ;
11 ;using namespace std;
12 ;
13 ;int main()
14 ;{
15 ;
16 ;    // Iniciamos el subsistema de video
17 ;
18 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
19 ;
20 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
21 ;        exit(1);
22 ;    }
23 ;
24 ;    atexit(SDL_Quit);
25 ;
26 ;    // Comprobamos que sea compatible el modo de video
27 ;
28 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
29 ;
30 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
31 ;        exit(1);
32 ;
33 ;    }
34 ;
35 ;
36 ;    // Antes de establecer el modo de video
37 ;    // Guardamos el nombre de la ventana y
```

10.3. Funciones para el manejo de ventanas

```
38 ; // cargaamos el icono
39 ;
40 ; char nombre_ventana[20];
41 ;
42 ; // Preguntamos el nombre para la ventana por consola
43 ;
44 ; cout << "Introduzca una palabra para el nombre de la ventana: ";
45 ; cin >> nombre_ventana;
46 ;
47 ; // Cargamos la imagen del icono
48 ;
49 ; SDL_Surface *icono;
50 ;
51 ; icono = SDL_LoadBMP("./Imagenes/icono.bmp");
52 ;
53 ; if(icono == NULL) {
54 ;
55 ;     cerr << "No se puede cargar el icono "
56 ;         << SDL_GetError() << endl;
57 ;     exit(1);
58 ;
59 ;
60 ;
61 ; // Establecemos el nombre de la ventana y el icono
62 ;
63 ; SDL_WM_SetCaption(nombre_ventana, NULL);
64 ; SDL_WM_SetIcon(icono, NULL); // Compatible con MS Windows
65 ;
66 ; // Establecemos el modo de video
67 ;
68 ; SDL_Surface *pantalla;
69 ;
70 ; pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
71 ;
72 ; if(pantalla == NULL) {
73 ;
74 ;     cerr << "No se pudo establecer el modo de video: "
75 ;         << SDL_GetError();
76 ;
77 ;     exit(1);
78 ;
79 ;
80 ; cout << "Pulsa ESC para terminar. " << endl;
81 ;
82 ; // Variables auxiliares
83 ;
84 ; SDL_Event evento;
85 ;
86 ; // Bucle infinito
87 ;
88 ; for( ; ; ) {
89 ;
90 ;     while(SDL_PollEvent(&evento)) {
```

10. Gestor de Ventanas

```
91 ;  
92 ;           if(evento.type == SDL_KEYDOWN) {  
93 ;  
94 ;               if(evento.key.keysym.sym == SDLK_ESCAPE)  
95 ;                   return 0;  
96 ;               }  
97 ;           }  
98 ;       }  
99 ;   }  
100 ;}  
101 ;};
```

Vamos a explicar lo novedoso del ejemplo. Lo primero que hacemos una vez inicializada SDL es pedir una palabra para que sea colocada en el título de la ventana. Hemos limitado el tamaño de la misma a 20 caracteres ya que lo consideramos un tamaño razonable.

Cargamos el ícono a mostrar en la ventana en una variable de tipo superficie. Una vez que tenemos todos los datos listos pasamos a realizar la llamada a las funciones *SDL_WM_SetCaption()* y *SDL_WM_SetIcon()* que establecen el título y el ícono de la ventana respectivamente.

Una vez realizadas las llamadas a estas funciones establecemos el modo de video para así mostrar la ventana de nuestra aplicación. El resto del código se destina a capturar el evento que provoca la terminación de dicha aplicación.

10.3.4. Minimizando la ventana

Una de las tareas que nos facilita SDL es la de minimizar la aplicación mientras se ejecuta. No suele ser un aspecto interesante a la hora de desarrollar un videojuego pero está presente por si lo necesitamos en algún momento. La función que realiza esta tarea en SDL es:

```
int SDL_WM_IconifyWindow(void);
```

Si no se realiza con éxito el minimizado la función devolverá el valor 0, y en caso de minimizar la ventana devolverá un valor distinto de 0. Una vez devuelto este valor la aplicación recibirá y activará el evento *SDL_APPACTIVE*.

10.3.5. Maximizando la ventana

Existe una función que nos permite alternar entre modo ventana y modo a pantalla completa. El prototipo de dicha función es:

```
int SDL_WM_ToggleFullScreen(SDL_Surface *surface);
```

10.3. Funciones para el manejo de ventanas

Como parámetro debemos pasarle el puntero de la superficie principal de la pantalla que obtuvimos mediante *SDL_SetVideoMode()*. En nuestros ejemplos a este nombre le solemos asignar el nombre de pantalla. Esta función devuelve 1 si el cambio es exitoso y 0 en caso de fallo.

10.3.6. Ejemplo 2

Vamos a añadir estas funciones a nuestro ejemplo anterior y así dotarlo de una mayor funcionalidad. Aquí tienes el listado:

```
1 ;// Ejemplo 2
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Añadimos a la ventana de nuestra aplicación
5 ;// un título e ícono personalizado
6 ;
7 ;
8 ;#include <iostream>
9 ;#include <SDL/SDL.h>
10 ;
11 ;using namespace std;
12 ;
13 ;int main()
14 ;{
15 ;
16 ;    // Iniciamos el subsistema de video
17 ;
18 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
19 ;
20 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
21 ;        exit(1);
22 ;    }
23 ;
24 ;    atexit(SDL_Quit);
25 ;
26 ;    // Comprobamos que sea compatible el modo de video
27 ;
28 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
29 ;
30 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
31 ;        exit(1);
32 ;
33 ;    }
34 ;
35 ;
36 ;    // Antes de establecer el modo de video
37 ;    // Establecemos el ícono y el nombre de la ventana
38 ;
39 ;    // Cargamos la imagen del ícono
40 ;
41 ;    SDL_Surface *icono;
```

10. Gestor de Ventanas

```
42 ;  
43 ;     icono = SDL_LoadBMP("./Imagenes/icono.bmp");  
44 ;  
45 ;     if(icono == NULL) {  
46 ;  
47 ;         cerr << "No se puede cargar el ícono "  
48 ;             << SDL_GetError() << endl;  
49 ;         exit(1);  
50 ;     }  
51 ;  
52 ;  
53 ;     // Establecemos el nombre de la ventana y el ícono  
54 ;  
55 ;     SDL_WM_SetCaption("Prueba", NULL);  
56 ;     SDL_WM_SetIcon(icono, NULL); // Compatible con MS Windows  
57 ;  
58 ;     // Establecemos el modo de video  
59 ;  
60 ;     SDL_Surface *pantalla;  
61 ;  
62 ;     pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);  
63 ;  
64 ;     if(pantalla == NULL) {  
65 ;  
66 ;         cerr << "No se pudo establecer el modo de video: "  
67 ;             << SDL_GetError();  
68 ;  
69 ;         exit(1);  
70 ;     }  
71 ;  
72 ;     cout << "Pulsa ESC para terminar. " << endl;  
73 ;     cout << "Pulsa m para minimizar. " << endl;  
74 ;     cout << "Pulsa f para cambiar a pantalla completa. " << endl;  
75 ;  
76 ;     // Variables auxiliares  
77 ;  
78 ;     SDL_Event evento;  
79 ;  
80 ;     // Bucle infinito  
81 ;  
82 ;     for( ; ; ) {  
83 ;  
84 ;         while(SDL_PollEvent(&evento)) {  
85 ;  
86 ;             if(evento.type == SDL_KEYDOWN) {  
87 ;  
88 ;                 if(evento.key.keysym.sym == SDLK_ESCAPE)  
89 ;                     return 0;  
90 ;  
91 ;                 if(evento.key.keysym.sym == SDLK_m) {  
92 ;  
93 ;                     // Si pulsamos m, minimizamos  
94 ;
```

10.4. Convertir la entrada de usuario en exclusiva

```
95 ;           if(!SDL_WM_IconifyWindow())
96 ;               cout << "No se puede minimizar." << endl;
97 ;
98 ;           if(evento.key.keysym.sym == SDLK_f) {
99 ;
100 ;               // Si pulsamos f pasamos a pantalla completa
101 ;
102 ;               if(!SDL_WM_ToggleFullScreen(pantalla))
103 ;
104 ;                   cerr << "No se puede pasar a pantalla completa." << endl;
105 ;
106 ;
107 ;           }
108 ;
109 ;
110 ;       }
111 ;
112 ;}
```

Las novedades que presentamos en este listado son dos casos nuevos en la parte del *game loop* que gestionamos la entrada del teclado por eventos. Estos casos son el que gestiona la pulsación de la tecla 'm' y de la tecla 'f'.

Cuando pulsemos la tecla 'm' la ventana de nuestra aplicación se minimizará por lo que en ella hemos realizado la llamada a la función *SDL_WM_IconifyWindow()*. Cuando hagamos lo propio con la tecla 'f' pasaremos a modo de pantalla completa. Para que se produzca este efecto llamaremos a la función *SDL_WM_ToggleFullScreen()* pasándole como parámetro la variable *pantalla* que almacena la superficie principal de nuestra aplicación.

10.4. Convertir la entrada de usuario en exclusiva

En este tutorial hemos hecho referencia varias veces al término de foco. Existen dos tipos de focos: el foco de entrada, referente normalmente a la entrada por teclado, y el foco de ratón, que hace referencia al momento en el que el ratón está sobre nuestra aplicación. SDL proporciona la capacidad de hacer que ambos tipos de foco sean exclusivos de nuestra aplicación “enganchando” todas las entradas de eventos.

Cuando decidimos “enganchar” la entrada en nuestra aplicación SDL es la única que recibirá eventos. Aunque en un primer momento pueda parecer que esto puede ser contraproducente no es algo malo, si no más bien correcto, sobre todo para los videojuegos. Las aplicaciones de videojuegos suelen ser codiciosas en cuanto al consumo de recursos y no les gusta compartir el sistema. Así podemos “apropiarnos” de los dispositivos los que nos ahorrará más

10. Gestor de Ventanas

de un dolor de cabeza.

La función para “enganchar” la entrada, o lo que es lo mismo, disponerla en exclusividad es:

```
SDL_GrabMode SDL_WM_GrabInput(SDL_GrabMode mode);
```

Esta función recibe como parámetro el modo deseado para captar la entrada y devuelve el método de enganchar la entrada que está establecido actualmente. Las constates de este método son `SDL_GRAB_ON`, `SDL_GRAB_OFF` y `SDL_GRAB_QUERY`. Las dos primeras activan o desactivan el modo de entrada exclusiva mientras que la tercera se usa para consultar el estado actual de la disposición de la entrada.

Con esta función conseguiremos que los dos tipos de focos sean exclusivos de nuestra aplicación. La idea es parecida a la de tener un formulario modal que no nos permitiese centrar la atención en un programa diferente al nuestro.

10.5. Ejercicio 1

Vamos a poner en práctica el uso de esta función. Crea un programa en SDL en una ventana que pulsando la tecla ‘g’ nos permita tener la exclusividad de la entrada para que veas el efecto que esto produce en el sistema. Permite también que se pueda desactivar esta opción.

Cuando pulsemos dicha tecla tanto el ratón como el teclado, y los demás dispositivos de entrada que tengamos conectados, sólo podrán ser utilizados en el área de nuestra aplicación.

Aquí tienes la solución a este ejercicio:

```
1 ;// Ejercicio 1
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de prueba
5 ;// El aspecto fundamental de esta aplicación es capturar el
6 ;// foco de entrada en exclusividad
7 ;
8 ;
9 ;
10 ;#include <iostream>
11 ;#include <SDL/SDL.h>
12 ;
13 ;using namespace std;
14 ;
```

10.5. Ejercicio 1

```
15 ;int main()
16 ;{
17 ;
18 ;    // Iniciamos el subsistema de video
19 ;
20 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
21 ;
22 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
23 ;        exit(1);
24 ;
25 ;
26 ;    atexit(SDL_Quit);
27 ;
28 ;    // Comprobamos que sea compatible el modo de video
29 ;
30 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
31 ;
32 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
33 ;        exit(1);
34 ;
35 ;
36 ;
37 ;
38 ;    // Antes de establecer el modo de video
39 ;    // Establecemos el icono y el nombre de la ventana
40 ;
41 ;    // Cargamos la imagen del icono
42 ;
43 ;    SDL_Surface *icono;
44 ;
45 ;    icono = SDL_LoadBMP("./Imagenes/icono.bmp");
46 ;
47 ;    if(icono == NULL) {
48 ;
49 ;        cerr << "No se puede cargar el icono "
50 ;            << SDL_GetError() << endl;
51 ;        exit(1);
52 ;
53 ;
54 ;
55 ;    // Establecemos el nombre de la ventana y el icono
56 ;
57 ;    SDL_WM_SetCaption("Prueba", NULL);
58 ;    SDL_WM_SetIcon(icono, NULL); // Compatible con MS Windows
59 ;
60 ;    // Establecemos el modo de video
61 ;
62 ;    SDL_Surface *pantalla;
63 ;
64 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
65 ;
66 ;    if(pantalla == NULL) {
67 ;
```

10. Gestor de Ventanas

```
68 ;         cerr << "No se pudo establecer el modo de video: "
69 ;             << SDL_GetError() << endl;
70 ;
71 ;         exit(1);
72 ;
73 ;
74 ;     cout << "Pulsa ESC para terminar. " << endl;
75 ;     cout << "Pulsa m para minimizar. " << endl;
76 ;     cout << "Pulsa f para cambiar a pantalla completa. " << endl;
77 ;     cout << "Pulsa g para tomar en exclusividad la entrada." << endl;
78 ;
79 ; // Variables auxiliares
80 ;
81 ;     SDL_Event evento;
82 ;
83 ; // Bucle infinito
84 ;
85 ; for( ; ; ) {
86 ;
87 ;     while(SDL_PollEvent(&evento)) {
88 ;
89 ;         if(evento.type == SDL_KEYDOWN) {
90 ;
91 ;             if(evento.key.keysym.sym == SDLK_ESCAPE)
92 ;                 return 0;
93 ;
94 ;             if(evento.key.keysym.sym == SDLK_m) {
95 ;
96 ;                 // Si pulsamos m, minimizamos
97 ;
98 ;                 if(!SDL_WM_SetIconifyWindow())
99 ;                     cout << "No se puede minimizar." << endl;
100 ;
101 ;             if(evento.key.keysym.sym == SDLK_f) {
102 ;
103 ;                 // Si pulsamos f pasamos a pantalla completa
104 ;
105 ;                 if(!SDL_WM_ToggleFullScreen(pantalla))
106 ;
107 ;                     cout << "No se puede pasar a pantalla completa."
108 ;                         << endl;
109 ;             }
110 ;
111 ;             if(evento.key.keysym.sym == SDLK_g) {
112 ;
113 ;
114 ;                 // Si tenemos la entrada en exclusividad
115 ;
116 ;                 if(SDL_GRAB_ON == SDL_WM_GrabInput(SDL_GRAB_QUERY)) {
117 ;
118 ;                     SDL_WM_GrabInput(SDL_GRAB_OFF);
119 ;
120 ;                     cout << "Entrada exclusiva OFF" << endl;
```

10.6. Capturando los Eventos del Windows Manager

```
121 ;
122 ;           } else {
123 ;
124 ;               SDL_WM_GrabInput(SDL_GRAB_ON) ;
125 ;
126 ;               cout << "Entrada exclusiva ON" << endl;
127 ;           }
128 ;
129 ;       }
130 ;
131 ;   }
132 ; }
133 ;
134 ; }
135 ;
136 ; }
```

10.6. Capturando los Eventos del Windows Manager

Para terminar vamos a estudiar como capturar los eventos producidos por el gestor de ventanas o *windows manager*. Estos eventos son específicos de cada plataforma, debes saber que si decides responder a este tipo de eventos estarás limitando la portabilidad de tu aplicación. Por esta razón no será éste un tema de especial interés en nuestro tutorial.

Para poder consultar los eventos del windows manager tenemos de activar esta opción con una función que ya hemos visto. Se trata de:

```
Uint8 SDL_EventState(Uint8 type, int state);
```

Como recordarás esta función recibe dos parámetros. El primero, en *type*, es el tipo de evento mientras que el parámetro *state* es el estado de respuesta de dicho evento. Podemos pasar como parámetros cualquier tipo de evento, pero para este caso en concreto, el caso del gestor de ventanas, deberemos de utilizar la macro **SDL_SYSWMEVENT** en el parámetro *type*.

El estado del evento puede ser uno de estos tres valores: **SDL_ENABLE**, **SDL_IGNORE** o **SDL_QUERY**. Si el estado del evento es **SDL_ENABLE** el tipo de evento pasado como parámetro será añadido a la cola de eventos, mientras que si el valor es **SDL_IGNORE** dicho evento no será añadido a la cola. Si el valor es **SDL_QUERY** es que el estado actual del evento será devuelto por **SDL_EventState()**.

Una vez activada la recepción de eventos del gestor de ventanas los recibiremos como los otros tipos de eventos gestionados en SDL. Como podrás ver a continuación la única información de sobre los eventos del windows

10. Gestor de Ventanas

manager que recibiremos será una estructura del tipo `SDL_Event` con el dato de que evento a ocurrido. Ninguna maravilla.

Para poder obtener más información de estos eventos necesitamos la función:

```
int SDL_GetWMInfo(SDL_SysWMinfo *info);
```

El parámetro de esta función es un puntero a la estructura `SDL_SysWMinfo`. El contenido exacto de esta estructura depende de la plataforma donde hayamos compilado nuestra aplicación, por lo que no es precisamente una buena idea responder a este tipo de eventos. Si necesitas responder a eventos en un entorno WIN32 la estructura está definida de la siguiente forma:

```
1 ;typedef struct {
2 ;     SDL_version version;
3 ;     HWND window;
4 ;} SDL_SysWMinfo;
```

La estructura tiene dos miembros. Uno que indica la versión de la librería SDL que estamos utilizando. El otro es un campo de tipo `HWND` que es la pantalla principal de la aplicación. Si necesitas código específico para WIN32 este es el camino para hacerlo. Realmente, no te lo aconsejo.

10.7. Recopilando

En este capítulo hemos aprendido a interactuar con el gestor de ventanas en el que se ejecuta nuestra aplicación. Ya somos capaces de personalizar nuestra ventana así como de realizar varias acciones como las de maximizar, minimizar o tomar en exclusiva la entrada de usuario desde nuestra aplicación SDL.

Recuerda que responder a eventos de un gestor de ventanas específico no es una buena idea ya que limitará la portabilidad de tu aplicación.

Capítulo 11

SDL_image. Soporte para múltiples formatos.

11.1. Introducción. Conocimientos Previos

En este capítulo a estudiar una librería totalmente diferente a las demás por su composición. Esta librería auxiliar consta de... ¡una sola función! Sí, sorprendente. Es la librería auxiliar `SDL_image`.

Esta librería nos ofrece gran versatilidad a la hora de trabajar con imágenes. Hasta ahora sólo podíamos cargar en superficies imágenes en formato *bmp* lo que supone tener almacenada las imágenes en un formato que no ayuda nada a optimizar el espacio en consumido disco. Es una buena idea tener una librería que nos permita trabajar con otros formatos de imagen.

Esta librería posee una única función que nos permite manejar ficheros en formato *bmp*, *gif*, *jpg*, *png*, *tga*, *pnm*, *xpm* y *lhm*.

Cada uno de estos formatos tienen unas características que los hacen mejores o peores según sea la imagen que queremos guardar. Por ejemplo actualmente el formato *jpg* es el más utilizado para guardar imágenes porque utiliza un algoritmo de compresión bueno para almacenar fotografías. El formato *gif* o el *png* puede ser más adecuado para guardar cierto tipo de dibujos, aunque la versatilidad de *png* es importante. En definitiva el tema de los formatos de imagen es muy interesante y no estaría demás adquiririeses un un profundo conocimiento de los mismos. Te animo a que investigues qué formato es el que más te conviene en cada momento para tu proyecto.

Para poder utilizar esta librería auxiliar debemos de tenerla instalada. Si no la tienes configurada vuelve al apartado de instalación de este tutorial donde podrás encontrar los pasos a seguir para instalar correctamente esta librería.

11. SDL_image. Soporte para múltiples formatos.

11.2. Objetivo

El objetivo de este capítulo es familiarizarnos con todo lo necesario para utilizar esta librería. Desde cómo compilar con ella hasta cómo utilizar la función que nos proporciona.

11.3. Compilando con SDL_image

Cuando creamos nuestra aplicación y hacemos uso de esta librería dos son las cuestiones que tenemos que tener en cuenta antes de realizar la compilación. La primera es que es necesario realizar el `include` de dicha librería en los ficheros fuente que sea necesario. Tenemos que incluir `#include <SDL/SDL_image.h>` para que el compilador pueda comprobar tipos.

La segunda cosa a tener en cuenta es que tenemos que indicarle al compilador que enlace contra esta librería. Esto lo indicaremos con la opción `-lSDL_image`. En nuestro caso, al trabajar con *makefiles* sólo tendremos que añadir esta opción a la variable que usamos para especificar las librerías como puedes observar en el siguiente ejemplo:

```
1 ;CXX = g++
2 ;CXXFLAGS = -ansi -Wall
3 ;LIB = -lSDL -lSDL_image
4 ;EXE = test
5 ;OBJETOS = main.o
6 ;
7 ;${EXE}: ${OBJETOS}
8 ;      ${CXX} -o ${LIB} ${OBJETOS}
9 ;
10 ;${OBJETOS}:
11 ;
12 ;clean:
13 ;      ${RM} ${EXE} ${OBJETOS} *~ *#
;
```

11.4. Usando SDL_image

Como ya hemos visto `SDL_image` sólo tiene una función y responde al siguiente prototipo:

```
SDL_Surface *IMG_Load(const char *file);
```

Como podrás ver, el prototipo es muy parecido al de la función `SDL_LoadBMP()`. Recibe como parámetro donde está alojado el fichero, es decir, la ruta y nombre del fichero que queremos cargar, y devuelve un puntero a la superficie donde vamos a tener almacenada dicha imagen en

11.4. Usando `SDL_image`

formato de superficie. Si ocurre algún problema esta función devolverá el valor `NULL`.

La única diferencia en la definición de esta función con respecto a `SDL_LoadBMP()` es que permite cargar varios tipos de gráficos que no soportaba la función original.

Como puedes ver esta librería no supone complejidad adicional alguna.

11.4.1. Ejemplo 1

Vamos a realizar un sencillo ejemplo que cargue imágenes en los distintos formatos soportados para comprobar que `SDL_image` funciona correctamente. En la figura 11.1 tienes un esquema que ilustra el objetivo del ejemplo.

```
1 ;// Ejemplo 1
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Utilizando SDL_image
5 ;
6 ;
7 ;#include <iostream>
8 ;#include <SDL/SDL.h>
9 ;
10 ;#include <SDL/SDL_image.h>
11 ;
12 ;using namespace std;
13 ;
14 ;int main()
15 ;{
16 ;    // Iniciamos el subsistema de video
17 ;
18 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
19 ;
20 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
21 ;        exit(1);
22 ;    }
23 ;
24 ;    atexit(SDL_Quit);
25 ;
26 ;    // Comprobamos que sea compatible el modo de video
27 ;
28 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
29 ;
30 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
31 ;        exit(1);
32 ;
33 ;    }
34 ;
35 ;    // Antes de establecer el modo de video
```

11. SDL_image. Soporte para múltiples formatos.

```
36 ; // Establecemos el nombre de la ventana
37 ;
38 ;     SDL_WM_SetCaption("Ejemplo 1. SDL_image", NULL);
39 ;
40 ; // Establecemos el modo
41 ;
42 ;     SDL_Surface *pantalla;
43 ;
44 ;     pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
45 ;
46 ;     if(pantalla == NULL) {
47 ;
48 ;         cerr << "No se pudo establecer el modo de video: "
49 ;             << SDL_GetError();
50 ;
51 ;         exit(1);
52 ;
53 ;
54 ; // Variables auxiliares
55 ;
56 ;     SDL_Event evento;
57 ;
58 ; // Cargamos la imagen en formato GIF
59 ;
60 ;     SDL_Surface *imagen = IMG_Load("Imagenes/negritos.gif");
61 ;
62 ; // La mostramos por pantalla
63 ;
64 ;     SDL_BlitSurface(imagen, NULL, pantalla, NULL);
65 ;     SDL_Flip(pantalla);
66 ;
67 ;     cout << "\n Cargada imagen en formato GIF" << endl;
68 ;     cout << "Pulsa una tecla para cargar otro formato de imagen. " << endl;
69 ;
70 ;
71 ; // Esperamos que se produzca un evento de teclado
72 ;
73 ;     do {
74 ;
75 ;         SDL_WaitEvent(&evento);
76 ;
77 ;     } while(evento.type != SDL_KEYDOWN);
78 ;
79 ;
80 ; //*****/*****
81 ;
82 ; // Cargamos la imagen en formato JPG
83 ;
84 ;     imagen = IMG_Load("Imagenes/negritos.jpg");
85 ;
86 ; // La mostramos por pantalla
87 ;
88 ;     SDL_BlitSurface(imagen, NULL, pantalla, NULL);
```

11.4. Usando `SDL_image`

```
89 ;     SDL_Flip(pantalla);
90 ;
91 ;     cout << "\n Cargada imagen en formato JPG" << endl;
92 ;     cout << "Pulsa una tecla para cargar otro formato de imagen. " << endl;
93 ;
94 ;
95 ;     // Esperamos que se produzca un evento de teclado
96 ;
97 ;     do {
98 ;
99 ;         SDL_WaitEvent(&evento);
100 ;
101 ;     } while(evento.type != SDL_KEYDOWN);
102 ;
103 ;
104 ;     /*****/
105 ;
106 ;     // Cargamos la imagen en formato png
107 ;
108 ;     imagen = IMG_Load("Imagenes/negritos.png");
109 ;
110 ;     // La mostramos por pantalla
111 ;
112 ;     SDL_BlitSurface(imagen, NULL, pantalla, NULL);
113 ;     SDL_Flip(pantalla);
114 ;
115 ;     cout << "\n Cargada imagen en formato PNG" << endl;
116 ;     cout << "Pulsa una tecla para cargar otro formato de imagen. " << endl;
117 ;
118 ;     // Esperamos que se produzca un evento de teclado
119 ;
120 ;     do {
121 ;
122 ;         SDL_WaitEvent(&evento);
123 ;
124 ;     } while(evento.type != SDL_KEYDOWN);
125 ;
126 ;
127 ;     /*****/
128 ;
129 ;     // Cargamos la imagen en formato tga
130 ;
131 ;     imagen = IMG_Load("Imagenes/negritos.tga");
132 ;
133 ;     // La mostramos por pantalla
134 ;
135 ;     SDL_BlitSurface(imagen, NULL, pantalla, NULL);
136 ;     SDL_Flip(pantalla);
137 ;
138 ;     cout << "\n Cargada imagen en formato TGA" << endl;
139 ;     cout << "Pulsa una tecla para cargar otro formato de imagen. " << endl;
140 ;
141 ;     // Esperamos que se produzca un evento de teclado
```

11. SDL_image. Soporte para múltiples formatos.

```
142 ;
143 ;    do {
144 ;
145 ;        SDL_WaitEvent(&evento);
146 ;
147 ;    } while(evento.type != SDL_KEYDOWN);
148 ;
149 ;    //***** ****
150 ;
151 ;    // Cargamos la imagen en formato XPM
152 ;
153 ;    imagen = IMG_Load("Imagenes/negritos.xpm");
154 ;
155 ;    // La mostramos por pantalla
156 ;
157 ;    SDL_BlitSurface(imagen, NULL, pantalla, NULL);
158 ;    SDL_Flip(pantalla);
159 ;
160 ;    cout << "\n Cargada imagen en formato XPM" << endl;
161 ;    cout << "Pulsa una tecla para terminar. " << endl;
162 ;
163 ;    // Esperamos que se produzca un evento de teclado
164 ;
165 ;    do {
166 ;
167 ;        SDL_WaitEvent(&evento);
168 ;
169 ;    } while(evento.type != SDL_KEYDOWN);
170 ;
171 ;    return 0;
172 ;}
```

Como puedes ver en el código la aplicación consiste en cargar la misma imagen en diferentes formato de forma secuencial. Cada vez que pulsemos una tecla se cargará una imagen en un formato diferente hasta pasar por todos los formatos soportados pos *SDL_image*.

11.5. Recopilando

En este capítulo hemos aprendido a utilizar la librería adicional *SDL_image* que nos permitirá utilizar una amplio tipo de formatos de ficheros de imágenes en nuestra aplicación lo que nos posibilita ganar en potencia en cuanto a la gestión de recursos.

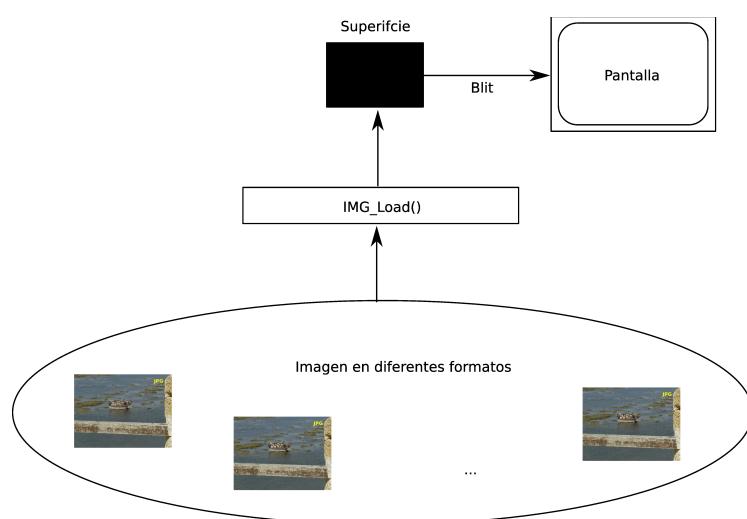


Figura 11.1: Ejemplo 1

11. SDL_image. Soporte para múltiples formatos.

Capítulo 12

SDL_ttf. Escritura de textos sobre superficies

12.1. Introducción

La librería SDL nos permite trabajar con gráficos, sonidos... controlar la entrada mediante varias técnicas... Una función que le falta a esta librería es la de dibujar texto en pantalla. Los textos son comunes en los videojuegos, aunque cuanto menores sean, menos problemas de localización tendremos. Como localización se entiende la configuración de la aplicación para un entorno concreto, con un lenguaje natural concreto y una simbología para dicho lenguaje muy determinada.

No todo el mundo habla inglés, español o coreano, por eso es una buena idea minimizar la cantidad de texto a utilizar en nuestro videojuego, a no ser que sea imprescindible para el desarrollo del mismo.

En lo visto hasta ahora, para insertar un texto en nuestra aplicación gráfica teníamos que utilizar nuestro editor de imágenes favorito y crear un texto en mapa de bits. Seguidamente cargar la imagen como superficie en nuestro programa y colocarla donde fuese necesario.

Esta librería nos libra de todo ese trabajo. Nos permite escribir texto en la pantalla gráfica directamente. Nos ofrece dibujar el texto que queramos en una superficie SDL utilizando el tipo de letra que deseemos, eso sí, el tipo de formato de letra tiene que ser compatible con las fuentes true type (ttf).

Para poder utilizar esta librería auxiliar debemos de tenerla instalada. Si no la tienes instalada vuelve al apartado de instalación de este tutorial donde podrás encontrar los pasos a seguir para instalar correctamente esta librería.

La librería SDL_ttf tiene 31 funciones disponibles. Vamos a dividir las funciones en varios grupos para proceder a su estudio. Puedes identificar a todas las funciones de esta librería auxiliar fácilmente porque comienzan por

TTF_.

12.2. Objetivos

1. Aprender a manejar las funciones que nos proporciona la librería.
2. Adquirir una cultura básica sobre tipografías.
3. Aprender a gestionar los recursos tipográficos con SDL.
4. Conocer los distintos tipos de renderizados que nos ofrece esta librería.

12.3. Compilando

No debes de olvidar que antes de compilar una aplicación que haga uso de esta librería debes indicarle al compilador que enlace con dicha librería. Para esto incluiremos *-lSDL_ttf* a la hora de compilar nuestra aplicación. En nuestros *makefiles* la incluiremos en la variable que controla las librerías que añadimos a la hora de compilar.

En los ficheros donde vayamos a hacer uso de funciones de esta librería debemos de incluir el fichero de cabecera de la librería mediante `#include <SDL/SDL_ttf.h>` para que el compilador pueda realizar las comprobaciones pertinentes.

12.4. Inicialización

Para utilizar esta nueva librería, e igualmente que con otros aspectos de SDL, debemos de inicializarla. La función que nos permite realizar esta tarea es:

```
int TTF_Init(void);
```

La función no recibe ningún parámetro y devuelve 0 si la operación se realiza con éxito, devuelve -1 en caso de existir algún problema. Una vez utilizada la librería debemos de cerrarla o liberarla. La función que se encarga de cerrar esta librería es:

```
void TTF_Quit(void);
```

Como vemos esta función no recibe ni devuelve ningún valor por lo que es perfecta para utilizarla como parámetro en la función *atexit()*, lo que nos evitara tener que llamar a la función *TTF_Quit(void)* directamente.

Cuando quieras hacer uso de la librería puedes utilizar una estructura selectiva que compruebe si se ha inicializado correctamente dicha librería, y de ser así, haga la llamada a la función *atexit()*. Nosotros utilizaremos esta forma de actuar en los ejemplos para que tengas un ejemplo que seguir.

12.5. Creando y Destruyendo

La estructura que mantiene la información de esta librería auxiliar es la estructura *TTF_font*. Los detalles de implementación de esta estructura están totalmente ocultos y no son necesarios para trabajar con la librería. Sólo necesitamos trabajar con punteros a este tipo de estructura pasándolo como parámetro en distintas funciones.

Para elegir el tipo de letra a utilizar SDL, o como dice la documentación de *SDL_ttf*, para “crear” una fuente la librería proporciona dos funciones a usar dependiendo del número de fuentes almacenadas en un fichero. En el caso de que el fichero contenga almacenada un “sólo” tipo de letra la función a utilizar es la siguiente:

```
TTF_Font *TTF_OpenFont(const char *file, int ptsize);
```

Esta función recibe como primer parámetro la ubicación del fichero donde está almacenada la fuente. El otro parámetro de entrada es *ptsiz* que es el tamaño de letra que queremos aplicar cuando sea mostrada por pantalla en puntos. Un punto es una setenta y dos parte de una pulgada, pero depende del tipo del sistema que estemos utilizando. En un monitor convencional hay de 72 a 96 puntos por pulgada. En lo que concierne a gráficos a mostrar por pantalla un punto es prácticamente de igual tamaño que un píxel. La función devuelve un puntero a la estructura *TTF_Font* que hemos comentado en el párrafo anterior.

La composición de la estructura devuelta no es de especial relevancia para el uso de estas funciones ya que el valor devuelto cuando creamos la fuente lo utilizaremos íntegramente.

En un mismo fichero .ttf se pueden guardar varias fuentes, *SDL_ttf* nos permite utilizar este tipo de ficheros. Para estos casos utilizaremos la siguiente función:

```
TTF_Font *TTF_OpenFontIndex(const char *file, int ptsize, long index);
```

Respecto a la anterior esta función recibe un parámetro más. El nuevo parámetro sirve para indicar a la función el número de índice de la fuente que

12. SDL_ttf. Escritura de textos sobre superficies

queremos utilizar a elegir dentro de las que posee el fichero en cuestión, es decir, indica fuente queremos cargar.

Cuando hayamos terminado de utilizar el tipo de letra deberemos de cerrarla, o lo que es lo mismo para esta librería auxiliar, destruirla o liberarla. Para esto utilizaremos la siguiente función:

```
void TTF_Close(TTF_Font *font);
```

Esta función recibe como parámetro el puntero obtenido como devolución en la función con la que cargábamos la fuente en memoria. Esta función se encarga de cerrarla y liberar la memoria que ocupa dicha fuente.

12.6. Obteniendo Información

El mundo del trabajo con fuentes está lleno de jerga propia de imprenta. Necesitamos conocer parte de esta jerga para poder manejar con comodidad la información que vamos a tratar sobre las fuentes. Como ya sabes el tipo de dato TTF_Font está oculto al programador-usuario. Para acceder a cualquier tipo de información almacenada en ella deberemos de hacer uso de funciones que nos permitan realizar esta tarea.

El primer aspecto que vamos a consultar sobre las fuentes es el tamaño de la misma. El tamaño de la fuente es la altura máxima que pueden alcanzar los caracteres. Normalmente suele ser igual al tamaño en puntos especificado en las funciones que se encargan de abrir las fuentes. Si no coinciden ambos valores estarán muy próximos. Para conocer el tamaño de una fuente utilizaremos la función:

```
int TTF_FontHeight(TTF_Font *font);
```

Esta función recibe como parámetro el puntero devuelto por la función de apertura de la fuente y devuelve el tamaño o alutura de la fuente en píxeles.

Cuando escribimos sobre papel marcado, ya sea con cuadros o líneas, puedes observar como letras como la *g* y la *q* traspasan la linea sobre la que estamos escribiendo, mientras que letras como la *m*, la *n* o la *b* permanecen sobre dicha linea. Esta línea que tomamos de referencia es llamada linea base o *base line*. La parte que va desde la linea base hasta el punto más alto de la letra es conocido como *ascendente* o *subida*, mientras que la parte que sobre pasa la linea base, atravesándola, es la *bajada* o *descendente*. Puedes ver en la figura 12.1 un ilustración con estos conceptos.

Para concer el ascendente de una fuente SDL_ttf proporciona la función:



Figura 12.1: Anatomía de una fuente

```
int TTF_FontAscent(TTF_Font *font)
```

Mientras que para conocer el descendente deberemos de utilizar la función

```
int TTF_FontDescent(TTF_Font *font).
```

Ambas funciones reciben como parámetro el puntero a la estructura TTF_Font, como en las demás funciones de esta librería, y devuelven un entero con el valor de la característica cuestionada. Realmente la mayoría del tiempo no necesitaremos nada sobre el ascendente o descendente de nuestro tipo de letra cuando estemos trabajando con ella porque no es fundamental saber nada sobre estas características para trabajar con dicha fuente. Eso sí, si alguna vez necesitas este tipo de información ya conoces las funciones para consultar estos tipos de valores.

Existen características mucho más interesantes sobre las fuentes que estudiaremos cuando establezcamos el estilo de la fuente a utilizar en este capítulo.

12.7. Manejando las Fuentes

Como sabemos, la mayoría de las fuentes ttf permiten el uso de varios estilos como puede ser negrita, itálica... Para conocer el estilo aplicado a la fuente cargada utilizamos la siguiente función:

```
int TTF_GetFontStyle(TTF_Font *font);
```

Esta función recibe como parámetro el ya archiconocido puntero a TTF_Font y devuelve el estilo aplicado a dicha fuente. El valor devuelto por esta función es una combinación de banderas de bit que representan el estilo de la fuente. Las combinaciones de estilo se representan por TTF_STYLE_BOLD, TTF_STYLE_ITALIC y TTF_STYLE_UNDERLINE. Como podrás ver la primera constante corresponde al estilo “negrita”, la segunda al estilo “cursivo” y el último al “subrayado”. Si la fuente en cuestión no tiene ningún estilo aplicado

12. SDL_ttf. Escritura de textos sobre superficies

la función devolverá TTF_STYLE_NORMAL, equivalente a 0 o sin estilo.

Para que una fuente tenga un estilo aplicado debemos de poder especificar de que estilo queremos dotar a una función. SDL_ttf nos proporciona una función con este objetivo. Para establecer un estilo concreto utilizamos la función:

```
void TTF_SetFontStyle(TTF_Font *font, int style);
```

Como parámetros la función recibe el puntero a TTF_Font devuelto por la función que abría la fuente en cuestión. *style* es un campo de bits de banderas, como el devuelto en la función anterior, que puede tomar los valores:

- TTF_STYLE_BOLD: Para establecer el estilo **negrita**.
- TTF_STYLE_ITALIC: Establece el estilo *cursiva*.
- TTF_STYLE_UNDERLINE: Establece el estilo subrayado.
- TTF_STYLE_NORMAL: Restaura el estado normal.

Como puedes observar, y como es lógico, son los mismos valores que para *TTF_GetFontStyle*.

Otra de las características que podemos consultar del aspecto del texto presentado por esta librería es el espacio vertical entre líneas del texto. Si queremos dibujar una sola línea de texto esto no es importante, pero si influye en el momento que tenemos múltiples líneas. La función que nos da información acerca de esta separación entre líneas es:

```
int TTF_FontLineSkip(TTF_Font *font);
```

Esta función recibe como parámetro un puntero a un objeto TTF_Font y devuelve el número de píxeles que se deben respetar entre líneas de texto.

Para terminar vamos a presentar tres funciones que nos permiten conocer cuánto espacio va a ocupar el texto en pantalla. El espacio que ocupará nuestro texto en pantalla dependerá de la codificación de caracteres a utilizar. Un texto representado mediante UNICODE necesitará de campos de 16 bits donde pueden existir caracteres extraños, mientras que lo normal es usar caracteres de 8 bits. Existen tres funciones con este cometido dependiendo como estén representadas nuestras cadenas. Estas son:

- *int TTF_SizeText(TTF_Font *font, const char * text, int * w, int * h);*
- *int TTF_SizeUTF8(TTF_Font *font, const char * text, int * w, int * h);*

- `int TTF_SizeUNICODE(TTF_Font *font, const Uint16 *text, int *w, int *h);`

Estas funciones reciben como parámetro un puntero a TTF_Font como es habitual y como segundo parámetro la cadena de texto a medir. El tercer y cuarto parámetro son parámetros de salida donde en *w* se devuelve el ancho que necesitamos para poder mostrar el texto en píxeles y en *h* el alto medido en la misma unidad.

12.8. Rendering

Dibujar el texto en una superficie SDL conlleva un proceso. Las fuentes deben de convertirse en información interpretable a través de la librería SDL para ser mostradas por pantalla. Este proceso es conocido como render. El resto de funciones de SDL_ttf que vamos a estudiar están dedicadas a realizar este dibujado. Tres son las más habituales, dependiendo de la calidad de renderizado que queramos obtener:

- `SDL_Surface * TTF_RenderText_Solid(TTF_Font *font, const char *text, SDL_Color fg);`
- `SDL_Surface * TTF_RenderText_Shaded(TTF_Font *font, const char *text, SDL_Color fg, SDL_color bg);`
- `SDL_Surface * TTF_RenderText_Bladed(TTF_Font *font, const char *text, SDL_Color fg);`

Estas funciones se encargan de realizar el renderizado del texto. Reciben como parámetro la fuente que se desea utilizar, el texto que se desea dibujar y el color que se desea utilizar. Dicho color debe ser obtenido en el formato de píxel de la superficie mediante el tipo `SDL_Color`. La segunda función además recibe como parámetro el color de fondo que se le quiere aplicar a la fuente.

La diferencia entre estas tres funciones es la calidad de renderizados. Las mostramos ordenadas de menor calidad a mayor. Aplicar una función de mayor calidad supone una mayor complejidad de renderizado por lo que aumenta el tiempo que se tarda en realizar la operación ya que el consumo de recursos es mayor. Si utilizamos el renderizado *blended* tienes que ser consciente que supone una carga mayor para el sistema que un renderizado *solid*, lo que se traduce en una mayor calidad a costa de una mayor carga de trabajo que ralentiza el sistema.

Las tres funciones devuelven un puntero a una superficie que es la que deberemos volcar en la superficie principal para que sea mostrada por pantalla

12. SDL_ttf. Escritura de textos sobre superficies

como ya sabes.

Existen varias versiones de las funciones que hemos presentado anteriormente. Como pasaba en el apartado anterior podemos tener varias representaciones de las cadenas de caracteres dependiendo de las representaciones que tengamos de ellas. Esto es importante si queremos tratar el tema de la **localización** de nuestra aplicación. Un texto representado mediante UNICO-DE necesitará de campos de 16 bits, mientras que lo normal es usar caracteres de 8 bits. Existen las siguientes versiones de las funciones anteriores:

- *SDL_Surface * TTF_RenderUTF8_Solid(TTF_Font *font, const char *text, SDL_Color fg);*
- *SDL_Surface * TTF_RenderUNICODE_Solid(TTF_Font *font, const Uint16 *text, SDL_Color fg);*
- *SDL_Surface * TTF_RenderUTF8_Shaded(TTF_Font *font, const char *text, SDL_Color fg, SDL_color bg);*
- *SDL_Surface * TTF_RenderUNICODE_Shaded(TTF_Font *font, const Uint16 *text, SDL_Color fg, SDL_color bg);*
- *SDL_Surface * TTF_RenderUTF8_Bladed(TTF_Font *font, const char *text, SDL_Color fg, SDL_color bg);*
- *SDL_Surface * TTF_RenderUNICODE_Bladed(TTF_Font *font, const Uint16 *text, SDL_Color fg, SDL_color bg);*

Como puedes ver son funciones muy similares a las originales. Varía el tipo de representación de la cadena de caracteres en aquellas que tiene que hacerlo.

12.8.1. Ejemplo 1

Vamos a mostrar utilizar esta librería mostrando por pantalla el mensaje “Hola Mundo”. Vamos a estudiar el código:

```
1 ;// Ejemplo 1
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Utilizando fuentes ttf
5 ;
6 ;
7 ;#include <iostream>
8 ;#include <SDL/SDL.h>
9 ;
10 ;#include <SDL/SDL_ttf.h>
11 ;
```

```
12 ;using namespace std;
13 ;
14 ;int main()
15 ;{
16 ;    // Iniciamos el subsistema de video
17 ;
18 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
19 ;
20 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
21 ;        exit(1);
22 ;
23 ;
24 ;        atexit(SDL_Quit);
25 ;
26 ;    // Comprobamos que sea compatible el modo de video
27 ;
28 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
29 ;
30 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
31 ;        exit(1);
32 ;
33 ;
34 ;
35 ;    // Inicializamos la librería TTF
36 ;
37 ;    if(TTF_Init() == 0) {
38 ;
39 ;        atexit(TTF_Quit);
40 ;        cout << "TTF inicializada" << endl;
41 ;
42 ;
43 ;
44 ;    // Antes de establecer el modo de video
45 ;    // Establecemos el nombre de la ventana
46 ;
47 ;    SDL_WM_SetCaption("Hola Mundo. SDL_ttf", NULL);
48 ;
49 ;    // Establecemos el modo
50 ;
51 ;    SDL_Surface *pantalla;
52 ;
53 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
54 ;
55 ;    if(pantalla == NULL) {
56 ;
57 ;        cerr << "No se pudo establecer el modo de video: "
58 ;            << SDL_GetError();
59 ;
60 ;        exit(1);
61 ;
62 ;
63 ;    // Cargamos la fuente que vamos a utilizar de tamaño 40
64 ;}
```

12. SDL_ttf. Escritura de textos sobre superficies

```
65 ;     TTF_Font *fuente;
66 ;
67 ;     fuente = TTF_OpenFont("Fuentes/ep.ttf", 40);
68 ;
69 ;     // Mostramos información acerca de la fuente cargada
70 ;
71 ;     cout << "El tamaño de la fuente es " << TTF_FontHeight(fuente) << endl;
72 ;     cout << "El ascendente de la fuente es " << TTF_FontAscent(fuente) << endl;
73 ;     cout << "El descendente de la fuente es " << TTF_FontDescent(fuente) << endl;
74 ;     cout << "La separación entre líneas es " << TTF_FontLineSkip(fuente) << endl;
75 ;
76 ;     int w, h;
77 ;
78 ;     TTF_SizeUTF8(fuente, "Hola Mundo", &w, &h);
79 ;
80 ;     cout << "El mensaje Hola Mundo ocupará " << w << " píxeles de ancho"
81 ;         << " y " << h << " de alto." << endl;
82 ;
83 ;     // Vamos a escribir HolaMundo
84 ;
85 ;     SDL_Surface *texto;
86 ;     SDL_Color color;
87 ;
88 ;     // Establecemos el color para la fuente
89 ;
90 ;     color.r = 25;
91 ;     color.g = 150;
92 ;     color.b = 180;
93 ;
94 ;     // Renderizamos
95 ;
96 ;     texto = TTF_RenderText_Blended(fuente, "Hola Mundo", color);
97 ;
98 ;     // Establecemos la posición
99 ;
100 ;    SDL_Rect dest;
101 ;
102 ;    dest.x = 150;
103 ;    dest.y = 100;
104 ;    dest.h = texto->h;
105 ;    dest.w = texto->w;
106 ;
107 ;    // Mostramos el texto por pantalla
108 ;
109 ;    SDL_BlitSurface(texto, NULL, pantalla, &dest);
110 ;
111 ;    SDL_Flip(pantalla);
112 ;
113 ;    // Mostramos el resultado durante 5 seg
114 ;
115 ;    SDL_Delay(5000);
116 ;
117 ;
```

```
118 ;    return 0;  
119 ;}
```

Como puedes ver gran parte del código ya te es familiar. Con respecto a esta librería lo primero que hemos hecho es incluir su fichero de cabecera con `#include<SDL/SDL_ttf.h>`. Seguidamente inicializamos la librería haciendo uso de la función `TTF_Init()`.

Ahora bien. Vamos a mostrar un mensaje en una ventana haciendo uso de una fuente ttf. Cargamos dicha fuente mediante la función `TTF_OpenFont()` con un tamaño de 40 puntos. Una vez cargada mostramos toda la información disponible sobre el tipo de cadena que vamos a mostrar.

Llegados a este punto nos disponemos a realizar el renderizado de la cadena para convertirla en superficie. Utilizamos la versión blended a la que le pasamos, además de la fuente y de la cadena, el color elegido.

El resto del código es común a los demás ejemplos que hemos utilizado. Realizamos el blit sobre la pantalla principal y la mostramos. En este caso en vez de gestionar eventos para mantener el resultado en pantalla vamos a mostrarlo durante cinco segundos haciendo uso de una función para el manejo del tiempo de SDL.

12.9. Recopilando

En este capítulo hemos aprendido a utilizar fuentes ttf para mostrar textos en pantalla, algo muy común en el mundo de los videojuegos. Existen otras alternativas, como por ejemplo, cargar una imagen para cada letra y componer títulos. Como puedes ver esta alternativa es mucho menos correcta que la de utilizar fuentes ttf.

Algunas de estas fuentes dan problemas con la librería con lo que verás que en el videojuego final vamos a implementar funciones que nos permiten, por ejemplo, utilizar fuentes con espaciados mal definidos o saltos de línea.

Este es el inicio de trabajo con fuentes ttf, el límite está en tu imaginación. La carga de trabajo que supone el renderizado de una fuente es bastante importante. Es complicado crear textos decorativos con este método de trabajo. Por estos motivos en más de una ocasión optaremos por crear una fuente propia en un fichero mapa de bits bien definido utilizando nuestra herramienta favorita para crear fuentes personalizadas.

12. SDL_ttf. Escritura de textos sobre superficies

Capítulo 13

SDL_mixer. Gestión de sonidos.

13.1. Introducción. Conocimientos previos

El subsistema de audio es uno de los mas tediosos de utilizar de SDL. En nuestra ayuda acude la librería `SDL_mixer` para facilitarnos el manejo del sistema de sonido.

`SDL_mixer` es un complemento que mejora el subsistema de audio de `SDL`. Está preparada para manejar múltiples sonidos al mismo tiempo además de la música. Es más, si te sientes capaz, puede especificar la manera de mezclar la música y aplicar varios efectos (como el fade-out) en tu aplicación manejando punteros a funciones que realicen esta tarea.

`SDL_mixer` se encarga de realizar el mix o mezcla de canales de audio por nosotros de forma automática lo que nos ahorra el desarrollo de un sistema de mezclado. Para reproducir sonido en nuestra aplicación `SDL` diferencia el audio referente a la música del juego, a la que coloca en un canal independiente, de los sonidos de efectos propios del juego. Los formatos con los que trabaja esta librería son *wav*, *mp3*, *midi*, *Ogg Vorbis*, *MOD*, *IT*, *S3M* y *VOC*. Como podrás observar es compatible con muchos de los formatos deseables.

Antes de empezar debes saber que es un *chunk*. Un chunk no es más que un sonido producido en nuestro videojuego por algún efecto concreto, como puede ser el golpeo de una pelota en un juego de tenis o el choque de dos espadas en un juego de acción. Sabiendo esto vamos a empezar inicializando la librería.

13.2. Compilando con `SDL_mixer`

Como ocurre con las otras librerías adicionales, antes de compilar, debemos de incluir en nuestros ficheros fuente que vamos a usar esta librería mediante `#include<SDL/SDL_mixer.h>`. A la hora de compilar le debemos indicar que enlace contra esta librería mediante `-lSDL_mixer`. Como es habitual en la

13. SDL_mixer. Gestión de sonidos.

variable de nuestro makefile que controla las librerías que estamos usando agregaremos `-lSDL_mixer`.

13.3. Inicializando la librería

Para poder utilizar esta librería el subsistema de audio debe estar inicializado. Para ello se utiliza la bandera `SDL_INIT_AUDIO` en la función `SDL_Init()`. Una vez inicializado este subsistema debemos de iniciar la librería como en las demás librerías adicionales. Las funciones de `SDL_mixer` tienen en común que comienzan por `Mix_` por lo que son fácilmente diferenciables de las nativas de `SDL`.

Para inicializar librería llamaremos a la función:

```
int Mix_OpenAudio(int frequency, Uint16 format, int channels, int  
                  chunksizes);
```

En las nuevas versiones de `SDL` esta función se encarga también de abrir el dispositivo de audio por lo que no tenemos que inicializarlo previamente. Esta es la primera función de `SDL_mixer` a la que debemos llamar en nuestra aplicación. El primer parámetro que recibe la función es la frecuencia en hertzios (Hz) que queremos reproducir el sample. Los valores habituales para este parámetro los vimos estudiando el subsistema de audio de `SDL`. Ponemos aquí un recordatorio:

- 11025: Calidad telefónica.
- 22050: Calidad de radio. Es el máximo que permite recibir el oido humano.
- 44100: Calidad CD.

Como recordarás son valores idénticos a los utilizados en el subsistema de audio de `SDL` para el manejo del sonido. El campo `format` especifica los bits y el tipo del sample. Los posibles valores que puede tomar este campo son:

- `AUDIO_U8`: Sample de 8 bits sin signo.
- `AUDIO_S8`: Sample de 8 bits con signo.
- `AUDIO_U16` o `AUDIO_U16LSB`: Sample de 16 bits sin signo en formato little-endian.
- `AUDIO_S16` o `AUDIO_S16LSB`: Sample de 16 bits sin signo en formato little-endian.
- `AUDIO_16MSB`: Sample de 16 bits sin signo en formato big-endian.

- **AUDIO_U16SYS:** Dependiendo del diseño de nuestro sistema será **AUDIO_U16LSB** si es little-endian o **AUDIO_U16MSB** si el sistema es big-endian.
- **AUDIO_S16SYS:** Dependiendo del diseño de nuestro sistema será **AUDIO_S16LSB** si es little-endian o **AUDIO_S16MSB** si el sistema es big-endian.

Como en el parámetro anterior los valores son los mismos que puede tomar dicho parámetro en las funciones que manejan el sonido nativamente en SDL. En el parámetro *channels* indicamos en el número de canales que queremos trabajar, 1 para mono, 2 para estéreo. El último de los parámetros, *chunksize*, es el tamaño de chunk que queremos especificar en nuestra aplicación. La documentación aconseja usar un valor de 4096.

Esta función de inicialización realiza las mismas tareas que la función *SDL_OpenAudioSpec* que estudiamos en el manejo nativo del sonido por parte de SDL. Existen varias constantes que podemos utilizar al definir el formato y la frecuencia que queremos utilizar si no somos unos expertos. **MIX_DEFAULT_FREQUENCY** es el valor por defecto a utilizar cuando hablamos de frecuencia, esta constante contiene el valor 22050. El formato por defecto viene dado por **MIX_DEFAULT_FORMAT** que es equivalente a **AUDIO_S16SYS**. La última de las constantes es **MIX_DEFAULT_CHANNELS** que especifica el número de canales por defecto y que equivale a establecer dos canales.

Como es habitual en las funciones que trabajan con la SDL devuelve 0 en caso de éxito y -1 si hubo algún error. Una vez utilizada la librería tenemos que llamar a la función que se encarga de cerrar la librería es:

```
void Mix_CloseAudio(void);
```

Como ha pasado con otras funciones de cometido parecido esta función es una firme candidata a ser utilizada junto a *atexit()* y no tenemos que preocupar así de realizar la llamada a *Mix_CloseAudio()*.

Una vez inicializado el sistema si queremos consultar en qué modo hemos abierto el mismo podemos utilizar la función:

```
int Mix_QuerySpec(int *frequency, Uint16 *format, int *channels);
```

Esta función devuelve 0 si ocurrió un error y un valor distinto de 0 si todo fue correctamente. Almacena en los punteros que recibe como parámetros los valores de la frecuencia, el formato y los canales que hayamos establecido para el sistema de audio. Esta función permite que comprobemos que la configuración del sistema de audio es la adecuada para los datos que carguemos como

13. SDL_mixer. Gestión de sonidos.

información de audio.

Vamos a empezar presentando conceptos que debemos conocer sobre los sonidos en el mundo del videojuego y como podemos aprovechar `SDL_mixer` para aplicar dichos conceptos a nuestra aplicación.

13.4. Los Sonidos. Chunks

Normalmente en los videojuegos existen gran cantidad de efectos sonoros. Todos los sonidos de efectos del videojuego se abstraen individualmente como un *chunk*. Siempre trabajaremos con punteros al tipo de datos *chunk*, ya que `SDL_mixer` almacena cada efecto en un *chunk*. Los chunks pueden ser cargados de ficheros de disco o desde la memoria. Normalmente se almacenan en un fichero de formato WAV o VOC. Para que nos entendamos mejor, cuando se produce una explosión o un disparo en un videojuego, el sonido que produce, eso es un chunk. Existen numerosos ejemplos de chunks.

Todos los sonidos de nuestra aplicación serán chunks. La estructura que soporta este concepto se define como

```
1 ;typedef struct {
2     int allocated;
3     Uint8 *abuf;
4     Uint32 alen;
5     Uint8 volume;
6 } Mix_Chunk;
```

Esta estructura será la encargada de almacenar los chunks. Es relativamente simple. El campo *allocated* almacena el lugar donde está el chunk, *abuf* es un puntero a un entero sin signo de 8 bits donde comienzan los datos del audio, mientras que *alen* especifica la longitud del búffer. El volumen del chunk en cuestión es almacenado en el campo *volume*.

Nunca trabajaremos directamente con la estructura *Mix_Chunk* aunque la tienes disponible para hacerlo si así lo decides. Normalmente utilizaremos punteros a ella que nos servirán de parámetros en las funciones para manejar este tipo de dato.

Una vez cargado el chunk la reproducción del sonido se hará a través de un canal elegido manualmente o bien dejaremos a `SDL_Mixer` que seleccione el que crea oportuno, ya que cada canal sólo puede reproducir un sonido en un momento dado. Para reproducir varios sonidos simultáneos tenemos que hacer uso de varios canales.

13.4.1. Cargando Chunks

Normalmente cargaremos ficheros WAV en chunks. Para cargar un fichero en su correspondiente *chunk* utilizamos la función con el prototipo:

```
Mix_Chunk *Mix_LoadWAV(char *file);
```

Le pasamos a la función como parámetro el fichero WAV que queremos cargar y esta se encarga de devolvernos un puntero a la estructura *Mix_Chunk* con el que trabajar. Si ocurre algún error la función devolverá el valor `NULL`. Alternativamente, puedes cargar un fichero WAV que ya esté almacenado en memoria con una llamada a la siguiente función:

```
Mix_Chunk *Mix_QuickLoad_WAV(UINT8 *mem);
```

El parámetro que recibe esta función es un puntero a la zona de memoria que contiene el fichero WAV. Hay muchos avisos en la documentación de `SDL_mixer` que te aconsejan que no uses esta función si no estás totalmente seguro de lo que estás haciendo y que realmente esté el sonido en esa zona de memoria. Es peligroso para la estabilidad de nuestra aplicación.

Otra función que nos permite leer directamente de memoria es la siguiente:

```
Mix_Chunk *Mix_QuickLoad_RAW(UINT8 *mem, UINT32 len);
```

Esta función recibe un puntero a memoria donde se almacena la información de audio y la longitud de los datos RAW (o en crudo). Es decir, esta función trabaja al más bajo nivel por lo que si no estás capacitado cien por cien para ello, es mejor no utilizarla.

13.4.2. Liberando Chunks

Una vez utilizado, y si ya no es necesaria su presencia en memoria principal, podemos liberar el chunk mediante la función:

```
void Mix_FreeChunk(Mix_Chunk *chunk);
```

Como puedes ver sólo tenemos que pasarle como parámetro el *chunk* a liberar.

13. SDL_mixer. Gestión de sonidos.

13.4.3. Estableciendo el Volumen

Es posible que queramos reproducir sonidos a distintos niveles de volumen. La siguiente función establece el volumen de la reproducción del sonido:

```
int Mix_VolumeChunk(Mix_Chunk *chunk, int volume);
```

Esta función recibe como parámetro el puntero al *chunk* al que queremos aplicar el volumen que le pasamos como segundo parámetro. El valor del volumen está dentro del rango entre 0 y **MIX_MAX_VOLUME**. El valor de esta constante es 128 y representa al mayor volumen posible, mientras que 0 representa la ausencia de sonido.

13.5. Canales

En inglés *channels*. Los canales permiten reproducir más de un sonido en un mismo momento. Cada canal puede reproducir un chunk diferente a la vez. Puedes decidir el número de canales que vayan a estar disponibles en el sistema con la funciones de inicialización y cambiarlo en un momento dado de la ejecución del programa. Hay opciones aplicables a cada canal que contenga un chunk, como por ejemplo el número de repeticiones, especificar cuánto tiempo se va a reproducir el sonido, o producir un efecto de fade en el canal de un determinado chunk. Podemos especificar en qué canal queremos reproducir un determinado sonido o *SDL_mixer* elegirá por nosotros la mejor opción de los que estén libres. Una vez que el sonido esté reproduciéndose podremos pausar, reanudar o parar uno de los canales, independientemente de los demás.

Los canales afectan a los sonidos de efectos pero no a la música del videojuego.

Antes de comenzar a reproducir sonidos vamos a ver qué posibilidades nos ofrecen los canales. Es importante que *SDL_Mixer* conozca el número de canales que queremos utilizar. Necesitaremos tantos canales como sonidos simultáneos queramos reproducir, es importante no quedarse corto, pero éste es un parámetro a optimizar. Hay que tener en cuenta que cuantos más canales utilicemos más recursos del sistema tendremos ocupados. Para realizar esta tarea hacemos uso de la función:

```
int Mix_AllocateChannels(int numchannels);
```

Esta función recibe como parámetro el número de canales que queremos configurar. Podemos llamar a esta función cuando queramos para modificar este número de canales existentes al mismo tiempo. No es una buena idea cambiar el número de canales con mucha frecuencia ya que si especificamos un número menor de canales de los que estaban preparados en un momento

dado tendrán que ser parados con la correspondiente carga para el sistema. Si a esta función le pasamos como parámetro 0 canales, la reproducción en todos los canales será parada.

13.5.1. Asignando Canales

Ya tenemos el sonido cargado en un *chunk*. El siguiente paso que tenemos que seguir es el de reproducirlo en un canal de sonido. *SDL_mixer* tiene cuatro funciones dedicadas a la tarea de reproducir sonidos. La primera función que se encarga de esta tarea es:

```
int Mix_PlayChannel(int channel, Mix_Chunk *chunk, int loops);
```

Como podrás observar esta función reproduce el sonido del parámetro *chunk* en el canal especificado en *channel* y si queremos reproducirlo sólo una vez deberemos de pasarle al parámetro *loops* el valor 0. Si queremos que el sonido de reproduzca una y otra vez, indefinidamente, pasaremos -1 en el parámetro *loops*.

Como comentábamos antes *SDL_mixer* tiene la capacidad de poder seleccionar el canal de reproducción automáticamente. Para conseguir esto debemos de pasarle a la función precedente el valor -1 en el parámetro *channel*.

Además de esta función, como comentamos anteriormente, *SDL_mixer* proporciona otras tres funciones para reproducir sonidos. Estas funciones son:

```
int Mix_PlayChannelTimed(int channel, Mix_Chunk *chunk, int loops, int ticks);
```

Esta función es idéntica a *Mix_PlayChannel* pero el sonido se reproducirá durante los milisegundos indicados en el parámetro *ticks*. Si pasamos el valor -1 en el parámetro *ticks* el sonido se reproducirá indefinidamente, según la configuración propuesta en los otros parámetros. La configuración de los otros parámetros es exactamente igual que en *Mix_PlayChannel*.

13.5.2. Aplicando Efectos

La siguiente función a estudiar añade un efecto a la reproducción de sonidos. El prototipo de la función es:

```
int Mix_FadeInChannel(int channel, Mix_Chunk *chunk, int loops, int ms);
```

13. SDL_mixer. Gestión de sonidos.

Esta función produce un efecto ascendente del nivel del volumen del sonido o *chunk*. El volumen del sonido irá ascendiendo desde 0 hasta el correspondiente valor definido en el *chunk*. Este aumento de volumen se hará manera gradual creando el efecto fade-in. En el parámetro *ms* indicamos los milisegundos que queremos que tarde el sonido en llegar a su volumen “normal”. Un -1 en los parámetros *loops* or *channel* significa lo mismo que en la función *Mix_PlayChannel*.

La cuarta función que nos permite la reproducción de sonidos combina las posibilidades de las dos funciones anteriores. Podemos reproducir un sonido iterativamente, durante unos milisegundos establecidos con un degradado de volumen de entrada de una duración determinada. El prototipo de la función es el siguiente:

```
int Mix_FadeInChannelTimed(int channel, Mix_Chunk *chunk, int loops,
                            int ms, int ticks);
```

El valor de los parámetros es el mismo que el de las otras funciones estudiadas.

13.5.3. Parando la reproducción

SDL_Mixer nos proporciona además la posibilidad de pausar, reanudar, parar o realizar un *fade out* el sonido que estamos reproduciendo. No podría de ser de otra forma ya que si no tendríamos una API incompleta para el manejo de sonidos. Para ello utilizamos las siguientes funciones:

```
void Mix_Pause(int channel);
void Mix_Resume(int channel);
int Mix_HaltChannel(int channel);
int Mix_FadeOutChannel(int channel, int ms);
```

Como habrás observado todas estas funciones trabajan sobre los canales de reproducción que reciben como parámetro. La primera función sirve para pausar la reproducción del canal específico pasado por parámetro. La segunda reanuda la reproducción de un sonido de un canal que previamente estaba en pausa. Si le pasamos como parámetro el valor -1 a estas funciones aplicarán la acción que tienen encargada realizar a todos los canales. Es decir, que si pasamos el valor -1 a la función encargada de la pausa, pausará todos los canales.

Las dos últimas funciones paran la reproducción del canal. Mientras que *Mix_HaltChannel()* para la reproducción en seco la función *Mix_FadeOutChannel()* crea un efecto contrario al fade in que se utiliza para abandonar un sonido suavemente. Consiste en llevar el volumen el sonido

desde el valor actual hasta 0, este proceso se realiza en tantos milisegundos como se indique en el parámetro *ms*. Ambas funciones devuelven siempre el valor 0, sí, siempre.

Existe una función más que nos permite parar un determinado canal en un determinado momento. Esta función es:

```
int Mix_ExpireChannel(int channel, int ticks);
```

A esta función le indicamos qué canal queremos parar, en el parámetro *channel*, y en cuántos milisegundos queremos hacerlo, en el parámetro *ticks*. La reproducción en el canal parará antes de que termine este tiempo. Si pasamos el valor -1 antes de expirar el tiempo serán parados todos los canales.

13.5.4. Ejemplo 1

A estas alturas hemos visto un número considerable de estructuras, conceptos y funciones. Es hora de analizar algún ejemplo que nos permita ver como utilizar estos en un programa SDL. Vamos a empezar por algo básico. Vamos a cargar un sonido y vamos a conseguir que se reproduzca por nuestros altavoces.

Vamos a analizar el código:

```

1 ;// Ejemplo 1
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Reproduciendo sonidos
5 ;
6 ;
7 ;#include <iostream>
8 ;#include <SDL/SDL.h>
9 ;
10 ;#include <SDL/SDL_mixer.h>
11 ;
12 ;using namespace std;
13 ;
14 ;int main()
15 ;{
16 ;    // Iniciamos el subsistema de video
17 ;
18 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {
19 ;
20 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
21 ;        exit(1);
22 ;
23 ;
24 ;    atexit(SDL_Quit);

```

13. SDL_mixer. Gestión de sonidos.

```
25 ;
26 ;    // Comprobamos que sea compatible el modo de video
27 ;
28 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
29 ;
30 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
31 ;        exit(1);
32 ;
33 ;    }
34 ;
35 ;    // Antes de establecer el modo de video
36 ;    // Establecemos el nombre de la ventana
37 ;
38 ;    SDL_WM_SetCaption("Prueba. SDL_mixer", NULL);
39 ;
40 ;    // Establecemos el modo
41 ;
42 ;    SDL_Surface *pantalla;
43 ;
44 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
45 ;
46 ;    if(pantalla == NULL) {
47 ;
48 ;        cerr << "No se pudo establecer el modo de video: "
49 ;            << SDL_GetError();
50 ;
51 ;        exit(1);
52 ;
53 ;    }
54 ;    // Inicializamos la librería SDL_Mixer
55 ;
56 ;    if(Mix_OpenAudio(MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT,\n
57 ;                    MIX_DEFAULT_CHANNELS, 4096) < 0) {
58 ;
59 ;        cerr << "Subsistema de Audio no disponible" << endl;
60 ;        exit(1);
61 ;
62 ;
63 ;    // Cargamos un sonido
64 ;
65 ;    Mix_Chunk *sonido;
66 ;
67 ;    sonido = Mix_LoadWAV("./Sonidos/space.wav");
68 ;
69 ;    if(sonido == NULL) {
70 ;
71 ;        cerr << "No se puede cargar el sonido" << endl;
72 ;        exit(1);
73 ;
74 ;    }
75 ;
76 ;    // Establecemos el volumen para el sonido
77 ;
```

```
78 ;     int volumen = 100;
79 ;
80 ;     Mix_VolumeChunk(sonido, volumen);
81 ;
82 ;
83 ;     // Creamos dos canales
84 ;
85 ;     Mix_AllocateChannels(2);
86 ;
87 ;     // Introducimos el sonido en uno de los canales
88 ;     // En el canal 1 con reproducción infinita (-1)
89 ;
90 ;     Mix_PlayChannel(1, sonido, -1);
91 ;
92 ;
93 ;     // Variables auxiliares
94 ;
95 ;     SDL_Event evento;
96 ;     SDL_EnableKeyRepeat(SDL_DEFAULT_REPEAT_DELAY,\n
97 ;                         SDL_DEFAULT_REPEAT_INTERVAL);
98 ;
99 ;     cout << "Pulse ESC para salir" << endl;
100 ;    cout << "Pulse Q para subir el volumen" << endl;
101 ;    cout << "Pulse A para bajar el volumen" << endl;
102 ;    cout << "Pulse P para pausar la reproducción" << endl;
103 ;    cout << "Pulse R para reanudar la reproducción" << endl;
104 ;    cout << "Pulsa H para mostrar la ayuda" << endl;
105 ;
106 ;     // Bucle infinito
107 ;
108 ;     for( ; ; ) {
109 ;
110 ;         while(SDL_PollEvent(&evento)) {
111 ;
112 ;             if(evento.type == SDL_KEYDOWN) {
113 ;
114 ;                 if(evento.key.keysym.sym == SDLK_ESCAPE) {
115 ;
116 ;                     // Liberamos el sonido
117 ;
118 ;                     Mix_FreeChunk(sonido);
119 ;
120 ;                     // Cerramos el sistema de audio
121 ;                     // al terminar de trabajar con él
122 ;
123 ;                     atexit(Mix_CloseAudio);
124 ;
125 ;                     cout << "Gracias" << endl;
126 ;
127 ;
128 ;                     return 0;
129 ;                 }
130 ;             }
```

13. SDL_mixer. Gestión de sonidos.

```
131 ; // Manejo del Volumen
132 ;
133 ; if(evento.key.keysym.sym == SDLK_q) {
134 ;
135 ;     volumen += 2;
136 ;
137 ;     if(volumen < 128)
138 ;         Mix_VolumeChunk(sonido, volumen);
139 ;     else
140 ;         volumen = 128;
141 ;
142 ;     cout << "Volumen actual: " << volumen << endl;
143 ;
144 ; }
145 ;
146 ; if(evento.key.keysym.sym == SDLK_a) {
147 ;
148 ;     volumen -= 2;
149 ;
150 ;     if(volumen > -1)
151 ;         Mix_VolumeChunk(sonido, volumen);
152 ;     else
153 ;         volumen = 0;
154 ;
155 ;     cout << "Volumen actual: " << volumen << endl;
156 ;
157 ; }
158 ;
159 ; // Manejo de la reproducción
160 ;
161 ; if(evento.key.keysym.sym == SDLK_p) {
162 ;
163 ;     Mix_Pause(-1);
164 ;
165 ;     cout << "Reproducción pausada, pulse R para reproducir"
166 ;         << endl;
167 ;
168 ; }
169 ;
170 ; if(evento.key.keysym.sym == SDLK_r) {
171 ;
172 ;     Mix_Resume(-1);
173 ;
174 ;     cout << "Reproducción reanudada" << endl;
175 ;
176 ; }
177 ;
178 ; if(evento.key.keysym.sym == SDLK_h) {
179 ;
180 ;     cout << "\n == AYUDA == " << endl;
181 ;
182 ;     cout << "Pulse ESC para salir" << endl;
183 ;     cout << "Pulse Q para subir el volumen" << endl;
```

```

184 ;           cout << "Pulse A para bajar el volumen" << endl;
185 ;           cout << "Pulse P para pausar la reproducción" << endl;
186 ;           cout << "Pulse R para reanudar la reproducción" << endl;
187 ;           cout << "Pulsa H para mostrar la ayuda" << endl;
188 ;
189 ;
190 ;       }
191 ;
192 ;   }
193 ; }
194 ;
195 ;
```

Vamos a estudiar lo novedoso del ejemplo. Una vez establecido el modo de video nos disponemos a abrir el subsistema de audio. Para ello utilizamos la función *Mix_OpenAudio()*. Para llamar a esta función utilizamos los valores por omisión recomendados en la documentación de SDL.

Seguidamente creamos una variable de tipo *Mix_Chunk* sonido donde almacenaremos un fichero WAV mediante la función *Mix_LoadWav*. Establecemos el volumen para este sonido.

El tercer paso a seguir es asignar el sonido a un canal. Como no tenemos creados canales todavía nos ponemos manos a la obra. Con la función *Mix_AllocateChannels()* creamos los canales que creamos convenientes, en este caso para una prueba y mediante la función *Mix_PlayChannel* asignamos el sonido previamente cargado a un canal determinado en reproducción infinita.

Para terminar añadimos al bucle que controla los eventos casos que nos permitan pausar, reanudar el sonido así como nos casos más que nos permiten controlar el volumen de los sonidos que estemos reproduciendo.

13.5.5. Obteniendo información

La librería *SDL_mixer* proporciona funciones para conocer información acerca de la configuración y el estado de los canales. Cuando se reproduce un sonido en un canal pueden surgir varias preguntas como... ¿se está reproduciendo un sonido? ¿está un canal pausado? ¿qué chunk está siendo reproducido en un canal?

Para el control del audio será necesario conocer el estado de un canal con respecto a su estado de reproducción, pausa y otros estados en un momento dado. Para saber si un canal está reproduciendo algún sonido se nos proporciona la siguiente función:

```
int Mix_Playing(int channel);
```

13. SDL_mixer. Gestión de sonidos.

Le indicamos mediante un parámetro de entrada el canal que queremos consultar y devuelve 1 si dicho canal está reproduciendo algún sonido y 0 en caso de que no lo esté haciendo. Si pasamos el valor -1 como parámetro de la función la función nos devolverá el número de canales que están reproduciendo algún sonido en ese mismo momento.

Para conocer si un canal está pausado hacemos uso de la función:

```
int Mix_Paused(int channel);
```

Como en la función anterior le indicamos el canal que queremos consultar, devolverá 1 si está reproduciendo algun sonido y 0 si está en pausa. Si le pasamos el valor -1 como parámetro nos devolverá el número de canales que están pausados en un momento dado.

Para saber si un canal está haciendo fading usamos la función:

```
Mix_Fading Mix_FadingChannel(int which);
```

Como parámetro le pasamos el número de canal que queremos consultar. La función devuelve uno de estas tres constantes: **MIX_NO_FADE** que significa que no se está produciendo fading en ese canal, **MIX_FADE_OUT** que denota si se está produciendo un efecto de fading out en el canal y por último **MIX_FADE_IN** que como podrás intuir, significa que se está produciendo un efecto de fading in en el canal consultado.

Para terminar vamos presentar la función que nos permite conocer que *chunk* se está reproduciendo en un determinado canal. El prototipo de la función es el siguiente:

```
Mix_Chunk* Mix_GetChunk(int channel);
```

Esta función recibe como parámetro el número de canal a consultar y devuelve cual es último chunk que se ha reproducido en dicho canal. No es necesario que se esté reproduciendo dicho chunk en este canal ya que esto lo comprobamos con la función *Mix_Playing*.

13.5.6. Ejemplo 2

Vamos a añadir nuevas funcionalidades al ejemplo 1. Más concretamente vamos a mejorar la pausa/reproducción y vamos a completar el ejercicio añadiendo respuesta a dos eventos de teclado que nos permitan realizar un FadeIn y un FadeOut sobre el sonido que se está reproduciendo. La estructura del código es la misma con estos nuevos añadidos. Aquí tienes el resultado:

```
1 ;  
1 // Ejemplo 2  
2 ;//  
3 ;// Listado: main.cpp  
4 ;// Programa de pruebas. Reproduciendo sonidos  
5 ;  
6 ;  
7 ;#include <iostream>  
8 ;#include <SDL/SDL.h>  
9 ;  
10 ;#include <SDL/SDL_mixer.h>  
11 ;  
12 ;using namespace std;  
13 ;  
14 ;int main()  
15 ;{  
16 ;    // Iniciamos el subsistema de video  
17 ;  
18 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {  
19 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;  
20 ;        exit(1);  
21 ;    }  
22 ;  
23 ;  
24 ;    atexit(SDL_Quit);  
25 ;  
26 ;    // Comprobamos que sea compatible el modo de video  
27 ;  
28 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {  
29 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;  
30 ;        exit(1);  
31 ;    }  
32 ;  
33 ;  
34 ;    // Antes de establecer el modo de video  
35 ;    // Establecemos el nombre de la ventana  
36 ;  
37 ;    SDL_WM_SetCaption("Prueba. SDL_mixer", NULL);  
38 ;  
39 ;    // Establecemos el modo  
40 ;  
41 ;    SDL_Surface *pantalla;  
42 ;  
43 ;  
44 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);  
45 ;  
46 ;    if(pantalla == NULL) {  
47 ;  
48 ;        cerr << "No se pudo establecer el modo de video: "  
49 ;            << SDL_GetError();  
50 ;  
51 ;        exit(1);  
52 ;    }  
53 ;}
```

13. SDL_mixer. Gestión de sonidos.

```
54 ; // Inicializamos la librería SDL_Mixer
55 ;
56 ; if(Mix_OpenAudio(MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT,\ 
57 ;                   MIX_DEFAULT_CHANNELS, 4096) < 0) {
58 ;
59 ;     cerr << "Subsistema de Audio no disponible" << endl;
60 ;     exit(1);
61 ;
62 ;
63 ; // Cargamos un sonido
64 ;
65 ; Mix_Chunk *sonido;
66 ;
67 ; sonido = Mix_LoadWAV("./Sonidos/space.wav");
68 ;
69 ; if(sonido == NULL) {
70 ;
71 ;     cerr << "No se puede cargar el sonido" << endl;
72 ;     exit(1);
73 ;
74 ; }
75 ;
76 ;
77 ; // Establecemos el volumen para el sonido
78 ;
79 ; int volumen = 100;
80 ;
81 ; Mix_VolumeChunk(sonido, volumen);
82 ;
83 ;
84 ; // Creamos dos canales
85 ;
86 ; Mix_AllocateChannels(2);
87 ;
88 ; // Introducimos el sonido en uno de los canales
89 ; // En el canal 1 con reproducción infinita (-1)
90 ;
91 ; Mix_PlayChannel(1, sonido, -1);
92 ;
93 ;
94 ; // Variables auxiliares
95 ;
96 ; SDL_Event evento;
97 ; SDL_EnableKeyRepeat(SDL_DEFAULT_REPEAT_DELAY,\ 
98 ;                     SDL_DEFAULT_REPEAT_INTERVAL);
99 ;
100 ; cout << "Pulse ESC para salir" << endl;
101 ; cout << "Pulse Q para subir el volumen" << endl;
102 ; cout << "Pulse A para bajar el volumen" << endl;
103 ; cout << "Pulse P para pausar la reproducción" << endl;
104 ; cout << "Pulse R para reanudar la reproducción" << endl;
105 ; cout << "Pulse F para producir un fade out" << endl;
106 ; cout << "Pulse I para producir un fade in" << endl;
```

```
107 ;     cout << "Pulse H para mostrar la ayuda" << endl;
108 ;
109 ; // Bucle infinito
110 ;
111 ; for( ; ; ) {
112 ;
113 ;     // RECUERDA: -1 en SDL_Mixer en las llamadas a función
114 ;     // simboliza infinito
115 ;
116 ;     while(SDL_PollEvent(&evento)) {
117 ;
118 ;         if(evento.type == SDL_KEYDOWN) {
119 ;
120 ;             if(evento.key.keysym.sym == SDLK_ESCAPE) {
121 ;
122 ;                 // Liberamos el sonido
123 ;
124 ;                 Mix_FreeChunk(sonido);
125 ;
126 ;                 // Cerramos el sistema de audio
127 ;                 // al terminar de trabajar con él
128 ;
129 ;                 atexit(Mix_CloseAudio);
130 ;
131 ;                 cout << "Gracias" << endl;
132 ;
133 ;
134 ;             return 0;
135 ;         }
136 ;
137 ;         // Manejo del Volumen
138 ;
139 ;         if(evento.key.keysym.sym == SDLK_q) {
140 ;
141 ;             volumen += 2;
142 ;
143 ;             if(volumen < 128)
144 ;                 Mix_VolumeChunk(sonido, volumen);
145 ;             else
146 ;                 volumen = 128;
147 ;
148 ;             cout << "Volumen actual: " << volumen << endl;
149 ;
150 ;         }
151 ;
152 ;         if(evento.key.keysym.sym == SDLK_a) {
153 ;
154 ;             volumen -= 2;
155 ;
156 ;             if(volumen > -1)
157 ;                 Mix_VolumeChunk(sonido, volumen);
158 ;             else
159 ;                 volumen = 0;
```

13. SDL_mixer. Gestión de sonidos.

```
160 ;
161 ;           cout << "Volumen actual: " << volumen << endl;
162 ;
163 ;       }
164 ;
165 ;       // Manejo de la reproducción
166 ;
167 ;       if(evento.key.keysym.sym == SDLK_p) {
168 ;
169 ;           if(Mix_Playing(-1) > 0) {
170 ;
171 ;               Mix_Pause(-1);
172 ;
173 ;               cout << "Reproducción pausada, pulse R para reproducir"
174 ;                   << endl;
175 ;
176 ;           } else {
177 ;
178 ;               cout << "La reproducción ya está pausada" << endl;
179 ;           }
180 ;
181 ;       }
182 ;
183 ;       if(evento.key.keysym.sym == SDLK_r) {
184 ;
185 ;           if(Mix_Paused(-1) > 0) {
186 ;
187 ;               Mix_Resume(-1);
188 ;
189 ;               cout << "Reproducción reanudada" << endl;
190 ;
191 ;           } else {
192 ;
193 ;               cout << "La reproducción ya está activada" << endl;
194 ;
195 ;           }
196 ;
197 ;       }
198 ;       // Efectos
199 ;
200 ;       if(evento.key.keysym.sym == SDLK_f) {
201 ;
202 ;           Mix_FadeOutChannel(-1, 5000);
203 ;
204 ;           cout << "FadeOut: Pulse i para FadeIn" << endl;
205 ;
206 ;       }
207 ;
208 ;       if(evento.key.keysym.sym == SDLK_i) {
209 ;
210 ;           Mix_FadeInChannel(-1, sonido, -1, 5000);
211 ;
212 ;           cout << "FadeIn: Realizando FadeIn" << endl;
```

```

213 ;
214 ;
215 ;
216 ;           if(evento.key.keysym.sym == SDLK_h) {
217 ;
218 ;               cout << "\n == AYUDA == " << endl;
219 ;
220 ;               cout << "Pulse ESC para salir" << endl;
221 ;               cout << "Pulse Q para subir el volumen" << endl;
222 ;               cout << "Pulse A para bajar el volumen" << endl;
223 ;               cout << "Pulse P para pausar la reproducción" << endl;
224 ;               cout << "Pulse R para reanudar la reproducción" << endl;
225 ;               cout << "Pulse F para producir un fade out" << endl;
226 ;               cout << "Pulse I para producir un fade in" << endl;
227 ;               cout << "Pulse H para mostrar la ayuda" << endl;
228 ;
229 ;           }
230 ;
231 ;       }
232 ;
233 ;   }
234 ; }
235 ;
236 ;}
;
```

13.6. Grupos

SDL_mixer nos permite agrupar un número de canales que formen un grupo. Esto nos permite usar este grupo para reproducir un tipo determinado de sonidos de nuestra aplicación. Por ejemplo, podemos separar los canales que vamos a usar para reproducir sonido de efectos (SFX) de los canales utilizados, por ejemplo, para reproducir voz (VOC). Los grupos se utilizan para manejar varios canales a la vez, pudiendo parar, reproducir, pausar... varios de ellos al mismo tiempo. Es una buena práctica utilizar grupos para organizar nuestros canales.

13.6.1. Configurando los grupos

Lo primero que tenemos que hacer es configurar los canales para que no sean dominados cuando una función recibe el parámetro -1, por ejemplo para la reproducción. Para hacer esto SDL_mixer proporciona la función:

int Mix_ReserveChannels(int num);

La función recibe como parámetro un número de canal. El valor devuelto es el número de canales “reservados”, es decir, que no pueden ser dominados por el canal o parámetro -1. El siguiente paso a realizar es agrupar los canales.

13. SDL_mixer. Gestión de sonidos.

Podemos realizar esta tarea canal a canal o mediante rangos de canales. Para hacerlo canal a canal SDL_mixer proporciona la siguiente función:

```
int Mix_GroupChannel(int which, int tag);
```

Esta función recibe como parámetro un número de canal *which* y el grupo al que queremos añadir el canal *tag*. La función devuelve 0 en el caso de que haya existido algún problema y 1 si se agrupa el canal correctamente. Si marcamos o añadimos un canal al grupo -1 le estamos quitando la exclusividad dejando de ser reservado.

EJEMPLO 215 FOSDL

Para realizar la misma tarea pero utilizando rango de canales podemos usar la siguiente función:

```
int Mix_GroupChannels(int from, int to, int tag);
```

Esta función recibe como parámetros el inicio del rango *from*, el último canal a agrupar *to* y el grupo al que queremos que pertenezcan *tag*. Esta última función es mucho más eficiente que su predecesora.

13.6.2. Obteniendo información

Ya sabemos configurar grupos. Existen varias funciones que nos permiten obtener información acerca de los grupos existentes. La primera de exxas nos permite saber cuántos canales pertenecen a un determinado grupo. SDL_mixer proporciona con este fin la siguiente función:

```
int Mix_GroupCount(int tag);
```

Esta función recibe como parámetro el grupo a consultar (*tag*) y devuelve el número de canales que pertenecen a dicho grupo. Si el valor devuelto es 0 significa que no existen canales en ese grupo. Si pasamos como parámetro -1 la función devolverá el número total de canales.

Si tenemos un número límitado de canales puede ser interesante conocer que canal dentro de un grupo de canales lleva más tiempo reproduciendo un sonido con la idea de pararlo y reutilizarlo. Para obtener esta información y la de cual es el último grupo en reproducir un sonido SDL proporciona dos funciones:

```
int Mix_GroupOldest(int tag); int Mix_GroupNewer(int tag);
```

Las dos funciones reciben como parámetro el grupo a consultar. La primera función devuelve cuál es el canal del grupo que lleva más tiempo reproduciendo un sonido. Si el valor devuelto es -1 existen dos posibles significados. El primero que no haya canales en el grupo. El segundo que no haya reproducido ninguno algún sonido todavía.

La segunda función devuelve cual es el canal dentro del grupo que lleva menos tiempo reproduciendo sonido, es decir, el más nuevo. Si la función devuelve -1 el significado es idéntico al de la función anterior.

13.6.3. Efectos sobre los grupos

Para terminar vamos a presentar dos funciones que van a ser fundamentales si realizamos separación de sonidos en canales de efectos y de voz, por ejemplo. Podemos realizar el efecto de fade-out de un grupo de canales mediante la función:

```
int Mix_FadeOutGroup(int tag, int ms);
```

El parámetro *tag* indica que grupo es al que queremos aplicar el efecto, mientras que en el parámetro *ms* indicamos durante cuantos milisegundos queremos que se realice dicho efecto. La función devuelve el número de canales a los que se les va a aplicar el efecto en cuestión.

Para parar la reproducción de los canales del grupo *SDL_mixer* proporciona la función:

```
int Mix_HaltGroup(int tag);
```

Como es habitual la función recibe como parámetro el número que identifica al grupo de canales que queremos parar. Esta función devuelve siempre 0 por lo que no es necesario que comprobemos el valor devuelto.

13.6.4. Ejemplo 3

Antes de empezar con el tema referente a la música o banda sonora del videojuego vamos a poner en práctica las funciones que hemos estudiado. Aquí tienes un ejemplo similar a los anteriores pero aplicando los efectos/acciones a grupos de sonidos:

```
1 ;// Ejemplo 3
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Reproduciendo sonidos en grupos
```

13. SDL_mixer. Gestión de sonidos.

```
5 ;
6 ;
7 ;#include <iostream>
8 ;#include <SDL/SDL.h>
9 ;
10 ;#include <SDL/SDL_mixer.h>
11 ;
12 ;using namespace std;
13 ;
14 ;int main()
15 ;{
16 ;    // Iniciamos el subsistema de video
17 ;
18 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {
19 ;
20 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
21 ;        exit(1);
22 ;    }
23 ;
24 ;    atexit(SDL_Quit);
25 ;
26 ;    // Comprobamos que sea compatible el modo de video
27 ;
28 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
29 ;
30 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
31 ;        exit(1);
32 ;
33 ;    }
34 ;
35 ;    // Antes de establecer el modo de video
36 ;    // Establecemos el nombre de la ventana
37 ;
38 ;    SDL_WM_SetCaption("Prueba. SDL_mixer", NULL);
39 ;
40 ;    // Establecemos el modo
41 ;
42 ;    SDL_Surface *pantalla;
43 ;
44 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
45 ;
46 ;    if(pantalla == NULL) {
47 ;
48 ;        cerr << "No se pudo establecer el modo de video: "
49 ;            << SDL_GetError();
50 ;
51 ;        exit(1);
52 ;    }
53 ;
54 ;    // Inicializamos la librería SDL_Mixer
55 ;
56 ;    if(Mix_OpenAudio(MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT,\n
57 ;                    MIX_DEFAULT_CHANNELS, 4096) < 0) {
```

13.6. Grupos

```
58 ;
59 ;     cerr << "Subsistema de Audio no disponible" << endl;
60 ;     exit(1);
61 ;
62 ;
63 ; // Cargamos un sonido
64 ;
65 ; Mix_Chunk *sonido1, *sonido2;
66 ;
67 ; sonido1 = Mix_LoadWAV("./Sonidos/space.wav");
68 ; sonido2 = Mix_LoadWAV("./Sonidos/aplauso.wav");
69 ;
70 ; if(sonido1 == NULL || sonido2 == NULL) {
71 ;
72 ;     cerr << "Error al cargar los sonidos" << endl;
73 ;     exit(1);
74 ;
75 ; }
76 ;
77 ;
78 ; // Establecemos el volumen para el sonido
79 ;
80 ; int volumen = 100;
81 ;
82 ; Mix_VolumeChunk(sonido1, volumen);
83 ; Mix_VolumeChunk(sonido2, volumen);
84 ;
85 ;
86 ; // Creamos cuatro canales
87 ;
88 ; Mix_AllocateChannels(4);
89 ;
90 ;
91 ; // Reservo los cuatro canales
92 ; // Para que no estén a disposición de SDL_mixer
93 ;
94 ; Mix_ReserveChannels(4);
95 ;
96 ; // Creamos dos grupos de canales
97 ;
98 ; // UNO: Canales 0 y 1
99 ;
100 ; Mix_GroupChannel(0, 1);
101 ; Mix_GroupChannel(1, 1);
102 ;
103 ; // DOS: Canales 2 y 3
104 ;
105 ; Mix_GroupChannels(2, 3, 2);
106 ;
107 ; cout << "Canales creados" << endl;
108 ;
109 ; // Mostramos información de los grupos
110 ;
```

13. SDL_mixer. Gestión de sonidos.

```
111 ;     cout << "Información de los canales" << endl;
112 ;
113 ;     cout << "-----" << endl;
114 ;
115 ;     cout << "Grupo 1: " << Mix_GroupCount(1) << " canales" << endl;
116 ;     cout << "Grupo 2: " << Mix_GroupCount(2) << " canales" << endl;
117 ;
118 ;     cout << "-----" << endl;
119 ;
120 ;
121 ; // Introducimos los sonidos en canales
122 ;
123 ;     Mix_PlayChannel(1, sonido1, -1);
124 ;     Mix_PlayChannel(3, sonido2, -1);
125 ;
126 ; // Variables auxiliares
127 ;
128 ;     SDL_Event evento;
129 ;     SDL_EnableKeyRepeat(SDL_DEFAULT_REPEAT_DELAY,\n
130 ;                         SDL_DEFAULT_REPEAT_INTERVAL);
131 ;
132 ;     cout << "Pulse ESC para salir" << endl;
133 ;     cout << "Pulse Q para subir el volumen" << endl;
134 ;     cout << "Pulse A para bajar el volumen" << endl;
135 ;     cout << "Pulse P para pausar la reproducción del grupo 1" << endl;
136 ;     cout << "Pulse O para pausar la reproducción del grupo 2" << endl;
137 ;     cout << "Pulse F para producir un fade out del grupo 1" << endl;
138 ;     cout << "Pulse I para producir un fade out del grupo 2" << endl;
139 ;     cout << "Pulse H para mostrar la ayuda" << endl;
140 ;
141 ; // Bucle infinito
142 ;
143 ; for( ; ; ) {
144 ;
145 ;     // RECUERDA: -1 en SDL_Mixer en las llamadas a función
146 ;     // simboliza infinito
147 ;
148 ;     while(SDL_PollEvent(&evento)) {
149 ;
150 ;         if(evento.type == SDL_KEYDOWN) {
151 ;
152 ;             if(evento.key.keysym.sym == SDLK_ESCAPE) {
153 ;
154 ;                 // Liberamos los sonidos
155 ;
156 ;                 Mix_FreeChunk(sonido1);
157 ;                 Mix_FreeChunk(sonido2);
158 ;
159 ;                 // Cerramos el sistema de audio
160 ;                 // al terminar de trabajar con él
161 ;
162 ;                 atexit(Mix_CloseAudio);
163 ;
```

13.6. Grupos

```
164 ;           cout << "Gracias" << endl;
165 ;
166 ;
167 ;       return 0;
168 ;
169 ;
170 ;   // Manejo del Volumen
171 ;
172 ;   if(evento.key.keysym.sym == SDLK_q) {
173 ;
174 ;       volumen += 2;
175 ;
176 ;       if(volumen < 128) {
177 ;           Mix_VolumeChunk(sonido1, volumen);
178 ;           Mix_VolumeChunk(sonido2, volumen);
179 ;       }
180 ;       else
181 ;           volumen = 128;
182 ;
183 ;       cout << "Volumen actual: " << volumen << endl;
184 ;
185 ;   }
186 ;
187 ;   if(evento.key.keysym.sym == SDLK_a) {
188 ;
189 ;       volumen -= 2;
190 ;
191 ;       if(volumen > -1) {
192 ;           Mix_VolumeChunk(sonido1, volumen);
193 ;           Mix_VolumeChunk(sonido2, volumen);
194 ;       }
195 ;       else
196 ;           volumen = 0;
197 ;
198 ;       cout << "Volumen actual: " << volumen << endl;
199 ;
200 ;   }
201 ;
202 ;   // Manejo de la reproducción
203 ;
204 ;   if(evento.key.keysym.sym == SDLK_p) {
205 ;
206 ;       Mix_HaltGroup(1);
207 ;       cout << "Grupo 1 parado" << endl;
208 ;   }
209 ;
210 ;   if(evento.key.keysym.sym == SDLK_o) {
211 ;
212 ;       Mix_HaltGroup(2);
213 ;       cout << "Grupo 2 parado" << endl;
214 ;   }
215 ;
216 ;   // Efectos
```

13. SDL_mixer. Gestión de sonidos.

```
217 ;
218 ;           if(evento.key.keysym.sym == SDLK_f) {
219 ;
220 ;               int num;
221 ;
222 ;               num = Mix_FadeOutGroup(1, 5000);
223 ;
224 ;               cout << "FadeOut 1: " << num << " canales" << endl;
225 ;
226 ;           }
227 ;
228 ;           if(evento.key.keysym.sym == SDLK_i) {
229 ;
230 ;               int num;
231 ;
232 ;               num = Mix_FadeOutGroup(2, 5000);
233 ;
234 ;               cout << "FadeOut 2: " << num << " canales" << endl;
235 ;
236 ;           }
237 ;
238 ;           if(evento.key.keysym.sym == SDLK_h) {
239 ;
240 ;               cout << "\n == AYUDA == " << endl;
241 ;
242 ;
243 ;               cout << "Pulse ESC para salir" << endl;
244 ;               cout << "Pulse Q para subir el volumen" << endl;
245 ;               cout << "Pulse A para bajar el volumen" << endl;
246 ;               cout << "Pulse P para pausar la reproducción del grupo 1" << endl;
247 ;               cout << "Pulse O para pausar la reproducción del grupo 2" << endl;
248 ;               cout << "Pulse F para producir un fade out del grupo 1" << endl;
249 ;               cout << "Pulse I para producir un fade out del grupo 2" << endl;
250 ;               cout << "Pulse H para mostrar la ayuda" << endl;
251 ;
252 ;
253 ;           }
254 ;
255 ;       }
256 ;
257 ;   }
258 ; }
259 ;
260 ;}
```

13.7. Música

La música en `SDL_mixer` es manejada de forma parecida a un efecto de sonido de la aplicación en aunque existe una diferencia importante. Sólo puede reproducirse una canción al mismo tiempo por lo que no puede ser separada

en canales y grupos. Con este propósito existen funciones que permiten cargar una gran variedad de tipos de ficheros de audio, como por ejemplo el MP3. Una vez cargada la música la podemos reproducir, pausar, continuar, parar, realizar un fade in, fade out y cambiar el momento de reproducción de la música.

13.7.1. Cargando la Banda Sonora

SDL_Mixer se encarga de reservar un canal exclusivo para la reproducción de música para nuestra aplicación. Como en la gestión de sonidos da soporte a multitud de formatos. Como ocurría con los chunks, la música tiene que ser cargada de un fichero y una vez que terminemos de trabajar con ella deberemos de liberar el recurso. Para cargar la música del videojuego utilizamos la función:

```
Mix_Music *Mix_LoadMUS(const char *file);
```

Esta función recibe como parámetro el fichero donde se encuentra la música que queremos cargar. La función devuelve un puntero a la estructura de datos que guarda la música del videojuego de tipo *Mix_Music*. La única diferencia con la función de cargar sonidos “convencionales” es que no tenemos que especificar el canal en el que queremos colocar el sonido. Como puedes observar existe un tipo exclusivo de datos dedicado a este tipo de sonidos. Como pasa con otros tipos de datos la definición de *Mix_Music* está oculta al usuario, lo que no nos supone ningún problema ya que trabajaremos siempre con un puntero a esa estructura y alterando los campos de la misma directamente.

Si ocurre algún error al cargar el fichero de música la función devuelve el valor **NULL**. El tipo de formato que podemos reproducir es muy amplio como comentábamos al principio del capítulo. Entre ellos se encuentran el MP3, MOD, WAV... y otros.

Como comentamos hace unas líneas cuando no necesitemos tener la música disponible en memoria principal deberemos liberar ésta para realizar una buena gestión de nuestros recursos. La función que permite realizar esta tarea es:

```
void Mix_FreeMusic(Mix_Music);
```

Esta función recibe como parámetro el puntero devuelto al cargar el fichero de música y como ya hemos dicho libera los recursos ocupados por ella.

13.7.2. Reproduciendo la Música

Igual que con los sonidos tenemos varias funciones que nos permiten controlar el estado de la reproducción. Para iniciarla SDL_mixer nos ofrece dos funciones. La primera de ella es:

13. SDL_mixer. Gestión de sonidos.

```
int Mix_PlayMusic(Mix_Music *music, int loops);
```

Esta función recibe como parámetro un puntero a la estructura *Mix_Music* devuelta por la función que carga la música en memoria. Además recibe en el parámetro *loops* el número de veces que ha de repetirse el nuestro tema. Si queremos reproducirlo una única vez pasamos 0 como valor y -1 si queremos hacerlo indefinidamente.

Existe otra función para iniciar la reproducción de la música. Esta añade el efecto fade-in al comienzo de la primera iteración que realice el tema. La función:

```
int Mix_FadeInMusic(Mix_Music *music, int loops, int ms);
```

Igual que la función que hemos visto cuando procesábamos los sonidos, reproduce la música creando un efecto *fade in* durante los milisegundos que le indiquemos en el parámetro *ms*. El parámetro *loops* tiene el mismo comportamiento que en la función anterior. Si todo fue bien la función devolverá 0, si por el contrario existió algún error la función devolverá -1. Existe otra función con exactamente las mismas características que esta pero que nos permite iniciar la reproducción de la música en una posición diferente a la del comienzo de la misma. La especificación es idéntica a la de la función anterior añadiendo un parámetro que controla la posición donde queremos comenzar. El prototipo de la función es:

```
int Mix_FadeInMusicPos(Mix_Music *music, int loops, int ms, double position);
```

El parámetro *position* es el que nos permite iniciar la reproducción desde un punto diferente al comienzo de la música y se especifica, normalmente, en segundos aunque depende del tipo de ficheros con el que vayamos a trabajar. Esto es muy útil si tenemos un fichero de música con varias canciones y queremos que se reproduzca una diferente por cada nivel.

Como con los canales podemos controlar varios aspectos de la reproducción de la música como pueden ser parar y reanudar la reproducción. Además podemos controlar el volumen como reiniciar la música desde el principio o establecer la reproducción en un punto distinto al actual. La primera que vamos a presentar es la que nos permite establecer el volumen de la música de la aplicación con la función:

```
int Mix_VolumeMusic(int volume);
```

Como parámetro le pasamos un valor dentro del rango de 0 a 128, donde 0 es el silencio y 128 el máximo volumen. Existe una constante, **MIX_MAX_VOLUME**, que equivale al valor máximo que puede tomar el volumen

de sonido. La función devuelve el valor previo al que estaba establecido el volumen. Si queremos consultar a qué volumen se encuentra establecida la música basta con pasar como parámetro a esta función el valor -1.

Una de los aspectos a tener en cuenta cuando estudiamos los sonidos era la capacidad de pausar la reproducción en determinados canales o grupos de canales. La función encargada de realizar esta acción en SDL es:

```
void Mix_PauseMusic();
```

Esta función no recibe ni devuelve ningún parámetro. Lo mismo ocurre con la siguiente función que es la encargada de reanudar la reproducción de la música una vez pausada:

```
void Mix_ResumeMusic();
```

SDL_Mixer ofrece un mayor control sobre la música que sobre los sonidos y añade a sus características la posibilidad de situar la reproducción de la música en el lugar que deseemos. Contamos con dos funciones que nos permiten realizar esta tarea. Si queremos reempezar la reproducción de la música, o lo que es lo mismo, comenzar a reproducir de nuevo desde el comienzo SDL_mixer proporciona una función para facilitarnos esta tarea:

```
void Mix_RewindMusic();
```

Esta función devuelve la reproducción de la música al principio de la misma. Como ocurría en las otras dos funciones anteriores la función no recibe ni devuelve ningún parámetro. Con la siguiente función podríamos conseguir el efecto que acabamos de presentar y otros tantos de unas características similares:

```
int Mix_SetMusicPosition(double position);
```

Esta función nos permite situar la posición de la reproducción en el lugar deseado. Este parámetro dependerá explícitamente del formato del archivo de música que estemos reproduciendo. En el caso de ser archivos de música digitalizada nos encontramos con varios casos. Si el fichero tiene el formato OGG la unidad de tiempo para el parámetro *position* es el milisegundo desde el comienzo de la música. Sin embargo si el fichero es MP3 especificaremos en el parámetro *position* el número de segundos que queremos saltar hacia adelante en la canción, nunca hacia atrás. Para un archivo MOD o MIDI la posición viene dada por el patrón o compás dentro del mismo fichero de audio. Como podrás ver el formato del fichero de música es mucho más importante de lo que puede parecer en un principio.

Las siguientes funciones a estudiar son las que nos permiten parar la reproducción de la música de nuestro videojuego. Éstas son:

13. SDL_mixer. Gestión de sonidos.

```
int Mix_HaltMusic();
int Mix_FadeOutMusic(int ms);
```

Estas funciones paran la reproducción de la música del videojuego o de la aplicación. La diferencia entre ellas te será familiar. La segunda recibe un parámetro *ms* dónde indicamos en milisegundos el tiempo que queremos que dure en efecto *fade out* mientras que en la primera la parada de la reproducción es inmediata en “seco”. Si existiese algún problema al intentar parar la reproducción *Mix_FadeOutMusic()* devolverá el valor -1 y en caso de éxito devolverá 0. La función *Mix_HaltMusic()* no recibe ningún parámetro y devuelve siempre el valor 0.

SDL_mixer nos permite configurar una función de callback para notificar cuando ha terminado la música de sonar, mejor dicho, nos notifica cuando la música ha parado. La función que nos permite establecer esta configuración es la siguiente:

```
void Mix_MixHookMusicFinished(void (*music_finished)());
```

Esta función recibe como parámetro un puntero a otra función que no reciba ningún parámetro y que no devuelva nada. Cuando la música pare la función pasada como parámetro será automáticamente llamada para su ejecución.

13.7.3. Obteniendo información de la Música

SDL_mixer proporciona funciones que te permiten conocer el estado de la música del videojuego en un determinado momento e información adicional. Por ejemplo, necesitamos pasar un parámetro *ms* a una función y no sabemos que unidad de tiempo utilizar ya que es dependiente del formato de fichero de audio que utilicemos. La función que nos permite conocer con qué tipo de fichero estamos trabajando es:

```
Mix_MusicType Mix_GetMusicType(const Mix_Music *music);
```

Esta función recibe un puntero a un objeto *Mix_Music* que no será modificado y devuelve una constante que define el tipo de música que se está reproduciendo. Los posibles valores de esta constante son: **MUS_CMD**, **MUS_WAV**, **MUS_MOD**, **MUS_MID**, **MUS_OGG** y **MUS_MP3** o **MUS_NONE**. Si el valor devuelto es **MUS_CMD** significará que se ha optado por utilizar un reproductor de música externo. El valor **MUS_NONE** indica que no hay música reproduciéndose. En todos los demás casos, las constantes especifican claramente el tipo de fichero de música del videojuego.

Si pasamos el valor `NULL` como parámetro la función intentará determinar que tipo de fichero de audio es el que se está reproduciendo en el canal de música.

Para conocer el estado del canal de música utilizamos las siguientes funciones:

```
int Mix_PlayingMusic(void);
```

Esta función sirve para consultar si se está reproduciendo la música en su canal, claro está. Devuelve 1 si se está reproduciendo música y 0 en caso de no estar reproduciéndola. La función no recibe ningún parámetro. Otra función que nos permite conocer el estado del canal es:

```
int Mix_PausedMusic(void);
```

Mediante esta función sabemos si la reproducción está pausada actualmente. La función devuelve 0 si la música no ha sido pausada y 1 si lo fue. Para conocer si al canal de música se le está aplicando un efecto fade utilizamos la función:

```
Mix_FadingMix_FadingMusic();
```

Esta función no recibe ningún parámetro y devuelve `MIX_NO_FADE`, `MIX_FADE_OUT` o `MIX_FADE_IN` para saber si existe *fade* en este momento y que tipo de *fade* se realiza.

13.7.4. Ejemplo 4

Ya tenemos suficiente material para afrontar otro ejercicio. Vamos a utilizar una gran parte de las funciones que hemos visto para el manejo de música con `SDL_mixer` y vamos a preparar un pequeño reproductor que nos muestre la información por consola capturando los eventos de la ventana de `SDL`.

```
1 ;// Ejemplo 4
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Añadiendo una BSO
5 ;
6 ;
7 ;#include <iostream>
8 ;#include <SDL/SDL.h>
9 ;
10 ;#include <SDL/SDL_mixer.h>
11 ;
12 ;using namespace std;
```

13. SDL_mixer. Gestión de sonidos.

```
13 ;
14 ;int main()
15 ;{
16 ;    // Iniciamos el subsistema de video
17 ;
18 ;    if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {
19 ;
20 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
21 ;        exit(1);
22 ;
23 ;
24 ;    atexit(SDL_Quit);
25 ;
26 ;    // Comprobamos que sea compatible el modo de video
27 ;
28 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
29 ;
30 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
31 ;        exit(1);
32 ;
33 ;
34 ;
35 ;    // Antes de establecer el modo de video
36 ;    // Establecemos el nombre de la ventana
37 ;
38 ;    SDL_WM_SetCaption("Prueba. SDL_mixer", NULL);
39 ;
40 ;    // Establecemos el modo
41 ;
42 ;    SDL_Surface *pantalla;
43 ;
44 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
45 ;
46 ;    if(pantalla == NULL) {
47 ;
48 ;        cerr << "No se pudo establecer el modo de video: "
49 ;            << SDL_GetError();
50 ;
51 ;        exit(1);
52 ;
53 ;
54 ;    // Inicializamos la librería SDL_Mixer
55 ;
56 ;    if(Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT,\n
57 ;                    MIX_DEFAULT_CHANNELS, 4096) < 0) {
58 ;
59 ;        cerr << "Subsistema de Audio no disponible" << endl;
60 ;        exit(1);
61 ;
62 ;
63 ;    // Al salir cierra el subsistema de audio
64 ;
65 ;    atexit(Mix_CloseAudio);
```

```

66 ;
67 ; // Cargamos un fichero para la BSO
68 ;
69 ; Mix_Music *bso;
70 ;
71 ; bso = Mix_LoadMUS("./Sonidos/bso.mp3");
72 ;
73 ; if(bso == NULL) {
74 ;
75 ;     cerr << "No se puede cargar el fichero bso.mp3" << endl;
76 ;     exit(1);
77 ; }
78 ;
79 ; // Variables auxiliares
80 ;
81 ; SDL_Event evento;
82 ; SDL_EnableKeyRepeat(SDL_DEFAULT_REPEAT_DELAY,\n
83 ;                     SDL_DEFAULT_REPEAT_INTERVAL);
84 ;
85 ; int volumen = 100;
86 ;
87 ; cout << "Pulse ESC para salir" << endl;
88 ; cout << "Pulse Q para subir el volumen" << endl;
89 ; cout << "Pulse A para bajar el volumen" << endl;
90 ; cout << "Pulse I para reproducir la música" << endl;
91 ; cout << "Pulse P para pausar la reproducción" << endl;
92 ; cout << "Pulse R para reproducir la música después del pause" << endl;
93 ; cout << "Pulse W para poner la música al inicio" << endl;
94 ; cout << "Pulse H para mostrar ayuda" << endl;
95 ;
96 ; // Bucle infinito
97 ;
98 ; for( ; ; ) {
99 ;
100 ;     // RECUERDA: -1 en SDL_Mixer en las llamadas a función
101 ;     // simboliza infinito
102 ;
103 ;     while(SDL_PollEvent(&evento)) {
104 ;
105 ;         if(evento.type == SDL_KEYDOWN) {
106 ;
107 ;             if(evento.key.keysym.sym == SDLK_ESCAPE) {
108 ;
109 ;                 // Cerramos el sistema de audio
110 ;                 // al terminar de trabajar con él
111 ;
112 ;                 Mix_FreeMusic(bso);
113 ;
114 ;                 cout << "Gracias" << endl;
115 ;
116 ;
117 ;                 return 0;
118 ;             }

```

13. SDL_mixer. Gestión de sonidos.

```
119 ;
120 ;           // Manejo del Volumen
121 ;
122 ;       if(evento.key.keysym.sym == SDLK_q) {
123 ;
124 ;           volumen += 2;
125 ;
126 ;           if(volumen < 128)
127 ;               Mix_VolumeMusic(volumen);
128 ;           else
129 ;               volumen = 128;
130 ;
131 ;           cout << "Volumen actual: " << volumen << endl;
132 ;
133 ;       }
134 ;
135 ;       if(evento.key.keysym.sym == SDLK_a) {
136 ;
137 ;           volumen -= 2;
138 ;
139 ;           if(volumen > -1)
140 ;               Mix_VolumeMusic(volumen);
141 ;           else
142 ;               volumen = 0;
143 ;
144 ;           cout << "Volumen actual: " << volumen << endl;
145 ;
146 ;       }
147 ;
148 ;           // Control de la reproducción
149 ;
150 ;       if(evento.key.keysym.sym == SDLK_i) {
151 ;
152 ;           Mix_PlayMusic(bso, -1);
153 ;
154 ;           cout << "Música iniciada" << endl;
155 ;
156 ;       }
157 ;
158 ;       if(evento.key.keysym.sym == SDLK_p) {
159 ;
160 ;           if(Mix_PlayingMusic() == 1) {
161 ;
162 ;               Mix_PauseMusic();
163 ;
164 ;               cout << "Música en pausa" << endl;
165 ;
166 ;           } else {
167 ;
168 ;               cout << "La música no está en reproducción" << endl;
169 ;
170 ;           }
171 ;
```

```

172 ; }
173 ;
174 ; if(evento.key.keysym.sym == SDLK_r) {
175 ;
176 ;     if(Mix_PausedMusic() == 1) {
177 ;
178 ;         Mix_ResumeMusic();
179 ;
180 ;         cout << "Música en reproducción" << endl;
181 ;     } else {
182 ;
183 ;         cout << "La música ya está en reproducción" << endl;
184 ;
185 ;     }
186 ;
187 ;
188 ;     if(evento.key.keysym.sym == SDLK_w) {
189 ;
190 ;         Mix_RewindMusic();
191 ;
192 ;         cout << "Música al principio" << endl;
193 ;
194 ;     }
195 ;
196 ;     if(evento.key.keysym.sym == SDLK_h) {
197 ;
198 ;         cout << "\n == AYUDA == " << endl;
199 ;         cout << "Pulse ESC para salir" << endl;
200 ;         cout << "Pulse Q para subir el volumen" << endl;
201 ;         cout << "Pulse A para bajar el volumen" << endl;
202 ;         cout << "Pulse I para reproducir la música" << endl;
203 ;         cout << "Pulse P para pausar la reproducción" << endl;
204 ;         cout << "Pulse R para reproducir la música después del pause" << endl;
205 ;         cout << "Pulse W para poner la música al inicio" << endl;
206 ;         cout << "Pulse H para mostrar ayuda" << endl;
207 ;
208 ;     }
209 ;
210 ; }
211 ; }
212 ;
213 ; }
214 ;
215 ;}
;
```

Como puedes ver, a estas alturas, todos los aspectos del código son conocidos. ¿Te atreves a proporcionar una interfaz gráfica?

13.8. Effects

El tema de los efectos de sonido es un aspecto muy interesante si disponemos de un hardware específico que nos permite tratar dichos sonidos adecuadamente. No vamos a tratar en profundidad este tema en el tutorial, pero sí vamos a realizar una pequeña introducción a la materia.

Esta librería te permite el uso de efectos predefinidos o creados por ti mismos. Como es de esperar, estos efectos son aplicados sobre los chunks que utilicemos en nuestra aplicación. Este es el último aspecto a tratar sobre *SDL_mixer* y trata sobre los efectos referentes a la posición virtual del sonido en 3D.

Los efectos son aplicados a canales individuales o a grupos indicando `MIX_CHANNEL_POST` en uno de los parámetros que veremos a continuación. Un concepto importante es que cada efecto surge con la idea de registrarse en un canal en particular produciendo los efectos en dicho canal. El efecto se mantendrá hasta que sea eliminado de dicho canal.

13.8.1. Efectos de Posicionamiento

Con los efectos de posicionamiento de *SDL_mixer* podemos especificar el volumen o *panning* de un canal, la distancia a la que será oido el sonido, la posición de la que llega el sonido, y el intercambio de los canales estéreos.

Para especificar el panning en un determinado canal utilizamos la función:

```
int Mix_SetPanning(int channel, Uint8 left, Uint8 right);
```

Esta función recibe tres parámetros. El primero el número de canal dónde vamos aplicar el efecto, el segundo el volumen para el altavoz izquierdo y el tercero se corresponde con el valor del volumen para el altavoz derecho. El rango de estos dos parámetros se establece entre 0 como silencio y 255 como el volumen más alto que se puede configurar. Esta función devuelve 0 si ocurrió algún error y un valor distinto si todo fue bien. Para quitar este efecto basta con establecer los volúmenes izquierdo y derecho a 255.

Si quieres establecer un sonido que venga de una distancia más o menos lejana puedes utilizar la función:

```
int Mix_SetDistance(int channel, Uint8 distance);
```

Esta función recibe como parámetro el número del canal donde registrar el efecto. El otro parámetro que recibe es la distancia a la que deseamos reproducir el sonido, evidentemente, una distancia virtual. El rango de este último parámetro va desde 0, que significa cerca y fuerte, hasta 255 que representa un sonido leve y alejado. Para desactivar este efecto basta con llamar a la función estableciendo el parámetro *distance* a 0.

Podemos simular una posición tridimensional mediante la función:

```
int Mix_SetPosition(int channel, Sint16 angle, Uint8 distance);
```

Esta función recibe un canal donde aplicar el efecto, el angulo medido en grados y la distancia a la que queremos colocar virtualmente el sonido. Si la función devuelve 0 es que existió algún error al establecer el efecto, y devolverá un valor distinto de 0 si no existió tal error. Si el sonido queremos que se sitúe a en frente nuestra el ángulo a aplicar es 0. Si queremos que aparezca detrás nuestra el ángulo establecido deberá ser 180, mientras que será 270 si lo queremos establecer a nuestra derecha. Para terminar si queremos que el sonido se reproduzca detrás nuestra deberemos de configurar el efecto con un ángulo de 180 grados.

El efecto de *reverse* consiste en intercambiar los canales estéreos uno por otro, es decir, el sonido del canal estéreo izquierdo se reproduciría por el canal derecho mientras que el derecho lo haría por el izquierdo. Para realizar este intercambio entre canales de sonido SDL_mixer proporciona la siguiente función:

```
int Mix_SetReverseStereo(int channel, int flip);
```

Esta función recibe como parámetro el número que identifica al canal que queremos que se aplique el efecto así como un parámetro que indica si el efecto está activo o no. El valor 0 en el parámetro *flip* desactiva y desregistra el efecto en un determinado canal mientras que el valor 1 activa y registra el efecto en el canal especificado. Esta función devuelve 0 en el caso de encontrar algún error y un valor distinto de cero si la activación del efecto ha sido correcta.

13.9. Recopilando

En este capítulo hemos aprendido a utilizar una librería adicional que nos permite dar un poco de potencia al subsistema de audio en SDL. De todas formas sigue siendo un subsistema que no posee toda la potencia que podría tener, pero es suficiente.

Hemos aprendido a manejar sonidos, canales, grupos de sonidos y la música para nuestra aplicación. Hemos realizado pequeñas versiones en consola para

13. SDL_mixer. Gestión de sonidos.

conseguir un código más limpio que nos permita centrarnos en los aspectos fundamentales.

Este es sólo el principio. Ahora puedes relacionar todo lo visto aquí con los demás subsistemas para conseguir desarrollar tu primera aplicación. No te impacientes, en breve realizaremos la nuestra en el tutorial.

Capítulo 14

SDL_net. Recursos de red

14.1. Introducción

SDL_net es una librería auxiliar de SDL preparada para proporcionar soporte sobre red a nuestros videojuegos. Esta librería es la favorita de muchos programadores dentro de las librerías auxiliares de SDL. Aunque no domines el tema de la programación en red puedes crear aplicaciones totalmente funcionales en SDL ya que esta librería auxiliar proporciona una metodología muy simple de utilizar. Es multiplataforma, como no podía ser de otra manera.

En este capítulo vamos a investigar que nos puede proporcionar SDL_net en la programación de nuestro videojuego. Para que puedas hacerte una idea cuando aprendimos a mostrar por pantalla un píxel o una imagen el subsistema de video estaba a nuestra disposición para crear lo que quisieramos poniendo el límite de nuestra creatividad. En esta librería cuando aprendamos a enviar un paquete de una aplicación a otra sabremos todo lo necesario para empezar a darle alas a nuestra creatividad.

14.2. Conceptos Básicos

Existen un par de conceptos básicos sobre redes que debes conocer antes de usar SDL_net. Si sueles jugar en red o eres un asiduo a Internet es posible que te sean familiares la mayoría de los términos y conceptos que vamos a detallar. Antes de nada debes de saber que existen muchísimas tecnologías de red a todos los niveles y que vamos a centrarnos sólo en aquellas que nos son útiles para el uso de esta librería. El mundo de las redes es fascinante y merece la pena que dediques tiempo a su estudio.

El primer concepto que vamos a presentar es el de *dirección IP*. IP significa Internet Protocol y es un estandar que permite identificar a cada ordenador dentro de una red, en este caso una red de redes. Una dirección IP está compuesta de una serie de números separados por puntos agrupadas en cuatro

14. SDL net. Recursos de red

campos tal que así 127.0.0.1. Cada uno de estos campos representa a un octeto de bits, es decir a 1 byte. Dependiendo del valor de estos campos la IP se engloba dentro de unos tipos o categorías. La IP representa a tu ordenador dentro de tu red o dentro de Internet.

Existen valores especiales para IP como el 127.0.0.1 destinados a pruebas en una misma máquina. Esta dirección IP es conocida como *localhost*.

El segundo concepto que tienes que tener claro es el de *socket*. Un socket es una conexión de un ordenador a otro, o mejor dicho, de una IP a otra. Los sockets permiten comunicar nuestros ordenadores y navegar por Internet.

Los siguientes términos que también te sonarán es el de *cliente* y *servidor*. Estos términos idéntifican el rol de los tipos de ordenadores o aplicaciones que hay en la red. El servidor es el encargado de proporcionar la información que se le pide una vez procesada. Un cliente es una aplicación o máquina que realiza las peticiones de información. Resumiendo, el servidor es una fuente de información consultada a través de un cliente.

El último término que vamos a introducir es el de *host*. Un host es la máquina central de un juego y contiene la información que necesitan otros clientes para jugar en red. Un host también puede ser un cliente. En definitiva es un concepto parecido al de servidor pero que no tiene porque tener la exclusividad de éste. Una traducción aproximada al término host en castellano puede ser la de anfitrión.

Todos estos términos y los conceptos sobre las redes podrían ocupar varios capítulos. Como no es el objetivo de este tutorial ser una base bien fundamentada sobre el mundo de las redes vamos a comentar en pocas líneas los tipos de redes que existen, ya que es necesario saber sobre que tipo de red vamos a trabajar a la hora de programar un videojuego. Siempre hablaremos de estructuras lógicas y no de la configuración física de la red ya que para nosotros es relevante su disposición a nivel lógico independientemente de como se haya conseguido dicha configuración.

Un tipo de red muy común hoy en día por la capacidad que tiene para compartir información entre usuario son las redes p2p o punto a punto. Este tipo de red es de las más simples ya que no existen servidores, mejor dicho, todas las máquinas que componen la red son tanto clientes como servidores de información.

Si queremos conectar varias computadoras mediante este sistema tendremos $(n) \times (n - 1)/2$ enlaces entre las redes. Esto quiere decir que para conectar los ordenadores tendremos un crecimiento cuadrático de enlaces, lo

que puede convertirse en insostenible. Las redes p2p puras pueden ser útiles para pequeñas redes pero no para proyectos de cierta envergadura en lo que se refiere al desarrollo de videojuegos. En la figura 14.1 puedes ver un ejemplo de configuración lógica de este tipo de redes.

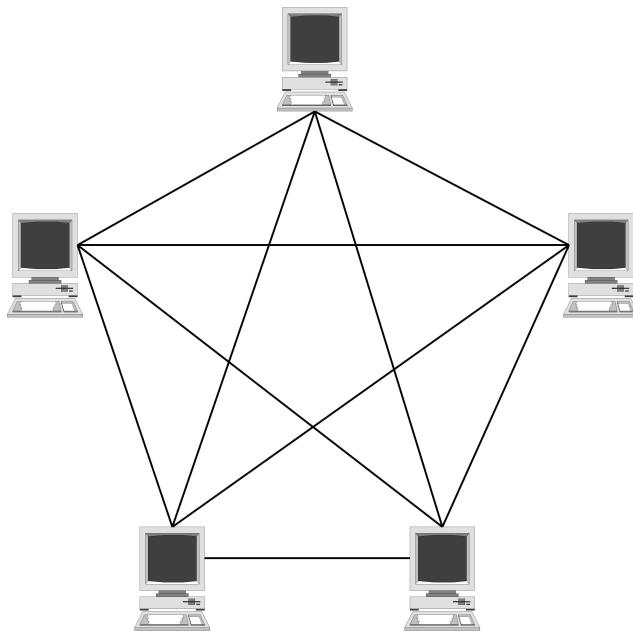


Figura 14.1: Red P2P

Para redes mayores la mejor configuración para utilizar en un videojuego es la de cliente-servidor. En este tipo de redes tendremos un servidor central con el que se comunicarán todas las máquinas y que actualizará la información a todos los clientes que lo necesiten. Puedes observar en la figura la disposición de este tipo de redes.

Cuando se establece este tipo de redes la comunicación entre clientes es indirecta ya que se realiza a través del servidor. Un cliente manda una información que queda almacenada en el servidor y que es enviada, ya sea a petición o no, al cliente correspondiente. Si se trata de un estado el servidor se encargará de hacer conocer esta nueva información a los clientes.

Cuando nuestro juego tiene que saltar a la red tenemos que tener en cuenta que el medio de red es lento y fácilmente saturable por lo que tenemos que optimizar al máximo el apartado de nuestra aplicación que vaya a controlar este aspecto. Hay muchos conceptos importantes en el mundo de las redes, como puede ser la distancia al servidor, que condicionará la respuesta y la información que dispondrán los clientes sobre el estado actual del videojuego.

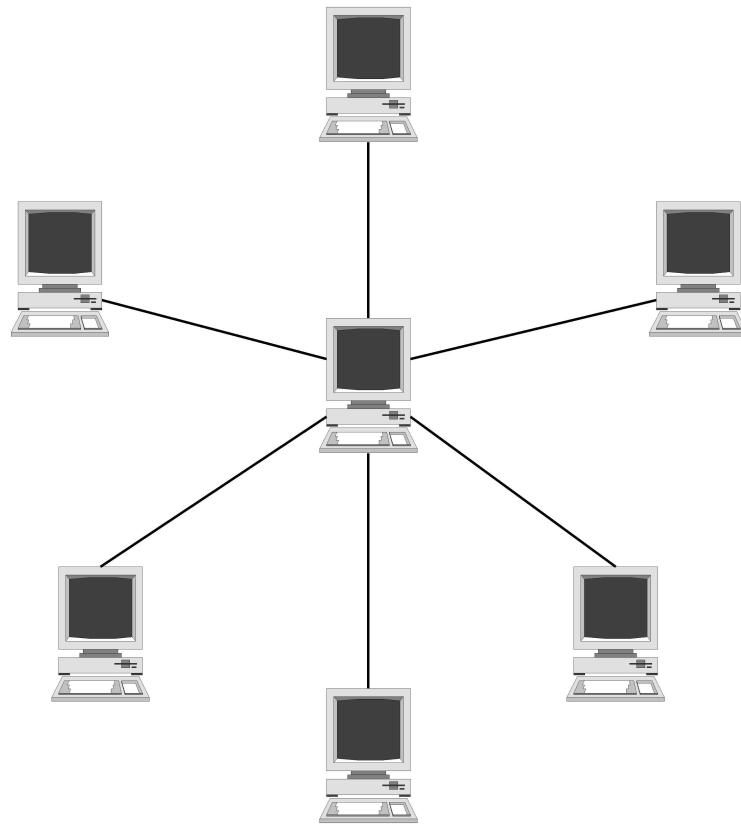


Figura 14.2: Red Cliente-Servidor

SDL_net está compuesta por cuatro partes diferentes asociadas a diferentes estructuras fundamentales. Estas son las direcciones IP, los socket TCP, los socket UDP y los conjuntos de socket.

14.3. Compilando con SDL_net

Como ya puedes suponer en este apartado no hay nada novedoso. Para hacer uso de *SD_net*, como no podía ser de otra forma, tenemos que incluir en los ficheros fuentes donde la utilicemos su fichero de cabecera mediante `#include <SDL/SDL_net.h>`.

A la hora de realizar la compilación tenemos que indicar al compilador que enlace contra la librería mediante `-lSDL_net`. En nuestros ejemplos, al hacer uso de *makefiles* añadiremos esta librería a la variable con la que controlamos contra qué librerías compilamos nuestra aplicación.

14.4. Inicializando

Como en SDL necesitamos inicializar `SDL_net` antes de poder utilizarla. La función que nos permite realizar esta acción es:

```
int SDLNet_Init(void);
```

Esta función no recibe ningún parámetro y devuelve 0 si todo fue correctamente. Si devuelve un valor distinto de cero sabremos que existió algún problema al iniciar la librería. Cuando terminemos de trabajar con la librería deberemos de cerrarla, para ello utilizaremos la siguiente función:

```
void SDLNet_Quit(void);
```

Como ocurría en las demás librerías adicionales esta función es perfecta para ser pasada como parámetro de la función `atexit()` lo que nos va a ahorrar el tener que preocuparnos más del cierre de la librería.

Las tareas de inicialización de esta librería componen las funciones más simples de `SDL_net`, todas las demás tienen una mayor dificultad.

14.5. Las Direcciones IP

Las direcciones IP nos permiten tener un control que nos permite conocer con qué computadoras nos estamos comunicando. Cada ordenador está identificado por una dirección IP de cuatro bytes y un número de puerto por el que se realizará la conexión. Un puerto no es más que un identificador, un número, de 16 bits.

Actualmente se trabaja con IPv4 que es la que estamos presentando. Existen nuevas implementaciones de este protocolo por lo que a medio plazo pueden existir variaciones en esta forma de trabajar. La estructura que soporta a las direcciones IP en `SDL_net` es el tipo de datos `IPaddress`.

La estructura `IPaddress` está definida de la siguiente forma:

```
1 ;typedef struct {
2 ;    Uint32 host;
3 ;    Uint16 port;
4 ;} IPaddress;
;
```

El campo `host` almacena los cuatro bytes que identifican a un ordenador, por esto es de tipo `Uint32`. Este campo puede tomar unos valores especiales como `INADDR_ANY(0)` y `INADDR_NONE(0xFFFFFFFF)`.

14. SDL_net. Recursos de red

El campo *port* contiene el puerto al que dirigiremos la conexión y en teoría puede tomar cualquier valor dentro del rango del tipo definido. En la práctica existen puertos que están reservados para aplicaciones específicas y que no es recomendable usar, como por ejemplo el 80, reservado para los navegadores web. Es importante que utilices un puerto mayor al 1024 ya que la mayoría de los reservados están por debajo de este número. Puedes encontrar en internet numerosas referencias a este aspecto.

14.5.1. Trabando con IP's

Sólo existen dos funciones en *SDL_net* para el trabajo con IP's. La primera de ellas es *SDLNet_ResolveHost()*. Puedes usar esta función para encontrar la dirección IP de un servidor con el que queramos conectar o inicializar la dirección IP para crear un servidor. El prototipo de la función es:

```
int SDLNet_ResolveHost(IPaddress *address, char * host, Uint16 port);
```

Esta función devuelve un entero. En el caso de que el valor devuelto sea 0 significará que la IP no puede ser resuelta. El primer parámetro que recibe la función es un puntero a una estructura *IPaddress*. Este campo es rellenoado con el dato proveniente del host resuelto. El segundo parámetro contiene una cadena con la dirección a la que nos queremos conectar. Esta puede ser 192.168.0.1 si queremos conectar a un host dentro de una red o directamente el nombre de un host disponible en un servidor de nombres. Si pasamos como parámetro en *host* el valor *NULL* la función creará una interfaz de red de escucha, justo lo que necesitamos para iniciar un servidor, y rellenará el campo *address* con la macro *INADDR_ANY*. El último de los parámetros *port* es el puerto donde el host deberá estar escuchando o en el que escuchará si estamos poniendo en marcha nuestro servidor.

Si necesitamos obtener la cadena identificadora asociada a una IP particular podemos usar la función:

```
char * SDLnet_ResolveIP(IPaddress * ip);
```

Esta función recibe un puntero a una estructura *IPaddress* y devuelve un identificador. Si *ip->host* es igual a *INADDR_ANY* la función devolverá el nombre de nuestro ordenador en la red, si no *SDL_net* buscará el nombre de dicha IP y lo devolverá.

14.6. TCP Socket

El socket TCP (*Transfer Control Protocol*) se utiliza para realizar una conexión entre dos ordenadores utilizando direcciones IP. Existen dos tipos de

sockets TCP, servidores y clientes. Usando TCP garantizamos que se recibirán los mensajes sean recibidos ya que este protocolo está orientado a la conexión y realiza comprobaciones por cada paquete que envía. Este tipo de socket está soportado en `SDL_net` por la estructura `_TCPsocket`, pero normalmente trabajaremos con un puntero al tipo `_TCPsocket`.

Curiosamente el tipo `TCPsocket` está definido en `SDL_net.h` como:

```
1 ;  
1 ;typedef struct _TCPsocket *TCPsocket;  
1 ;
```

La definición de la estructura `_TCPsocket` está oculta a los programadores. Esto es debido a que no necesitamos conocer esta estructura para poder trabajar con sockets TCP.

Antes de empezar a mandar datos a otro ordenador primero debes abrir un socket. El computador con el que abras el socket tiene que tener un servidor de sockets, por lo que en algún punto del código necesitarás inicializar el servidor de sockets o bien conectarte a uno existente. `SDL_net` proporciona una función que te permite crear ambos tipos de sockets, iniciando el servidor de sockets. Esta función es:

```
TCPsocket SDLNet_TCP_Open(IPaddress *ip);
```

Esta función recibe como parámetro un puntero a `IPaddress` y devuelve un socket TCP. Si la función devuelve el valor `NULL` es que algo habrá ido mal. Si el valor de `ip.host` es `INADDR_NONE` o `INADDR_ANY` un servidor de sockets será creado, si no, la función procurará conectar con dicho servidor.

Como es habitual, si abrimos un socket deberemos de cerrarlo cuando ya no lo necesitemos. La función que nos proporciona `SDL_net` para realizar esta tarea es:

```
void SDLNet_TCP_Close(TCPsocket sock);
```

Esta función cierra un socket TCP abierto, si es un servidor o no es irrelevante. Recibe como parámetro el socket a cerrar y no devuelve ningún valor.

Vamos a comentar algunas cosas sobre los servidores de sockets (creados con `INADDR_ANY` o `INADDR_NONE`) y los clientes de sockets. Vamos a suponer que queremos crear una aplicación de chat entre dos ordenadores de los que podemos elegir cual es el servidor y cual el cliente.

En el servidor necesitaremos un servidor de sockets tantos como número de clientes de sockets tengamos, es decir, uno por cada uno de los otros ordenadores conectados a nuestros servidor. En el cliente necesitamos un sólo

14. SDL_net. Recursos de red

cliente de socket con el comunicarnos con el servidor.

¿Porqué esto es así? Porque la única cosa que hace un servidor de sockets es escuchar a los clientes como inician una sesión (una sesión no es más que el periodo entre conexión y desconexión a un servidor). No utilizamos el servidor de sockets para enviar o recibir datos. Cuando un servidor de sockets tiene esos datos está listo para empezar a leer, usaremos la función:

```
TCPsocket SDLNet_TCP_Accept(TCPsocket server);
```

Esta función toma un servidor de sockets como parámetro y devuelve cierto socket. El valor devuelto es una conexión con una máquina remota que usó también esta función para conectar el ordenador con el servidor de sockets. Después tenemos que conectarnos a el nuevo ordenador, podemos buscar la dirección IP del ordenador llamando a:

```
IPaddress * SDLNet_TCP_GetPeerAddress(TCPsocket sock);
```

Esta función toma como parámetro un socket y devuelve la dirección IP que estábamos buscando. Vamos con las funciones que permiten establecer la comunicación. La primera de ella es:

```
int SDLNet_TCP_Send(TCPsocket sock, void * data, int len);
```

Esta función recibe un socket no perteneciente a un servidor de sockets, un puntero a los datos que queramos enviar y un entero que debe especificar la longitud de los datos a enviar. Esta función devuelve la cantidad de datos que están siendo enviados. Si el valor no es igual al del parámetro *len* es que existe algún tipo de error. En el otro lado tenemos la función:

```
int SDLNet_TCP_Recv(TCPsocket sock, void * data, int maxlen);
```

Esta función toma un socket que no pertenezca al servidor, un puntero al búffer de datos dónde vamos a almacenar lo recibido y la longitud máxima que pueden tener esos datos. El valor devuelto por la función representa cuantos datos han sido leidos, y deben ser menos que *maxlen*. En el caso de que la función devuelva un 0 o un valor menor es que existe un error.

Aunque sea increible con solo dos estructuras y ocho funciones puedes hacer la aplicación de red que quieras siendo tu creatividad el límite. *SDL_net* hace que esto sea fácil.

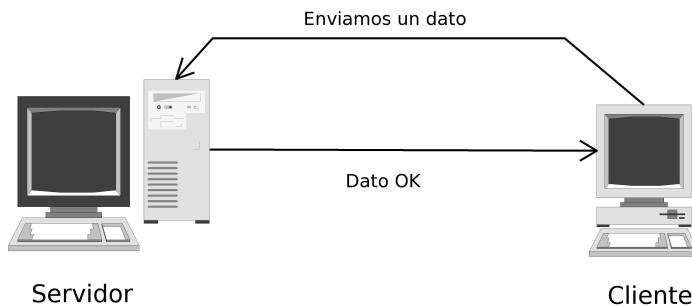


Figura 14.3: Conexión TCP

14.6.1. Implementación de un servidor TCP

Vamos a implementar un servidor TCP haciendo uso de la librería `SDL_net`. Si necesitas una conexión fiable, aunque lenta, TCP es el protocolo que necesitas. Como ya hemos comentado está orientado a la conexión y no dejará que se escape ninguno de los datos que enviamos. Ninguna información enviada a través de TCP será corrupta. Las conexiones TCP pueden llegar a ser realmente lentas, pero es la mejor alternativa si la prioridad es la validez de los datos.

Para crear nuestro servidor vamos a dar los siguientes pasos:

1. Inicializamos la librería `SDL_net`
2. Configuramos un puerto de escucha
3. Abrimos un socket asociado a este puerto
4. Esperamos una conexión a aceptar
5. Recibimos datos y los manejamos
6. Cerramos y terminamos

Vamos a ver el código que nos permite realizar todas estas tareas. Ya hemos visto en la referencia todas las funciones que vamos a utilizar por lo que no necesitan una mayor explicación:

```

;
1 ;// Listado: servidor.cpp
2 ;//
3 ;// Servidor TCP usando SDL_net
4 ;
5 ;#include <iostream>
6 ;#include <SDL/SDL.h>
7 ;
8 ;#include <SDL/SDL_net.h>
```

14. SDL_net. Recursos de red

```
9 ;
10 ;using namespace std;
11 ;
12 ;int main() {
13 ;
14 ;    // Inicializamos SDL_net
15 ;
16 ;    if(SDLNet_Init() < 0) {
17 ;
18 ;        cerr << "SDL_Init: " << SDLNet_GetError();
19 ;        exit(1);
20 ;
21 ;    }
22 ;
23 ;    atexit(SDLNet_Quit);
24 ;
25 ;    // Modo servidor
26 ;
27 ;    IPaddress ip;
28 ;
29 ;    // Configuramos el servidor para escuchar el puerto 2000
30 ;
31 ;    if(SDLNet_ResolveHost(&ip, NULL, 2000) < 0) {
32 ;
33 ;        cerr << "SDLNet_ResolveHost(): " << SDLNet_GetError();
34 ;        exit(1);
35 ;
36 ;    }
37 ;
38 ;    // Abrimos una conexión
39 ;
40 ;    TCPsocket socket;
41 ;
42 ;    socket = SDLNet_TCP_Open(&ip);
43 ;
44 ;    if(!socket) {
45 ;
46 ;        cerr << "SDLNet_TCP_Open(): " << SDLNet_GetError();
47 ;        exit(1);
48 ;    }
49 ;
50 ;
51 ;    cout << "Servidor activo" << endl;
52 ;
53 ;    // Bucle de control de la conexión
54 ;
55 ;    bool salir = false;
56 ;    TCPsocket socket_cliente;
57 ;
58 ;    while(salir == false) {
59 ;
60 ;        // ¿Tenemos una conexión pendiente?
61 ;        // La aceptamos
```

```

62 ;
63 ;     if((socket_cliente = SDLNet_TCP_Accept(socket))) {
64 ;
65 ;         // Ahora realizamos la comunicación con el cliente
66 ;
67 ;         IPAddr *ip_remota;
68 ;
69 ;         // Mostramos la información
70 ;
71 ;         if((ip_remota = SDLNet_TCP_GetPeerAddress(socket_cliente)))
72 ;
73 ;             cout << "Cliente conectado "
74 ;                 << SDLNet_Read32(&ip_remota->host)
75 ;                 << " : "
76 ;                 << SDLNet_Read16(&ip_remota->port) << endl;
77 ;
78 ;     else
79 ;
80 ;         cerr << "SDLNet_TCP_GetPeerAddress(): " << SDLNet_GetError() << endl;
81 ;
82 ;
83 ;         // Mostramos lo que envía el cliente
84 ;
85 ;         bool terminar = false;
86 ;         char buffer[512];
87 ;
88 ;         while(terminar == false) {
89 ;
90 ;             // Leemos de la conexión
91 ;
92 ;             if (SDLNet_TCP_Recv(socket_cliente, buffer, 512) > 0) {
93 ;
94 ;                 cout << "Cliente dice: " << buffer << endl;;
95 ;
96 ;                 // Si da orden de salir, cerramos
97 ;
98 ;                 if(strcmp(buffer, "exit") == 0) {
99 ;
100 ;                     terminar = true;
101 ;                     cout << "Desconectando" << endl;
102 ;
103 ;                 }
104 ;
105 ;                 if(strcmp(buffer, "quit") == 0) {
106 ;
107 ;                     terminar = true;
108 ;                     salir = true;
109 ;                     cout << "Server Down" << endl;
110 ;                 }
111 ;             }
112 ;         }
113 ;
114 ;         // Cierro el socket

```

14. SDL_net. Recursos de red

```
115 ;
116 ;           SDLNet_TCP_Close(socket_cliente);
117 ;
118 ;
119 ;       }
120 ;
121 ;       SDLNet_TCP_Close(socket);
122 ;
123 ;       return 0;
124 ;}
```

14.6.2. Implementación de un cliente TCP

Una vez implementado el programa servidor necesitamos un cliente que interactúe con él. Para desarrollar este cliente vamos a seguir los siguientes pasos:

1. Inicializaremos la librería `SDL_net`
2. Conectaremos con la dirección del servidor
3. Abrimos un socket
4. Leemos datos del usuario, en este caso cadenas de texto
5. Enviamos los datos al servidor
6. Cerramos y terminamos

Vamos a ver el código del cliente. No tiene mucha más complicación que la inicialización de los valores a enviar al servidor.

```
;_____
1 ;// Listado: cliente.cpp
2 ;//
3 ;// Cliente TCP usando SDL_net
4 ;
5 ;
6 ;#include <iostream>
7 ;#include <SDL/SDL_net.h>
8 ;
9 ;using namespace std;
10 ;
11 ;
12 ;int main(int argc, char **argv)
13 ;{
14 ;
15 ;    // Comprobamos los parámetros
16 ;
17 ;    if (argc < 3) {
18 ;
```

```

19 ;         cerr << "Uso: "<< argv[0] << " servidor puerto" << endl;
20 ;         exit(1);
21 ;
22 ;     }
23 ;
24 ;     // Inicializamos SDL_net
25 ;
26 ;     if (SDLNet_Init() < 0) {
27 ;
28 ;         cerr << "SDLNet_Init(): " << SDLNet_GetError() << endl ;
29 ;         exit(1);
30 ;     }
31 ;
32 ;     // Resolvemos el host (servidor)
33 ;
34 ;     IPaddress ip;
35 ;
36 ;     if (SDLNetResolveHost(&ip, argv[1], atoi(argv[2])) < 0) {
37 ;
38 ;         cerr << "SDLNetResolveHost(): " << SDLNet_GetError() << endl;
39 ;         exit(1);
40 ;
41 ;     }
42 ;
43 ;     // Abrimos una conexión con la IP provista
44 ;
45 ;     TCPsocket socket;
46 ;
47 ;     if (!(socket = SDLNet_TCP_Open(&ip))) {
48 ;
49 ;         cerr << "SDLNet_TCP_Open: " << SDLNet_GetError() << endl;
50 ;         exit(1);
51 ;     }
52 ;
53 ;     // Enviamos los mensaje
54 ;
55 ;     bool terminar = false;
56 ;     char buffer[512];
57 ;     int longitud;
58 ;
59 ;     while(terminar == false) {
60 ;
61 ;         cout << "Escribe algo :> " ;
62 ;         cin >> buffer;
63 ;
64 ;         // Lo escrito + terminador
65 ;
66 ;         longitud = strlen(buffer) + 1;
67 ;
68 ;         // Lo enviamos
69 ;
70 ;         if (SDLNet_TCP_Send(socket, (void *)buffer, longitud) < longitud) {
71 ;

```

14. SDL_net. Recursos de red

```
72 ;         cerr << "SDLNet_TCP_Send: " << SDLNet_GetError() << endl;;
73 ;         exit(1);
74 ;
75 ;
76 ;         // Si era salir o cerrar, terminamos con la conexión
77 ;
78 ;         if(strcmp(buffer, "exit") == 0)
79 ;             terminar = true;
80 ;
81 ;         if(strcmp(buffer, "quit") == 0)
82 ;             terminar = true;
83 ;
84 ;
85 ;         SDLNet_TCP_Close(socket);
86 ;         SDLNet_Quit();
87 ;
88 ;     return 0;
89 ;}
```

Como ocurría con el servidor todas las funciones utilizadas han sido previamente estudiadas por lo que, en principio, no necesitan más explicación.

14.7. UDP Socket

Los sockets UDP (*User Datagrama Protocol*) es un tipo de socket parecido al TCP. La principal diferencia es que este tipo de sockets no garantiza la entrega del paquete una vez que se ha enviado. Suelen usarse cuando se necesita una mayor fluidez de datos y no es importante que se pierda alguno de ellos. Este tipo de socket son soportados en sdl por la estructura `_UDPSocket`, pero como pasaba con los sockets TCP, trabajaremos con un puntero al tipo `UDPSocket`.

SDL_net tiene funciones que nos permiten usar UDP para enviar mensajes a través de la red con el problema de que estos paquetes que enviamos no serán confiables.

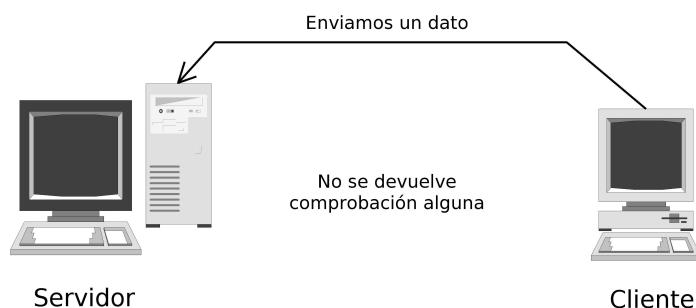


Figura 14.4: Conexión UDP

14.7.1. Implementando un servidor UDP

No encontramos en un caso parecido al de la implementación del servidor pero con las características especiales de que vamos a utilizar UDP. La forma de proceder va a ser la siguiente:

1. Inicializaremos la librería `SDL_net`
2. Abriremos un socket en un puerto específico
3. Reservaremos memoria para los paquetes
4. Esperaremos un paquete del cliente
5. Gestionaremos el paquete
6. Liberaremos memoria y terminaremos

Vamos a ver el código:

```

1 ;// Listado: servidor.cpp
2 ;//
3 ;// Servidor UDP usando SDL_net
4 ;
5 ;#include <iostream>
6 ;#include <SDL/SDL.h>
7 ;
8 ;#include <SDL/SDL_net.h>
9 ;
10 ;using namespace std;
11 ;
12 ;int main() {
13 ;
14 ;    // Inicializamos SDL_net
15 ;
16 ;    if(SDLNet_Init() < 0) {
17 ;
18 ;        cerr << "SDL_Init(): " << SDLNet_GetError();
19 ;        exit(1);
20 ;
21 ;    }
22 ;
23 ;    atexit(SDLNet_Quit);
24 ;
25 ;    // Abrimos una conexión en el puerto 2000
26 ;
27 ;    UDPsocket socket;
28 ;
29 ;    socket = SDLNet_UDP_Open(2000);
30 ;
31 ;    if(!socket) {
32 ;

```

14. SDL_net. Recursos de red

```
33 ;         cerr << "SDLNet_UDP_Open(): " << SDLNet_GetError();  
34 ;         exit(1);  
35 ;     }  
36 ;  
37 ;  
38 ;     cout << "Servidor activo" << endl;  
39 ;  
40 ;     // Reservamos espacio para los paquetes  
41 ;  
42 ;     UDPpacket *p;  
43 ;  
44 ;     if (!(p = SDLNet_AllocPacket(512))) {  
45 ;  
46 ;         cerr << "SDLNet_AllocPacket:" << SDLNet_GetError() << endl;  
47 ;         exit(1);  
48 ;     }  
49 ;  
50 ;     // Bucle de control de la conexión  
51 ;  
52 ;     bool salir = false;  
53 ;  
54 ;     while(salir == false) {  
55 ;  
56 ;         if (SDLNet_UDP_Recv(socket, p)) {  
57 ;  
58 ;             // Mostramos información de los datos recibidos  
59 ;  
60 ;             cout << "Paquete UDP recibido" << endl;  
61 ;             cout << "\tCanal: " << p->channel << endl;  
62 ;             cout << "\tDatos: " << (char *)p->data << endl;;  
63 ;             cout << "\tLongitud: " << p->len << endl;  
64 ;             cout << "\tMaxlen: " << p->maxlen << endl;  
65 ;             cout << "\tEstado: " << p->status << endl;  
66 ;             cout << "\tDirección: " << p->address.host << " " << p->address.port << endl;  
67 ;  
68 ;             // Si el paquete contiene la palabra quit, salimos  
69 ;  
70 ;             if (!strcmp((char *)p->data, "quit"))  
71 ;  
72 ;                 salir = true;  
73 ;             }  
74 ;  
75 ;         }  
76 ;  
77 ;  
78 ;         // Liberamos memoria y salimos  
79 ;  
80 ;         SDLNet_FreePacket(p);  
81 ;         SDLNet_Quit();  
82 ;  
83 ;         return 0;  
84 ;     }
```

Como puedes ver no hay nada nuevo en el código.

14.7.2. Implementando un cliente UDP

Una vez el implementado el programa servidor necesitamos un cliente que nos permita comprobar el funcionamiento del mismo. Por esto y para que tengas una guía de como trabajar con el tipo de conexión UDP tanto en modo cliente como en el modo servidor. La forma de proceder es la siguiente:

1. Inicializamos la librería SDL_net
2. Abrimos un puerto al azar que pueda ser usado
3. Resolvemos la dirección del servidor
4. Reservamos memoria para los paquetes
5. Rellenamos y enviamos los paquetes
6. Liberamos memoria y terminamos

Vamos a ver el código:

```
1 ;// Listado: cliente.cpp
2 ;//
3 ;// Cliente UDP usando SDL_net
4 ;
5 ;
6 ;#include <iostream>
7 ;#include <SDL/SDL_net.h>
8 ;
9 ;using namespace std;
10 ;
11 ;
12 ;int main(int argc, char **argv)
13 ;{
14 ;
15 ;    // Comprobamos los parámetros
16 ;
17 ;    if (argc < 3) {
18 ;
19 ;        cerr << "Uso: "<< argv[0] << " servidor puerto" << endl;
20 ;        exit(1);
21 ;
22 ;    }
23 ;
24 ;    // Inicializamos SDL_net
25 ;
26 ;    if (SDLNet_Init() < 0) {
27 ;
```

14. SDL_net. Recursos de red

```
28 ;         cerr << "SDLNet_Init(): " << SDLNet_GetError() << endl ;
29 ;         exit(1);
30 ;
31 ;
32 ;     // Abrimos un socket en un puerto aleatorio (0)
33 ;
34 ;     UDPsocket socket;
35 ;
36 ;     if (!(socket = SDLNet_UDP_Open(0))) {
37 ;
38 ;         cerr << "SDLNet_UDP_Open(): " << SDLNet_GetError() << endl;
39 ;         exit(1);
40 ;
41 ;
42 ;     // Resolvemos el host (servidor)
43 ;
44 ;     IPaddress ip_server;
45 ;
46 ;     if (SDLNetResolveHost(&ip_server, argv[1], atoi(argv[2])) < 0) {
47 ;
48 ;         cerr << "SDLNetResolveHost(): " << SDLNet_GetError() << endl;
49 ;         exit(1);
50 ;
51 ;
52 ;
53 ;     // Reservamos memoria para los paquetes
54 ;
55 ;     UDPpacket *p;
56 ;
57 ;     if (!(p = SDLNet_AllocPacket(512))) {
58 ;
59 ;         cerr << "SDLNet_AllocPacket(): " << SDLNet_GetError() << endl;
60 ;         exit(1);
61 ;
62 ;
63 ;
64 ;     // Enviamos los mensajes
65 ;
66 ;     bool terminar = false;
67 ;
68 ;     while(terminar == false) {
69 ;
70 ;         cout << "Rellena el búffer :> " ;
71 ;         cin >> ((char *) p->data);
72 ;
73 ;         p->address.host = ip_server.host; // Establecemos el destino
74 ;         p->address.port = ip_server.port; // y el puerto
75 ;
76 ;         // Longitud + 1
77 ;
78 ;         p->len = strlen((char *)p->data) + 1;
79 ;
80 ;         SDLNet_UDP_Send(socket, -1, p); // Establecemos el canal
```

14.8. Sockets Genéricos. Conjuntos de Sockets

```
81 ;  
82 ;      // Si el paquete contiene "quit" salimos  
83 ;  
84 ;      if (!strcmp((char *)p->data, "quit"))  
85 ;          terminar = true;  
86 ;      }  
87 ;  
88 ;  
89 ;      SDLNet_FreePacket(p);  
90 ;      SDLNet_Quit();  
91 ;  
92 ;      return 0;  
93 ;}
```

14.8. Sockets Genéricos. Conjuntos de Sockets

Los conjuntos o colección de sockets son utilizados tanto con TCP como con UDP para buscar datos que lleguen a un determinado socket. *SDLNet_SocketSet*, igual que *TCPsocket*, está definido como un puntero a una estructura a la que se nos oculta su definición. Existe otro tipo de con *SDLNet_SocketSet* llamado *SDLNet_GenericSocket* que está definido de la siguiente manera:

```
1 ;  
1 ;typedef struct {  
2 ;    int ready;  
3 ;} *SDLNet_GenericSocket;
```

Este tipo permite utilizar varios tipos de sockets en un conjunto de sockets. Para usar un conjunto de sockets primero tenemos que asignarlo o mejor dicho, crearlo. Para crear un conjunto de sockets utilizamos la siguiente función:

```
SDLNet_SocketSet SDLNet_AllocSocketSet(int maxsockets);
```

Esta función toma como parámetros el número de sockets que queremos contener en él y devuelve el conjunto de sockets. Como es habitual una vez que terminemos de trabajar con este conjunto deberemos de liberar los recursos que utiliza. Para esto utilizamos la función:

```
void SDLNet_FreeSocketSet(SDLNet_SocketSet set);
```

Esta función no devuelve ningún valor y toma como parámetro un conjunto de sockets que queremos liberar. Está claro que antes de liberarlo, si hemos creado dicho conjunto, es que vamos a utilizarlo. Tenemos que añadir los sockets que queramos a este conjunto de sockets. Para eso usaremos *SDLNet_AddSocket*:

14. SDL_net. Recursos de red

```
int SDLNet_AddSocket(SDLNet_SocketSet set, SDLNet_GenericSocket  
                      sock);
```

Esta función recibe como primer parámetro el conjunto de sockets a llenar y un puntero genérico que nos permite introducir cualquier tipo de socket en dicho conjunto realizando un casting en el parámetro. Existen un par de macros en *SDL_net* para ayudarnos con esta tarea. Si vamos a añadir un socket TCP podemos usar *SDLNet_TCP_AddSocket* en lugar de la función anterior realizando la misma acción. Para sockets UDP podemos trabajar de la misma forma.

Para eliminar un elemento dentro del SocketSet utilizaremos la función:

```
int SDLNet_DelSocket(SDLNet_SocketSet set, SDLNet_GenericSocket  
                      sock);
```

Esta función recibe como parámetros, primero el conjunto de sockets que vamos a eliminar y luego el socket que queremos eliminar dentro de dicho conjunto. Como ocurría con *SDLNet_AddSocket* si funcionamos sólo con sockets TCP podemos utilizar la función *SDLNet_TCP_DelSocket* evitando el casting. De forma análoga existe la misma función que realiza la misma acción para UDP.

Ahora vamos a presentar una de las funciones para realizar una de las tareas más importantes a realizar con los grupos de sockets. Debemos de tener una función que nos permite comprobar los datos que existen en dicho grupo de sockets. Podemos conseguirlo con la función:

```
int SDLNet_CheckSockets(SDLNet_SocketSet set, Uint32 timeout);
```

Esta función recibe como parámetros el grupo de sockets a consultar y un tiempo en el parámetro *timeout*. El valor para *timeout* puede ser 0, lo que significaría que el grupo de sockets va a realizar un polling rápido sobre sus datos. Si pasamos otro valor tiene como significado el tiempo durante el cual la función va a hacer poll de los datos en los sockets.

El valor devuelto es el número de sockets en el conjunto que tienen datos listos. Si el valor que nos devuelve la función es -1 es que existió algún error al realizar las operaciones pertinentes.

Después de llamar a esta función podemos ver los datos de un socket en concreto que tuviese datos preparados llamando a *SDLNet_SocketReady* que no es una función si no una macro definida como:

```
#define SDLNet_SocketReady(sock)  
(sock != NULL) && ((SDLNet_GenericSocket)sock)->ready)
```

14.9. Recopilando

En este capítulo hemos aprendido a utilizar la librería adicional `SDL_net` creando un servidor y un cliente UDP y TCP. Con las agrupaciones de sockets podremos utilizar gran cantidad de sockets, de diferentes tipos pudiendo organizarlos de la mejor manera.

Este es el punto de partida para el uso de las redes en la programación de los videojuegos. Para lograr sacar partido de esta librería es necesario un buen diseño de mensajística para el videojuego. Puedes mandar cualquier cosa, ahora está en tu mano.

14. SDL _ net. Recursos de red

Capítulo 15

Los Sprites y los Personajes

15.1. Introducción

A estas alturas de curso tenemos suficientes herramientas para enfrentarnos a la creación de un videojuego. Con un buen diseño y un poco de habilidad podremos crear nuestros propios movimientos, animaciones, control de colisiones...

En este y los capítulos sucesivos vamos a intentar que el esfuerzo que tengas que hacer para desarrollar tu primer videojuego no sea tan costoso. Vamos a ayudarte a crear animaciones, moverlas por la pantalla, interactuar con otros elementos del videojuego. En resumen, proporcionarte herramientas para que puedas llevar a cabo tus tareas pero teniendo presente que esto sólo es el principio.

Vamos a empezar presentando algunos conceptos y realizando nuestras primeras animaciones.

15.2. Objetivos

Los objetivos de este capítulo son:

1. Conocer el concepto de sprite y los asociados a éste.
2. Aprender a controlar los distintos tipos de animaciones.
3. Obtener la capacidad de crear una galería.
4. Ser capaces de controlar las colisiones en nuestra aplicación.

15.3. Sprites

Si es tu primera incursión en el desarrollo de videojuegos seguramente no sabrás qué es un *sprite*. El sprite se considera un de las unidades fundamen-

15. Los Sprites y los Personajes

tales en el desarrollo de videojuegos de dos dimensiones. Existen numerosas definiciones para este término. Nosotros nos vamos a conformar con tener un concepto sencillo y claro de lo que es un sprite.

Un sprite es un objeto que podemos visualizar en pantalla que tiene asociados ciertos atributos como puede ser la posición, velocidad, estado... Por ejemplo, un sprite puede ser un personaje de un videojuego, el ítem que recoge del suelo, un decorado... en definitiva cualquier representación gráfica que hagamos en el videojuego.

Los sprites en su origen fueron un tipo concreto de mapa de bits que se podían dibujar directamente por un hardware específico que liberaba de esta tarea a la CPU reduciendo la carga del sistema. Los sprites, como las imágenes que cargamos en SDL, son rectangulares. El uso de transparencias o colores clave nos permiten que estos sprites o imágenes tengan una forma distinta a la rectangular. Esta tarea era implementada por el hardware que hemos comentando realizando operaciones AND y OR entre la imagen original y la de volcado dejando el resultado en una posición de memoria que el hardware gráfico se encargaría de mostrar por pantalla.



Figura 15.1: Ejemplo de Sprite

Con el paso de los años el uso del término *sprite* se ha extendido a cualquier pequeño mapa de bits que se dibuje en pantalla sin tener en cuenta quien es el encargado de dibujarlo previamente. La creación de sprites ha evolucionado tanto como el mundo de la creación de videojuegos pero nosotros vamos a crear nuestros propios sprites a mano, punto por punto, digamos de una manera más “tradicional”.

Actualmente los juegos en tres dimensiones copan el mercado. Esto provoca que el uso de sprite sea menor que antiguamente aunque sigue siendo fundamental en el desarrollo de gran número de juegos. La aparición de juegos para pequeños dispositivos y en formato flash provoca que “sprite” siga siendo un concepto vivo.

Como puedes ver el concepto de sprite es muy amplio. A todos esta información tenemos que añadir que también se le llama sprite a una animación, normalmente en GIF o PNG, que nos permita simular el movimiento de un personaje sea o no de videojuegos.

Los fanáticos de los sprites han protagonizado evolución que ha generado un arte propio, el conocido *pixel Art*. Existen numerosas comunidades dedicadas a este tipo de arte en sus distintas vertientes con un gran número de adeptos.

Y ahora lo que verdaderamente nos importa. En este curso cuando hagamos uso de la palabra sprite estaremos haciendo referencia las imágenes que representan a un personaje o a un ítem de nuestro videojuego.

Según su movimiento existen dos tipos de sprites:

- Estáticos: Son aquellos que están fijos en la pantalla como puede ser el fondo del juego, un decorado, un objeto a coger o bien un ítem informativo.
- Dinámicos: Son aquellos que se pueden mover por la pantalla o que estando estáticos tienen alguna animación asociada. Ejemplo de este tipo de sprites son los personajes de videojuegos, un objeto que realice un movimiento determinado o un objeto estático animado.

Los sprites, sobre todo los animados, pueden estar compuestos de una o varias imágenes. Cada una de estas imágenes es conocida como cuadros de animación. Comunmente en el mundo del desarrollo de videojuego cada una de estas imágenes es conocida como **frame**. Las animaciones de los personajes no son más que una secuencia de frames, que vistas una tras otra a una velocidad adecuada producen el efecto de movimiento o bien el de alguna acción particular.

El personaje principal de nuestro videojuego es un sprite de los llamados animados. Cuando se programan videojuegos mediante otras tecnologías, como puede ser OpenGL para tres dimensiones, se utilizan técnicas distintas ya que para simular el movimiento la metodología es totalmente diferente. En este caso, más que una secuencia de imágenes, habría que realizar cálculos matemáticos que respondiesen a determinados eventos de los dispositivos de entrada o del mismo videojuego.

En un sprite se distinguen dos tipos de movimiento que debemos controlar. El movimiento externo o del sprite por la pantalla, y el movimiento interno o de animación del sprite. Para tomar la posición del sprite utilizamos la misma referencia que cuando trabajamos con imágenes ya que en definitiva se trata del mismo concepto. Recuerda que la referencia la referencia del juego se sitúa en la esquina superior izquierda de la pantalla, partiendo ahí el origen tanto del eje *x* como del eje *y* para nuestra aplicación.

15. Los Sprites y los Personajes

El curso va introduciéndose cada vez más en el manejo del lenguaje C++. Para realizar un correcto manejo lo de los sprites es necesario introducirnos en la programación orientada a objetos ya que proporciona unas características deseables para este tipo de objetos y nos facilitará mucho la generación de aplicaciones dedicadas al videojuego. Si no lo has hecho aún es el momento de ponerte las pilas con este lenguaje para que el seguimiento del curso te sea más sencillo. De todas formas iremos explicando todas las novedades que aparezcan en el código.

15.4. Animando un Sprite

Para seguir este capítulo debes de tener muy presente todos los conceptos que estudiamos en el apartado dedicado al subsistema de video ya que es el punto de partida para poder realizar una animación. Vamos a manejar imágenes, superficies... No es un desarrollo complicado pero necesita que tengas asentadas las ideas que expusimos en aquel capítulo.

15.4.1. Qué es una animación

Podemos definir animación como una simulación de movimiento mediante una secuencia de imágenes. Al mostrar estas imágenes (llamadas cuadros o *frames*) sucesivamente en una misma posición producen una ilusión de movimiento que no existió en realidad. Entendemos también como animación cualquier movimiento de translación que se haga sobre una superficie aunque el sprite que la realice no simule ningún tipo de acción.

En este principio se basan el cine o la televisión. Muestran imágenes una detrás de otra con pequeñas variaciones mediante las cuales (por el **fenómeno phi**) nuestro cerebro construye un movimiento llenando los huecos entre una imagen y la siguiente. Junto con la persistencia de la retina son la base de la teoría de la representación de movimiento en cine, ordenador o televisión.

En definitiva, una animación es una secuencia de imágenes que son capaces de simular un movimiento. La vista humana tiene la limitación de poder percibir 24 imágenes por segundo lo que nos establece un punto de partida sobre el número de imágenes que necesitaremos para percibir un movimiento fluido.

15.4.2. Nuestra primera animación

Ya sabemos qué es una animación. Ahora vamos a crear nuestra primera animación. Para poder crear una animación necesitamos tantas imágenes como

variaciones queramos que tenga dicha animación. Una vez tengamos estas imágenes iremos sustituyendo unas por otras en pantalla creando un efecto de movimiento. Fácil ¿no? Vamos a implementar una pequeña aplicación que nos permita ver como conseguimos un efecto de animación.

Para preparar estas imágenes hemos usado el editor *The Gimp*. Este editor es, además de gratuito, libre. Es una de las aplicaciones de software libre que está sufriendo un mayor evolución en el mundo del tratamiento de imágenes y es más que suficiente para poder crear nuestros personajes.

15.4.2.1. Implementando una animación

Vamos a crear una animación a partir de 30 imágenes mostrándolas por pantalla una detrás de otra.

```
1 ;// Ejemplo 1
2 ;//
3 ;// Listado: main.cpp
4 ;// Programa de pruebas. Animación básica
5 ;// Primera versión. Poco eficiente
6 ;
7 ;
8 ;#include <iostream>
9 ;#include <iomanip>
10 ;
11 ;#include <SDL/SDL.h>
12 ;#include <SDL/SDL_image.h>
13 ;
14 ;#define TEMPO 80
15 ;
16 ;using namespace std;
17 ;
18 ;int main()
19 ;{
20 ;
21 ;    // Iniciamos el subsistema de video
22 ;
23 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
24 ;
25 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
26 ;        exit(1);
27 ;
28 ;    }
29 ;
30 ;
31 ;    atexit(SDL_Quit);
32 ;
33 ;    // Comprobamos que sea compatible el modo de video
34 ;
35 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
36 ;
```

15. Los Sprites y los Personajes

```
37 ;         cerr << "Modo no soportado: " << SDL_GetError() << endl;
38 ;         exit(1);
39 ;
40 ;     }
41 ;
42 ;
43 ; // Establecemos el modo de video
44 ;
45 ;     SDL_Surface *pantalla;
46 ;
47 ;     pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
48 ;
49 ;     if(pantalla == NULL) {
50 ;
51 ;         cerr << "No se pudo establecer el modo de video: "
52 ;             << SDL_GetError() << endl;
53 ;
54 ;         exit(1);
55 ;     }
56 ;
57 ;
58 ; // Cargamos las 30 imágenes de la animación
59 ; // 30 accesos a disco (muy mejorable)
60 ;
61 ;     SDL_Surface *imagenes[30];
62 ;
63 ;     imagenes[29] = NULL;
64 ;
65 ;     imagenes[0] = IMG_Load("Imagenes/1.png");
66 ;     imagenes[1] = IMG_Load("Imagenes/2.png");
67 ;     imagenes[2] = IMG_Load("Imagenes/3.png");
68 ;     imagenes[3] = IMG_Load("Imagenes/4.png");
69 ;     imagenes[4] = IMG_Load("Imagenes/5.png");
70 ;     imagenes[5] = IMG_Load("Imagenes/6.png");
71 ;     imagenes[6] = IMG_Load("Imagenes/7.png");
72 ;     imagenes[7] = IMG_Load("Imagenes/8.png");
73 ;     imagenes[8] = IMG_Load("Imagenes/9.png");
74 ;     imagenes[9] = IMG_Load("Imagenes/10.png");
75 ;     imagenes[10] = IMG_Load("Imagenes/11.png");
76 ;     imagenes[11] = IMG_Load("Imagenes/12.png");
77 ;     imagenes[12] = IMG_Load("Imagenes/13.png");
78 ;     imagenes[13] = IMG_Load("Imagenes/14.png");
79 ;     imagenes[14] = IMG_Load("Imagenes/15.png");
80 ;     imagenes[15] = IMG_Load("Imagenes/16.png");
81 ;     imagenes[16] = IMG_Load("Imagenes/17.png");
82 ;     imagenes[17] = IMG_Load("Imagenes/18.png");
83 ;     imagenes[18] = IMG_Load("Imagenes/19.png");
84 ;     imagenes[19] = IMG_Load("Imagenes/20.png");
85 ;     imagenes[20] = IMG_Load("Imagenes/21.png");
86 ;     imagenes[21] = IMG_Load("Imagenes/22.png");
87 ;     imagenes[22] = IMG_Load("Imagenes/23.png");
88 ;     imagenes[23] = IMG_Load("Imagenes/24.png");
89 ;     imagenes[24] = IMG_Load("Imagenes/25.png");
```

15.4. Animando un Sprite

```
90 ;     imagenes[25] = IMG_Load("Imagenes/26.png");
91 ;     imagenes[26] = IMG_Load("Imagenes/27.png");
92 ;     imagenes[27] = IMG_Load("Imagenes/28.png");
93 ;     imagenes[28] = IMG_Load("Imagenes/29.png");
94 ;
95 ; // Variables auxiliares
96 ;
97 ;     SDL_Event evento;
98 ;     int i = 0;
99 ;     Uint32 negro = SDL_MapRGB(pantalla->format, 0, 0, 0);
100 ;
101 ;    Uint32 t0 = SDL_GetTicks();
102 ;    Uint32 t1;
103 ;
104 ; // Bucle "infinito"
105 ;
106 ;    for( ; ; ) {
107 ;
108 ;        t1 = SDL_GetTicks();
109 ;
110 ;        // Mostramos una imagen cada medio segundo
111 ;
112 ;        if((t1 - t0) > TEMPO) {
113 ;
114 ;            if(i > 28) {
115 ;
116 ;                SDL_FillRect(pantalla, NULL, negro);
117 ;                SDL_Flip(pantalla);
118 ;
119 ;                i = 0;
120 ;
121 ;            } else {
122 ;
123 ;                SDL_BlitSurface(imagenes[i], NULL, pantalla, NULL);
124 ;
125 ;                SDL_Flip(pantalla);
126 ;
127 ;                t0 = SDL_GetTicks();
128 ;
129 ;                i++;
130 ;            }
131 ;        }
132 ;
133 ;        while(SDL_PollEvent(&evento)) {
134 ;
135 ;            if(evento.type == SDL_KEYDOWN) {
136 ;
137 ;                if(evento.key.keysym.sym == SDLK_ESCAPE)
138 ;
139 ;                    return 0;
140 ;
141 ;                if(evento.key.keysym.sym == SDLK_f)
142 ;                    SDL_WM_ToggleFullScreen(pantalla);
```

15. Los Sprites y los Personajes

```
143 ;  
144 ; }  
145 ;  
146 ; if(evento.type == SDL_QUIT)  
147 ;  
148 ;     return 0;  
149 ;  
150 ;  
151 ; }  
152 ; }  
153 ;};
```

Vamos a lo novedoso del código. Como puedes ver hemos creado un vector de bajo nivel donde almacenamos cada uno de las imágenes que van a formar parte de nuestra animación. Cada vez que almacenamos una de estas imágenes necesitamos realizar un acceso a disco además de tener que especificar para cada uno de los cuadros el fichero dónde está almacenado dicho fichero.

Nos situamos en 30 lecturas de disco lo que es una carga de trabajo considerable para el sistema. Este aspecto lo mejoraremos en el siguiente apartado haciendo uso de las rejillas o panel de imágenes.

Otro aspecto destacable (negativamente) es que todo el control de la animación hemos tenido que hacer externamente. Es decir, no tenemos ningún tipo de estructura que indicando lo que queremos mostrar nos establezca una secuencialidad, un control del tiempo... y otros aspectos propios de una animación. Hemos hecho uso de las capacidades que provee SDL para el manejo del tiempo para controlar que la respuesta de la animación sea la misma en cualquier sistema donde la ejecutemos siempre que dicho sistema soporte la carga de esta aplicación. Como ya estudiamos es fundamental controlar la temporalidad de las animaciones que creamos con SDL.

En definitiva con este ejemplo no hemos más que emular el comportamiento de un gif animado con SDL pero podemos llegar mucho más lejos.

15.4.3. Animación Interna

Una animación interna está compuesta de un número determinado de fotogramas o imágenes sueltas que definen el comportamiento del personaje como una unidad. La animación interna se refiere a los efectos que se producen en la imagen del personaje cuando se le asocia una acción. Por ejemplo, si un personaje está caminando por la pantalla el hecho de trasladarse por dicha pantalla lo conocemos como animación externa mientras el proceso de mover las piernas, una detrás de otra, lo conocemos como animación interna.

Para almacenar estas imágenes utilizaremos una rejilla lo suficientemente grande. En ella tendremos almacenadas todas las frames de las acciones o

estados que puede tener nuestro personaje. Cada una de estas animaciones tendrá asociada un número determinado de fotogramas dentro de esta rejilla de imágenes. Si lo consideramos oportuno (no te lo aconsejo) podemos tener en un solo fichero imágenes de más de un personaje o ítem. Tenemos que darle soporte a esta manera de trabajo.

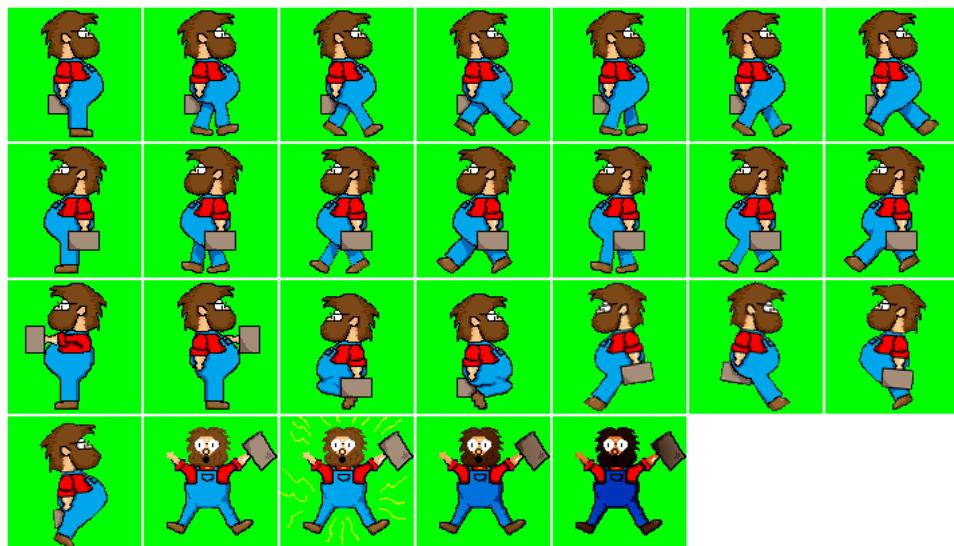


Figura 15.2: Ejemplo de rejilla. Nuestro personaje principal Jacinto

Vamos a situarnos. Tenemos una rejilla con todas las imágenes que componen un personaje. Estas imágenes son de diferentes animaciones. Necesitamos una estructura que almacene una secuencia de cuadros para cada una de las animaciones, así como debe mostrar la imagen de la animación en un instante dado con lo que conseguiremos el efecto de animación deseado. Es decir necesitamos mostrar la secuencialidad de las imágenes para conseguir la animación.

Cada vez que mostremos una imagen por pantalla deberemos de borrar la anterior sino queremos ir dejando un rastro de imágenes antiguas por la superficie principal. Recuerda que el blit lo realizamos entre la superficie principal y una imagen actual. Estas imágenes van quedándose “grabadas” en dicha superficie mostrando un número de imágenes en pantalla que no tiene sentido ninguno.

La solución a este problema es intuitiva. Se trata de borrar el frame anterior antes de dibujar el siguiente, con lo que conseguiremos no dejar residuos en la pantalla. Depende del juego que estemos implementando tendremos a bien actualizar sólo una porción de pantalla o bien toda ella, teniendo en cuenta el trabajo adicional que supone para la máquina realizar cada una de estas acciones.

15. Los Sprites y los Personajes

Necesitamos una estructura que nos almacene qué imágenes pertenecen a qué animación y en qué secuencia deben ser mostradas, es decir, lo que se conoce como control lógico de la animación interna. Recuerda que la animación interna es la propia del personaje mientras que la externa es el movimiento por un determinado superficie o mapa.

15.4.4. Implementando el Control de la animación interna

Para implementar este control tenemos que hacer uso del concepto de clase. Vamos a usar parte de la potencia que nos brinda el lenguaje C++ para realizar un mejor trabajo de codificación que con el ejemplo anterior. Tenemos que crear una clase que nos permita, a partir de una rejilla, definir varios tipos de animaciones que mostrar por pantalla. Todo este proceso lo utilizaremos para nuestro proyecto final.

La siguiente clase que vamos a presentar es el control lógico de la animación. Esta clase se encarga de llevar el control de qué posiciones o figuras de la rejilla pertenecen a una animación concreta. Todo los métodos que implementamos en ella están destinados a ofrecer la suficiente potencia para realizar esta tarea. Veamos el fichero de cabecera que define la clase:

```
1 ;// Listado: Control_Animacion.h
2 ;//
3 ;// Esta clase controla la secuencia de animación de los personajes de la
4 ;// aplicación
5 ;
6 ;#ifndef _CONTROL_ANIMACION_H_
7 ;#define _CONTROL_ANIMACION_H_
8 ;
9 ;const int MAX_NUM CUADROS = 30;
10 ;
11 ;class Control_Animacion {
12 ; public:
13 ;
14 ;    // Constructor
15 ;    Control_Animacion(char *frames);
16 ;
17 ;    // Consultoras
18 ;    int cuadro(void);
19 ;    bool es_primer_cuadro(void);
20 ;
21 ;    // Modificadoras
22 ;    int avanzar(void);
23 ;    void reiniciar(void);
24 ;
25 ;    // Destructor
26 ;    ~Control_Animacion();
```

```

27 ;
28 ; private:
29 ;
30 ;     int cuadros[MAX_NUM_CUADROS];
31 ;     int paso;
32 ;};
33 ;
34 ;#endif
;
```

Vamos a estudiar la definición de esta clase. Entre los elementos públicos de la clase encontramos:

- **El constructor (ControlAnimacion()):** Recibe una cadena separada por comas indicando las posiciones de la rejilla que pertenecen a la animación que estamos creando. Un ejemplo de una cadena de este tipo es, por ejemplo, “1, 3, 5, 7”. Con esta cadena definiríamos que la animación que estamos creando se corresponde con las posiciones 1, 3, 5 y 7 de la rejilla, que han de mostrarse en este orden, seguramente, en varias repeticiones. El parámetro *retardo* hace referencia al espaciado de tiempo que vamos a establecer entre mostrar un frame de la animación y el siguiente.

▪ Funciones consultoras

- *int cuadro(void)*: Esta función devuelve el cuadro actual que debe ser mostrado en para seguir la secuencialidad de la animación.
- *bool es_primer_cuadro(void)*: Esta función nos devuelve si el cuadro a mostrar es el primero de la secuencia.

▪ Funciones modificadoras

- *int avanzar(void)*: Avanza un cuadro de la animación.
- *void reiniciar(void)*: Coloca la animación en el primer cuadro.

En la parte privada de la clase nos encontramos con varios tipos de variables. La variable *paso* controla la situación actual de la animación, es decir, el paso en el que se encuentra. En el vector *cuadros* almacenamos la secuencialidad de las imágenes de la animación. Las otras dos variables se destinan a controlar el retardo entre frame y frame que en esta implementación no vamos a tomar en cuenta.

Veamos la implementación de estas funciones:

```

;_____
1 ;// Listado: Control_Animacion.cpp
2 ;//
3 ;// Implementación de la clase Control_Animacion
4 ;
5 ;#include <iostream>
```

15. Los Sprites y los Personajes

```
6 ;#include "Control_Animacion.h"
7 ;
8 ;
9 ;using namespace std;
10 ;
11 ;
12 ;Control_Animacion::Control_Animacion(char *frames) {
13 ;
14 ;    int i = 0;
15 ;    char frames_tmp[1024];
16 ;    char *proximo;
17 ;
18 ;
19 ;    strcpy(frames_tmp, frames);
20 ;
21 ;    // Trabajamos con una copia de los cuadros indicados
22 ;
23 ;    for(proximo = strtok(frames_tmp, ","); proximo; i++){
24 ;
25 ;        // Desmembramos la cadena separada por comas
26 ;
27 ;        this->cuadros[i] = atoi(proximo);
28 ;        proximo = strtok(NULL, ",\0");
29 ;    }
30 ;
31 ;    // Inicializamos las variables
32 ;
33 ;    this->cuadros[i] = -1;
34 ;    this->paso = 0;
35 ;
36 ;#ifdef DEBUG
37 ;    cout << "Control_Animacion::Control_Animacion()" << endl;
38 ;#endif
39 ;
40 ;
41 ;
42 ;int Control_Animacion::cuadro(void) {
43 ;
44 ;    return cuadros[paso];
45 ;}
46 ;
47 ;
48 ;int Control_Animacion::avanzar(void) {
49 ;
50 ;    if(cuadros[++paso] == -1) {
51 ;        paso = 0;
52 ;        return 1;
53 ;    }
54 ;
55 ;    return 0;
56 ;}
57 ;
58 ;
```

15.4. Animando un Sprite

```
59 ;void Control_Animacion::reiniciar(void) {
60 ;
61 ;    // Volvemos al principio
62 ;
63 ;    paso = 0;
64 ;}
65 ;
66 ;
67 ;bool Control_Animacion::es_primer_cuadro(void) {
68 ;
69 ;    if(paso == 0)
70 ;        return true;
71 ;
72 ;    return false;
73 ;}
74 ;
75 ;Control_Animacion::~Control_Animacion() {
76 ;
77 ;#ifdef DEBUG
78 ;    cout << "Control_Animacion::~Control_Animacion()" << endl;
79 ;#endif
80 ;}
```

La función que tiene una mayor carga de código es el constructor porque implementamos el bucle que nos permite separar la cadena pasada por el usuario en elementos unarios que almacenar en el vector. Lo demás no tiene mayor complejidad. En la última posición de *cuadros* almacenamos un -1 para saber cuando ha llegado el final de los elementos del vector.

Vamos a implementar un programa de ejemplo que nos permita hacer uso de esta clase y así comprobar la potencia de la misma. En el programa inicializaremos una secuencia cualquiera que mostraremos repetidamente. Aquí tienes el código de la misma:

```
;_____
1 ;// Listado: main.cpp
2 ;//
3 ;// Programa de prueba de la clase Control_Animación
4 ;
5 ;#include <iostream>
6 ;
7 ;#include "Control_Animacion.h"
8 ;
9 ;using namespace std;
10 ;
11 ;int main() {
12 ;
13 ;    Control_Animacion animacion("0, 2, 4, 6, 8");
14 ;
15 ;    cout << "Secuencia: " ;
16 ;
17 ;    for(int i = 0; i < 100; i++) {
```

15. Los Sprites y los Personajes

```
18 ;
19 ;     cout << " - " << animacion.cuadro();
20 ;
21 ;     animacion.avanzar();
22 ;
23 ; }
24 ;
25 ;     cout << endl;
26 ;
27 ; return 0;
28 ;
29 ;}
```

En este programa de ejemplo hemos inicializado una variable del tipo *Control_Animacion* con unos valores determinados que marcan la secuencialidad de la animación. Luego hemos mostrado la secuencia que devuelve esta estructura para comprobar si es la correcta. Lo hacemos repetidamente para comprobar su correcto funcionamiento y efectivamente, por mucho que repitamos la consulta, nos devuelve la misma secuencia que poco más adelante será la que defina el orden de la animación.

15.4.5. Gestionando una rejilla de imágenes

Bien, ya tenemos una clase que nos permite controlar la secuencialidad de las imágenes de, por ejemplo, una rejilla de imágenes. Ahora necesitamos una clase que nos permita cargar una rejilla de imágenes y escoger una de las imágenes de dicha rejilla para ser mostrada por pantalla.

La idea es la siguiente. Vamos a cargar la rejilla entera en memoria. Esto nos ahorrará un número considerable de accesos a disco en comparación con la técnica utilizada en nuestra primera animación. La ventaja de este método es, principalmente, esa. Tendremos todas las imágenes en memoria en todo momento y podremos utilizarlas con cierta rapidez.

La principal desventaja del método es que ocupamos bastante mayor cantidad de memoria que con el método de cargar las imágenes sueltas. Como bien sabes el orden de velocidad de la memoria principal en un ordenador es bastante menor que el de la velocidad de la memoria secundaria por lo que la diferencia es considerable. Esto, unido a que la memoria principal actualmente no es artículo de lujo, nos lleva a evaluar esta como la mejor opción que se nos presenta.

El fichero de cabecera que nos permite gestionar las rejillas es el siguiente:

```
1 ;// Listado: Imagen.h
2 //
```

15.4. Animando un Sprite

```
3 // Clase para gestionar el trabajo con imágenes y rejillas
4 ;
5 ifndef _IMAGEN_H_
6 define _IMAGEN_H_
7 ;
8 #include <SDL/SDL.h>
9 ;
10 ;
11 class Imagen {
12 ;
13 public:
14 ;
15 // Constructor
16 ;
17 Imagen(char *ruta, int filas = 1, int columnas = 1,\n
18     Uint32 r = 0, Uint32 g = 255, Uint32 b = 0);
19 ;
20 void dibujar(SDL_Surface *superficie, int i, int x, int y, int flip = 1);
21 ;
22 // Consultoras
23 ;
24 int anchura(void);
25 int altura(void);
26 int cuadros(void);
27 ;
28 // Destructor
29 ;
30 ~Imagen();
31 ;
32 private:
33 ;
34 SDL_Surface *imagen;
35 SDL_Surface *imagen_invertida;
36 ;
37 // Propiedades de la rejilla de la imagen
38 int columnas;
39 int filas;
40 ;
41 // Ancho y alto por frame o recuerdo de la animación
42 int w, h;
43 ;
44 // Invierte la imagen en horizontal
45 SDL_Surface * invertir_imagen(SDL_Surface *imagen);
46 ;
47 // Color clave
48 Uint32 colorkey;
49 };
50 ;
51 endif
```

Vamos a estudiar la definición de la clase. En la parte pública de la clase encontramos lo siguiente:

15. Los Sprites y los Personajes

- **Constructor Imagen()**: El constructor recibe como parámetros primero la ruta donde está almacenada la imagen que contiene la rejilla, segundo el número de filas de imágenes que contiene el fichero a cargar y el tercero el número de columnas de imágenes que contiene dicho fichero. Si queremos cargar una imagen simple basta con indicar que hay una fila de imágenes y una sola columna, o como por omisión el valor de estos campos es uno, no sería necesario indicarlo.

En los campos r, g, b podemos definir que color queremos utilizar como *color key* para nuestras imágenes para cuando realicemos el blit sobre la superficie principal. Se puede establecer un color key por clase si lo consideramos oportuno. Por omisión será el color verde.

- *void dibujar(...)*: Esta función dibujar en la superficie que recibe como primer parámetro la imagen número *i* en la posición (x, y) de dicha superficie. El parámetro flip indica si queremos que la imagen sea la original o rotada verticalmente. Esto es especialmente útil para realizar animaciones en dos sentidos. No tenemos que crear al personaje volteado para que “ande” hacia el lado contrario de donde lo dibujamos.
- *int anchura(void)*: Devuelve la anchura de la imagen cargada en memoria.
- *int altura(void)*: Devuelve la altura de la imagen cargada en memoria.
- *int cuadros(void)*: Devuelve el número de cuadros o frames que posee la imagen cargada en memoria.
- **Destructor Imagen()**: Libera la memoria de los elementos utilizados.

En la parte privada de la clase nos encontramos variables que almacenan información de esta clase como el número de filas y de columnas o el ancho y alto de la superficie que almacena la clase. También almacenamos la imagen invertida para los efectos de movimiento.

En esta parte de la clase se encuentra también la función encargada de obtener la imagen invertida ya que es una función interna que la clase no va a ofrecer como método de la misma.

Veamos la implementación de las funciones:

```
1 ;// Listado: Imagen.cpp
2 ;//
3 ;// Implementación de la clase imagen para la
4 ;// gestión de imágenes y rejillas
5 ;
```

15.4. Animando un Sprite

```
6 ;#include <iostream>
7 ;#include <SDL/SDL_image.h>
8 ;
9 ;#include "Imagen.h"
10 ;
11 ;using namespace std;
12 ;
13 ;
14 ;Imagen::Imagen(char *ruta, int filas, int columnas, \
15 ;                  Uint32 r, Uint32 g, Uint32 b) {
16 ;
17 ;    this->filas = filas;
18 ;    this->columnas = columnas;
19 ;
20 ;#ifdef DEBUG
21 ;    cout << "-> Cargando" << ruta << endl;
22 ;#endif
23 ;
24 ;    // Cargamos la imagen
25 ;
26 ;    imagen = IMG_Load(ruta);
27 ;
28 ;    if(imagen == NULL) {
29 ;
30 ;        cerr << "Error: " << SDL_GetError() << endl;;
31 ;        exit(1);
32 ;
33 ;    }
34 ;
35 ;    // Convertimos a formato de pantalla
36 ;
37 ;    SDL_Surface *tmp = imagen;
38 ;
39 ;    imagen = SDL_DisplayFormat(tmp);
40 ;
41 ;    SDL_FreeSurface(tmp);
42 ;
43 ;    if(imagen == NULL) {
44 ;
45 ;        cerr << "Error: " << SDL_GetError() << endl;
46 ;        exit(1);
47 ;
48 ;    }
49 ;
50 ;    // Calculamos el color transparente, en nuestro caso el verde
51 ;
52 ;    colorkey = SDL_MapRGB(imagen->format, r, g, b);
53 ;
54 ;    // Lo establecemos como color transparente
55 ;
56 ;    SDL_SetColorKey(imagen, SDL_SRCCOLORKEY, colorkey);
57 ;
58 ;
```

15. Los Sprites y los Personajes

```
59 ; // Hallamos la imagen invertida utilizada en el mayor de los casos
60 ; // para las imágenes de vuelta
61 ;
62 ;     imagen_invertida = invertir_imagen(imagen);
63 ;
64 ;     if(imagen_invertida == NULL) {
65 ;
66 ;         cerr << "No se pudo invertir la imagen: " << SDL_GetError() << endl;
67 ;         exit(1);
68 ;
69 ;     }
70 ;
71 ;     // El ancho de una imagen es el total entre el número de columnas
72 ;     w = imagen->w / columnas;
73 ;
74 ;     // El alto de una imagen es el total entre el número de filas
75 ;     h = imagen->h / filas;
76 ;
77 ;}
78 ;
79 ;
80 ;
81 ;void Imagen::dibujar(SDL_Surface *superficie, int i, int x, int y, int flip)
82 ;{
83 ;
84 ;    SDL_Rect destino;
85 ;
86 ;    destino.x = x;
87 ;    destino.y = y;
88 ;
89 ;    // No se usan
90 ;
91 ;    destino.h = 0;
92 ;    destino.w = 0;
93 ;
94 ;    // Comprobamos que el número de imagen indicado sea el correcto
95 ;    if(i < 0 || i > (filas * columnas)) {
96 ;
97 ;        cerr << "Imagen::Dibujar = No existe el cuadro: " << i << endl;
98 ;        return;
99 ;    }
100 ;
101 ;    SDL_Rect origen;
102 ;
103 ;    // Separaciones de 2 píxeles dentro de las rejillas para observar
104 ;    // bien donde empieza una imagen y donde termina la otra
105 ;
106 ;    origen.w = w - 2;
107 ;    origen.h = h - 2;
108 ;
109 ;    // Seleccionamos cual de las imágenes es la que vamos a dibujar
110 ;    switch(flip) {
111 ;
```

15.4. Animando un Sprite

```
112 ;     case 1:
113 ;
114 ;         // Cálculo de la posición de la imagen
115 ;         // dentro de la rejilla
116 ;
117 ;         origen.x = ((i % columnas) * w) + 2;
118 ;         origen.y = ((i / columnas) * h) + 2;
119 ;
120 ;         SDL_BlitSurface(imagen, &origen, superficie, &destino);
121 ;
122 ;         break;
123 ;
124 ;     case -1:
125 ;
126 ;         // Cálculo de la posición de la imagen
127 ;         // dentro de la rejilla invertida
128 ;
129 ;         origen.x = ((columnas-1) - (i % columnas)) * w + 1;
130 ;         origen.y = (i / columnas) * h + 2;
131 ;
132 ;         // Copiamos la imagen en la superficie
133 ;
134 ;         SDL_BlitSurface(imagen_invertida, &origen, superficie, &destino);
135 ;
136 ;         break;
137 ;
138 ;     default:
139 ;
140 ;         cerr << "Caso no válido: Imagen invertida o no" << endl;
141 ;         break;
142 ;     }
143 ;}
144 ;
145 ;
146 ;// Devuelve la anchura de un cuadro de la rejilla
147 ;
148 ;int Imagen::anchura() {
149 ;
150 ;    return w;
151 ;}
152 ;
153 ;
154 ;// Devuelve la altura de un cuadro de la rejilla
155 ;
156 ;int Imagen::altura() {
157 ;
158 ;    return h;
159 ;}
160 ;
161 ;// Devuelve el número de cuadros de la rejilla de la imagen
162 ;
163 ;int Imagen::cuadros() {
164 ;
```

15. Los Sprites y los Personajes

```
165 ;     return columnas * filas;
166 ;
167 ;
168 ;Imagen::~Imagen()
169 ;{
170 ;
171 ;    SDL_FreeSurface(imagen);
172 ;    SDL_FreeSurface(imagen_invertida);
173 ;
174 ;#ifdef DEBUG
175 ;    cout << "<- Liberando imagen" << endl;
176 ;#endif
177 ;
178 ;
179 ;
180 ;
181 ;SDL_Surface * Imagen::invertir_imagen(SDL_Surface *imagen) {
182 ;
183 ;    SDL_Rect origen;
184 ;    SDL_Rect destino;
185 ;
186 ;
187 ;    // Origen -> ancho una línea
188 ;    // Comienzo de copia por el principio
189 ;
190 ;    origen.x = 0;
191 ;    origen.y = 0;
192 ;    origen.w = 1;
193 ;    origen.h = imagen->h;
194 ;
195 ;    // Destino -> ancho una línea
196 ;    // Comienzo de 'pegado' por el final
197 ;    // Para lograr la inversión
198 ;
199 ;    destino.x = imagen->w;
200 ;    destino.y = 0;
201 ;    destino.w = 1;
202 ;    destino.h = imagen->h;
203 ;
204 ;    SDL_Surface *invertida;
205 ;
206 ;    // Pasamos imagen a formato de pantalla
207 ;
208 ;    invertida = SDL_DisplayFormat(imagen);
209 ;
210 ;    if(invertida == NULL) {
211 ;
212 ;        cerr << "No podemos convertir la imagen al formato de pantalla" << endl;
213 ;        return NULL;
214 ;    }
215 ;
216 ;    // Preparamos el rectángulo nuevo vacío del color transparente
217 ;
```

15.4. Animando un Sprite

```
218 ;     SDL_FillRect(invertida, NULL, SDL_MapRGB(invertida->format, 0, 255, 0));
219 ;
220 ;     // Copiamos linea vertical a linea vertical, inversamente
221 ;
222 ;     for(int i = 0; i < imagen->w; i++) {
223 ;
224 ;         SDL_BlitSurface(imagen, &origen, invertida, &destino);
225 ;
226 ;         origen.x = origen.x + 1;
227 ;         destino.x = destino.x - 1;
228 ;     }
229 ;
230 ;     return invertida;
231 ;
232 ;}
```

Como puedes ver todas las funciones están detalladamente comentadas. La implementación no tiene mayor complejidad.

Vamos a crear un programa de prueba que nos permita comprobar el funcionamiento de esta clase. Se trata de cargar una rejilla y mostrar diferentes imágenes de la misma en pantalla y así mostrar la potencia de la clase. Aquí tienes el código de la aplicación:

```
;-----  
1 ;// Listado: main.cpp  
2 ;//  
3 ;// Programa de prueba de la clase Imagen  
4 ;  
5 ;#include <iostream>  
6 ;#include <SDL/SDL.h>  
7 ;  
8 ;#include "Imagen.h"  
9 ;  
10 ;using namespace std;  
11 ;  
12 ;int main() {  
13 ;  
14 ;  
15 ;    // Iniciamos el subsistema de video  
16 ;  
17 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {  
18 ;  
19 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;  
20 ;        exit(1);  
21 ;    }  
22 ;  
23 ;  
24 ;    atexit(SDL_Quit);  
25 ;  
26 ;    // Comprobamos que sea compatible el modo de video  
27 ;  
28 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {  
-----
```

15. Los Sprites y los Personajes

```
30 ;
31 ;         cerr << "Modo no soportado: " << SDL_GetError() << endl;
32 ;         exit(1);
33 ;
34 ;     }
35 ;
36 ;
37 ;     // Establecemos el modo de video
38 ;
39 ;     SDL_Surface *pantalla;
40 ;
41 ;     pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
42 ;
43 ;     if(pantalla == NULL) {
44 ;
45 ;         cerr << "No se pudo establecer el modo de video: "
46 ;             << SDL_GetError() << endl;
47 ;
48 ;         exit(1);
49 ;
50 ;
51 ;
52 ;     // Creamos un elemento de la clase Imagen
53 ;
54 ;     Imagen personaje("./Imagenes/jacinto.bmp", 4, 7);
55 ;
56 ;
57 ;     // Mostramos por consola información de la imagen
58 ;
59 ;     cout << "La imagen tiene: " << endl;
60 ;     cout << "- anchura: " << personaje.anchura() << " px" << endl;
61 ;     cout << "- altura : " << personaje.altura() << " px" << endl;
62 ;     cout << "- cuadros: " << personaje.cuadros() << endl;
63 ;
64 ;     // Mostramos varios "frames"
65 ;
66 ;     // El primero normal y rotado
67 ;
68 ;     personaje.dibujar(pantalla, 0, 0, 0);
69 ;     personaje.dibujar(pantalla, 0, 100, 0, -1);
70 ;
71 ;     // El 10 normal y rotado
72 ;
73 ;     personaje.dibujar(pantalla, 10, 100, 100);
74 ;     personaje.dibujar(pantalla, 10, 200, 100, -1);
75 ;
76 ;     // El 20 normal y rotado
77 ;
78 ;     personaje.dibujar(pantalla, 20, 200, 200);
79 ;     personaje.dibujar(pantalla, 20, 300, 200, -1);
80 ;
81 ;     // El 12 normal y rotado
82 ;
```

15.4. Animando un Sprite

```
83 ;     personaje.dibujar(pantalla, 12, 300, 300);
84 ;     personaje.dibujar(pantalla, 12, 400, 300, -1);
85 ;
86 ;     // Actualizamos la superficie principal
87 ;
88 ;     SDL_Flip(pantalla);
89 ;
90 ;
91 ;
92 ;     // Variables auxiliares
93 ;
94 ;     SDL_Event evento;
95 ;
96 ;     // Bucle "infinito"
97 ;
98 ;     for( ; ; ) {
99 ;
100 ;         while(SDL_PollEvent(&evento)) {
101 ;
102 ;             if(evento.type == SDL_KEYDOWN) {
103 ;
104 ;                 if(evento.key.keysym.sym == SDLK_ESCAPE)
105 ;                     return 0;
106 ;
107 ;             }
108 ;
109 ;             if(evento.type == SDL_QUIT)
110 ;                 return 0;
111 ;
112 ;         }
113 ;     }
114 ;}
```

En este programa de prueba lo primero que hacemos es inicializar la librería `SDL` y establecer un modo de vídeo. Seguidamente creamos una instancia de la clase `Imagen` `personaje` con los valores correspondientes.

Lo primero que hacemos con dicha clase es mostrar toda la información que tenemos de ella. A continuación, y a modo de ejemplo, mostramos por pantalla varios de los *frames* que almacena la imagen que cargamos en dicha clase. Como puedes ver esta clase nos libera de varias tareas, como la de definir una variable del tipo `SDL_Rect` para indicar el lugar donde mostrar la imagen cargada o establecer el *color key* que hemos definido a verde por omisión ya que es el que vamos a utilizar en nuestras aplicaciones.

En definitiva. Esta clase nos permite trabajar tanto con imágenes individuales como con rejillas de imágenes. Tiene la suficiente potencia para liberarnos de muchas tareas tediosas y repetitivas. Ahora bien, haciendo uso de las dos últimas clases, ¿serías capaz de implementar una clase que reproduciese una animación?

15. Los Sprites y los Personajes

15.4.6. Clase Animación Interna

Vamos a unir las dos últimas clases que hemos implementado en una que nos permita crear una animación, ya sea de un personaje principal que vayamos a controlar, de un enemigo o una simple presentación a pantalla completa.

Dicha animación se basará en las imágenes de una rejilla que mostrará secuencialmente según sea el efecto que queramos poner de manifiesto. Deberá de tener un control sobre la velocidad de dicha animación para que no se convierta en un garabato en la pantalla.

Vamos a ver el fichero de cabecera de la clase:

```
;  
1 ;// Listado: Animacion.h  
2 ;//  
3 ;// Clase Animación  
4 ;  
5 ;#ifndef _ANIMACION_H_  
6 ;#define _ANIMACION_H_  
7 ;  
8 ;#include <SDL/SDL.h>  
9 ;  
10 ;// Declaración adelantada  
11 ;  
12 ;class Imagen;  
13 ;class Control_Animacion;  
14 ;  
15 ;// Clase  
16 ;  
17 ;class Animacion {  
18 ;  
19 ; public:  
20 ;  
21 ;     // Constructor  
22 ;  
23 ;     Animacion(char *ruta_imagen, int filas, int columnas,\br/>24 ;                 char *frames, int retardo);  
25 ;  
26 ;     // Animación fija e infinita en pantalla  
27 ;  
28 ;     void animar(SDL_Surface *pantalla, int x, int y, int flip = 1);  
29 ;  
30 ;     // Dibuja la animación paso por paso  
31 ;  
32 ;     void paso_a_paso(SDL_Surface *pantalla, int x, int y, int flip = 1);  
33 ;  
34 ;     Uint32 retardo();  
35 ;  
36 ; private:  
37 ;  
38 ;     Imagen *imagen;
```

15.4. Animando un Sprite

```
39 ;     Control_Animacion *control_animacion;
40 ;     Uint32 retardo_;
41;};
42;
43;#endif
```

Vemos que la clase Animación en su parte pública tiene cuatro funciones. La primera de ella es el constructor al que le pasaremos la ruta de la imagen que contiene la rejilla, el número de filas y columnas de dicha rejilla, una cadea especificando qué imágenes de la rejilla formarán la animación y un retardo, que será el tiempo que transcurra entre fotograma y fotograma de la animación.

La siguiente función *animar()* reproduce una animación estática e infinita en la superficie que recibe como parámetro. Esta función dibujará la animación en la posición (x, y) de la pantalla.

La función *paso_a_paso()* imprime en pantalla el siguiente fotograma de la animación. Si en un momento dado hemos mostrado por pantalla el tercer fotograma de la animación, al llamar a esta función, nos mostrará el cuarto fotograma en la posición (x, y) . Esta función nos va a ser muy útil a la hora de combinar la animación interna con la externa.

En la parte privada encontramos varias variables. Como no podía ser de otra forma en *imagen* y *control_animacion* ponemos de manifiesto el uso de estas clases dentro de nuestra nueva clase. La variable *retardo_* almacena el retardo que el usuario ha establecido para las secuencias de la animación.

Vamos a estudiar la implementación de estas funciones. Veamos el código fuente de la clase:

```
1; // Listado: Animacion.cpp
2;//
3; // Implementación de la clase animación
4;
5;#include <iostream>
6;
7;#include "Animacion.h"
8;#include "Imagen.h"
9;#include "Control_Animacion.h"
10;
11;using namespace std;
12;
13;Animacion::Animacion(char *ruta_imagen, int filas, int columnas,\n                      char *frames, int retardo_) {
14;
15;
16;    // Inicializamos las variables de la clase
17;
18;    this->imagen = new Imagen(ruta_imagen, filas, columnas);
19;    this->control_animacion = new Control_Animacion(frames);
```

15. Los Sprites y los Personajes

```
20 ;     this->retardo_ = retardo_;
21 ;
22 ;}
23 ;
24 ;void Animacion::animar(SDL_Surface *pantalla, int x, int y, int flip) {
25 ;
26 ;    Uint32 t0;
27 ;    Uint32 t1;
28 ;
29 ;    for( ; ; ) {
30 ;
31 ;        imagen->dibujar(pantalla, control_animacion->cuadro(),x, y, flip);
32 ;        control_animacion->avanzar();
33 ;        SDL_Flip(pantalla);
34 ;
35 ;        SDL_FillRect(pantalla, NULL, \
36 ;                      SDL_MapRGB(pantalla->format, 0, 0, 0));
37 ;
38 ;        t0 = SDL_GetTicks();
39 ;        t1 = SDL_GetTicks();
40 ;
41 ;        while((t1 - t0) < retardo_) {
42 ;
43 ;            t1 = SDL_GetTicks();
44 ;
45 ;        }
46 ;
47 ;    }
48 ;
49 ;}
50 ;
51 ;// El control de la temporalidad tiene que ser externo
52 ;
53 ;void Animacion::paso_a_paso(SDL_Surface *pantalla, int x, int y, int flip){
54 ;
55 ;    imagen->dibujar(pantalla, control_animacion->cuadro(), x, y, flip);
56 ;    control_animacion->avanzar();
57 ;    SDL_Flip(pantalla);
58 ;
59 ;}
60 ;
61 ;Uint32 Animacion::retardo() {
62 ;
63 ;    return retardo_;
64 ;}
```

Como puedes ver el constructor se limita a inicializar las variables que vamos a utilizar en el control y muestra por pantalla de la animación. La función *animar()* al mostrar una animación sin movimiento externo, controla su propio tiempo de retardo entre cuadro y cuadro haciendo uso de las funciones SDL que nos permiten tomar señales de tiempo. Va mostrando fotograma a fotograma en una posición fija.

La función *paso_a_paso()* no lleva el control del tiempo. No sería lógico que lo llevarse. Esta función muestra un sólo fotograma, el siguiente al actual. Esto nos permite temporizar exteriormente la animación, lo que unido a que cada fotograma podemos colocarlo en la posición que más nos convenga, es la mejor opción para controlar la animación si pensamos en añadirle translación sobre la superficie principal.

El control del tiempo es fundamental en la animación. No podemos ser dependientes de la máquina en la que se esté ejecutando nuestra aplicación para obtener una respuesta u otra de la misma. Debemos de realizar el pertinente control de tiempo. Desde que se difundió el uso de ordenadores personales es habitual encontrar en las aplicaciones de juego que se realice este control pero tiempo atrás, cuando existía una mayor variedad de sistemas propietarios, no era tan fundamental este control ya que todas las máquinas donde iba a ser ejecutado el videojuego tenían unas características similares lo que permitía que el comportamiento del videojuego fuera totalmente determinista y relativamente fácil de controlar.

Cuando desarrolles un videojuego con SDL será fundamental que se realice este control para que el videojuego sea jugado tal como diseñaste. En nuestro caso y gracias a la portabilidad que nos ofrece SDL a diferentes sistemas operativos este control se presenta aún más crítico que con otras herramientas uniplataforma. Podemos encontrarnos el caso de ejecutar nuestra aplicación en dos computadoras exactamente igual a nivel hardware pero con diferentes sistemas operativos y no tener la misma respuesta del sistema. Siempre podremos retener una acción para que produzca dentro del intervalo de tiempo que nosotros queremos lo que no podemos mejorar con esta técnica es lo referente a la necesidad de unos recursos mayores.

La única forma que podemos solucionar el tema de falta de recursos es realizando un buen diseño e implementación de la aplicación para conseguir un videojuego lo más eficiente posible.

Para terminar esta sección hemos creado un pequeño programa de prueba para la clase Animación que nos permite mostrar diferentes animaciones en pantalla. El código es el siguiente:

```
1 ;// Listado: main.cpp
2 ;//
3 ;// Programa de prueba de la clase Imagen
4 ;
5 ;#include <iostream>
6 ;#include <SDL/SDL.h>
7 ;
```

15. Los Sprites y los Personajes

```
8 ;#include "Animacion.h"
9 ;
10 ;using namespace std;
11 ;
12 ;int main() {
13 ;
14 ;
15 ;    // Iniciamos el subsistema de video
16 ;
17 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
18 ;
19 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
20 ;        exit(1);
21 ;
22 ;    }
23 ;
24 ;
25 ;    atexit(SDL_Quit);
26 ;
27 ;    // Comprobamos que sea compatible el modo de video
28 ;
29 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
30 ;
31 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
32 ;        exit(1);
33 ;
34 ;    }
35 ;
36 ;
37 ;    // Establecemos el modo de video
38 ;
39 ;    SDL_Surface *pantalla;
40 ;
41 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
42 ;
43 ;    if(pantalla == NULL) {
44 ;
45 ;        cerr << "No se pudo establecer el modo de video: "
46 ;            << SDL_GetError() << endl;
47 ;
48 ;        exit(1);
49 ;    }
50 ;
51 ;    // Creamos animaciones para llenar la pantalla
52 ;
53 ;    Animacion animacion("./Imagenes/jacinto.bmp", 4, 7,\n
54 ;                        "0,1,2,3,4,3,2,1,0", 120);
55 ;
56 ;    Animacion animacion1("./Imagenes/jacinto.bmp", 4, 7,\n
57 ;                        "0,15,15,0,0,15", 120);
58 ;
59 ;    Animacion animacion2("./Imagenes/jacinto.bmp", 4, 7,\n
60 ;                        "22,23,24,25", 120);
```

15.4. Animando un Sprite

```
61 ;
62 ;    // animacion.animar(pantalla, 100, 100);
63 ;
64 ;
65 ;    // Variables auxiliares
66 ;
67 ;    SDL_Event evento;
68 ;
69 ;    // Para controlar el tiempo
70 ;
71 ;    Uint32 t0 = SDL_GetTicks();
72 ;    Uint32 t1;
73 ;
74 ;    // Bucle "infinito"
75 ;
76 ;    for(int x = 0, y = 0 ; ; ) {
77 ;
78 ;        // Si se sale de la pantalla volvemos a introducirlo
79 ;
80 ;        if(x == 640)
81 ;            x = 0;
82 ;
83 ;        // Referencia de tiempo
84 ;
85 ;        t1 = SDL_GetTicks();
86 ;
87 ;        if((t1 - t0) > animacion.retardo()) {
88 ;
89 ;            // Nueva referencia de tiempo
90 ;
91 ;            t0 = SDL_GetTicks();
92 ;
93 ;            // Movimiento del personaje
94 ;
95 ;            x += 4;
96 ;
97 ;            // Limpiamos la pantalla
98 ;            // Sería mejor limpiar el anterior
99 ;
100 ;            SDL_FillRect(pantalla, NULL,\n
101 ;                         SDL_MapRGB(pantalla->format, 0, 0, 0));
102 ;
103 ;            // Mostramos el siguiente paso de todas las animaciones
104 ;
105 ;            animacion.paso_a_paso(pantalla, x, y);
106 ;            animacion1.paso_a_paso(pantalla, 200, 300);
107 ;            animacion2.paso_a_paso(pantalla, 300, 300);
108 ;
109 ;
110 ;
111 ;            // Control de la entrada
112 ;
113 ;            while(SDL_PollEvent(&evento)) {
```

15. Los Sprites y los Personajes

```
114 ;  
115 ;           if(evento.type == SDL_KEYDOWN) {  
116 ;  
117 ;               if(evento.key.keysym.sym == SDLK_ESCAPE)  
118 ;                   return 0;  
119 ;  
120 ;           }  
121 ;  
122 ;           if(evento.type == SDL_QUIT)  
123 ;               return 0;  
124 ;  
125 ;       }  
126 ;   }  
127 ;}
```

Vamos a lo novedoso. Una vez inicializada SDL y establecido el modo de video creamos tres variables animación para tres animaciones diferentes. Las inicializamos con los valores correspondientes observando en la imagen que almacena la rejilla qué figuras son necesarias para cada una de las animaciones.

Una vez definidas creamos unas variables que nos van a permitir controlar el tiempo de ejecución de la aplicación. Una vez en el bucle principal tendremos dos variables x e y que nos van a aportar un control sobre el movimiento lineal (externo) que le vamos a dar a las animaciones para ir introduciéndonos en la siguiente sección.

Controlamos que ninguna de estas variables se salgan de la pantalla para que no perdamos de vista nuestras animaciones. Mediante t1 y t0 controlaremos el tiempo. Si el intervalo de tiempo es lo suficientemente grande dibujaremos el siguiente frame de nuestras animaciones en pantalla. Así de simple. Antes de dibujar los siguientes cuadros limpiaremos la pantalla para no ir dejando residuos en la misma. Este bucle se repetirá hasta que el usuario se haya cansado de observar las tres animaciones y decida salir de la aplicación.

Ya sabemos como animar un sprite en nuestra aplicación. Ahora bien, un personaje de un videojuego puede tener numerosos estados. Un estado define un comportamiento. Por ejemplo cuando un personaje anda tiene asociado un estado que es “andando”. Este personaje puede en ese momento pararse, saltar, golpear... realizar ciertas acciones que lo lleven a un cambio de estado. Lo primero que tenemos que hacer al diseñar un personaje es saber en qué estados vamos a representar al personaje. Todo esto está sujeto a una base teórica que trata sobre autómatas que estudiaremos una vez hallamos visto todos los tipos de animaciones.

15.4.7. Animación Externa. Actualización Lógica

Mucha de la bibliografía diferencia entre animación interna y externa. Nosotros vamos a respetar el término de animación externa aún no estando de acuerdo totalmente con él.

La animación externa es aquella que realiza el personaje al moverse por la pantalla o por el nivel. Mediante posiciones de coordenadas (x, y) podemos controlar la posición del personaje en todo momento. Para los enemigos podemos establecer un movimiento por la pantalla dado por funciones matemáticas dependiendo de la inteligencia que le queramos dotar. Para mover al personaje principal necesitaremos un control de un dispositivo de entrada para traducir las acciones sobre él en movimientos sobre la pantalla. Podemos decir que la animación externa es el cambio de posición del personaje u objeto dentro de la superficie principal que define la pantalla.



Figura 15.3: Representación de la animación externa.

Para mover un personaje del punto (x_0, y_0) al punto (x_1, y_1) deberemos de realizar un traslado continuo de un punto a otro con n pasos intermedios y a su vez deberemos de ir reproduciendo la animación interna del personaje dependiendo siempre del tipo de movimiento que estemos llevando a cabo, como puede ser andar, saltar, correr, agacharse... La combinación de ambos tipo de animación, siempre que se haga adecuadamente, produce un efecto de movimiento en la pantalla.

15. Los Sprites y los Personajes

El control de este tipo de movimiento debe ser desarrollado en íntima colaboración con el desarrollo de niveles. Si estamos desarrollando un juego de plataformas deberemos de dotar al personaje de cierta gravedad a la hora de realizar saltos y cambios de nivel. Sin embargo si dotamos al juego de una perspectiva aérea tal vez tengamos que dotar al movimiento de otras fuerzas físicas diferentes a la anterior.

En un nivel de, por ejemplo, un juego de plataformas tendremos que definir que zonas de la pantalla serán tomadas como sólidas para que el personaje pueda saltar de una plataforma a otra o simplemente moverse entre ellas.

15.4.8. Implementando el control de la animación externa

La implementación de este control es muy dependiente del tipo de videojuego que vayamos a desarrollar. A la hora de utilizar un dispositivo de entrada, como puede ser un teclado o un joystick, para controlar el movimiento de un sprite en la pantalla tenemos dos alternativas en cuanto al conocer que acción quiere realizar el usuario.

La primera opción es trabajar con el subsistema de eventos. Los eventos son producidos cada vez que pulsamos una tecla, movemos el ratón o interactuamos con el joystick. Tienes un apartado dedicado a la gestión de este tipo de procesos que puedes repasar para afrontar la tarea de manejar tu personaje. Las ventajas que ofrece este tipo de gestión de acciones es que no tenemos que estar realizando una espera activa para saber que ocurre si no funcionar una vez disparado un evento. El problema más grave para el manejo de personajes en un videojuego es el tiempo de vida del evento. Este tiempo de vida no nos permite, habitualmente, tener una reacción correcta a los eventos producidos por el usuario cuando interacciona con el dispositivo de entrada. El proceso de estos eventos hace que el uso general de esta técnica no sea atractivo para controlar el personaje. Para teclas especiales tales como salir de la aplicación, volver a un menú, cambiar a pantalla completa y otras es adecuado utilizar este método ya que será una forma de dar preferencia a estas teclas frente a las de uso común de manejo del protagonista.

Para implementar la interacción del jugador con el usuario es más interesante un método que nos permita reaccionar a la acción actual del usuario independientemente de si hemos procesado toda la entrada o no. La segunda opción y la elegida es hacer un polling continuo al estado del teclado. Con esta técnica obtenemos un mapa instantáneo del estado del teclado con lo que podemos reaccionar a diferentes acciones. La clase que envuelverá este proceso nos permite consultar si una tecla está presionada en un

15.4. Animando un Sprite

determinado momento, que es algo más lógico que preguntar si se está produciendo un evento en dicho instante ya que los eventos están pensados para ser una acción que produzcan una reacción a partir ellos y no un proceso continuo.

A favor de esta segunda opción está la implementación que vamos a presentar de los diagrama de estados de las acciones del personaje principal, que cambiará de un estado a otro según se cumplan ciertas condiciones en el dispositivo de entrada. Los diagramas de estados no son algo trivial. Estudiaremos en este curso los conceptos asociados a estos diagramas sin entrar en profundidad, sólo lo justamente necesario para realizar una correcta implementación. Lo haremos una vez estudiadas las animaciones para dar respuesta a los eventos de teclado que han de actuar sobre la animación interna. Este es un tema muy interesante en el que sería bueno que profundizases para tu formación.

A diferencia de la animación interna no vamos a implementar una clase que nos controle el movimiento de cualquier personaje ya que por la naturaleza de las variables que necesitamos no se ajustan a la creación de una clase “Movimiento”.

Vamos a crear una pequeña clase personaje, con los elementos básicos de posición, que unidos a una clase que nos controle la entrada de teclado nos permita mover nuestro personaje a través de la superficie principal. El tipo de movimiento del personaje lo definiremos en la clase del mismo personaje ya que será en esta donde tendremos el comportamiento que deba tener el personaje.

La primera clase que vamos a presentar es la que controla el teclado. Es una clase bastante sencilla que hemos implementado de una manera muy simple. El fichero donde definimos la clase es el siguiente:

```
1 ;_____
2 ;// Listado: Teclado.h
3 ;//
4 ;// Control del dispositivo de entrada
5 ;
6 ;#ifndef _TECLADO_H_
7 ;#define _TECLADO_H_
8 ;
9 ;#include <SDL/SDL.h>
10 ;
11 ;
12 ;const int NUM_TECLAS = 9;
13 ;
14 ;
15 ;class Teclado {
16 ;
17 ;    public:
```

15. Los Sprites y los Personajes

```
18 ;    // Constructor
19 ;    Teclado();
20 ;
21 ;    // Teclas a usar en la aplicación
22 ;
23 ;    enum teclas_configuradas {
24 ;        TECLA_SALIR,
25 ;        TECLA_SUBIR,
26 ;        TECLA_BAJAR,
27 ;        TECLA_ACEPTAR,
28 ;        TECLA_DISPARAR,
29 ;        TECLA_IZQUIERDA,
30 ;        TECLA_DERECHA,
31 ;        TECLA_SALTAR,
32 ;        TECLA_GUARDAR
33 ;    };
34 ;
35 ;    // Consultoras
36 ;    void actualizar(void);
37 ;    bool pulso(enum teclas_configuradas tecla);
38 ;
39 ; private:
40 ;    Uint8* teclas;
41 ;    SDLKey teclas_configuradas[NUM_TECLAS];
42;};
43 ;
44 ;#endif
```

Como puedes observar definimos un enumerado que nos sirve para independizarnos de las constantes que usa SDL para utilizar el teclado. Hacen de interfaz entre nuestra clase y SDL. Los métodos que contiene son simples. El primero, *actualizar()*, es otra interfaz que nos libera de tener que llamar a la función que actualiza el estado del teclado.

El segundo, *pulso()*, comprueba si está pulsada una tecla que hemos pasado como parámetro. Si es así devuelve *true* y en caso contrario devuelve *false*. En cuanto a la parte privada de la clase tenemos dos variables. La primera, *teclas*, nos permite conocer el estado del teclado en un momento dado y la segunda *teclas_configuradas* nos permite conocer qué teclas son las que hemos configurado para el manejo del personaje principal.

Vamos a ver la implementación de la clase:

```
1 ;// Listado: Teclado.cpp
2 ;//
3 ;// Implementación de la clase teclado
4 ;
5 ;#include <iostream>
6 ;
7 ;#include "Teclado.h"
```

15.4. Animando un Sprite

```
8 ;
9 ;using namespace std;
10 ;
11 ;
12 ;Teclado::Teclado() {
13 ;
14 ;#ifdef DEBUG
15 ;    cout << "Teclado::Teclado()" << endl;
16 ;#endif
17 ;
18 ;
19 ;    // Configuramos la teclas que usaremos en la aplicación
20 ;
21 ;    teclas_configuradas[TECLA_SALIR] = SDLK_ESCAPE;
22 ;    teclas_configuradas[TECLA_SUBIR] = SDLK_UP;
23 ;    teclas_configuradas[TECLA_BAJAR] = SDLK_DOWN;
24 ;    teclas_configuradas[TECLA_ACEPTAR] = SDLK_RETURN;
25 ;    teclas_configuradas[TECLA_DISPARAR] = SDLK_SPACE;
26 ;    teclas_configuradas[TECLA_IZQUIERDA] = SDLK_LEFT;
27 ;    teclas_configuradas[TECLA_DERECHA] = SDLK_RIGHT;
28 ;    teclas_configuradas[TECLA_SALTAR] = SDLK_UP;
29 ;    teclas_configuradas[TECLA_GUARDAR] = SDLK_s;
30 ;
31 ;
32 ;
33 ;
34 ;void Teclado::actualizar(void) {
35 ;
36 ;    // Actualizamos el estado del teclado mediante mapeo
37 ;    teclas = SDL_GetKeyState(NULL);
38 ;
39 ;
40 ;
41 ;bool Teclado::pulso(enum teclas_configuradas tecla) {
42 ;
43 ;    if(teclas[teclas_configuradas[tecla]])
44 ;        return true;
45 ;    else
46 ;        return false;
47 ;}
```

Como puedes ver no hay nada especial en la implementación. Cuando necesites implementar algo hazlo lo más sencillo que puedas, y una vez conseguido, simplificalo.

En cuanto a la clase Personaje hemos creado una clase base que nos permita principalmente tener una imagen asociada al personaje y poder establecer y modificar la posición de dicha imagen en la pantalla. Veamos la definición de la clase:

```
1 ;// Listado: Personaje.h
```

15. Los Sprites y los Personajes

```
2 ;//  
3 ;// Clase Personaje  
4 ;  
5 ;#ifndef _PERSONAJE_H_  
6 ;#define _PERSONAJE_H_  
7 ;  
8 ;#include <SDL/SDL.h>  
9 ;  
10 ;  
11 ;class Personaje {  
12 ;  
13 ; public:  
14 ;  
15 ;     // Constructor  
16 ;  
17 ;     Personaje(char *ruta, int x = 0, int y = 0);  
18 ;  
19 ;  
20 ;     // Consultoras  
21 ;  
22 ;     int pos_x(void);  
23 ;     int pos_y(void);  
24 ;  
25 ;     void dibujar(SDL_Surface *pantalla);  
26 ;  
27 ;     // Modificadoras  
28 ;  
29 ;     void pos_x(int x);  
30 ;     void pos_y(int y);  
31 ;  
32 ;     // Modifica la posición del personaje con respecto al eje X  
33 ;  
34 ;     void avanzar_x(void);  
35 ;     void retrasar_x(void);  
36 ;  
37 ;     // Modifica la posición del personaje con respecto al eje Y  
38 ;  
39 ;     void bajar_y(void);  
40 ;     void subir_y(void);  
41 ;  
42 ;  
43 ;  
44 ; private:  
45 ;  
46 ;     // Posición  
47 ;     int x, y;  
48 ;  
49 ;     SDL_Surface *imagen;  
50 ;  
51 ;};  
52 ;  
53 ;#endif // _PERSONAJE_H_
```

15.4. Animando un Sprite

En la implementación no añadimos ninguna novedad. En el constructor de la clase inicializamos las variables de la misma cargando la imagen en la variable correspondiente y asignando un color clave para que no se muestre el fondo verde de la imagen. El método *dibujar()* se encarga de pintar en la superficie que le pasemos como parámetro el personaje en las coordenadas donde se encuentre en ese momento. Las demás funciones modifican la posición del personaje según sea necesario. Establecemos el intervalo de movimiento del personaje de cuatro en cuatro píxeles para obtener un movimiento acorde con la naturaleza del personaje. La implementación de la clase es la siguiente:

```
1 ;// Listado: Personaje.cpp
2 ;//
3 ;// Implementación de la clase Personaje
4 ;
5 ;#include <iostream>
6 ;#include <SDL/SDL_image.h>
7 ;
8 ;#include "Personaje.h"
9 ;
10 ;using namespace std;
11 ;
12 ;
13 ;// Constructor
14 ;
15 ;Personaje::Personaje(char *ruta, int x, int y) {
16 ;
17 ;    this->x = x;
18 ;    this->y = y;
19 ;
20 ;    // Cargamos la imagen
21 ;
22 ;    imagen = IMG_Load(ruta);
23 ;
24 ;    if(imagen == NULL) {
25 ;
26 ;        cerr << "Error: " << SDL_GetError() << endl;;
27 ;        exit(1);
28 ;
29 ;    }
30 ;
31 ;    // Calculamos el color transparente, en nuestro caso el verde
32 ;
33 ;    Uint32 colorkey = SDL_MapRGB(imagen->format, 0, 255, 0);
34 ;
35 ;    // Lo establecemos como color transparente
36 ;
37 ;    SDL_SetColorKey(imagen, SDL_SRCCOLORKEY, colorkey);
38 ;
39 ;}
40 ;
41 ;
```

15. Los Sprites y los Personajes

```
42 ;// Consultoras
43 ;
44 ;int Personaje::pos_x(void) {
45 ;
46 ;    return x;
47 ;
48 ;}
49 ;
50 ;int Personaje::pos_y(void) {
51 ;
52 ;    return y;
53 ;
54 ;}
55 ;
56 ;void Personaje::dibujar(SDL_Surface *pantalla) {
57 ;
58 ;    SDL_Rect rect;
59 ;
60 ;    rect.x = x;
61 ;    rect.y = y;
62 ;
63 ;    SDL_BlitSurface(imagen, NULL, pantalla, &rect);
64 ;
65 ;
66 ;}
67 ;
68 ;
69 ;// Modificadoras
70 ;
71 ;void Personaje::pos_x(int x) {
72 ;
73 ;    this->x = x;
74 ;
75 ;}
76 ;
77 ;
78 ;void Personaje::pos_y(int y) {
79 ;
80 ;    this->y = y;
81 ;
82 ;}
83 ;
84 ;// El movimiento de la imagen se establece
85 ;// de 4 en 4 pixeles
86 ;
87 ;void Personaje::avanzar_x(void) {
88 ;
89 ;    x += 4;
90 ;}
91 ;
92 ;void Personaje::retrasar_x(void) {
93 ;
94 ;    x -= 4;
```

```

95 ;}
96 ;
97 ;void Personaje::bajar_y(void) {
98 ;
99 ;    y += 4;
100 ;
101 ;}
102 ;
103 ;void Personaje::subir_y(void) {
104 ;
105 ;    y -= 4;
106 ;
107 ;}

```

15.4.9. Actualización Lógica

Ahora vamos va combinar el uso de ambas clases en un programa principal que nos permita mover el personaje por la pantalla. Primero, como es habitual inicializaremos SDL y definiremos un objeto para cada una de las clases a utilizar. En el *game loop* del juego consultaremos el estado del teclado para los movimientos del personaje y manejaremos los eventos referentes a las teclas especiales como la de salir del juego.

Cada vez que el jugador pulse una tecla se llevará a cabo lo que se conoce como **actualización lógica** del juego. Esto es, actualizaremos el valor de la posición sin mostrarlo en pantalla. Podemos pulsar varias teclas y que el personaje se desplace lógicamente hacia arriba a la derecha en dos posiciones. Estos valores serán almacenados en la clase personaje y por cada vuelta del bucle se hará una actualización en pantalla de la posición de personaje.

Esto nos permite diferenciar dos tareas que, aunque estén relacionadas, son totalmente independientes. ¿Por qué hacemos esto así y no dibujamos el personaje cada vez que se mueve? La respuesta es simple. Imagínate que tenemos veinte elementos en pantalla que debemos de redibujar en pantalla cada vez que se mueven. Ahora vamos a suponer que se mueven una vez por segundo. En un sólo segundo tendríamos que realizar veinte actualizaciones de pantalla lo que supone un desperdicio de recursos cuando podemos trabajar de forma más eficiente.

La alternativa escogida es la siguiente. En cada iteración del *game loop* se realiza como mucho una actualización de la pantalla. Vamos actualizando la posición lógica de todos los personajes, objetos... que se han movido o que se están animando y al final del bucle mostramos todas las modificaciones de una sola vez. Además añadimos una comprobación, y si no ha existido variaciones en los elementos que componen la escena no actualizamos la pantalla con lo que podemos ahorrarnos varios ciclos de actualización.

15. Los Sprites y los Personajes

En el siguiente programa de ejemplo podemos ver como aplicar esta técnica:

```
1 ;// Listados: main.cpp
2 ;//
3 ;// Programa de prueba
4 ;
5 ;#include <iostream>
6 ;#include <SDL/SDL.h>
7 ;
8 ;#include "Teclado.h"
9 ;#include "Personaje.h"
10 ;
11 ;using namespace std;
12 ;
13 ;int main() {
14 ;
15 ;    // Iniciamos el subsistema de video
16 ;
17 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
18 ;
19 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
20 ;        exit(1);
21 ;
22 ;    }
23 ;
24 ;
25 ;    atexit(SDL_Quit);
26 ;
27 ;    // Comprobamos que sea compatible el modo de video
28 ;
29 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
30 ;
31 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
32 ;        exit(1);
33 ;
34 ;    }
35 ;
36 ;
37 ;    // Establecemos el modo de video
38 ;
39 ;    SDL_Surface *pantalla;
40 ;
41 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
42 ;
43 ;    if(pantalla == NULL) {
44 ;
45 ;        cerr << "No se pudo establecer el modo de video: "
46 ;            << SDL_GetError() << endl;
47 ;
48 ;        exit(1);
49 ;    }
```

15.4. Animando un Sprite

```
50 ;
51 ;
52 ;    // Teclado para controlar al personaje
53 ;
54 ;    Teclado teclado;
55 ;
56 ;    // Cargamos un personaje
57 ;
58 ;    Personaje principal("./Imagenes/jacinto.bmp");
59 ;
60 ;
61 ;    // Lo mostramos por pantalla
62 ;
63 ;    principal.dibujar(pantalla);
64 ;
65 ;    SDL_Flip(pantalla);
66 ;
67 ;
68 ;    // Variables auxiliares
69 ;
70 ;    SDL_Event evento;
71 ;
72 ;    bool terminar = false;
73 ;
74 ;    int x0, y0;
75 ;
76 ;    // Game loop
77 ;
78 ;    while(terminar == false) {
79 ;
80 ;        // Actualizamos el estado del teclado
81 ;
82 ;        teclado.actualizar();
83 ;
84 ;        // Variables de control para saber si
85 ;        // tenemos que refrescar la pantalla o no
86 ;
87 ;        x0 = principal.pos_x();
88 ;        y0 = principal.pos_y();
89 ;
90 ;        // Actualización lógica de la posición
91 ;
92 ;        if(teclado.pulso(Teclado::TECLA_SUBIR)) {
93 ;
94 ;            principal.subir_y();
95 ;
96 ;        }
97 ;
98 ;        if(teclado.pulso(Teclado::TECLA_BAJAR)) {
99 ;
100 ;            principal.bajar_y();
101 ;
102 ;        }
```

15. Los Sprites y los Personajes

```
103 ;
104 ;     if(teclado.pulso(Teclado::TECLA_IZQUIERDA)) {
105 ;
106 ;         principal.retrasar_x();
107 ;
108 ;     }
109 ;
110 ;     if(teclado.pulso(Teclado::TECLA_DERECHA)) {
111 ;
112 ;         principal.avanzar_x();
113 ;
114 ;     }
115 ;
116 ;
117 ;     // Si existe modificación dibujamos
118 ;
119 ;     if(x0 != principal.pos_x() || y0 != principal.pos_y()) {
120 ;
121 ;         cout << "= Posición actual del personaje" << endl;
122 ;         cout << "- X: " << principal.pos_x() << endl;
123 ;         cout << "- Y: " << principal.pos_y() << endl;
124 ;
125 ;         // Dibujamos al personaje en su posición nueva
126 ;
127 ;         Uint32 negro = SDL_MapRGB(pantalla->format, 0, 0, 0);
128 ;
129 ;         SDL_FillRect(pantalla, NULL, negro);
130 ;
131 ;         principal.dibujar(pantalla);
132 ;
133 ;         SDL_Flip(pantalla);
134 ;     }
135 ;
136 ;
137 ;
138 ;     // Control de Eventos
139 ;
140 ;     while(SDL_PollEvent(&evento)) {
141 ;
142 ;
143 ;         if(evento.type == SDL_KEYDOWN) {
144 ;
145 ;             if(evento.key.keysym.sym == SDLK_ESCAPE)
146 ;                 terminar = true;
147 ;
148 ;         }
149 ;
150 ;         if(evento.type == SDL_QUIT)
151 ;             terminar = true;
152 ;
153 ;     }
154 ;
155 ; }
```

```

156 ;
157 ;    return 0;
158 ;
159 ;}
;
```

Vamos a centrarnos en el bucle que controla el videojuego. Lo primero que hacemos es comprobar si queremos terminar con la aplicación. En caso de que queramos salir tendremos un único punto de salida que nos permitirá una salida ordenada de la aplicación. Seguidamente actualizamos el estado teclado para que se reflejen en la variable que controla este aspecto las variaciones producidas. Lo siguiente que hacemos es tomar una instantánea de la posición del personaje para comprobar con posterioridad si ha variado la posición del mismo.

Ahora nos encontramos con una serie de estructuras selectivas que comprueban cada una de las teclas que tenemos configuradas para saber si existen variaciones. De ser así actualizaremos la posición del personaje mediante los métodos que se nos proporciona para ello. La última estructura selectiva que encontramos comprueba si el personaje ha cambiado de posición. De ser así muestra en consola un mensaje con la nueva posición, limpiamos de la pantalla la imagen anterior y mostramos el personaje en su nueva posición. Si no existen variaciones en la posición simplemente no se hace esta actualización.

Después de estas estructuras tenemos un bucle que realiza el polling de eventos por si se ha pedido salir de la aplicación hacerlo de manera ordenada.

15.4.10. Ejercicio 1

Vamos realizar un ejercicio de programación. Se trata de implementar una animación que vaya rebotando en los límites de la pantalla pero que nunca se salga de la misma. Utiliza la clase de personaje del ejemplo anterior. Aunque se llame personaje puede representar a cualquier imagen estática que tenga que desplazarse por la pantalla. ¡Todo tuyo!

Utilizando las clases del ejemplo anterior la solución es la siguiente:

```

;_____
1 ;// Listados: main.cpp
2 ;//
3 ;// Ejercicio 1
4 ;
5 ;#include <iostream>
6 ;#include <SDL/SDL.h>
7 ;
8 ;#include "Personaje.h"
9 ;
```

15. Los Sprites y los Personajes

```
10 ;using namespace std;
11 ;
12 ;int main() {
13 ;
14 ;    // Iniciamos el subsistema de video
15 ;
16 ;    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
17 ;
18 ;        cerr << "No se pudo iniciar SDL: " << SDL_GetError() << endl;
19 ;        exit(1);
20 ;
21 ;    }
22 ;
23 ;
24 ;    atexit(SDL_Quit);
25 ;
26 ;    // Comprobamos que sea compatible el modo de video
27 ;
28 ;    if(SDL_VideoModeOK(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF) == 0) {
29 ;
30 ;        cerr << "Modo no soportado: " << SDL_GetError() << endl;
31 ;        exit(1);
32 ;
33 ;    }
34 ;
35 ;
36 ;    // Establecemos el modo de video
37 ;
38 ;    SDL_Surface *pantalla;
39 ;
40 ;    pantalla = SDL_SetVideoMode(640, 480, 24, SDL_HWSURFACE|SDL_DOUBLEBUF);
41 ;
42 ;    if(pantalla == NULL) {
43 ;
44 ;        cerr << "No se pudo establecer el modo de video: "
45 ;            << SDL_GetError() << endl;
46 ;
47 ;        exit(1);
48 ;    }
49 ;
50 ;
51 ;    // Cargamos la bola
52 ;
53 ;    Personaje principal("./Imagenes/bola.bmp");
54 ;
55 ;
56 ;    // Lo mostramos por pantalla
57 ;
58 ;    principal.dibujar(pantalla);
59 ;
60 ;    SDL_Flip(pantalla);
61 ;
62 ;
```

15.4. Animando un Sprite

```
63 ; // Variables auxiliares
64 ;
65 ;     SDL_Event evento;
66 ;
67 ;     bool terminar = false;
68 ;
69 ; // Controlará la dirección con
70 ; // respecto al eje x o y
71 ;
72 ;     bool direccion_x = true;
73 ;     bool direccion_y = false;
74 ;
75 ;     int x0, y0;
76 ;
77 ;     cout << "Pulse ESC para terminar" << endl;
78 ;
79 ; // Game loop
80 ;
81 ;     while(terminar == false) {
82 ;
83 ;         // Variables de control para saber si
84 ;         // tenemos que refrescar la pantalla o no
85 ;
86 ;         x0 = principal.pos_x();
87 ;         y0 = principal.pos_y();
88 ;
89 ;
90 ;         if(direccion_x == true && principal.pos_x() >= 540)
91 ;             direccion_x = false;
92 ;
93 ;         if(direccion_y == true && principal.pos_y() >= 380)
94 ;             direccion_y= false;
95 ;
96 ;         if(direccion_x == false && principal.pos_x() <= 0)
97 ;             direccion_x = true;
98 ;
99 ;         if(direccion_y == false && principal.pos_y() <= 0)
100 ;             direccion_y = true;
101 ;
102 ;
103 ;         if(direccion_x == true)
104 ;             principal.avanzar_x();
105 ;         else
106 ;             principal.retrasar_x();
107 ;
108 ;
109 ;         if(direccion_y == true)
110 ;             principal.bajar_y();
111 ;         else
112 ;             principal.subir_y();
113 ;
114 ;
115 ;
```

15. Los Sprites y los Personajes

```
116 ;
117 ;
118 ;
119 ;      // Si existe modificación dibujamos
120 ;
121 ;      if(x0 != principal.pos_x() || y0 != principal.pos_y()) {
122 ;
123 ;#ifdef DEBUG
124 ;          cout << "= Posición actual del personaje" << endl;
125 ;          cout << "- X: " << principal.pos_x() << endl;
126 ;          cout << "- Y: " << principal.pos_y() << endl;
127 ;#endif
128 ;
129 ;      // Dibujamos al personaje en su posición nueva
130 ;
131 ;      Uint32 negro = SDL_MapRGB(pantalla->format, 0, 0, 0);
132 ;
133 ;      SDL_FillRect(pantalla, NULL, negro);
134 ;
135 ;      principal.dibujar(pantalla);
136 ;
137 ;      SDL_Flip(pantalla);
138 ;
139 ;
140 ;
141 ;
142 ;      // Control de Eventos
143 ;
144 ;      while(SDL_PollEvent(&evento)) {
145 ;
146 ;
147 ;          if(evento.type == SDL_KEYDOWN) {
148 ;
149 ;              if(evento.key.keysym.sym == SDLK_ESCAPE)
150 ;                  terminar = true;
151 ;
152 ;          }
153 ;
154 ;          if(evento.type == SDL_QUIT)
155 ;              terminar = true;
156 ;
157 ;      }
158 ;
159 ;  }
160 ;
161 ;  return 0;
162 ;
163 ;}
```

Hemos dibujado un círculo que utilizaremos de “pelota”. Con dicha imagen instanciamos la clase Personaje en una variable que vamos a llamar *principal*. Como ves es un ejercicio más de programación que de SDL. La lógica que sigue es la siguiente:

Tenemos dos variables lógicas que controlan la dirección de la pelota. Si están a *true* significa que el movimiento es hacia abajo y a la derecha. Tenemos unas estructuras selectivas que comprueban la posición y tipo de movimiento del elemento en un momento dado. Según sea este le cambia el valor a las variables para que se haga el movimiento contrario, es decir, rebote. Si por ejemplo la pelota tiene un movimiento hacia la derecha y resulta que ha llegado al límite de la pantalla pues se cambiará la variable que marca su movimiento al valor contrario del que tuviese para que vaya en la otra dirección. Esta comprobación se hace para los dos ejes por lo que se gestiona cualquier tipo de “choque” con los bordes de la superficie principal.

15.4.11. Integrando las animaciones

Ya hemos visto como podemos producir y controlar los tipos de animaciones interna y externa. Normalmente se integran estas dos técnicas con el fin de obtener unos mejores resultados consiguiendo un juego más atractivo.

Tenemos la necesidad de integrar ambos tipos de animaciones pero todavía no lo sabemos todo para acometer este trabajo. Hay cosas importantes que debemos de estudiar antes como los diagramas de estados de un personaje (autómatas), el cálculo de colisiones y debemos de profundizar en temas como la temporización para obtener un resultado cercano a lo que se puede desear de un videojuego.

Estos temas son suficientes por ellos mismos para crear un tutorial para cada uno de ellos. Nostros vamos a dar una amplia introducción a cada uno de ellos que nos permita acometer nuestras aplicaciones SDL con ciertas garantías. Te recomiendo que profundices en cada una de estas temáticas ya que serán un gran complemento en tu formación y te dotarán de una base más sólida para acometer tus desarrollos en el mundo de la programación de videojuegos.

De los aspectos más complejos de controlar de un videojuego es su temporización. De ella depende gran parte de la respuesta de la aplicación por lo que será fundamental que no dejes pasar detalle en su estudio de la misma una vez lleguemos al capítulo donde trataremos la creación del videojuego de ejemplo ya que en él realizaremos la integración de todo lo visto en el tutorial.

15.5. Creando una galería

15.5.1. Introducción

La mejor manera de controlar las distintas animaciones e imágenes que tiene un personaje es crear una galería. Esta galería formará parte de la clase personaje ya que será esta la que tenga que controlar de una u otra manera el estado del mismo.

El concepto de galería no necesita más explicación ya que su propia definición lo dice todo. Vamos a implementar una clase que almacene un conjunto de animaciones e imágenes que se asocien a un personaje para que tengas un ejemplo de como poder crear una galería para tu videojuego.

En el siguiente capítulo ampliaremos esta galería para que nos sea útil en nuestro videojuego de ejemplo.

15.5.2. Implementación de una galería

Como es habitual en el tutorial vamos a utilizar una implementación simple para desarrollar la galería. Vamos a usar aplicaciones de alto nivel y usar iteradores pero el objetivo es que captes la idea de la implementación independientemente de los detalles concretos del lenguaje o su nivel de abstracción.

Para codificar esta clase vamos a hacer uso de las clases que hemos creado a lo largo del capítulo para completar esta galería.

La definición de la clase es la siguiente:

```
1 ;// Listado: Galeria.h
2 ;
3 ;#ifndef _GALERIA_H_
4 ;#define _GALERIA_H_
5 ;
6 ;#include <map>
7 ;
8 ;// Declaración adelantada
9 ;
10 ;class Imagen;
11 ;class Fuente;
12 ;
13 ;
14 ;using namespace std;
15 ;
16 ;class Galeria {
17 ;
18 ; public:
```

15.5. Creando una galería

```
19 ;  
20 ;    // Tipos de imágenes contenidas en la galería  
21 ;  
22 ;    enum codigo_imagen {  
23 ;  
24 ;        //... Nos permite indexar las imágenes  
25 ;    };  
26 ;  
27 ;    // Fuentes almacenadas en la galería  
28 ;  
29 ;    enum codigo_fuente {  
30 ;  
31 ;        //... Nos permite indexar las fuentes  
32 ;    };  
33 ;  
34 ;  
35 ;    // Constructor  
36 ;  
37 ;    Galeria ();  
38 ;  
39 ;    // Consultoras  
40 ;  
41 ;    Imagen *imagen(codigo_imagen cod_ima);  
42 ;    Fuente *fuente(codigo_fuente indice);  
43 ;  
44 ;    ~Galeria();  
45 ;  
46 ;    // Conjunto de imágenes y de fuentes  
47 ;    // que vamos a utilizar en la aplicación  
48 ;  
49 ;    map<codigo_imagen, Imagen *> imagenes;  
50 ;    map<codigo_fuente, Fuente *> fuentes;  
51 ;};  
52 ;  
53 ;#endif
```

En la definición de la galería tenemos varios enumerados que nos van a permitir indexar los elementos de la galería de una manera sencilla y cómoda. La clase ofrece, además del constructor y el destructor, elementos consultores que extraen la información de la galería.

Esta es una definición básica para la galería ya que supone que ésta sea estática con unos elementos insertados desde inicio. Es muy sencillo añadir a esta clase un par de métodos que nos permitan añadir elementos de forma dinámica gracias a los contenedores que utilizamos de la STL. Adapta esta clase como creas conveniente.

La implementación de la clase es la siguiente:

```
1 ;// Listado: Galeria.cpp  
2 ;// Implementación de la clase galería del videojuego
```

15. Los Sprites y los Personajes

```
3 ;
4 ;#include <iostream>
5 ;
6 ;#include "Galeria.h"
7 ;#include "Imagen.h"
8 ;#include "Fuente.h"
9 ;
10 ;using namespace std;
11 ;
12 ;
13 ;Galeria::Galeria() {
14 ;
15 ;    // Cargamos las rejillas en la galería para las animaciones
16 ;    // y las imágenes fijas
17 ;
18 ;    imagenes[ /* código imagen */ ] = new Imagen( ... );
19 ;
20 ;    // Cargamos las fuentes en la galería
21 ;
22 ;    fuentes[ /* código fuente */ ] = new Fuente(...);
23 ;
24 ;}
25 ;
26 ;
27 ;Imagen *Galeria::imagen(codigo_imagen cod_ima) {
28 ;
29 ;    // Devolvemos la imagen solicitada
30 ;
31 ;    return imagenes[cod_ima];
32 ;}
33 ;
34 ;
35 ;Fuente *Galeria::fuente(codigo_fuente indice) {
36 ;
37 ;    // Devolvemos la fuente solicitada
38 ;
39 ;    return fuentes[indice];
40 ;}
41 ;
42 ;
43 ;
44 ;Galeria::~Galeria() {
45 ;
46 ;    // Descargamos la galería
47 ;
48 ;    delete imagenes[ /* código */ ];
49 ;    ...
50 ;
51 ;    delete fuentes[ /* código */ ];
52 ;    ...
53 ;
54 ;}
```

15.6. La animación interna y los autómatas

Como puedes ver la implementación de la clase es muy sencilla. Se basa en trabajar con las aplicaciones que contiene la información deseada e insertar y extraer la información de ellas.

15.6. La animación interna y los autómatas

15.6.1. Introducción

La teoría de autómatas es una materia muy interesante a la que podríamos dedicarle años de estudio. En esta sección vamos a presentar una serie de conceptos muy básicos que nos van a permitir diseñar los estados en los que queremos que tenga nuestro personaje así como qué debe ocurrir para que este pase de un estado a otro.

Los personajes de videojuegos suelen realizar unos movimientos determinados como pueden ser andar, saltar, golpear... Estos movimientos suelen obtenerse al pulsar una determinada tecla o realizar una acción bien definida. En algunos videojuegos la cantidad de movimientos que se pueden obtener es mayor combinando varias acciones con diferentes teclas.

El proceso de desarrollo se hace más complicado cuantas más acciones es capaz de hacer nuestro personaje. En esta sección vamos a estudiar como utilizar los autómatas finitos para representar el conjunto de movimientos que puede realizar el personaje. Este autómata nos será de ayuda en la codificación del mismo.

Vamos a comenzar con el concepto de estado:

15.6.2. Estados

Como hemos dicho un personaje, normalmente, puede realizar varios tipos de movimientos. Un **estado** representa el comportamiento de un personaje, objeto u otra entidad en un momento dado. Cada estado suele ser resultado de realizar o no una acción. Por ejemplo, si no pulsamos ninguna tecla ni ningún botón el personaje estará en estado de reposo o parado, pero será un estado. Cuando pulsamos la flecha a la derecha seguramente nuestro personaje andará hacia esa posición por lo que pasará al estado “andando”.

Un estado puede tener asociada una imagen fija o una animación (de las conocidas como internas). Cuando el personaje está parado seguramente tenemos una imagen fija que lo represente o bien una animación con un gesto que nos haga entender que el personaje está en reposo. Como regla general para los estados que representan movimientos se suelen usar animaciones y para los

15. Los Sprites y los Personajes

estados de reposo imágenes fijas aunque no es una norma que haya que cumplir.

Los estados se unen y asocian formando autómatas. La representación de un estado en el diagrama de transiciones es un círculo o circunferencia.

15.6.3. Autómata

Un autómata finito se suele representar mediante un diagrama llamado de estados o de transiciones. Este diagrama es un modelo lógico asociado a una serie de propiedades. Estas propiedades son un conjunto de estados, un alfabeto que nos permite pasar de un estado a otro mediante las relaciones de transición o cambio. Una transición no es más que un cambio de estado producido por una acción o letra del alfabeto. Un alfabeto no es mas que un conjunto de símbolos. Este conjunto de símbolos puede estar definido sobre el abecedario, las teclas de un ordenador... cualquier simbología es válida.

Formalmente existen varios tipos de autómatas: autómata finito determinista (AFD), los finito no deterministas(AFND) y los autómatas finitos no deterministas con transiciones ε . No vamos a entrar en profundidad sobre el estudio de los autómatas por lo que no vamos a detallar en qué consiste cada uno de ellos. Es suficiente con que conozcamos que existen diversos tipos de autómatas y que, como referencia, vamos a utilizar autómatas finitos no deterministas con transiciones ε con el fin de diseñar los distintos estados en los que puede estar nuestro personaje y los cambios o transiciones entre ellos.

Podemos interpretar un autómata como una máquina que se encuentra en un estado determinado y que reacciona a diferentes acciones cambiando (transicionando) de un estado a otro. Esta máquina nos dirá si una entrada es válida o no dependiendo si el estado final en que se encuentre es un estado de los conocidos como de terminación o validación. En nuestro caso no vamos a utilizar esta máquina para que nos compruebe si una entrada es válida si no para tener un control sobre el comportamiento de un personaje o un objeto dentro del videojuego para que realice sólo aquellas acciones que le sean permitidas en un determinado momento.

Vamos a crear nuestro primer **diagrama de transiciones**. Nuestro personaje va a tener varios estados: parado, salto, andar, golpear y agachar. En nuestro caso vamos a diferenciar los estados de cuando nuestro personaje esté mirando hacia la derecha de cuando lo esté haciendo a la izquierda. Crear un diagrama de estados no es complicado. Lo primero que debes hacer es dibujar tantos círculos como estados quieras controlar. Lo siguiente es estudiar cada uno de esos estados. Céntrate en un estado y piensa a que estados es lógico que llegue tomando como partida dicho estado. Una vez lo tengas claro

15.6. La animación interna y los autómatas

dibuja una flecha hacia ese estado y etiquétala con la acción que debe ocurrir para que el personaje cambie del estado inicial a este nuevo estado final. Este proceso lo repetiremos con cada uno de los estados que dibujamos inicialmente.

Al terminar marca con un ángulo el estado que quieras tomar como inicial y hazle otra circunferencia al estado que tomes como final. Con este proceso obtendrás un diagrama parecido al de la figura 15.4. Cuando quieras que se pueda pasar de un estado a otro sin que ocurra ninguna acción en la flecha que une ambos estados márcala con un ε lo que denotará una transición como la que quieras representar.

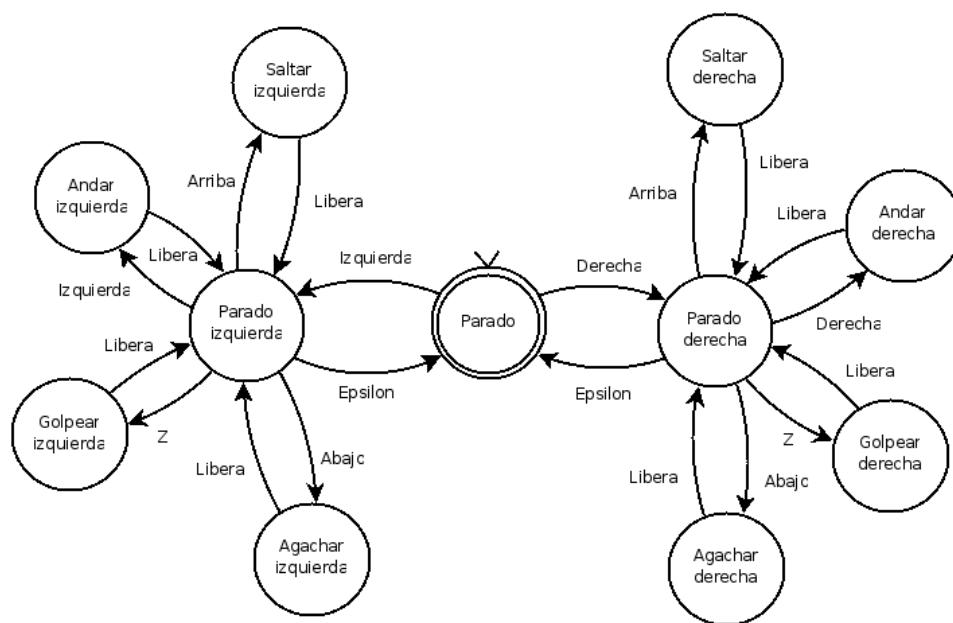


Figura 15.4: Ejemplo de autómata de nuestro personaje principal

De la figura 15.4 podemos interpretar las siguientes acciones:

1. El estado inicial y final, marcados con un doble círculo y una semiflecha, es el de parado.
2. Para situarnos en las acciones que se hacen mirando hacia la izquierda tenemos que pulsar la flecha de la izquierda mientras que para las acciones que se realizan mirando hacia la derecha tenemos que pulsar al menos una vez la tecla de la flecha a la derecha.
3. Una vez en los estados parados a la izquierda o la derecha están determinadas las acciones (o teclas a pulsar) que hay que realizar para llegar a cada uno de los estados conectados con el anterior. Por ejemplo si pulsamos la tecla Z estando parado mirando hacia la derecha el personaje

15. Los Sprites y los Personajes

golpeará en ese sentido y una vez soltemos dicha tecla volverá al estado parado en esa dirección.

Un diagrama de transiciones no es complicado de interpretar ni de diseñar siempre que utilicemos una definición relajada del mismo. Vamos a realizar la implementación de dicho autómata.

15.6.4. Implementando un autómata

Existen varios métodos para implementar un autómata. Nosotros vamos a optar por una implementación clara y sencilla del mismo, como hemos hecho en el resto del tutorial, ya que el objetivo es centrarnos conocer técnicas que nos permitan implementar este tipo estructuras. Los algoritmos que utilicemos serán muy optimizables. Esta tarea ya es un ejercicio de programación que está en tu mano mejorar.

La idea es la siguiente. Vamos a implementar un algoritmo que, primero, se repita un número indeterminado de veces, es decir, hasta que el usuario quiera salir de la aplicación. Segundo, que en cada iteración permita conocer qué acción estamos realizando, en nuestro caso, sobre el teclado para reaccionar según el diseño del autómata.

Para ofrecer una respuesta correcta tenemos que ampliar nuestra clase personaje añadiéndole las animaciones que vamos a utilizar en cada uno de los estados en los que se va a encontrar. Haremos uso para ello de las clases **Animacion**, **Imagen** y **Control_Animacion** presentadas en ejemplos anteriores. Como puedes ver los programas de ejemplo cada vez son más complejos pero estamos en el camino para crear un auténtico videojuego.

Una de las técnicas para implementar un autómata se conoce como el método de los *cases*. Este método se basa en utilizar una estructura selectiva del tipo *switch case* que tenga como discriminador el estado en el que nos encontramos. Si estamos en un determinado estado podremos realizar las acciones que se encuentre dentro de dicho case, como cambiar de estado u otra cosa. En nuestro caso si estamos en reposo estaremos en este caso del estado y podremos andar, agacharnos...

Vamos a ponernos manos a la obra. Vamos a estudiar la implementación del ejemplo que vamos a tratar. El primer fichero de código que vamos a estudiar es el de la propia implementación de lo que sería el autómata. Lo hemos incluido en el programa principal en la parte que diferenciamos como el bucle del juego. El código es el siguiente:

```
1 ;// Listados: main.cpp
```

15.6. La animación interna y los autómatas

```
2 ;//  
3 ;// Programa de prueba  
4 ;// Implementación INTUITIVA de autómata  
5 ;// Nos acerca a la implementación del autómata pero no es la  
6 ;// metodología más correcta  
7 ;  
8 ;#include <iostream>  
9 ;#include <SDL/SDL.h>  
10 ;  
11 ;#include "Teclado.h"  
12 ;#include "Personaje.h"  
13 ;#include "Miscelanea.h"  
14 ;  
15 ;using namespace std;  
16 ;  
17 ;int main() {  
18 ;  
19 ;    // Inicializamos SDL y Establecemos el modo de video  
20 ;  
21 ;    SDL_Surface *pantalla;  
22 ;  
23 ;    if(initializar	SDL(&pantalla, "Automata")) {  
24 ;  
25 ;        cerr << "No se puede inicializar SDL" << SDL_GetError() << endl;  
26 ;        exit(1);  
27 ;    }  
28 ;  
29 ;  
30 ;  
31 ;    // Teclado para controlar al personaje  
32 ;  
33 ;    Teclado teclado;  
34 ;  
35 ;    // Cargamos un personaje  
36 ;  
37 ;    Personaje principal;  
38 ;  
39 ;  
40 ;    // Variables auxiliares  
41 ;  
42 ;    SDL_Event evento;  
43 ;    SDL_Rect antiguo;  
44 ;  
45 ;    bool terminar = false;  
46 ;  
47 ;    int x0, y0;  
48 ;  
49 ;    estados_personaje s0;  
50 ;  
51 ;    Uint32 negro = SDL_MapRGB(pantalla->format, 0, 0, 0);  
52 ;  
53 ;    // Controlo la dirección: true = derecha & false = izquierda  
54 ;
```

15. Los Sprites y los Personajes

```
55 ;     bool direccion = true;
56 ;
57 ;     // Game loop
58 ;
59 ;     while(terminar == false) {
60 ;
61 ;         // Actualizamos el estado del teclado
62 ;
63 ;         teclado.actualizar();
64 ;
65 ;         // Variables de control para saber si
66 ;         // tenemos que refreshcar la pantalla o no
67 ;
68 ;         x0 = principal.pos_x();
69 ;         y0 = principal.pos_y();
70 ;         s0 = principal.estado_actual();
71 ;
72 ;         // Actualización lógica de la posición
73 ;         // y el estado
74 ;
75 ;         if(teclado.pulso(Teclado::TECLA_SUBIR)) {
76 ;
77 ;             if(direccion)
78 ;                 principal.cambio_estado(SALTAR_DERECHA);
79 ;             else
80 ;                 principal.cambio_estado(SALTAR_IZQUIERDA);
81 ;
82 ;         } else if(teclado.pulso(Teclado::TECLA_BAJAR)) {
83 ;
84 ;             if(direccion)
85 ;                 principal.cambio_estado(AGACHAR_DERECHA);
86 ;             else
87 ;                 principal.cambio_estado(AGACHAR_IZQUIERDA);
88 ;
89 ;         } else if(teclado.pulso(Teclado::TECLA_IZQUIERDA)) {
90 ;
91 ;             if(principal.pos_x() > -10) {
92 ;
93 ;                 principal.retrasar_x();
94 ;                 principal.cambio_estado(ANDAR_IZQUIERDA);
95 ;
96 ;             }
97 ;
98 ;             direccion = false;
99 ;
100 ;        } else if(teclado.pulso(Teclado::TECLA_DERECHA)) {
101 ;
102 ;            if(principal.pos_x() < 580) {
103 ;
104 ;                principal.avanzar_x();
105 ;                principal.cambio_estado(ANDAR_DERECHA);
106 ;
107 ;            }
108 ;        }
109 ;    }
110 ;}
```

15.6. La animación interna y los autómatas

```
108 ;  
109 ;         dirección = true;  
110 ;  
111 ;     } else if(teclado.pulso(Teclado::TECLA_DISPARAR)) {  
112 ;  
113 ;         if(dirección)  
114 ;             principal.cambio_estado(GOLPEAR_DERECHA);  
115 ;         else  
116 ;             principal.cambio_estado(GOLPEAR_IZQUIERDA);  
117 ;  
118 ;     } else {  
119 ;  
120 ;         if(dirección)  
121 ;             principal.cambio_estado(PARADO_DERECHA);  
122 ;         else  
123 ;             principal.cambio_estado(PARADO_IZQUIERDA);  
124 ;     }  
125 ;  
126 ;  
127 ;     // Si existe modificación dibujamos  
128 ;  
129 ;     if(x0 != principal.pos_x() || y0 != principal.pos_y()  
130 ;        || s0 != principal.estado_actual() ) {  
131 ;  
132 ;#ifdef DEBUG  
133 ;         cout << "= Posición actual del personaje" << endl;  
134 ;         cout << "- X: " << principal.pos_x() << endl;  
135 ;         cout << "- Y: " << principal.pos_y() << endl;  
136 ;#endif  
137 ;  
138 ;     // Dibujamos al personaje en su posición nueva  
139 ;     // Borramos la antigua  
140 ;  
141 ;     antiguo.x = x0;  
142 ;     antiguo.y = y0;  
143 ;  
144 ;  
145 ;     SDL_FillRect(pantalla, NULL, negro);  
146 ;  
147 ;     principal.dibujar(pantalla);  
148 ;  
149 ;     SDL_Flip(pantalla);  
150 ; }  
151 ;  
152 ;     // Control de Eventos  
153 ;  
154 ;     while(SDL_PollEvent(&evento)) {  
155 ;  
156 ;  
157 ;         if(evento.type == SDL_KEYDOWN) {  
158 ;  
159 ;             if(evento.key.keysym.sym == SDLK_ESCAPE)  
160 ;                 terminar = true;
```

15. Los Sprites y los Personajes

```
161 ;
162 ;        }
163 ;
164 ;    if(evento.type == SDL_QUIT)
165 ;        terminar = true;
166 ;
167 ;    }
168 ;
169 ;}
170 ;
171 ;    return 0;
172 ;
173 ;}
```

Como puedes ver para inicializar SDL y establecer el modo de video hemos implementado una función que nos libera de realizar esta tarea. En este tutorial no hemos sido partidarios de utilizar esta función para todo ya que el objetivo de este tutorial es manejar con cierta soltura SDL y si utilizamos una función que nos libera de realizar tareas fundamentales no practicamos con dichas funciones básicas. En este caso, como la carga de código es bastante mayor, hemos decidido hacer uso de ella. Como en todos los ejemplos tienes disponible el código de dicha función.

Lo segundo que hacemos en la aplicación es crear una instancia de Teclado y de Personaje ya que son los dos elementos que vamos a utilizar para simular el autómata. Veremos las modificaciones que hemos tenido que hacer en la clase Personaje utilizada hasta ahora para que acepte animaciones ya que en los demás ejemplos en los que hemos usado esta clase la hemos inicializado con imágenes estáticas.

Antes de entrar en el bucle o *game loop* del juego declaramos un número considerable de variables auxiliares que vamos a tener que usar con diferentes propósitos. Te preguntarás, ¿cómo controlamos la temporización?

Lo más lógico es procesar la temporización en la clase Personaje donde se realice la animación del mismo. Esto nos permite personalizar la velocidad de cada personaje. Cuando desarrollemos un videojuego tendremos una clase Participante de la que heredarán varias clases hijas definiendo así los personajes principales, los adversarios y otros elementos. La temporización no podrá ser controlada de manera global a todas las instancias de las clases ya que esto limitaría la configuración del comportamiento. Necesitamos poder definir comportamientos diferentes para cada uno de los integrantes de la aplicación.

Para no complicar más el ejercicio hemos decidido utilizar un control de la temporización de manera local al personaje definiendo su comportamiento en el constructor para cada una de las acciones que realiza. En este ejemplo

15.6. La animación interna y los autómatas

vamos a centrarnos en la implementación “informal” del autómata pero es interesante ver como hemos controlado el tiempo. Para controlar el tiempo hemos modificado el método *paso_a_paso()* perteneciente a la clase **Animación**. La nueva implementación es la siguiente:

```
1 ;// Esta función lleva el control del tiempo
2 ;
3 ;void Animacion::paso_a_paso(SDL_Surface *pantalla, int x, int y, int flip) {
4 ;
5 ;    Uint32 t0 = SDL_GetTicks();
6 ;    Uint32 t1 = SDL_GetTicks();
7 ;
8 ;    if(control_animacion->numero_cuadros() < 2) {
9 ;
10 ;        imagen->dibujar(pantalla, control_animacion->cuadro(), x, y, flip);
11 ;        control_animacion->avanzar();
12 ;
13 ;    } else {
14 ;
15 ;        do {
16 ;
17 ;            t1 = SDL_GetTicks();
18 ;
19 ;            } while((t1 - t0) < retardo_);
20 ;
21 ;        imagen->dibujar(pantalla, control_animacion->cuadro(), x, y, flip);
22 ;        control_animacion->avanzar();
23 ;
24 ;    }
25 ;}
```

El código es bastante simple. Si tenemos más de un cuadro es que estamos en ante una animación y no una imagen fija. En este caso mediante un *do while* dejamos pasar el tiempo hasta que pase la cantidad de tiempo que hemos definido como retardo. Una vez haya cumplido este tiempo realizamos las acciones habituales de avanzar la posición del personaje y dibujar el frame actual.

Seguidamente actualizaremos el estado del teclado y tomaremos una “foto” del momento actual del personaje, tanto de su posición actual como del estado en el que se encuentra. Esto nos permitirá determinar si tenemos que realizar un repintado del personaje en pantalla. Siempre que el personaje no varíe de estado o de posición no se realizará dicho repintado.

Después de la inicialización de estas variables tenemos la lógica que pertenece a la implementación del autómata. Hemos usado una estructura selectiva diferente al *switch - case* pero igualmente válida. La lógica es la siguiente se estudia cada uno de las acciones que puede producir el usuario sobre el teclado y se reacciona según sea la acción realizada. Por ejemplo, si

15. Los Sprites y los Personajes

pulsamos la tecla definida para andar hacia la derecha precesaremos dicha orden moviendo el personaje hacia la derecha cambiando su estado y su posición.

Esta es una manera intuitiva de implementar el autómatana pero **no es la más correcta**. El método de los *cases* se basa en el estudio del estado actual del personaje y no en el de las acciones que produzcan transiciones ya que necesitamos saber en qué estado estamos para reaccionar de una manera o de otra. Vamos a abordar la implementación correcta para el autómata.

Entonces, ¿cómo se construye el método de los cases? Lo primero que tenemos que hacer es construir, con una estructura selectiva, una esquema que nos permita tratar todos los posibles estados en los que pueda estar nuestro personaje. En nuestro caso se trata de una estructura con la siguiente forma:

```
;  
1 ;switch(estado) {  
2 ;  
3 ;    case PARADO:  
4 ;        break;  
5 ;  
6 ;    case PARADO_DERECHA:  
7 ;        break;  
8 ;  
9 ;    case PARADO_IZQUIERDA:  
10 ;        break;  
11 ;  
12 ;    case SALTAR_DERECHA:  
13 ;        break;  
14 ;  
15 ;    ...  
16 ;  
17 ;    case AGACHAR_IZQUIERDA:  
18 ;        break;  
19 ;  
20 ;    default:  
21 ;  
22 ;}
```

Una vez tengamos codificada esta estructura tenemos que añadirle comportamiento. ¿Cómo hacemos esto? En este momento deberíamos de hacer una tabla con todos los posibles estados y las transiciones entre ellos para estudiar los posibles casos de aceptación del autómata. Como en este caso el autómata no es una máquina verificadora si no que es una herramienta para implementar el comportamiento de un personaje vamos a realizar el estudio caso por caso fijándonos en el diagrama de transiciones que expusimos al principio de la sección.

Vamos a empezar estudiando el caso *Parado*. Cuando nos encontramos en este estado pueden ocurrir dos acciones válidas: que pulsemos la tecla

15.6. La animación interna y los autómatas

definida para realizar un movimiento a la derecha o que pulsemos la tecla con el mismo objetivo pero hacia la izquierda. Si pulsamos la tecla definida para el movimiento a la derecha el personaje cambiará de estado a *Parado derecha* y si pulsamos la tecla de movimiento hacia la izquierda pasará al estado *Parado izquierda*. Si no pulsamos ninguna tecla se quedará como estaba. Fácil ¿no?. Ahora tenemos que incluir este estudio en nuestra estructura selectiva. Nos vamos al caso donde estudiamos este estado e incluimos la lógica que nos permite simular el comportamiento que acabamos de describir. El resultado es el siguiente:

```
;  
1 ; case PARADO:  
2 ;  
3 ;     if(teclado.pulso(Teclado::TECLA_DERECHA)  
4 ;  
5 ;         // Si "->" cambio de estado a parado derecha  
6 ;  
7 ;         principal.cambio_estado(PARADO_DERECHA);  
8 ;  
9 ;     else if(teclado.pulso(Teclado::TECLA_IZQUIERDA))  
10 ;  
11 ;         // Si "<-" cambio de estado a parado izquierda  
12 ;  
13 ;         principal.cambio_estado(PARADO_IZQUIERDA);  
14 ;  
15 ;  
16 ;break;  
;
```

Como puedes ver no es algo complicado. De manera análoga completamos los demás casos. El resultado del estudio del diagrama de transiciones es el siguiente:

```
;  
1 ;switch(principal.estado_actual()) {  
2 ;  
3 ; case PARADO:  
4 ;  
5 ;     if(teclado.pulso(Teclado::TECLA_DERECHA)  
6 ;  
7 ;         // Si "->" cambio de estado a parado derecha  
8 ;  
9 ;         principal.cambio_estado(PARADO_DERECHA);  
10 ;  
11 ;     else if(teclado.pulso(Teclado::TECLA_IZQUIERDA))  
12 ;  
13 ;         // Si "<-" cambio de estado a parado izquierda  
14 ;  
15 ;         principal.cambio_estado(PARADO_IZQUIERDA);  
16 ;  
17 ;     break;  
18 ;
```

15. Los Sprites y los Personajes

```
19 ; case PARADO_DERECHA:  
20 ;  
21 ;     if(teclado.pulso(Teclado::TECLA_SUBIR))  
22 ;  
23 ;         principal.cambio_estado(SALTAR_DERECHA);  
24 ;  
25 ;     else if(teclado.pulso(Teclado::TECLA_DERECHA)) {  
26 ;  
27 ;         principal.avanzar_x();  
28 ;         principal.cambio_estado(ANDAR_DERECHA);  
29 ;  
30 ;     } else if(teclado.pulso(Teclado::TECLA_DISPARAR))  
31 ;  
32 ;         principal.cambio_estado(GOLPEAR_DERECHA);  
33 ;  
34 ;     else if(teclado.pulso(Teclado::TECLA_BAJAR))  
35 ;  
36 ;         principal.cambio_estado(AGACHAR_DERECHA);  
37 ;  
38 ;     else  
39 ;  
40 ;         principal.cambio_estado(PARADO);  
41 ;  
42 ;  
43 ;     break;  
44 ;  
45 ; case PARADO_IZQUIERDA:  
46 ;  
47 ;     if(teclado.pulso(Teclado::TECLA_SUBIR))  
48 ;  
49 ;         principal.cambio_estado(SALTAR_IZQUIERDA);  
50 ;  
51 ;     else if(teclado.pulso(Teclado::TECLA_IZQUIERDA)) {  
52 ;  
53 ;         principal.retroceder_x();  
54 ;         principal.cambio_estado(ANDAR_IZQUIERDA);  
55 ;  
56 ;     } else if(teclado.pulso(Teclado::TECLA_DISPARAR))  
57 ;  
58 ;         principal.cambio_estado(GOLPEAR_IZQUIERDA);  
59 ;  
60 ;     else if(teclado.pulso(Teclado::TECLA_BAJAR))  
61 ;  
62 ;         principal.cambio_estado(AGACHAR_IZQUIERDA);  
63 ;  
64 ;     else  
65 ;  
66 ;         principal.cambio_estado(PARADO);  
67 ;  
68 ;  
69 ;     break;  
70 ;  
71 ;
```

15.6. La animación interna y los autómatas

```
72 ; case SALTAR_DERECHA:  
73 ;     principal.cambio_estado(PARADO);  
75 ;  
76 ;     break;  
77 ;  
78 ; case SALTAR_IZQUIERDA:  
79 ;  
80 ;     principal.cambio_estado(PARADO);  
81 ;  
82 ;     break;  
83 ;  
84 ; case ANDAR_DERECHA:  
85 ;  
86 ;     principal.cambio_estado(PARADO);  
87 ;  
88 ;     break;  
89 ;  
90 ; case ANDAR_IZQUIERDA:  
91 ;  
92 ;     principal.cambio_estado(PARADO);  
93 ;  
94 ;     break;  
95 ;  
96 ; case GOLPEAR_DERECHA:  
97 ;  
98 ;     principal.cambio_estado(PARADO);  
99 ;  
100 ;    break;  
101 ;  
102 ;  
103 ; case GOLPEAR_IZQUIERDA:  
104 ;  
105 ;     principal.cambio_estado(PARADO);  
106 ;  
107 ;     break;  
108 ;  
109 ; case AGACHAR_DERECHA:  
110 ;  
111 ;     principal.cambio_estado(PARADO);  
112 ;  
113 ;     break;  
114 ;  
115 ; case AGACHAR_IZQUIERDA:  
116 ;  
117 ;     principal.cambio_estado(PARADO);  
118 ;  
119 ;     break;  
120 ;  
121 ; default:  
122 ;  
123 ;     cerr << "No se conoce este estado" << endl;  
124 ;
```

15. Los Sprites y los Personajes

```
125 ;  
126 ; }
```

Como puedes ver la estructura toma una dimensión importante. No hemos separado en funciones los comportamientos a realizar en cada uno de los estados para que se viese las analogías y diferencias entre ellos. Esta es una manera más correcta de implementar el autómata pero a la hora de programar videojuegos usaremos aquella que nos sea más útil en un momento dado. El que las transiciones del autómata sean controladas por eventos de teclado produce que en ocasiones necesitemos discriminar el comportamiento por la entrada que recibimos de usuario más que por el estado en el que se encuentre el personaje en un momento dado.

Puedes ver que al implementar el autómata más formalmente hemos hecho desaparecer apaños que necesitábamos en la anterior implementación, como el de control hacia que lado estaba mirando el personaje en un momento dado o el de iniciar la aplicación en vez de en el estado parado en el que posicionaba a nuestro personaje mirando hacia la izquierda.

Para poder implementar el comportamiento del autómata hemos tenido que completar la clase personaje con varios métodos que nos permiten controlar el estado del personaje así como asociar animaciones a cada uno de estos estados. La implementación realizada es muy concreta para este caso en particular. A la hora de desarrollar nuestro videojuego de ejemplo realizaremos una implementación más general que nos permita reutilizar la clase sea cual sea el personaje que queramos crear. Vamos a ver las novedades de la implementación de esta clase:

```
;_____  
1 ;// Listado: Personaje.h  
2 ;//  
3 ;// Clase Personaje  
4 ;  
5 ;#ifndef _PERSONAJE_H_  
6 ;#define _PERSONAJE_H_  
7 ;  
8 ;#include <SDL/SDL.h>  
9 ;  
10 ;class Animacion; // Declaración adelantada  
11 ;  
12 ;// Enumerado con los posibles estados del personaje  
13 ;  
14 ;enum estados_personaje {  
15 ;  
16 ;    PARADO,  
17 ;    PARADO_DERECHA,  
18 ;    SALTAR_DERECHA,  
19 ;    ANDAR_DERECHA,  
20 ;    GOLPEAR_DERECHA,
```

15.6. La animación interna y los autómatas

```
21 ;    AGACHAR_DERECHA,
22 ;    PARADO_IZQUIERDA,
23 ;    SALTAR_IZQUIERDA,
24 ;    ANDAR_IZQUIERDA,
25 ;    GOLPEAR_IZQUIERDA,
26 ;    AGACHAR_IZQUIERDA,
27 ;    MAX_ESTADOS
28 ;
29 ;};
30 ;
31 ;class Personaje {
32 ;
33 ;    public:
34 ;
35 ;        // Constructor
36 ;
37 ;        Personaje(void);
38 ;
39 ;
40 ;        // Consultoras
41 ;
42 ;        int pos_x(void);
43 ;        int pos_y(void);
44 ;
45 ;        estados_personaje estado_actual(void);
46 ;
47 ;        void dibujar(SDL_Surface *pantalla);
48 ;
49 ;        // Modificadoras
50 ;
51 ;        void cambio_estado(estados_personaje status);
52 ;
53 ;
54 ;        void pos_x(int x);
55 ;        void pos_y(int y);
56 ;
57 ;        // Modifica la posición del personaje con respecto al eje X
58 ;
59 ;        void avanzar_x(void);
60 ;        void retrasar_x(void);
61 ;
62 ;        // Modifica la posición del personaje con respecto al eje Y
63 ;
64 ;        void bajar_y(void);
65 ;        void subir_y(void);
66 ;
67 ;
68 ;
69 ;    private:
70 ;
71 ;        // Posición
72 ;
73 ;        int x, y;
```

15. Los Sprites y los Personajes

```
74 ;
75 ;     estados_personaje estado;
76 ;
77 ;     // Galería de animaciones
78 ;
79 ;     Animacion *galeria_animaciones[MAX_ESTADOS];
80 ;
81 ;
82 ;};
83 ;
84 ;#endif
;
```

Lo primero que hemos incluido en la definición de la clase es un conjunto de estados del personaje. Asociados a estos estados tenemos dos funciones miembro. Una nos permite cambiar el estado del personaje mientras que la otra nos devuelve el estado actual del mismo.

En la parte privada de la clase hemos incluido una variable que controlará el estado actual del personaje y una galería de animaciones. Esta galería de animaciones tendrá el tamaño del número de estados disponible y asociará a cada estado una imagen o animación que reproducir cuando el personaje esté en dicho estado.

Veamos la nueva implementación de la clase:

```
;_____
1 ;// Listado: Personaje.cpp
2 ;//
3 ;// Implementación de la clase Personaje
4 ;
5 ;#include <iostream>
6 ;#include <SDL/SDL_image.h>
7 ;
8 ;#include "Personaje.h"
9 ;#include "Animacion.h"
10 ;
11 ;using namespace std;
12 ;
13 ;
14 ;// Constructor
15 ;
16 ;Personaje::Personaje(void) {
17 ;
18 ;    // Inicializamos las variables
19 ;
20 ;    this->x = 0;
21 ;    this->y = 380;
22 ;
23 ;    estado = PARADO_DERECHA;
24 ;
25 ;    galeria_animaciones[PARADO_DERECHA] =
```

15.6. La animación interna y los autómatas

```
26 ;         new Animacion("./Imagenes/jacinto.bmp", 4, 7, "0", 0);
27 ;
28 ;     galeria_animaciones[SALTAR_DERECHA] =
29 ;         new Animacion("./Imagenes/jacinto.bmp", 4, 7, "21", 0);
30 ;
31 ;     galeria_animaciones[ANDAR_DERECHA] =
32 ;         new Animacion("./Imagenes/jacinto.bmp", 4, 7,\n
33 ;                         "1,2,3,2,1,0,4,5,6,5,4,0", 18);
34 ;
35 ;     galeria_animaciones[GOLPEAR_DERECHA] =
36 ;         new Animacion("./Imagenes/jacinto.bmp", 4, 7, "15", 0);
37 ;
38 ;     galeria_animaciones[AGACHAR_DERECHA] =
39 ;         new Animacion("./Imagenes/jacinto.bmp", 4, 7, "17", 2);
40 ;
41 ;     galeria_animaciones[PARADO_IZQUIERDA] =
42 ;         new Animacion("./Imagenes/jacinto.bmp", 4, 7, "7", 0);
43 ;
44 ;     galeria_animaciones[SALTAR_IZQUIERDA] =
45 ;         new Animacion("./Imagenes/jacinto.bmp", 4, 7, "20", 0);
46 ;
47 ;     galeria_animaciones[ANDAR_IZQUIERDA] =
48 ;         new Animacion("./Imagenes/jacinto.bmp", 4, 7, "8,9,10,9,8,7,11,12,13,12,11,7", 18);
49 ;
50 ;     galeria_animaciones[GOLPEAR_IZQUIERDA] =
51 ;         new Animacion("./Imagenes/jacinto.bmp", 4, 7, "14", 0);
52 ;
53 ;     galeria_animaciones[AGACHAR_IZQUIERDA] =
54 ;         new Animacion("./Imagenes/jacinto.bmp", 4, 7, "16", 2);
55 ;
56 ;
57 ;}
58 ;
59 ;
60 ;// Consultoras
61 ;
62 ;int Personaje::pos_x(void) {
63 ;
64 ;    return x;
65 ;
66 ;}
67 ;
68 ;int Personaje::pos_y(void) {
69 ;
70 ;    return y;
71 ;
72 ;}
73 ;
74 ;estados_personaje Personaje::estado_actual(void) {
75 ;
76 ;    return estado;
77 ;
78 ;}
```

15. Los Sprites y los Personajes

```
79 ;
80 ;void Personaje::dibujar(SDL_Surface *pantalla) {
81 ;
82 ;    // Según sea el estado del personaje
83 ;    // en un momento determinado
84 ;    // dibujaremos una animación u otra
85 ;
86 ;    switch(estado) {
87 ;
88 ;        case PARADO:
89 ;            estado = PARADO_DERECHA;
90 ;
91 ;        case PARADO_DERECHA:
92 ;            galeria_animaciones[PARADO_DERECHA]->paso_a_paso(pantalla, x, y);
93 ;            break;
94 ;
95 ;        case SALTAR_DERECHA:
96 ;            galeria_animaciones[SALTAR_DERECHA]->paso_a_paso(pantalla, x, y);
97 ;            break;
98 ;
99 ;        case ANDAR_DERECHA:
100 ;            galeria_animaciones[ANDAR_DERECHA]->paso_a_paso(pantalla, x, y);
101 ;            break;
102 ;
103 ;        case GOLPEAR_DERECHA:
104 ;            galeria_animaciones[GOLPEAR_DERECHA]->paso_a_paso(pantalla, x, y);
105 ;            break;
106 ;
107 ;        case AGACHAR_DERECHA:
108 ;            galeria_animaciones[AGACHAR_DERECHA]->paso_a_paso(pantalla, x, y);
109 ;            break;
110 ;
111 ;        case PARADO_IZQUIERDA:
112 ;            galeria_animaciones[PARADO_IZQUIERDA]->paso_a_paso(pantalla, x, y);
113 ;            break;
114 ;
115 ;        case SALTAR_IZQUIERDA:
116 ;            galeria_animaciones[SALTAR_IZQUIERDA]->paso_a_paso(pantalla, x, y);
117 ;            break;
118 ;
119 ;        case ANDAR_IZQUIERDA:
120 ;            galeria_animaciones[ANDAR_IZQUIERDA]->paso_a_paso(pantalla, x, y);
121 ;            break;
122 ;
123 ;        case GOLPEAR_IZQUIERDA:
124 ;            galeria_animaciones[GOLPEAR_IZQUIERDA]->paso_a_paso(pantalla, x, y);
125 ;            break;
126 ;
127 ;        case AGACHAR_IZQUIERDA:
128 ;            galeria_animaciones[AGACHAR_IZQUIERDA]->paso_a_paso(pantalla, x, y);
129 ;            break;
130 ;
131 ;    default:
```

15.6. La animación interna y los autómatas

```
132 ;         cerr << "Personaje::dibujar() " << endl;
133 ;
134 ;     } // end switch(estado)
135 ;
136 ;
137 ;     SDL_Rect rect;
138 ;
139 ;     rect.x = x;
140 ;     rect.y = y;
141 ;
142 ;     SDL_BlitSurface(NULL, NULL, pantalla, &rect);
143 ;
144 ;
145 ;}
146 ;
147 ;
148 ;// Modificadoras
149 ;
150 ;void Personaje::cambio_estado(estados_personaje status) {
151 ;
152 ;    estado = status;
153 ;
154 ;}
155 ;
156 ;void Personaje::pos_x(int x) {
157 ;
158 ;    this->x = x;
159 ;
160 ;}
161 ;
162 ;
163 ;void Personaje::pos_y(int y) {
164 ;
165 ;    this->y = y;
166 ;
167 ;}
168 ;
169 ;// El movimiento de la imagen se establece
170 ;// de 4 en 4 píxeles
171 ;
172 ;void Personaje::avanzar_x(void) {
173 ;
174 ;    x += 6;
175 ;}
176 ;
177 ;void Personaje::retrasar_x(void) {
178 ;
179 ;    x -= 6;
180 ;}
181 ;
182 ;void Personaje::bajar_y(void) {
183 ;
184 ;    y += 4;
```

15. Los Sprites y los Personajes

```
185 ;
186 ;
187 ;
188 ;void Personaje::subir_y(void) {
189 ;
190 ;    y -= 4;
191 ;
192 ;}
```

En el constructor de la clase hemos añadido la inicialización de las nuevas variables que contiene la clase. En la galería de animaciones hemos asociado las animaciones a los estados haciendo uso de la clase `Animacion` para manejarlas. En cuanto a las demás funciones hay pocas diferencias con respecto a la clase personaje antes implementada. La principal es la nueva función dibujar que muestra el frame o cuadro perteneciente a la animación asociada al estado actual del personaje.

Como puedes ver en el programa principal hemos hecho uso de una pequeña librería que hemos creado para la inicialización de SDL. No es más que la agrupación funcional de las tareas que debemos realizar al inicio con SDL por lo que puedes consultarla en los listados.

15.6.5. Conclusión

La mayor parte del trabajo dedicado a construir un personaje consiste en diseñar un autómata que responda correctamente a los comportamientos que queramos que tenga nuestro personaje. La implementación con o sin SDL es bastante sencilla y fácil de modificar. Es importante que le dediques una buena parte del tiempo al diseño lo que te ahorrará tener que modificar codificación cuando estés inmerso en el desarrollo del proyecto.

15.7. Colisiones

15.7.1. Introducción

¿Qué sería de un videojuego dónde el personaje principal no pudiese realizar más acciones que su propio movimiento? La respuesta es rápida: Nada. La interacción de un personaje con los elementos que componen la escena de un videojuego es fundamental. Cuando un personaje se encuentra con otro debemos de tener bien definidas las acciones que queramos que ocurran en respuesta a dicha “colisión”.

Por ejemplo si nuestro personaje se encuentra con un malvado villano y este nos alcanza lo más lógico es que hagamos descender nuestro nivel de vida.

Si pasamos sobre un objeto tendremos que ser capaces de cogerlo o realizar alguna acción con él. De esto tratan las colisiones.

15.7.2. ¿Qué es una colisión? Detectando colisiones

Podemos definir una colisión entre dos elementos como un choque entre estos dos elementos. En el mundo del videojuego el significado de esta palabra es el mismo añadiéndole algunos matices. Se produce una colisión cuando dos personajes u objetos del videojuego chocan entre ellas.



Figura 15.5: Ejemplo de colisión.

Dos personajes chocan entre ellos lógicamente, lo que se produce en realidad en SDL, a un nivel físico, es el solapamiento de superficies. En SDL, como recordarás, las superficies son unos elementos rectangulares donde tenemos una determinada información gráfica. Estas superficies tienen una posición determinada por su esquina superior derecha, una altura y una anchura determinada.

Dos superficies habrán colisionado cuando algún píxel de alguna de ellas esté contenido dentro de un píxel de la otra superficie. No es un concepto complicado pero conlleva una serie de problemas. El primero es que la gran mayoría de personajes y elementos no son rectangulares. Cuando dibujamos a los integrantes de nuestra aplicación lo haremos sobre un lienzo de un color que habremos escogido como *color clave* para que no se reproduzca durante el blitting. El tamaño de este lienzo, que también es rectangular, es mayor que el propio elemento. Al cargar esta imagen en pantalla cargamos el rectángulo completo. Esto provoca que se detecte la colisión antes de que se produzca por el exceso de tamaño del lienzo con respecto al personaje que tenemos

15. Los Sprites y los Personajes

dibujado en él.

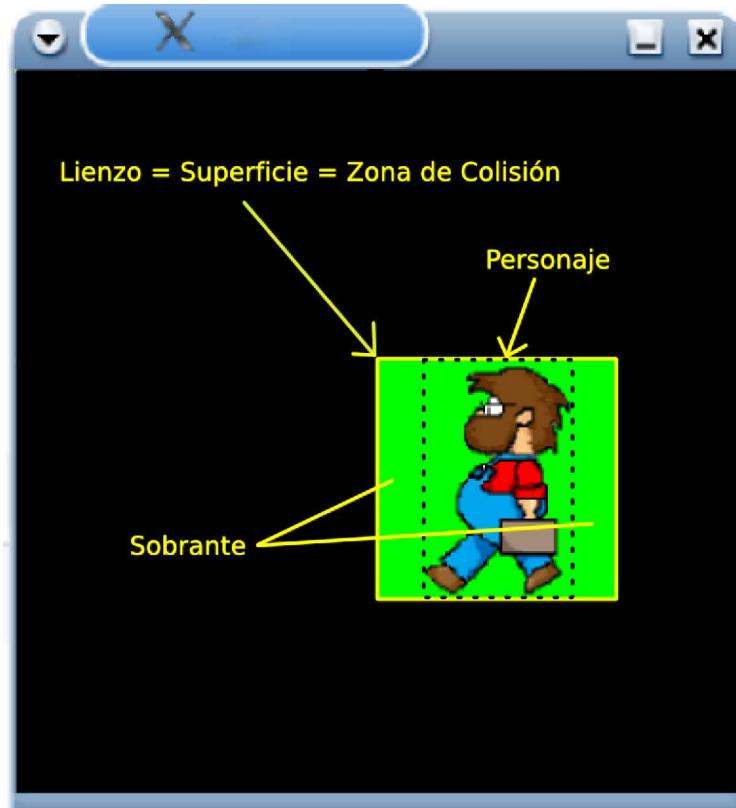


Figura 15.6: Elementos de una imagen SDL.

La **detección** de colisiones es una técnica que consiste en implementar unas funciones que nos permitan conocer cuando ha existido una colisión entre dos elementos. Este va a ser el núcleo de nuestro trabajo en cuanto a las colisiones.

Existen varias formas de solventar en cierta medida los problemas que acarrea el que las superficies sean rectangulares. La primera de ellas es recortar todo lo posible el lienzo sobrante así tendremos menos exceso por lo que el comportamiento de la colisión será más adecuado. Esto muchas veces no está en nuestras manos. Como programadores, y en proyectos de cierta envergadura, no nos haremos cargo del diseño de los personajes y seguramente no tengamos tiempo de ajustar todos los lienzos que pueden existir en una animación con el trabajo que supone retocar una rejilla de imágenes.

La segunda forma de evitar un mal comportamiento de las colisiones es realizar una buena implementación de las funciones dedicadas a detectar dichas colisiones. Podemos añadir información a nuestro personaje que ciña la superficie de colisión lo más posible al personaje para conseguir una mejor

respuesta. Esta es la opción más pausible y la que vamos a desarrollar.

En definitiva, una colisión es un solapamiento de píxeles al que, normalmente, deberemos de reaccionar. Para que esta reacción se haga en el momento correcto tendremos que implementar unas funciones dedicadas a la detección de colisiones que veremos a continuación.

15.7.3. Tipos de colisiones

Existen varias clasificaciones para las colisiones según sobre qué aspecto queramos discriminar las diferencias entre unas y otras. La primera y más común es la clasificación que diferencia entre el tipo de elemento con el que se produce la colisión. Según dicha clasificación las colisiones pueden ser:

Entre personajes Estas colisiones se producen entre los elementos “activos” de la aplicación. Cuando vamos por el mapa de nuestro juego y el personaje principal se encuentra con un enemigo y aproximan lo suficiente se produce una colisión de este tipo. De la misma manera que si nuestro personaje principal tiene alguna clase de arma y dispara cuando dicho disparo llegue al enemigo este producirá una colisión con el adversario.

Con el escenario Este tipo de colisiones son el que se producen entre el personaje y los elementos que componen el escenario o el nivel en cuestión. En un videojuego de plataformas cada una de las superficies donde puede subirse nuestro personaje y el propio personaje producen una colisión que provoca que podamos mantenerlo subida en ella. Si no se produjese dicha colisión tendríamos que usar otra técnica para que nuestro programa supiese que dicha plataforma es suelo y debe interpretarlo como tal.

También podemos clasificar las colisiones según su forma. Existen colisiones de un cuerpo que son aquellas en las que se utiliza una forma geométrica para representar al personaje o al objeto en cuestión para realizar la detección de colisiones y las colisiones compuestas que son aquellas en las que los integrantes que van a colisionar han sido divididos en varios rectángulos que nos van a permitir diferenciar la figura real del personaje. En cuantas más figuras geométricas dividamos a nuestro personaje mejor comportamiento obtendremos de las colisiones.

Las clasificaciones de las colisiones son un aspecto meramente informativo que nos van a ayudar a plantear la detección de colisiones para determinar de qué manera actuar según el caso que nos encontremos.

15. Los Sprites y los Personajes

15.7.4. Implementando la detección de colisiones

Vamos a implementar diferentes tipos de detección de colisiones desde los métodos más sencillos hasta algunos más complejos. Es importante saber diferenciar cuando usar un método de detección de colisiones u otro. El compromiso entre tiempo de procesamiento y resultado tiene que estar siempre presente.

Después de estudiar este apartado vas a ser capaz de crear tu propio método de detección de colisiones. Imagina que tienes un ítem que es una pequeña esfera que le va a proporcionar puntos a tu personaje principal. Esta esfera está almacenada en una imagen cuadrada. Ahora tienes varias opciones, la primera es implementar la colisión tomando como superficie el cuadrado que envuelve a la esfera. Seguramente si la esfera es pequeña en el transcurso de la partida no notarás que la colisión no se produce al cien por cien del acercamiento.

La otra alternativa es implementar una función que divida la esfera en cien pequeños rectángulos pareados horizontalmente. Cada vez que se produzca una vuelta en el *game loop* habrá que comprobar si existe colisión con este elemento. Esto provocará un fuerte aumento de los recursos para obtener una respuesta que casi teníamos con el método anteriormente descrito. Estas son decisiones de diseño que debes de tomar a la hora de crear un videojuego.

Vamos a empezar con una implementación de las colisiones sencilla.

15.7.5. Detección de colisiones de superficie única

Existen varios casos de detección de colisiones de superficies únicas. El primero es el que hemos planteado en esta sección como paradigma del mal que en ocasiones puede resultar ventajoso. Se trata de utilizar como superficie de colisión el rectángulo que envuelve a nuestros personajes.

Vamos a realizar una primera versión de lo que sería la implementación del método de las colisiones. Vamos a tratar las colisiones que se producen entre las superficies sin considerar que parte de ellas están o no dibujadas. Este método es válido cuando utilicemos dibujos que rellenen toda la superficie donde han sido almacenados.

Vamos a tomar de referencia nuestra clase personaje sin animaciones asociadas para simplificar todo lo posible el código. A esta clase vamos a añadirle dos nuevos métodos *ancho()* y *alto()* que nos permiten conocer el ancho y alto de la imagen que va a almacenar como representación del personaje, o lo que es lo mismo, la superficie que cumple con este objetivo. Estos métodos nos van a ayudar a la hora de realizar el cálculo de la colisión.

Además hemos comentado la parte de la clase donde se establecía el colo clave para se pueda observar cuando entran en contacto las superficies.

La implementación de la función que nos permite calcular la colisión es la siguiente:

```

1 ;_____
1 ;// Listados: Colisiones.cpp
2 ;///
3 ;// Implementación de funciones dedicadas a la detección
4 ;// de colisiones
5 ;
6 ;// Esta función devuelve true si existe colisión entre los dos
7 ;// personajes que recibe como parámetros
8 ;
9 ;#include <iostream>
10 ;#include "Personaje.h"
11 ;#include "Colision_superficie.h"
12 ;
13 ;using namespace std;
14 ;
15 ;
16 ;
17 ;bool colision_superficie(Personaje &uno, Personaje &otro) {
18 ;
19 ;    int w1, h1, w2, h2, x1, y1, x2, y2;
20 ;
21 ;    w1 = uno.ancho();
22 ;    h1 = uno.alto();
23 ;
24 ;    w2 = otro.ancho();
25 ;    h2 = otro.alto();
26 ;
27 ;    x1 = uno.pos_x();
28 ;    y1 = uno.pos_y();
29 ;
30 ;    x2 = otro.pos_x();
31 ;    y2 = otro.pos_y();
32 ;
33 ;    if( ((x1 + w1) > x2) &&
34 ;        ((y1 + h1) > y2) &&
35 ;        ((x2 + w2) > x1) &&
36 ;        ((y2 + h2) > y1))
37 ;
38 ;        return true;
39 ;
40 ;
41 ;    return false;
42 ;
43 ;}
;
```

Para que sea más fácil de entender hemos definido unas variables enteras para almacenar tanto la posición como el ancho y alto de los dos personajes

15. Los Sprites y los Personajes

de los que tenemos que comprobar si existe colisión o no. La técnica es la siguiente. Se trata de comprobar si alguno de los puntos de una superficie están contenidos en la otra.

Con $x1$ y $w1$ tenemos cubierto la parte horizontal del primer personaje. Para que exista colisión $x2$, que es la posición en x del personaje *otro*, tiene que ser menor que la suma de $x1$ y $w1$. Este razonamiento lo extendemos a cada uno de los lados de las dos superficies y tenemos la función que detecta las colisiones.

La función, como no podía ser de otra forma, devolverá *true* en caso de existir colisión y *false* en caso de que no exista. En el programa de ejemplo hemos hecho uso de la librería adicional *SDL_ttf* para mostrar un mensaje cada vez que exista colisión.

Al ejecutar la aplicación puedes ver como la colisión se produce mucho antes de que los personajes se encuentren en realidad. Esto hace que el comportamiento de esta implementación no sea aceptable en muchos casos. Vamos a mejorar esta implementación.

Vamos a proponer una nueva solución que nos permita seguir utilizando un sólo rectángulo por superficie pero que tenga un mejor comportamiento. Se trata de añadir información en la clase personaje de manera que especifiquemos mediante una posición inicial, altura y anchura el rectángulo de la superficie que ocupa el personaje en realidad.

La implementación de la detección de colisiones no variará más que en cambiar las variables de estudio de cada uno de los personajes. Vamos a ver los cambios en la clase personaje:

```
1 ;// Listado: Personaje.h
2 ;//
3 ;// Clase Personaje complementada para la detección de colisiones
4 ;
5 ;#ifndef _PERSONAJE_H_
6 ;#define _PERSONAJE_H_
7 ;
8 ;#include <SDL/SDL.h>
9 ;
10 ;
11 ;class Personaje {
12 ;
13 ;    public:
14 ;
15 ;        // Constructor
16 ;
17 ;        Personaje(char *ruta, SDL_Rect &real, int x = 0, int y = 0);
18 ;
```

```

19 ;
20 ;    // Consultoras
21 ;
22 ;    int pos_x(void);
23 ;    int pos_y(void);
24 ;
25 ;    int ancho(void);
26 ;    int alto(void);
27 ;
28 ;    int pos_x_real(void);
29 ;    int pos_y_real(void);
30 ;    int ancho_real(void);
31 ;    int alto_real(void);
32 ;
33 ;    void dibujar(SDL_Surface *pantalla);
34 ;
35 ;    // Modificadoras
36 ;
37 ;    void pos_x(int x);
38 ;    void pos_y(int y);
39 ;
40 ;    // Modifica la posición del personaje con respecto al eje X
41 ;
42 ;    void avanzar_x(void);
43 ;    void retrasar_x(void);
44 ;
45 ;    // Modifica la posición del personaje con respecto al eje Y
46 ;
47 ;    void bajar_y(void);
48 ;    void subir_y(void);
49 ;
50 ;
51 ;
52 ; private:
53 ;
54 ;    // Posición
55 ;    int x, y;
56 ;
57 ;    SDL_Rect superficie_real;
58 ;
59 ;    SDL_Surface *imagen;
60 ;
61 ;};
62 ;
63 ;#endif
;
```

Como puedes ver hemos incluido una nueva variable de tipo rectángulo en la parte privada de la clase que controlará qué área dentro de la superficie de la imagen, o de la animación que tenga asociada, es válida para la realizar la detección de colisiones, es decir, donde está dibujado realmente el personaje. En el caso de tener una animación y existir una diferencia notable entre la parte de la superficie que se encuentra el personaje entre un cuadro y

15. Los Sprites y los Personajes

otro tendríamos que definir una secuencia que según el paso en el que se encontrase la animación utilizase un cuadro u otro para el cálculo de la colisión.

Además hemos incluido en la clase cuatro nuevos métodos que nos van a permitir conocer la posición real del personaje en pantalla, así como su anchura altura. Como ya hemos dicho lo más probable es que el personaje no ocupe toda la superficie donde está almacenado y estas funciones nos permiten saber donde se encuentra la imagen actual del personaje. La implementación de dichas funciones es:

```
1 ;// Funciones para la implementación de la detección de colisiones
2 ;
3 ;int Personaje::pos_x_real(void) {
4 ;
5 ;    // Desplazamiento con respecto a la posición actual
6 ;
7 ;    return superficie_real.x + x;
8 ;
9 ;}
10 ;
11 ;int Personaje::pos_y_real(void) {
12 ;
13 ;    return superficie_real.y + y;
14 ;
15 ;}
16 ;
17 ;// Alto y ancho ajustados al personaje
18 ;
19 ;int Personaje::ancho_real(void) {
20 ;
21 ;    return superficie_real.w;
22 ;
23 ;}
24 ;
25 ;int Personaje::alto_real(void) {
26 ;
27 ;    return superficie_real.h;
28 ;
29 ;}
```

Como puedes ver estas funciones calculan el desplazamiento interno que produce que el personaje no ocupe toda la superficie donde es almacenado. En cuanto a la nueva función que calcula la colisión es muy parecida a la primera que hemos visto pero esta vez usa las funciones que proporcionan las dimensiones reales del personaje. Aquí tienes la nueva implementación:

```
1 ;// Listados: Colisiones.cpp
2 ;//
3 ;// Implementación de funciones dedicadas a la detección
```

```

4 ;// de colisiones
5 ;
6 ;// Esta función devuelve true si existe colisión entre los dos
7 ;// personajes que recibe como parámetros
8 ;//
9 ;// Superficies únicas ajustada
10 ;
11 ;#include <iostream>
12 ;#include "Personaje.h"
13 ;#include "Colision_superficie_ajustada.h"
14 ;
15 ;using namespace std;
16 ;
17 ;bool colision_superficie_ajustada(Personaje &uno, Personaje &otro) {
18 ;
19 ;    int w1, h1, w2, h2, x1, y1, x2, y2;
20 ;
21 ;    w1 = uno.ancho_real();
22 ;    h1 = uno.alto_real();
23 ;
24 ;    w2 = otro.ancho_real();
25 ;    h2 = otro.alto_real();
26 ;
27 ;    x1 = uno.pos_x_real();
28 ;    y1 = uno.pos_y_real();
29 ;
30 ;    x2 = otro.pos_x_real();
31 ;    y2 = otro.pos_y_real();
32 ;
33 ;    if( ((x1 + w1) > x2) &&
34 ;        ((y1 + h1) > y2) &&
35 ;        ((x2 + w2) > x1) &&
36 ;        ((y2 + h2) > y1))
37 ;
38 ;        return true;
39 ;
40 ;
41 ;    return false;
42 ;
43 ;}
;
```

Como puedes ver al ejecutar el programa de ejemplo hemos conseguido una respuesta bastante ajustada de la colisión, podríamos decir que casi milimétrica. En muchos casos no nos interesaría hacer una implementación pormenorizada de la colisión si no que será suficiente con un comportamiento de este estilo que no sobrecarge al sistema. Lo comentábamos al comienzo de la sección. Se trata de establecer una línea bien definida en el compromiso rendimiento/respuesta.

Vamos a estudiar otros métodos para la detección de colisiones.

15.7.6. Detección de colisiones de superficies compuestas

Hemos conseguido una buena implementación para calcular las colisiones entre dos de nuestros personajes. Pero que ocurriría si nuestros personajes tuviesen forma triangular. No habría forma de aproximar la forma del rectángulo a dicho triángulo. Vamos a utilizar el ejemplo anterior con dos triángulos para ver como respondería la implementación de las colisiones.

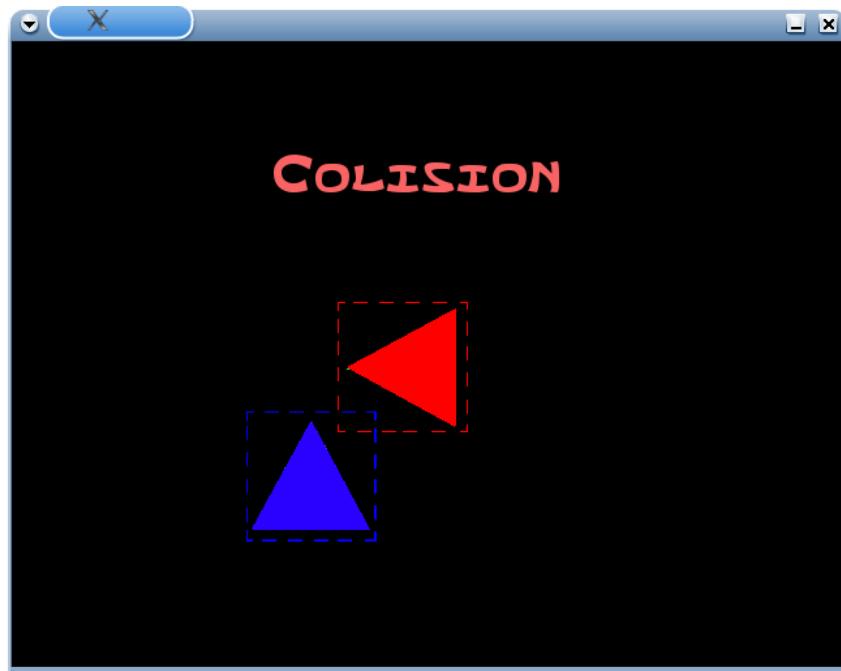


Figura 15.7: Colisión entre triángulos simple.

En el ejemplo 10 hemos utilizado el mismo código que en el primer caso para detectar las colisiones. Se puede observar como el resultado no es aceptable. Los dos triángulos no se llegan ni a acercar cuando la función informa ya de una colisión. Tenemos que buscar una solución.

En el apartado anterior ajustamos el rectángulo a la figura lo que nos proporcionó una solución válida. En este caso no vale. No podemos ajustar un sólo rectángulo a la figura de un triángulo. Existen varias posibles soluciones para este problema.

La primera que vamos a afrontar es un método generalista que nos va a permitir solucionar mucho de los problemas que genera la detección de colisiones. Se trata de dividir la figura que queramos “colisionar” en rectángulos lo suficientemente pequeños como para que la respuesta del sistema sea válida. Es importante que dichos recuadros tengan un tamaño adecuado ya que es un problema si son excesivamente pequeños o excesivamente grandes.

Cuando comprobemos una colisión por este método tendremos que estudiar cada uno de los rectángulos que componen las figuras que colisionan. Si los rectángulos en los que hemos dividido la figura son demasiado pequeños la carga del sistema será alta, muchas veces, sin necesidad. Sin embargo si no tienen un tamaño lo suficientemente pequeño puede ser que no consigamos el comportamiento deseado.

Para la implementación de este nuevo método necesitamos ampliar la clase Personaje para que nos ofrezca suficiente información sobre la composición de la superficie que lo representa. En un vector vamos a almacenar los rectángulos que van a componer la superficie. Además añadimos un método que nos permite añadir elementos a dicho vector.

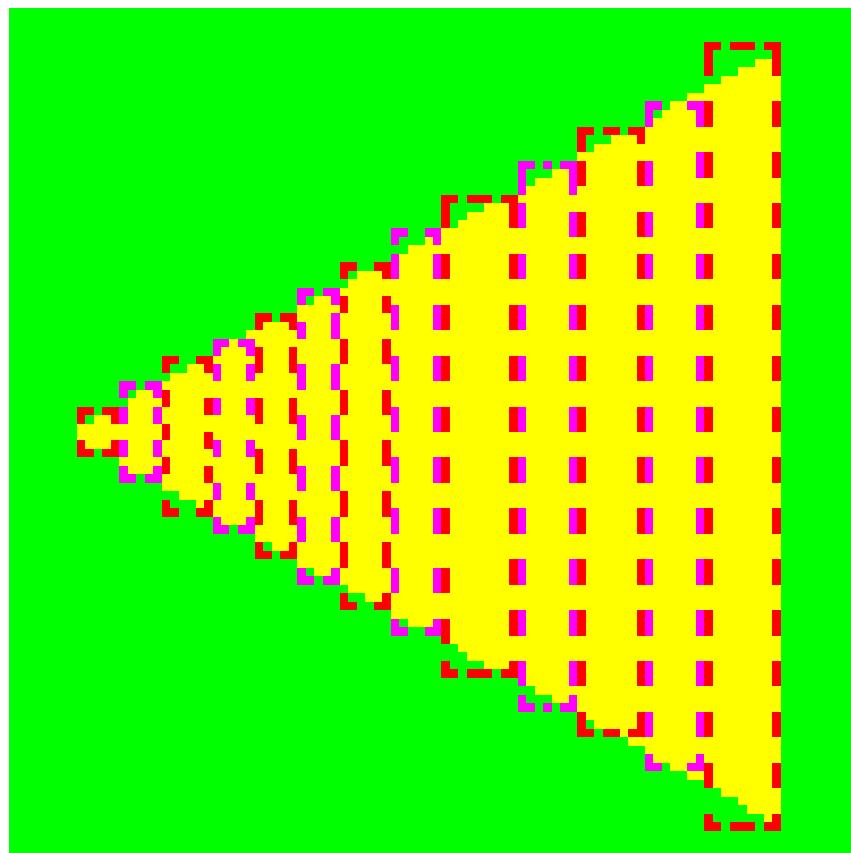


Figura 15.8: Triángulo dividido en rectángulos

En el programa principal tenemos que construir las superficies de colisión a partir de la imagen que hemos dividido en rectángulos para poder utilizar la función que nos permite detectar las colisiones haciendo uso de dicho método. El código resultante es el siguiente:

15. Los Sprites y los Personajes

```
1 ; // Vamos a construir el área de colisión a partir de rectángulos
2 ; // para el personaje principal
3 ;
4 ;     SDL_Rect rect_principal[12] = {{3, 82, 92, 8},
5 ;                                         {10, 75, 78, 6},
6 ;                                         {14, 67, 71, 7},
7 ;                                         {17, 60, 64, 6},
8 ;                                         {26, 45, 48, 5},
9 ;                                         {29, 39, 40, 5},
10 ;                                         {32, 34, 34, 4},
11 ;                                         {35, 29, 28, 4},
12 ;                                         {38, 24, 22, 4},
13 ;                                         {40, 18, 18, 5},
14 ;                                         {44, 13, 11, 4},
15 ;                                         {47, 8, 5, 4}};
16 ;
17 ;
18 ;     for(int i = 0; i < 12; i++)
19 ;         principal.annadir_rectangulo(rect_principal[i]);
20 ;
21 ;
22 ; // Vamos a construir el área de colisión a partir de rectángulos
23 ; // para el adversario
24 ;
25 ;     SDL_Rect rect_adversario[12] = {{82, 3, 8, 92},
26 ;                                         {75, 10, 6, 78},
27 ;                                         {67, 14, 7, 71},
28 ;                                         {60, 17, 6, 64},
29 ;                                         {45, 26, 5, 48},
30 ;                                         {39, 29, 5, 40},
31 ;                                         {34, 32, 4, 34},
32 ;                                         {29, 35, 4, 28},
33 ;                                         {24, 38, 4, 22},
34 ;                                         {18, 40, 5, 18},
35 ;                                         {13, 44, 4, 11},
36 ;                                         {8, 47, 4, 5}};
37 ;
38 ;
39 ;     for(int i = 0; i < 12; i++)
40 ;         adversario.annadir_rectangulo(rect_adversario[i]);
```

Ahora vamos con lo que más nos interesa. ¿Cómo detectamos una colisión entre dos personajes que están compuesto por rectángulos? La base del razonamiento parte del primer ejemplo de detección de colisiones. En ese caso comprobábamos si algún punto de la superficie de uno de los personajes estaba contenido en la superficie del otro personaje. Para este caso vamos a extender el razonamiento. Tenemos dos superficies cuyas áreas de colisión están compuestas por múltiples rectángulos para saber si existe una colisión basta con comprobar si algún punto de alguna de las áreas que compone cada personaje está contenida en alguna de las superficies que componen al otro personaje.

Para realizar esta comprobación tenemos que anidar dos bucles que nos permitan recorrer por cada una de las superficies del primer personaje todas las del segundo participante en la posible colisión. El resultado de la implementación de este método es el siguiente:

```

1 ;// Listado: Colisiones.cpp
2 ;//
3 ;// Implementación de funciones dedicadas a la detección
4 ;// de colisiones
5 ;
6 ;// Esta función devuelve true si existe colisión entre los dos
7 ;// personajes que recibe como parámetros.
8 ;//
9 ;// Estos personajes estarán compuestos por rectángulos
10 ;
11 ;#include <iostream>
12 ;#include "Personaje.h"
13 ;#include "Colisiones.h"
14 ;
15 ;using namespace std;
16 ;
17 ;
18 ;// Colisión entre dos personajes divididos en rectángulos
19 ;// Devuelve true en caso de colisión
20 ;// O(n^2)
21 ;
22 ;
23 ;bool colision(Personaje &uno, Personaje &otro) {
24 ;
25 ;    int w1, h1, w2, h2, x1, y1, x2, y2;
26 ;
27 ;    // Todos los rectángulos del primer personaje
28 ;
29 ;    for(size_t i = 0; i < uno.rectangulos.size(); i++) {
30 ;
31 ;        w1 = uno.rectangulos[i].w;
32 ;        h1 = uno.rectangulos[i].h;
33 ;        x1 = uno.rectangulos[i].x + uno.pos_x();
34 ;        y1 = uno.rectangulos[i].y + uno.pos_y();
35 ;
36 ;
37 ;        // Con todos los rectángulos del segundo personaje
38 ;
39 ;        for(size_t j = 0; j < otro.rectangulos.size(); j++) {
40 ;
41 ;            w2 = otro.rectangulos[j].w;
42 ;            h2 = otro.rectangulos[j].h;
43 ;            x2 = otro.rectangulos[j].x + otro.pos_x();
44 ;            y2 = otro.rectangulos[j].y + otro.pos_y();
45 ;
46 ;
47 ;

```

15. Los Sprites y los Personajes

```
48 ;           // Si existe colisión entre alguno de los
49 ;           // rectángulos paramos los bucles
50 ;
51 ;           if( ((x1 + w1) > x2) &&
52 ;               ((y1 + h1) > y2) &&
53 ;               ((x2 + w2) > x1) &&
54 ;               ((y2 + h2) > y1))
55 ;
56 ;           return true;
57 ;
58 ;       }
59 ;
60 ;
61 ;   return false;
62 ;
63 ;}
```

Vemos en la implementación como hemos usado dos bucles *for* para resolver la detección de colisiones. En el más externo vamos recorriendo los recuadros que componen el primer personaje obteniendo la posición de cada uno de ellos en las variables (x_1, y_1) , mientras que en las variables w_1 y h_1 almacenamos el tamaño del recuadro en cuestión. Para cada uno de estos rectángulos recorremos cada uno de las superficies que componen al otro personaje en el bucle más interno. Utilizamos unas variables análogas de las que utilizamos para el primer personaje pero con el identificativo de 2. Por cada par de rectángulos realizamos una comprobación por si existe algún punto de uno de ellos que esté contenido en el otro. Si es así devolvemos el valor *true* para indicar que ha existido una colisión. En caso de recorrer todos los rectángulos sin que exista contención en ninguno de los sentidos devolvemos el valor *false*.

Con esta técnica podemos programar colisiones para cualquier tipo de superficies. Basta con dividir dicha superficie en rectángulos suficientes para tener un buen comportamiento. Vamos a realizar dos ejercicios que nos van a permitir practicar un poco con esta técnica.

15.7.7. Ejercicio 2

Utiliza la técnica que acabamos de estudiar para comprobar la colisión de dos esferas. Vamos a utilizar las clases implementadas en el último ejemplo. El trabajo de este ejercicio consiste en dividir la esfera en un número de partes racional de la mejor manera posible que nos ofrezca una comportamiento correcto en la detección de colisiones.

Puedes ver la solución en los listados de este tema en la carpeta ejercicio 2. La solución se centra en la división de la esfera. En este caso:

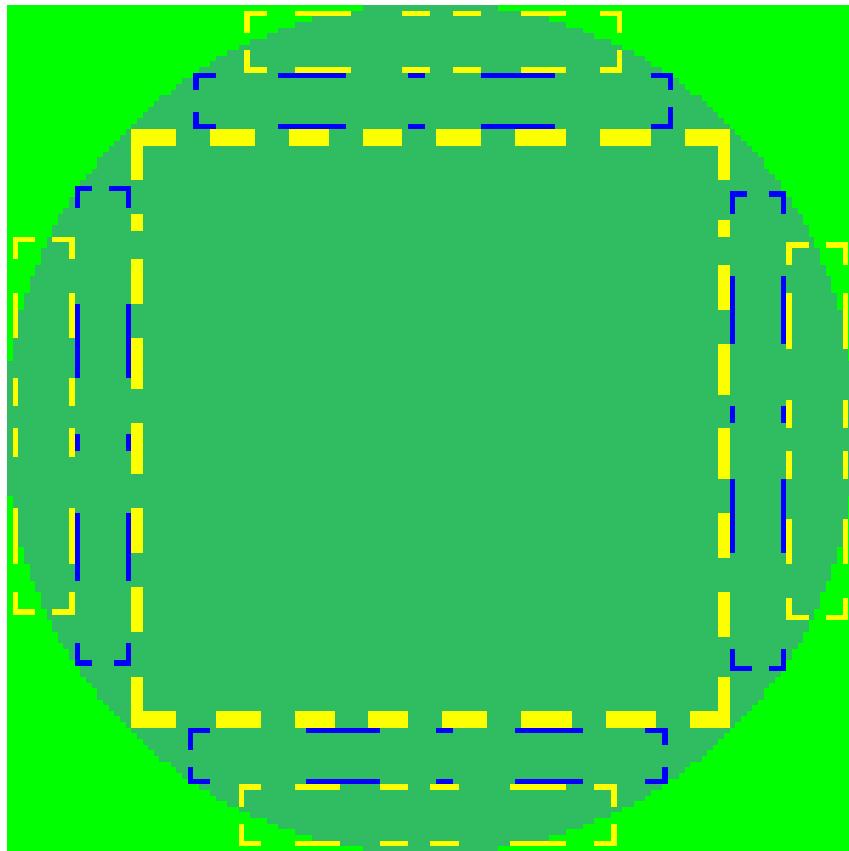


Figura 15.9: Esfera dividida en rectángulos

```
;  
1 ; // Vamos a construir el área de colisión a partir de rectángulos  
2 ; // para el personaje principal  
3 ;  
4 ;     SDL_Rect rect_principal[9] = {{22, 22, 107, 105},  
5 ;                         {33, 12, 84, 9},  
6 ;                         {42, 1, 66, 10},  
7 ;                         {12, 33, 9, 84},  
8 ;                         {1, 41, 10, 66},  
9 ;                         {32, 128, 84, 9},  
10 ;                        {41, 138, 66, 10},  
11 ;                        {128, 33, 9, 84},  
12 ;                        {138, 41, 10, 66}};  
13 ;  
14 ;  
15 ;     for(int i = 0; i < 9; i++)  
16 ;         principal.annadir_rectangulo(rect_principal[i]);
```

15. Los Sprites y los Personajes

15.7.8. Ejercicio 3

Vamos a realizar la misma práctica del ejercicio anterior pero esta vez con nuestro personaje principal. Utiliza el método que estamos aplicando para simular el comportamiento del segundo método de detección de colisiones para delimitar el área de colisiones del rival.



Figura 15.10: Jacinto dividido en secciones

El resultado completo lo tienes en el material del curso. Aquí puedes ver como hemos descompuesto al personaje principal.

```
1 ; // Componemos los personajes
2 ;
3 ; // Vamos a construir el área de colisión a partir de rectángulos
4 ; // para el personaje principal
5 ;
6 ;     SDL_Rect rect_principal[3] = {{27, 3, 41, 35},
7 ;                                         {34, 39, 37, 32},
8 ;                                         {30, 72, 46, 24}};
9 ;
10 ;
11 ;     for(int i = 0; i < 3; i++)
12 ;         principal.annadir_rectangulo(rect_principal[i]);
```

¿Sencillo no? El trabajo ha consistido principalmente en medir que proporciones queremos para cada personaje y de cuantos rectángulos lo vamos a componer.

Como puedes ver en este caso la respuesta es parecida a la que habíamos conseguido ajustando el tamaño de la superficie al personaje por lo que hay que sopesar si nos interesa aumentar la carga del sistema en la detección de colisiones en este caso.

15.7.9. Otros algoritmos de detección de colisiones

El objetivo de este capítulo es que conozcas diferentes técnicas de implementación de colisiones. Existen numerosos artículos escritos sobre este tema que es cuestión de profundizar si necesitas otro tipo de respuesta.

Un algoritmo infalible es que calcula si un píxel cuando colisiona con otra superficie esta es de la parte. La carga que supone tener que hacer este cálculo por cada uno de los píxeles que colisionan ha hecho que la descartemos como opción viable. Se trata de, una vez detectada colisión entre los rectángulos comprobar si los píxeles que colisionan son parte del color key o no.

Sobre la detección de colisiones existen numerosos artículos que te pueden ayudar a abrir la mente sobre este tema. Desde aquí te recomendamos un acercamiento a esta lectura que será muy enriquecedora para tu formación.

15.8. Recopilando

En este capítulo hemos tratado todo lo referente a los personajes y sprites que integrarán nuestro videojuego. Hemos visto como implementar los distintos tipos de animaciones de los participantes así como la manera de almacenar el contenido multimedia asociados a éstos en una galería.

Hemos presentado informalmente el concepto de autómata y su relación con los participantes del juego y con sus distintos tipos de animaciones.

Para terminar hemos estudiado el concepto de colisión así como diferentes técnicas para llevar a cabo su diseño y codificación.

15. Los Sprites y los Personajes

Capítulo 16

Un ejemplo del desarrollo software de un videojuego

16.1. Introducción

Llegados a este punto es el momento de realizar un pequeño videojuego de ejemplo que nos sirva para acoplar e interrelacional todos los conocimientos vistos en este tutorial. Vamos a seguir la misma filosofía que hemos llevado durante todo el tutorial. Vamos a implementar el videojuego con un código lo más asequible posible y cercano al C.

Vamos usar algunos aspectos integrados en el lenguaje C++ por motivos de comodidad y eficiencia pero sólo usando elementos muy básicos que sean de rápida y fácil compresión. Entre ellos está el concepto de clase (y todo lo que supone esto) y algunas estructuras de la STL.

En este capítulo vamos a plantear la historia, vamos a crear los personajes, los niveles con su correspondiente editor de niveles y la justificación de todas las decisiones que hemos tomado para el desarrollo del videojuego.

16.2. Conocimientos previos

Para llevar a cabo este capítulo deberás haber estudiado todos los capítulos previos del tutorial para que el esfuerzo que tengas que hacer por aprovechar el contenido de éste sea razonable.

Además de los conocimientos previos de SDL debes de manejarlo con cierta soltura en los aspectos básicos de C++ que hemos utilizado hasta la fecha ya que serán puestos en prácticas en este capítulos.

Vamos a realizar una pequeña introducción informal a la metodología UML que te será muy útil a la hora de plantearte el desarrollo de una

16. Un ejemplo del desarrollo software de un videojuego

aplicación con SDL.

16.3. Objetivos

Los objetivos de este capítulo son:

1. Conocer y comprender como integrar las distintas partes de SDL para realizar un videojuego.
2. Conseguir un acercamiento del lector al seguimiento de un patrón de diseño software como es UML.

16.4. Planteamiento informal de un videojuego

Vamos a tratar en primer lugar una pequeña introducción que nos va a permitir plantear informalmente un videojuego. Desde cómo elaborar una historia, pasando por los personajes, creación de niveles... Este proceso no difiere mucho en la mayoría de los casos de otros procesos de creación como el de una película o el relato de una novela complementados con el proceso de desarrollo software.

Una vez que hayamos realizado este recorrido por el planteamiento del videojuego vamos a proceder al análisis, diseño e implementación del mismo de una manera formal para que te acerques a los procesos y patrones formales necesarios para la creación de un juego y, probablemente, de cualquier aplicación.

16.5. La historia

16.5.1. Introducción. Cómo contar la historia

A la hora de crear un videojuego lo primero que tenemos que saber es qué historia queremos contar. Puede darse el caso de que queramos programar un videojuego que no necesite historia que lo envuelva y no necesitamos desarrollar este aspecto.

En cuanto al desarrollo de la historia no hay una metodología que nos permita crear historias claramente. Hay varios aspectos sobre la manera de escribir, hablar, mostrar cosas... que nos permiten provocar en el usuario tensión, felicidad, relax... Por ejemplo si queremos que una escena sea terrorífica de sentido común es oscurecer la escena, reproducir una música intrigantemente al mismo tiempo que utilizamos frases cortas y contundentes en

los textos que sean mostrados por pantalla.

Si por el contrario queremos que la escena sea alegre no hay nada como unos colores vivos, unos personajes felices y una música marchosa para conseguirlo. Como ya hemos dicho hay juegos que no necesitan de una historia para ser jugado. Estos juegos cuando son englobados en una trama son mucho más interesantes, veamos un ejemplo.

Imagina que has creado un videojuego del popular Sudoku y lo liberas. Si el videojuego es bueno puede que tenga una aceptación y que alguien afín al juego del Sudoku se haga con tu creación. Ahora imagina que has envuelto al Sudoku en una trabajada historia que empezara tal como así:

Al lejano oriente llegó un extranjero enamorado de una bella dama oriental. Él la amaba pero su familia no lo aceptaba. Un buen día el patriarca decidió dar una oportunidad a tan valiente foráneo. "Te casarás con mi hija si superas los retos que te imponga"

Como habrás podido suponer el reto será resolver varios Sudokus haciendo que la dificultad de los mismos sea gradual. Con este pequeño añadido al juego y una ambientación adecuada no sólo los amantes de los Sudokus valorarán tu obra si no que sumergerás en este mundo a otra clase de usuario no tan afín a este tipo de juegos. Esta táctica es muy utilizada actualmente para realizar remakes de juegos clásicos o de juegos de mesa transformados en videojuegos para complementarlos ofreciendo un mayor aliciente.

Una historia debe de tener unas partes bien definidas:

Situación: Es la introducción de la historia donde situamos al jugador en la época y le presentamos la historia en la que se va a envolver.

Desarrollo: Es la trama de la historia que debe ir avanzando paralelamente a los niveles que vaya completando nuestro jugador.

Desenlace: Es el final del juego. Existen numerosos artículos dedicados a la decepción que produce un videojuego con un mal desenlace. El jugador espera que después de haber dedicado parte de su tiempo a completar el videojuego el final de éste le proporcione un refuerzo positivo para aventurarse en un nuevo videojuego.

La calidad de la historia está en lo que tu imaginación sea capaz de producir.
¡Mucha suerte!

16. Un ejemplo del desarrollo software de un videojuego

16.5.2. Nuestra historia

La historia de nuestro videojuego es muy simple:

Jacinto es un electricista especialista en montajes. Un buen día su jefe le encargó el mantenimiento del cableado estructurado de un edificio de oficinas. Jacinto estudió mucho para conocer todas las especificaciones y usos de estos cableados siendo el RJ 45 su mejor aliado.

No sabía Jacinto que su trabajo iba a ser tan duro. Veía como todos los días le desaparecían las herramientas y que los cables aparecían rotos todas las mañanas y era un gran enigma para él. Un buen día decidió pernoctar en la oficina. Su sorpresa fue cuando vio a unas pequeñas ratas que se hacían con todo el material. Le robaban destornilladores, alicates... todo lo que veían. Aliadas de las ratas las motas de polvo impedían que Jacinto recuperase su material. ¡¡Ayuda a Jacinto!! Elimina a todas las ratas y a sus amigas las motas de polvo y recupera el material o su jefe lo despedirá.

Como puedes ver es una trama muy simple y escueta que nos servirá como punto de partida para desarrollar el videojuego. Ya sabemos que vamos a tener un personaje principal, que seguramente el juego sea de plataformas y que habrá al menos dos tipos de enemigos: las ratas y las motas de polvo. Como puedes ver sentarte a crear una historia puede ayudarte a plantear el juego que vamos a crear.

16.6. Los personajes

Una vez desarrollada la historia del juego será mucho más fácil plantear los personajes. Nuestro personaje principal debe de cumplir unos requisitos que lo hagan especial para el desarrollo del juego. Esto lo hará mucho más atractivo.

Entre los personajes podemos distinguir a tres tipos: el personaje principal, los adversarios y los objetos o ítems. El personaje principal debe de poder distinguirse claramente de los demás y tener unas animaciones lo más detalladas posibles que nos permitan disfrutar de su creación. Deberemos de diseñar todos los movimientos que puede realizar tal como vimos en el capítulo dedicado a los personajes.

Adversarios debe haberlos de varios tipos. La dificultad con la que son eliminados debe de ser fácilmente reconocible según el aspecto que tengan. Si un enemigo va a ser difícil de eliminar debe de tener un aspecto

robusto frente a otro que sea más fácil de hacer desaparecer. Si recuerdas algún juego de plataformas puedes ver como se pone de manifiesto este concepto.

Los objetos deben de ser fácilmente identificables y poseer alguna animación que los haga destacar del fondo del juego. Tendremos que decidir que nos aporta el obtener un objeto u otro.

En nuestro videojuego de ejemplo vamos a tener un personaje principal, nuestro electricista Jacinto. Las acciones y movimientos que puede realizar va a ser un subconjunto de las que diseñamos en el capítulo anterior que versaba sobre el diseño de personajes. Como enemigos tenemos a ratas y motas de polvo que tendrán un comportamiento similar y que deberemos de eliminar a base de golpes con el maletín.

Tendremos varios tipos de objetos que nos permitirán saltar, andar, navegar por el nivel y otros objetos o ítems que serán los que debamos ir recogiendo por el camino. Vamos a dejar como ejercicio el criterio por el que Jacinto podrá pasar de nivel. En unas líneas trataremos este tema.

16.7. Los niveles y el Editor

16.7.1. La creación de niveles

Depende del tipo de videojuego que queramos crear podremos diseñar los niveles de muy distinta forma. Si es un juego de lógica iremos complicando la estructura del juego hasta un nivel suficiente cada vez que pasemos un reto.

La creación de niveles tiene mucho contenido creativo como ocurría con el del desarrollo de la historia. Tenemos que diseñar niveles con distinto grado de dificultad que nos permitan ir avanzando por situaciones cada vez más difíciles que sortear. No existe una técnica definida para el diseño de estos niveles y tiene que ser nuestra capacidad de creación la que los determine.

En un nivel tendremos a nuestro personaje principal así como a los adversarios y los objetos. Para pasar de un nivel a otro deberemos de establecer unos objetivos que nos permitan promocionar por los distintos niveles.

En nuestra aplicación Jacinto no tiene definido que debe hacer para pasar de nivel y es una tarea que propondremos como ejercicio. Jacinto va a ir recogiendo diferentes herramientas y eliminando enemigos. Una de estas dos acciones puede ser suficiente para establecer un criterio de promoción de nivel. Por ejemplo podemos hacer que el personaje pase al nivel siguiente una vez

16. Un ejemplo del desarrollo software de un videojuego

recoja todo el material o bien cuando haya eliminado a todos los enemigos.

Es importante medir bien la dificultad de los niveles. Si creamos un juego demasiado complicado producirá una frustración en el jugador que hará que pierda interés en seguir con el mismo. Todos los aficionados a los videojuegos recordamos algún nivel o algún juego que se nos atragantó y abandonamos al ser superados por el mismo.

16.7.2. El editor de niveles. Los tiles

El editor de niveles está intimamente ligado al tipo de videojuego para el que se implementa éste. Nosotros para crear los distintos niveles hemos desarrollado un editor. Este editor nos permite establecer la posición de todos los elementos que integran el nivel tanto decorados, personajes, ítems...

La técnica que sigue el editor para construir el nivel es muy utilizada en el mundo de los videojuegos. Se divide el mapa del nivel en cuadrados con un tamaño fijo. Cada uno de estos recuadros es conocido como TILE. En un determinado tile sólo puede haber un elemento. En nuestro caso hemos decidido crear un tamaño de tile que es casi estándar (32×32). En un tile habrá un elemento del nivel como puede ser el suelo o un cuadro decorativo, será la posición inicial de los personajes del juego y determinarán la posición inicial de los elementos del juego.

Esto supone que un nivel no sea más que un conjunto de tiles de 32×32 que definen el contenido de cada parte de dicho nivel. Como almacenar y cargar estos ficheros gestionando la información de cada uno de estos niveles lo veremos en la implementación de las clases del videojuego.

No hay mejor manera de entender lo que es un tile y como se crean los niveles a partir de ellos que utilizar el editor de niveles que viene con el videojuego final. Es buen momento de que practiques y crees tus primeros niveles. En la figura 16.1 puedes ver un ejemplo de la construcción de un nivel a partir de tiles.

16.7.3. La lógica del juego

La lógica del juego sigue siendo la misma que estudiamos al principio del curso, no hay novedades importantes. Recuerda que se sigue un proceso de inicialización. En este caso la inicialización es más extensa. Hay que preparar todos los subsistemas que vamos a utilizar y determinar la posición de cada componente que verá dada por la información guardada en el fichero de niveles. En la figura 16.2 tienes un recordatorio de las fases que teníamos que

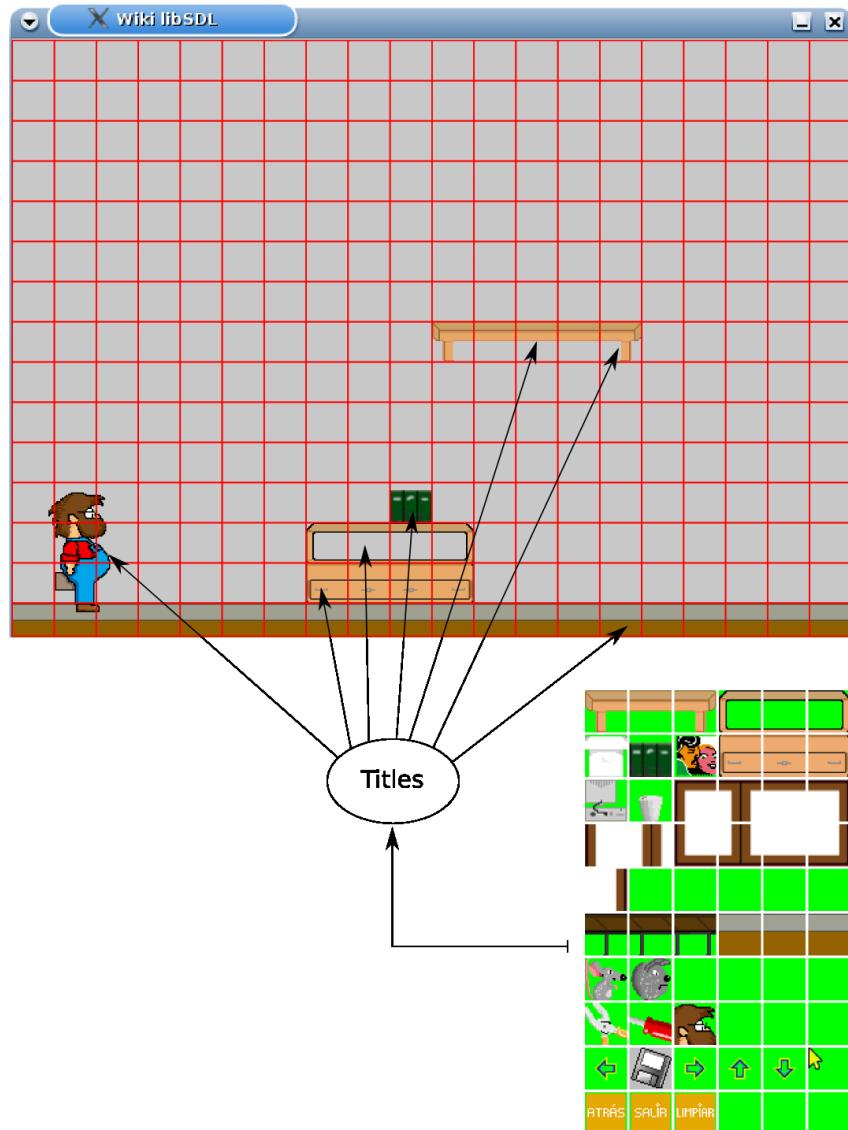


Figura 16.1: División en tiles

seguir a la hora de implementar un videojuego.

Una vez tengamos toda la información disponible mostraremos en pantalla nuestro juego. La lógica que sigue el juego, y que vamos a desarrollar a lo largo del capítulo, se basa en un bucle (el famoso *game loop*) que se va repitiendo a lo largo de la ejecución del programa hasta que el jugador decida salir de la aplicación. Recuerda que en este game loop se realiza una actualización lógica de las posiciones y estados de los personajes y demás elementos para luego para luego actualizar la pantalla con esta nueva información.

Durante la actualización lógica se comprueban las capturas de objetos, se gestionan las colisiones, se comprueba la temporización de la aplicación...

16. Un ejemplo del desarrollo software de un videojuego

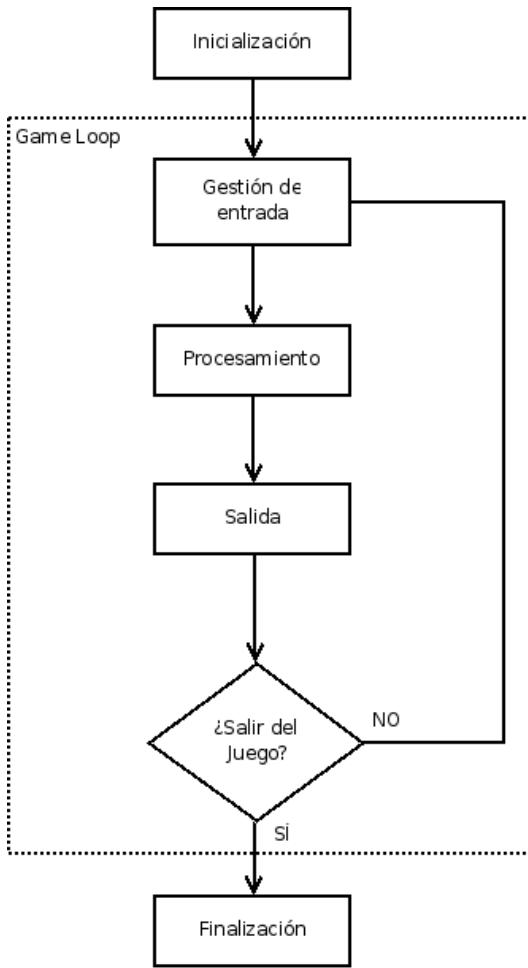


Figura 16.2: Lógica del juego

Como puedes observar estos elementos no son nuevos para nosotros ya que los hemos desarrollado en el tutorial con el objetivo de integrarlos ahora en una única aplicación. Recuerda que después de esta actualización lógica se deben de mostrar los cambios a través del interfaz de pantalla.

En este capítulo detallaremos la implementación y lógica de cada uno de los elementos que componen el videojuego desde el editor hasta la función de salida.

16.7.4. El control del tiempo

Como ya sabes el control del tiempo es una de las partes fundamentales que tenemos que tener en cuenta a la hora de desarrollar nuestro videojuego. Este es uno de los aspectos que más se descuida cuando diseñamos nuestros primeros videojuegos ya que no es un problema hasta que no probamos el resultado del mismo.

Ya conoces los conceptos relacionados al control del tiempo de capítulos anteriores. Para nuestro videojuego de ejemplo utilizaremos una espera activa que fije unos determinados “frames per second” que sean suficiente para mostrar una fluidez correcta del videojuego.

Como en los demás apartados estudiaremos detalladamente la implementación de esta técnica en el apartado de la codificación de la aplicación.

16.7.5. La detección de colisiones

Como ya expusimos en el tema anterior la detección y gestión de colisiones es fundamental para crear un videojuego de calidad. Vamos a aplicar las técnicas estudiadas teniendo en cuenta la relación respuesta/rendimiento del sistema.

En este caso vamos a utilizar un algoritmo simple de colisiones que da un resultado bastante fiable a la hora de ejecutar el videojuego. No tienes más que probarlo. Se trata de ajustar el rectángulo que envuelve a nuestro personaje lo más posible para que no se produzcan efectos no deseados. Sólo vamos a utilizar un rectángulo por participante ya que la carga de proceso que supone ampliar el número de áreas de colisiones no nos compensa con el resultado obtenido.

Según el tipo de colisión que se produzca deberemos de responder de manera diferente. Si nuestro personaje colisiona con un ítem deberá de eliminarlo del nivel y anotar dicho suceso. La colisión con estos ítems tiene que tener un propósito específico como puede ser el de aumentar nuestra puntuación, pasar de nivel al recogerlos todos o proporcionarnos una vida nueva.

En el caso de colisionar con enemigo deberemos de reflejar dos posibles casos. El primero sería que el personaje muriera. El número de vidas en principio lo contemplaremos como infinito para que puedas completar el videojuego dándole el comportamiento que más te guste. Basta con establecer un contador en el personaje que disminuya por colisión con el enemigo e incluir en el método al que llamamos cuando existe una colisión una llamada a salir al menú cuando sus vidas lleguen a cero.

El segundo caso a contemplar es que nuestro personaje esté en un determinado estado, golpeando por ejemplo, y procedamos a la eliminación del adversario en vez de la del protagonista. Dependiendo del comportamiento que vayamos a reflejar tendremos que avisar de esta colisión a una clase o a otra.

16. Un ejemplo del desarrollo software de un videojuego

Todos los detalles sobre las decisiones de diseño e implementación las veremos en el diseño y codificación del juego.

16.8. ¿Por qué Programación Orientada a Objetos?

La realización de un programa es la respuesta a un problema. Para resolver un problema tenemos que descomponerlo en casos más sencillos y en tareas individuales para luego volver a construir, a unir, todas estas partes dándole solución al problema.

El criterio de decomposición del sistema ha sido el funcional identificando las funciones del sistemas y sus partes teniendo así partes más simples. Este método da buen resultado cuando dichas funciones están bien definidas y son estables en el tiempo. Si un sistema recibe cambios estructurales grandes esta descomposición en funciones se tambalea teniendo que, seguramente, hacer cambios profundos en las aplicaciones que hayamos desarrollado.

La programación orientada a objetos integra las estructuras de datos y las funciones o métodos asociadas a ellas. Las funcionalidades se traducen en colaboraciones entre objetos que se realizan dinámicamente y las estructuras del programa no se ven amenazadas por un cambio en el mismo.

Uno de los principales motivos de utilizar programación orientada a objetos es la adecuación de este modelo a la programación de videojuegos. Podemos distinguir perfectamente los objetos que componen el videojuego y las funciones o capacidades asociadas a ellos lo que nos permite que este enfoque sea perfecto para crear una abstracción mediante clases del problema.

Además utilizamos programación orientada a objetos por:

- La orientación a objetos se aproxima más a la forma de pensar de las personas. Esto lo hace más comprensible y fácil de aplicar.
- Proporciona abstracción, ya que en cada momento se consideran sólo los elementos que nos interesan descartando los demás.
- Aumenta la consistencia interna al tratar los atributos y las operaciones como un todo.
- Permite expresar características comunes sin repetir la información.
- Facilita la reutilización de diseños y códigos.

16.8. ¿Por qué Programación Orientada a Objetos?

- Facilita la revisión y modificación de los sistemas desarrollados.
- Origina sistemas más estables y robustos.
- Se ha empleado con éxito para desarrollar aplicaciones tan diversas como compiladores, interfaces de usuario, sistemas de gestión de bases de datos, simuladores y, sobre todo, videojuegos.

16.8.1. Características de la Orientación a Objetos

Hoy en día el paradigma de la orientación a objetos ofrece una vista global de la Ingeniería del Software muy importante para realizar un desarrollo de calidad. Se trata de abordar los temas alrededor de los objetos, los participantes, de la aplicación y no entorno a las estructuras de datos.

Vamos a presentar brevemente las características de la orientación a objetos:

Abstracción: Las clases son la abstracción de un conjunto de objetos del mismo tipo.

Encapsulamiento: Permite proteger y englobar a los datos y la funcionalidad de dichos datos en una estructura que nos permite tenerlos asociados.

Ocultación de datos: Esta estructura permite tener varios tipos de datos. Entre ellos están los públicos y los privados ocultos al usuario.

Generalización: Una clase puede abstraerse en una clase más general y viceversa. Esto nos permite tener varios tipos de clases hijas de una clase madre según un discriminante con un comportamiento común o individual.

Polimorfismo: Nos permite funciones con el mismo nombre que se diferencien, por ejemplo, en el tipo de parámetros, que tengan comportamientos totalmente diferentes.

Clases y objetos: Un objeto es una instancia de una clase o lo que es lo mismo, es una clase inicializada. El objeto está próximo al mundo real mientras que la clase es una abstracción de éste.

Herramientas: Son cada una de las funcionalidades que nos aporta una clase para trabajar sobre tipos de datos como pueden ser sus atributos.

Atributos: Los atributos de un objeto son aquellas variables que definen dicho objeto. Las funciones de la clase deben de permitir trabajar con ellos para realizar las modificaciones pertinentes o simplemente para ser consultados ya que son la parte fundamental de dicho objeto.

16. Un ejemplo del desarrollo software de un videojuego

Todas estas propiedades y características de la programación orientada a objetos la hacen ideal para la programación de videojuegos. Este tipo de programación sobre un modelo de proceso de desarrollo en espiral que nos permita realizar un desarrollo incremental y el patrón de diseño UML es todo lo que necesitamos para enfrentarnos a la construcción de un software de calidad.

16.9. Modelado

El modelado de la aplicación comprende el análisis y el diseño del software que debemos de realizar antes de escribir el código. Creamos un conjunto de modelos, como si fuesen planos a seguir del software, que nos permiten describir distintos aspectos como los requisitos que debe cumplir, la estructura a utilizar y el comportamiento que debe de tener dicho sistema.

Es fundamental el uso de modelos para integrar nuestra aplicación en un proceso de ingeniería que nos permite crear un producto de calidad. Como podrás comprobar el modelado conlleva una preparación previa y un coste de tiempo considerable. Estos “inconvenientes” son compensados gracias a que el modelado proporciona un aumento de la productividad y calidad del software.

El modelado nos proporciona la documentación necesaria para desarrollar, implementar y mantener el proyecto así como un punto de partida para diseñar los planes de prueba del software.

La utilidad del modelado la podemos resumir en:

- Mapa o visualización de cómo es o cómo queremos que sea el sistema.
- Nos sirve para especificar el comportamiento y la estructura del sistema.
- Es una guía que nos permite encauzar la construcción del sistema software.
- Es una documentación válida acerca de las decisiones tomadas para la construcción del software.

La elección del modelado a utilizar es un aspecto crucial para el desarrollo software. El modelo a seguir tiene que ajustarse a nuestras necesidades reales. Un único modelo no es suficiente para analizar y diseñar un sistema. El desarrollo de un sistema se afronta mejor desde un conjunto de pequeños modelos casi independientes que nos proporcionen diferentes puntos de vista sobre el conjunto del sistema.

16.10. Especificación de los requisitos del sistema

En nuestro caso la elección es simple. Vamos a realizar una aplicación en C++ orientada a objetos por lo que vamos a utilizar el modelado que mejor se ajusta a esta tipología, el UML.

El UML es un lenguaje de especificación para definir un sistema software que detalla la funcionalidad del sistema y documenta el proceso de desarrollo del mismo. UML cuenta con varios tipos de diagramas que veremos a continuación.

En el mundo del videojuego es importantísimo realizar un modelado adecuado. Muchos tipos de juegos son prácticamente sistemas de tiempo real que necesitan un gran nivel de refinamiento mientras que otros son demasiado complejos para abordar su implementación sin un “mapa” que nos guíe en el proceso de codificación. Aunque para neófitos en la materia puede parecer una pérdida de tiempo está demostrado que un buen planteamiento del problema, llevado a cabo por el modelado, es fundamental para la creación de un software de calidad así como para que el resultado de nuestro desarrollo cumpla unas condiciones de mantenibilidad y ampliación que sería una tarea casi imposible sin dicho mapa del software.

Vamos a hacer una introducción al proceso de desarrollo software necesario para la creación del videojuego especificando toda y cada una de las partes necesarias del proceso del mismo.

16.10. Especificación de los requisitos del sistema

En esta toma primera toma de contacto con el software que vamos a desarrollar tenemos que hacer una lista detallada y lo más completa posible de los requisitos que debe de cumplir el software. Para esto debemos de recabar la máxima información posible. Existen distintas técnicas para realizar esta tarea (como entrevistas, prototipado, ...). En este caso será un proceso intrínseco en el que dedicaremos un tiempo a planetarnos que queremos que haga nuestro videojuego. Vamos a seguir un esquema que nos servirá para especificar los requisitos de una forma más metódica.

16.10.1. Requisitos de interfaces externas

Tenemos que describir de forma detallada los requisitos de conexión a otros sistemas hardware o software con los que vamos a interactuar así como los prototipos de las ventanas e interfaz de usuario así como de los informes que

16. Un ejemplo del desarrollo software de un videojuego

se hayan de generar.

El interfaz entre la aplicación y el hardware lo proporciona la librería SDL. Mediante esta librería vamos a acceder a las características necesarias que tenemos que modificar en los distintos dispositivos. Es un aspecto que no tendremos que analizar y diseñar ya que está preestablecido y sólo haremos uso de ello. SDL proporciona el interfaz con el teclado, el ratón, el joystick, el hardware gráfico y los dispositivos de sonido. Para interactuar con el sistema operativo utilizaremos funciones propias del lenguaje C++ ya que las tareas que realizaremos son triviales como reservar memoria o utilizar ficheros de almacenamiento.

Vamos a definir el interfaz entre el videojuego y el usuario. Todas las ventanas de la aplicación podrán ser mostradas a pantalla completa o en formato de ventana con una resolución de 640 x 480 píxeles. Existen tres tipos de ventanas con las que el usuario puede interactuar con la aplicación que pasamos a definir:

- Ventana de Menú: Esta ventana mostrará el título de la aplicación y un menú que nos permita elegir entre editar los niveles, jugar o salir de la aplicación. Este menú debe de tener un ítem o efecto que nos permita conocer sobre qué opción estamos situados. El usuario interactuará con la aplicación mediante las flechas cursoras arriba y abajo para navegar por las opciones y la tecla **enter** para seleccionar una de ellas. El prototipo de la ventana es el de la figura 16.3.



Figura 16.3: Prototipo: Ventana Menú

- Editor de Niveles: Esta ventana nos debe permitir modificar o crear cómodamente los niveles de la aplicación. Debe de tener un botón que nos

16.10. Especificación de los requisitos del sistema

permite limpiar el contenido del nivel así como guardar las modificaciones del mismo o descartarlas. Debe de tener un menú que nos permita seleccionar los elementos que queremos añadir al nivel que tenemos en el área de edición. El dispositivo de entrada para esta ventana será el ratón ya que se adapta perfectamente a las necesidades de esta parte de la aplicación. Debe de existir de indicador que nos permita conocer la posición del cursor en un momento dado así como un elemento que nos resalte qué nivel estamos editando. Para salir del editor dispondremos de un ícono así como la tecla ESC para volver al menú principal. El prototipo de la ventana de edición es el de la figura 16.4.

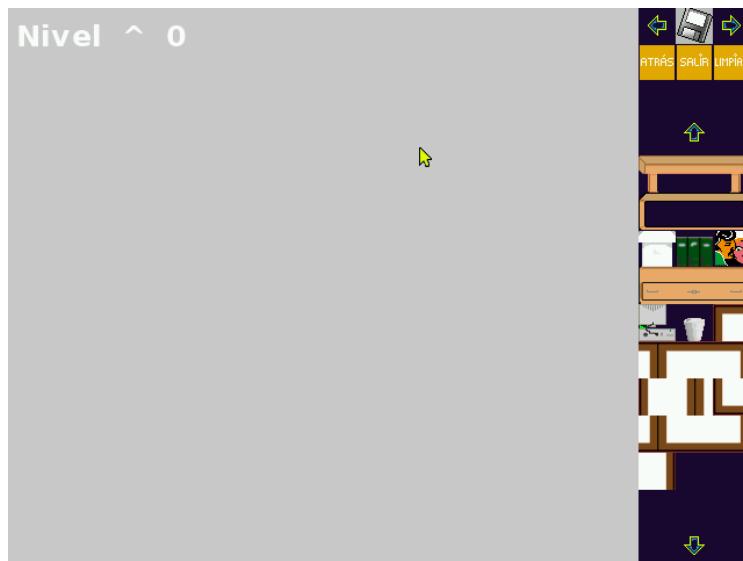


Figura 16.4: Prototipo: Editor de Niveles

- Ventana de Juego: Proporciona el interfaz para interactuar con los niveles diseñados en la aplicación. Es una ventana sin ningún tipo de elemento adicional que nos permite movernos por toda la superficie del mapa del nivel con las restricciones impuestas por el movimiento del personaje principal. La forma de interactuar con esta parte de la interfaz son las teclas cursoras que nos permitirán mover al protagonista por el nivel y la tecla ESC para volver al menú principal. No procede prototipo de esta ventana ya que no hay ningún requisito específico que mostrar.

16.10.2. Requisitos funcionales

En este apartado debemos de responder a la pregunta sobre qué debe de hacer la aplicación o sistema.

Nuestra aplicación tiene dos requisitos principales:

16. Un ejemplo del desarrollo software de un videojuego

- Gestionar los niveles de la aplicación y permitir que el usuario realice las modificaciones que cree pertinentes.
- Permitir que el usuario interactúe con los niveles y juegue con la aplicación.
- Debe de permitir salir de la aplicación en cualquier momento.

Se trata de definir las trazas a grandes rasgos de lo que debe de hacer el sistema. El cómo debe de hacerlo lo indicaremos en otra etapa del desarrollo.

16.10.3. Requisitos de rendimiento

En este apartado tenemos que indicar los requisitos relativos al rendimiento de la aplicación tal como tiempos de respuesta y otros aspectos.

Nuestra aplicación podemos considerarla de tiempo real blando ya que establecemos unos periodos que deben de cumplirse para una correcta funcionalidad de la aplicación pero el margen de error es flexible. Nos marcamos el objetivo de poder mostrar cien frames por segundo. La aplicación deberá estar optimizada sobre el parámetro del tiempo sacrificando el consumo de memoria principal.

16.10.4. Restricciones de diseño

Ahora vamos a especificar aquellas restricciones que se hayan identificado e influyan en el diseño del sistema.

El diseño de la aplicación tiene que primar los tiempos de respuesta sobre el consumo de recursos de espacio como la memoria principal o secundaria. Esta es la principal restricción que tendrá el diseño de nuestro videojuego. Las demás características tendrán que ser diseñadas basándose en este principio.

El grado de cohesión y acoplamiento de las clases puede verse afectado por esta restricción. Es necesario evaluar estos parámetros con el objetivo anteriormente expuesto.

16.10.5. Atributos del sistema software

En este apartado debemos de especificar todos los atributos con los que debe de cumplir nuestra aplicación.

Uno de los requisitos principales de la aplicación es que sea portable entre los sistemas compatibles con SDL. No podremos utilizar código dependiente

del sistema operativo donde desarrollemos nuestra aplicación. Debe de ser un código mantenable y ampliable que podamos mejorar en futuras versiones del mismo perfeccionando todos los aspectos que sean necesarios o que queramos modificar.

La aplicación debe ser robusta y fiable desde su diseño. La seguridad no es un tema relevante ya que no vamos a requerir ningún tipo de dato que sea conflictivo o susceptible de ser protegido.

16.10.6. Otros requisitos

En este apartado especificaremos todos los requisitos, que por su naturaleza, no hayan podido ser incluidos en otros apartados del tutorial.

Para este videojuego la aplicación, su diseño y codificación deberán de tener una fuerte componente didáctica que integre el temario visto en los distintos capítulos del tutorial con el fin de que el lector de dicho temario pueda tomar las implementaciones expuestas en este ejemplo como guía para elaborar su propio videojuego.

16.11. Análisis

En el análisis del sistema se realizan los modelos que nos ayudan a analizar y especificar el comportamiento del sistema. Existen varios tipos de modelado nosotros vamos a utilizar un enfoque orientado a objetos usando la notación UML.

En este apartado vamos a realizar un análisis del problema a resolver con el objetivo de describir qué debe de hacer el sistema software y no, todavía, cómo lo hace. Es un paso más entre el problema y su resolución final. Es fundamental a la hora de desarrollar un videojuego hacer un análisis detallado de lo que queremos que haga nuestro videojuego. Es importante que creemos un sistema potente pero que, si es uno de nuestros primeros proyectos, no se nos escape de las manos.

El análisis será el primer paso que nos guiará en la construcción de nuestro videojuego. Para este paso vamos a utilizar el modelado UML. UML es el lenguaje de modelado más utilizado en la actualidad. Es un lenguaje gráfico para visualizar, especificar, construir y documentar sistemas software. Nos permite especificar el sistema, que no describir métodos o procesos, y se define sobre una serie de diagramas que estudiaremos a continuación.

16. Un ejemplo del desarrollo software de un videojuego

16.11.1. Modelo de Casos de Uso

El modelo de casos de uso de UML especifica que comportamiento debe de tener el sistema software. Representa los requisitos funcionales del sistema centrándose en qué hace y no en cómo lo hace. Este modelo no es orientado a objetos por lo que podemos utilizarlo en proyectos que no lo sean.

16.11.1.1. Casos de uso

El caso de uso está compuesto de:

Conjunto de secuencia de acciones Cada una de estas secuencias representa un posible comportamiento del sistema.

Actores Roles o funciones que pueden adquirir cada usuario, dispositivo u otro sistema al interaccionar con nuestro sistema. El tiempo puede ser considerado un sistema por lo que puede ser un actor. Los actores no son parte del sistema en sí. Hay dos tipos de actores

Principales Demanda al sistema el cumplimiento de un objetivo.

Secundarios Se necesita de ellos para cumplir con un objetivo.

Variantes Casos especiales de comportamiento.

Escenarios Es una secuencia de interacciones entre actores y el sistema. Está compuesto de un flujo principal y flujos alternativos o excepcionales. Es una instancia de un caso de uso.

Los casos de uso son iniciados por un actor con un objetivo concreto y es completado con éxito cuando el sistema cumple con dicho objetivo. Existen secuencias alternativas que nos pueden llevar al éxito o al fracaso en el cumplimiento del objetivo. El conjunto completo de los casos de uso especifica todas las posibles formas de usar el sistema que no es más que el comportamiento requerido de dicho sistema.

Los casos de uso ofrecen un medio eficiente para capturar requisitos funcionales centrándose en las necesidades del usuario. Dirigen el proceso al desarrollo a que las actividades que conlleva este desarrollo se realizan a partir de los casos de uso.

Existen tres tipos de relaciones en los casos de uso:

Generalización Un caso de uso hereda el comportamiento y significado de otro.

Inclusión Un caso de uso incorpora explícitamente el comportamiento de otro en algún lugar de su secuencia.

Extensión Un caso de uso amplía la funcionalidad de otro incluyendo implícitamente el comportamiento de otro caso de uso.

16.11.1.2. Diagramas de casos de uso

Una vez estudiada esta pequeña introducción a los casos de uso vamos a realizar el diagrama que representa la funcionalidad de nuestro videojuego de ejemplo. Es importante que dediques un tiempo importante a este proceso antes de empezar a implementar el videojuego porque entre la documentación generada y la vista general del proyecto que te proporciona este diagrama te ahorrarán muchos quebraderos de cabeza.

Para obtener los casos de uso seguiremos el siguiente procedimiento:

1. Identificaremos a los usuarios del sistema y los roles que juegan en dicho sistema.
2. Para cada role identificaremos todas las maneras de interactuar con el sistema.
3. Creamos un caso de uso para cada objetivo que queramos cumplir.
4. Estructuraremos los casos de uso.

El diagrama de casos de uso es una ayuda visual pero lo realmente importante se encuentra en la descripción de los casos de uso. Vamos a seguir el esquema anterior para crear nuestro diagrama de casos de uso.

El usuario de nuestro sistema es único. El mismo será el que edite los niveles así como juegue a sus creaciones. No hay diferenciación de roles ya que siempre realizará la función de usuario del sistema software.

Ahora vamos a identificar las maneras que tiene el usuario de interactuar con el sistema. Son tres. La primera se produce cuando el jugador decide jugar a los niveles creados. La segunda es cuando el usuario decide editar los niveles del videojuego para modificar algún detalle del mismo o crear niveles nuevos. La tercera está relacionada con la petición de salir de la propia aplicación. Para cada uno de estos objetivos crearemos un caso de uso que mostraremos en el diagrama de la figura 16.5.

16.11.1.3. Descripción de los casos de uso

La descripción de un caso de uso es un texto que puede ser expresado de varias formas. Nosotros vamos a utilizar una notación formal usando

16. Un ejemplo del desarrollo software de un videojuego

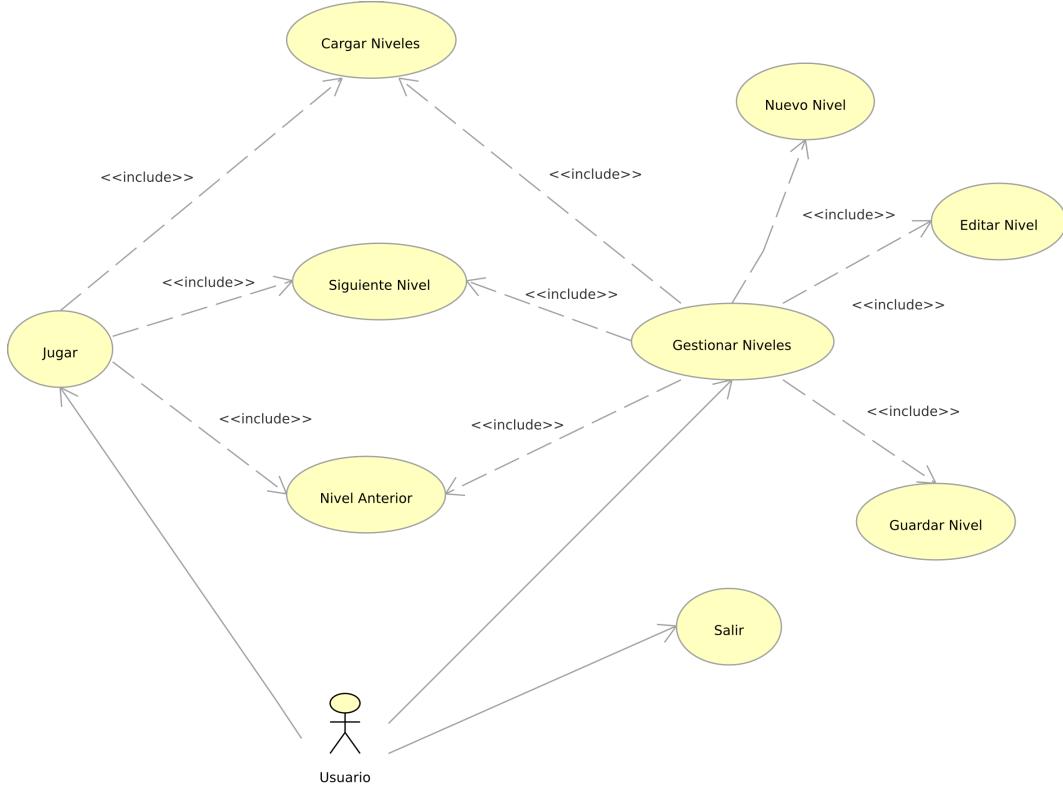


Figura 16.5: Diagrama de casos de uso: Videojuego de ejemplo

plantillas. Este texto debe ser legible y comprensible por un usuario que no sea experto. Para describir los casos de uso vamos a utilizar una plantilla en formato completo que nos permita dejar fuera de toda duda razonable los casos de uso requeridos en nuestro videojuego.

Vamos a proceder a describir los casos de uso de nuestro videojuego:

DESCRIPCIÓN CASO DE USO: Cargar Niveles

Caso de uso: Cargar Niveles

Descripción: Carga el fichero de niveles de juego en la aplicación desde un fichero de datos.

Actores: Usuario

Precondiciones: Para realizar dicha acción deben de existir niveles guardados en el fichero de niveles.

Postcondiciones: Los datos del primer nivel almacenado serán mostrados en pantalla para poder interaccionar con dicho nivel.

Escenario principal: Describimos el escenario principal:

1. El usuario demanda la carga de los niveles.
2. El sistema carga los niveles.
3. El sistema comprueba que el fichero existe.
4. El sistema comprueba que al menos el primer nivel existe.
5. El sistema carga el nivel en la aplicación.
6. El sistema muestra el primer nivel al usuario.

Extensiones (Flujo alternativo): Describimos el flujo alternativo:

- 1a Carga de nivel automática por secuencialidad del juego.
- 1b Carga de nivel a demanda del usuario mediante un interfaz.
- 3a El fichero no existe en el sistema.
 - a) El sistema muestra el error y cierra el sistema.
- 4a El nivel no existe en el fichero.
 - a) El sistema muestra el error y cierra el sistema.

16. Un ejemplo del desarrollo software de un videojuego

DESCRIPCIÓN CASO DE USO: Siguiente Nivel

Caso de uso: Siguiente Nivel

Descripción: Muestra el nivel siguiente al cargado actualmente.

Actores: Usuario

Precondiciones: Deben de existir niveles cargados en el sistema.

Postcondiciones: El siguiente nivel será mostrado en pantalla.

Escenario principal: Describimos el escenario principal:

1. El usuario demanda la carga del siguiente nivel.
2. El sistema comprueba que existe un siguiente nivel.
3. El sistema muestra el nivel al usuario.

Extensiones (Flujo alternativo): Describimos el flujo alternativo:

- 1a Carga de nivel automática por secuencialidad del juego.
- 1b Carga de nivel a demanda del usuario mediante un interfaz.
- 2a El nivel no existe en el sistema.
 - a) El sistema muestra el error y no se avanza de nivel.

No funcional: El interfaz del usuario proporcionará dos botones de navegación por los niveles del fichero fácilmente identificables.

DESCRIPCIÓN CASO DE USO: Nivel Anterior

Caso de uso: Nivel Anterior

Descripción: Muestra el nivel anterior al cargado actualmente.

Actores: Usuario

Precondiciones: Deben de existir niveles cargados en el sistema.

Postcondiciones: El nivel anterior será mostrado en pantalla.

Escenario principal: Describimos el escenario principal:

1. El usuario demanda la carga del nivel anterior.
2. El sistema comprueba que existe un nivel anterior.
3. El sistema muestra el nivel al usuario.

Extensiones (Flujo alternativo): Describimos el flujo alternativo:

- 1a El nivel no existe en el sistema.
 - a) El sistema muestra el error y no se retrasa de nivel.

16. Un ejemplo del desarrollo software de un videojuego

DESCRIPCIÓN CASO DE USO: Nuevo Nivel

Caso de uso: Nuevo Nivel

Descripción: Crea un nivel para ser editado.

Actores: Usuario

Precondiciones: Debe estar cargado el fichero de niveles.

Postcondiciones: Muestra un nivel vacío a editar por el usuario.

Escenario principal: Describimos el escenario principal:

1. El usuario demanda un nuevo nivel.
2. El sistema crea dicho nivel.
3. El sistema muestra el nivel al usuario.

Extensiones (Flujo alternativo): Describimos el flujo alternativo:

- 2a) El nivel no se puede crear en el sistema.
a) El sistema muestra el error y no se continúa.

DESCRIPCIÓN CASO DE USO: Editar Nivel

Caso de uso: Editar Nivel

Descripción: Editamos un nivel de los disponibles en el sistema.

Actores: Usuario

Precondiciones: El nivel a editar debe estar cargado en el sistema.

Postcondiciones: Modificamos un determinado nivel.

Escenario principal:

Escenario principal: Describimos el escenario principal:

1. El usuario demanda editar un nivel.
2. El sistema prepara dicho nivel.
3. El usuario edita el nivel.

Extensiones (Flujo alternativo): Describimos el flujo alternativo:

- 2a El nivel no está disponible en el sistema.
a) El sistema muestra el error y se reinicia la aplicación.

Cuestiones pendientes: Se debe guardar la modificación del nivel para que esté disponible en otros escenarios.

16. Un ejemplo del desarrollo software de un videojuego

DESCRIPCIÓN CASO DE USO: Guardar Nivel

Caso de uso: Guardar Nivel

Descripción: Guarda el nivel que muestra el sistema en el fichero de niveles.

Actores: Usuario

Precondiciones: Debe de existir un nivel cargado que guardar.

Postcondiciones: Guarda el nivel en el fichero de niveles.

Escenario principal: Describimos el escenario principal:

1. El usuario demanda guardar un nivel.
2. El sistema guarda el nivel.
3. El sistema muestra el resultado de la operación.

Extensiones (Flujo alternativo): Describimos el flujo alternativo:

- 3a El nivel no puede ser guardado.
 - a) El sistema muestra el error.
- 3b El nivel es guardado.
 - a) El sistema muestra OK.

DESCRIPCIÓN CASO DE USO: Jugar

Caso de uso: Jugar

Descripción: Nos permite interactuar con los niveles creados en el sistema.

Actores: Usuario

Precondiciones: Deben existir niveles en el sistema.

Postcondiciones: Se muestran los niveles y se nos permite interactuar con ellos secuencialmente.

Escenario principal: Describimos el escenario principal:

1. El usuario demanda interactuar con el sistema.
2. El sistema carga un nivel.
3. El usuario interacúa con el sistema.

Extensiones (Flujo alternativo): Describimos el flujo alternativo:

- 2a El nivel no puede ser cargado.
 - a) El sistema muestra el error y reinicia la aplicación.
- 3a El usuario completa un nivel.
 - a) Se vuelve al paso 3 y se carga otro nivel.
- *a El usuario decide terminar.
 - a) Se cierra la aplicación.

No funcional: Se dispondrá de un dispositivo de juegos para que el usuario pueda interactuar con el sistema.

16. Un ejemplo del desarrollo software de un videojuego

DESCRIPCIÓN CASO DE USO: Gestionar Niveles

Caso de uso: Gestionar Niveles

Descripción: Gestiona el fichero de niveles del sistema.

Actores: Usuario

Precondiciones:

Postcondiciones: Realiza una operación sobre el fichero de niveles.

Escenario principal: Describimos el escenario principal:

1. El usuario demanda una operación con los niveles.
2. El sistema realiza la operación.
3. El sistema muestra el resultado de la operación.

Extensiones (Flujo alternativo): Describimos el flujo alternativo:

- 2a La operación demandada es crear un nivel.
 - a) Incluir (Nuevo Nivel)
- 2b La operación demandada es cargar un nivel.
 - a) Incluir (Cargar Nivel)
- 2c La operación demandada es pasar de nivel.
 - a) Incluir (Siguiente Nivel)
- 2d La operación demandada es volver a un nivel anterior.
 - a) Incluir (Nivel Anterior)
- 2e La operación demandada es guardar un nivel.
 - a) Incluir (Guardar Nivel)
- 2f La operación demandada es editar un nivel.
 - a) Incluir (Editar Nivel)

16.11.2. Modelo conceptual de datos en UML

El siguiente paso del análisis de la aplicación que estamos realizando es realizar el modelo conceptual de datos. Este tipo de modelado sirve para especificar los requisitos del sistema y las relaciones estáticas que existen entre ellos.

Para realizar este modelo se utiliza como herramienta los diagramas de clase. En estos diagramas representamos las clases de objetos, las asociaciones entre dichas clases, los atributos que componen las clases y las relaciones de integridad.

16.11.2.1. Conceptos básicos

Vamos a presentar varios conceptos, que aunque ya los hemos utilizado en el tutorial, no está demás recordarlos. El primero de ellos es el concepto de **objeto**. Un objeto no es más que una entidad que existe en el mundo real bien distingible de las demás entidades. Un ejemplo de objeto es un bolígrafo, una persona, una factura...

Los objetos de un mismo tipo se abstraen en clases. Estas clases describen a un conjunto de objetos con las mismas propiedades, comportamientos comunes, con relaciones idénticas con los otros objetos y una semántica común.

Existen varios tipos de relaciones entre las distintas clases. Estas relaciones normalmente tienen asociadas unas restricciones de participación en dichas relaciones. Algunos ejemplos de estos tipos de relaciones son las agregaciones, las composiciones, las generalizaciones... El tema de las relaciones entre clases es un tema muy extenso e interesante en el que es necesario que profundices para poder realizar un diseño de calidad.

16.11.2.2. Descripción diagramas de clases conceptuales

En este punto debemos de tener claro que clases vamos a necesitar para implementar nuestro videojuego. Es el momento de presentarlas describiendo brevemente las características de cada una de ellas para, posteriormente, realizar el diagrama de clases.

Imagen: Esta clase nos permite controlar todos los aspectos referentes a las imágenes necesarias en la aplicación.

Música: Esta clase nos permite controlar la música del videojuego.

Sonido: La clase sonido nos permite gestionar los sonidos del videojuego.

Fuente: Nos permite utilizar una fuente para rotular en la aplicación.

Texto: Con esta clase creamos y controlamos los textos necesarios en la aplicación.

Galería: La galería englobará todos los contenidos multimedia de la aplicación. Será la encarga de gestionar todos los aspectos de este contenido, desde la inicialización de los elementos, hasta la utilización de éstos por parte de las demás clases de la aplicación.

Participante: Esta clase será el interfaz de todos los participantes del juego. Nos permitirá realizar las actualizaciones lógicas y gráficas de todos los elementos existentes en un nivel.

16. Un ejemplo del desarrollo software de un videojuego

Protagonista: Subclase de Participante que nos permitirá controlar los aspectos del personaje principal del juego.

Item: Clase heredada de Participante que nos permite controlar los objetos e ítems del nivel y su comportamiento.

Enemigo: Clase hija de Participante que controla todos los aspectos de los enemigos.

Control Movimiento: Clase auxiliar que nos permite establecer un tipo de movimiento a diferentes clases del sistema.

Control Animación: Controla las animaciones internas de todos los elementos de la aplicación a partir de una rejilla de imágenes.

Control Juego: Lleva el control de la lógica y los elemertos del juego. Gestiona las colisiones así como los elementos existentes en un determinado nivel, sus actualizaciones lógicas y gráficas.

Menu: Esta clase controla el menú principal de la aplicación y sus distintas opciones.

Juego: Esta clase controla los aspectos relevantes de la aplicación en tiempo de juego.

Editor: Con esta clase controlaremos los aspectos relevantes de la edición y gestión de niveles.

Interfaz: Esta clase hace de interfaz entre las diferentes escenas de la aplicación para la navegación de ellas que no son otras que Menu, Juego y Editor.

Nivel: Controla las propiedades y proporciona las capacidades para trabajar con niveles.

Ventana: Esta clase nos permitirá mostrar sólo una porción adecuada de la superficie del nivel

Universo: Es la clase principal que interrelaciona todos los aspectos del juego.

Teclado: Esta clase nos permite gestionar la entrada de teclado.

Apuntador: Con la clase Apuntador podremos utilizar el dispositivo de ratón en el editor de niveles.

16.11.3. Diagrama de clases

Una vez descritas todas las clases y las relaciones entre ellas vamos a presentar los diagramas de clases que nos van a permitir tener un mapa conceptual de la globalidad de la aplicación para afrontar su desarrollo. Vamos a presentar los diagramas por subconjuntos. Así podremos estudiar los detalles de cada uno de ellos. Para terminar incluiremos un diagrama donde observar las relaciones entre todas las clases que intervendrán en la aplicación.

Para descomponer el diagrama hemos tomado el criterio de funcionalidad. Mostraremos subconjuntos de clases que tengan un fin común.

16. Un ejemplo del desarrollo software de un videojuego

El primer subdiagrama que presentamos incluye a las clases que gestionarán los elementos multimedia de la aplicación. Podemos decir que la clase principal de este grupo es la clase Galería que estará compuesta de las clases que están asociadas a ella. El diagrama es el de la figura 16.6.

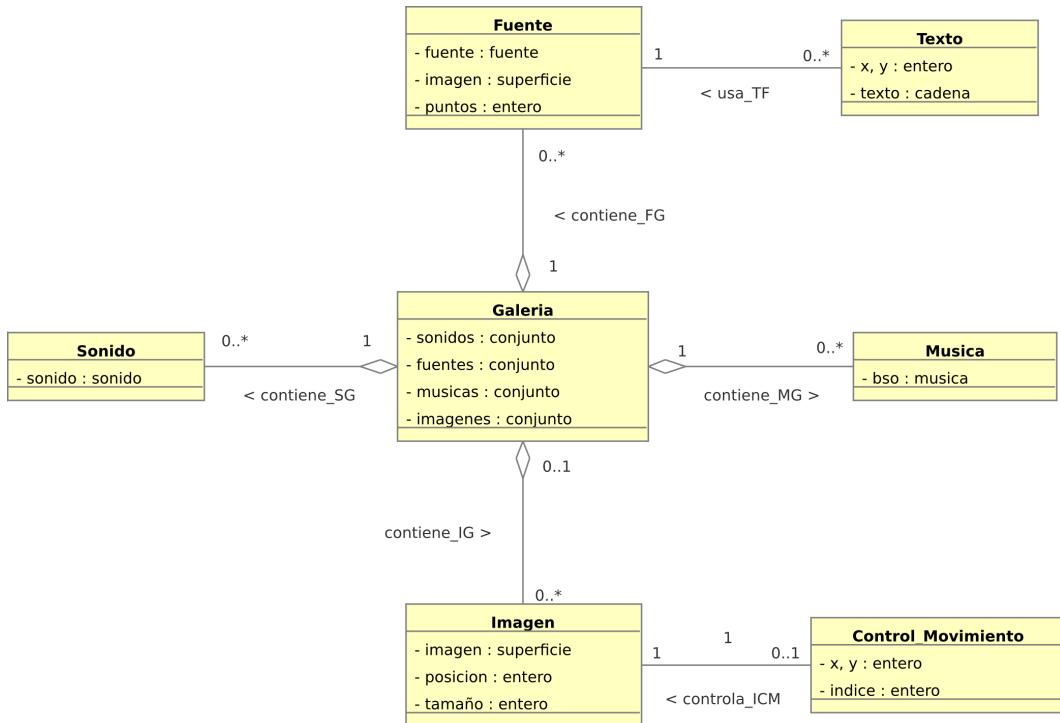


Figura 16.6: Diagrama de clases: Componentes multimedia

En este diagrama se muestran siete clases. Como hemos comentado Galería es la clase que gestiona a las demás.

16.11. Análisis

El segundo diagrama que presentamos relaciona a las clases que nos permiten crear y modificar niveles en la aplicación. En este caso la clase que permite relacionar a las demás componentes del diagrama es la clase Universo aunque la que contendrá toda la lógica para realizar esta tarea es la clase Editor.

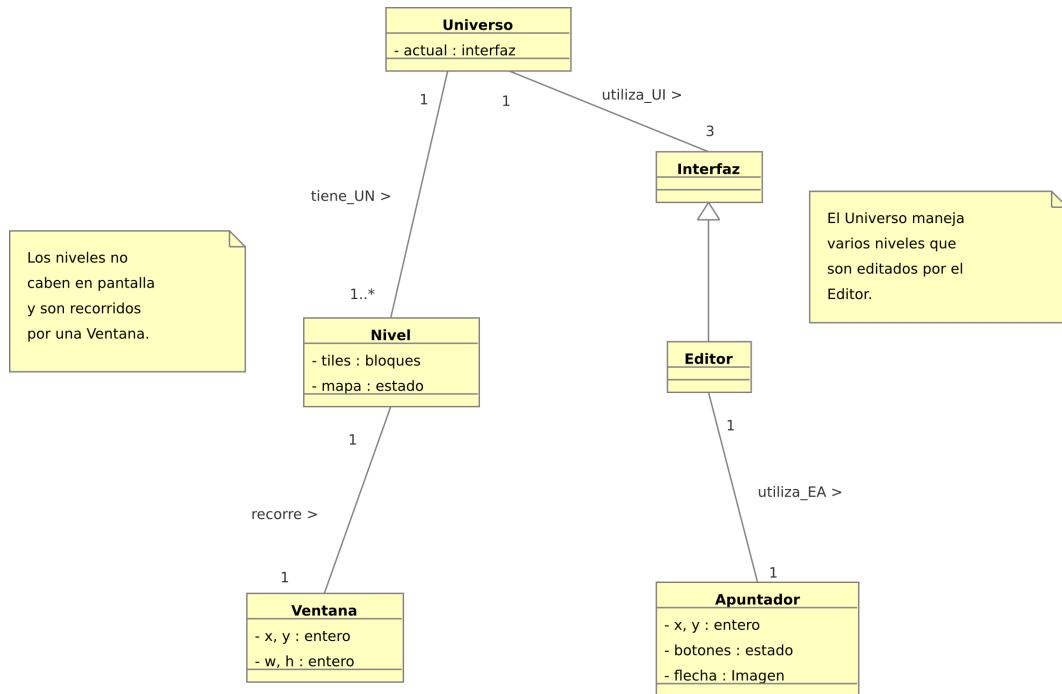


Figura 16.7: Diagrama de clases: Editando niveles

16. Un ejemplo del desarrollo software de un videojuego

El tercer diagrama que vamos a estudiar engloba las clases que están asociadas directamente con la clase Participante que recoge todos los elementos activos que participan en los niveles del juego.

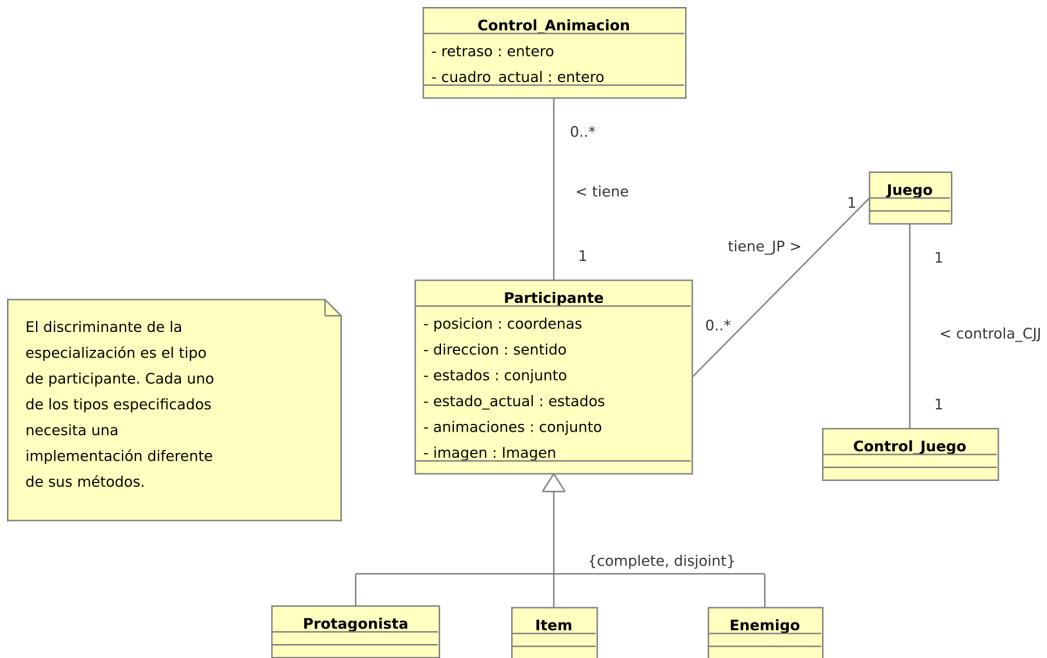


Figura 16.8: Diagrama de clases: Participantes

16.11. Análisis

Ahora vamos a estudiar uno de los diagramas de clases más importantes del análisis. Se trata de la clase Universo encargada de relacionar las clases del videojuego, los elementos de control y los elementos multimedia que representa a cada una de estas partes.

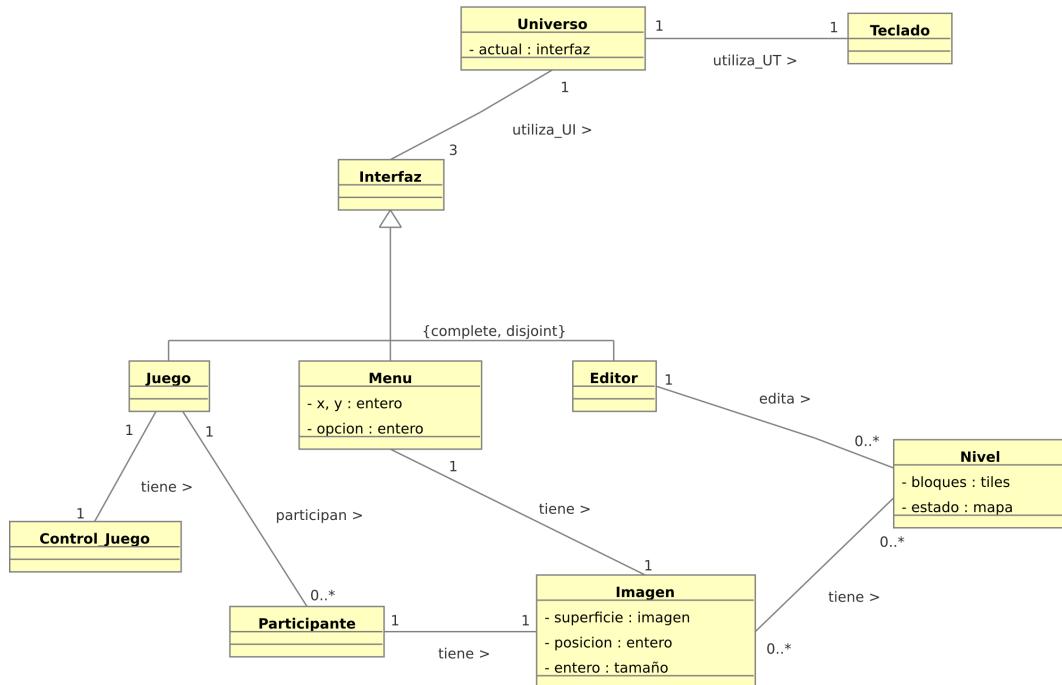


Figura 16.9: Diagrama de clases: Universo

16. Un ejemplo del desarrollo software de un videojuego

Para terminar con este apartado vamos a presentar el diagrama de clases que relaciona todos los demás diagramas. Al ser excesivamente grande no vamos a detallar cada uno de sus componentes ya que esta tarea la realizamos en los anteriores diagramas.

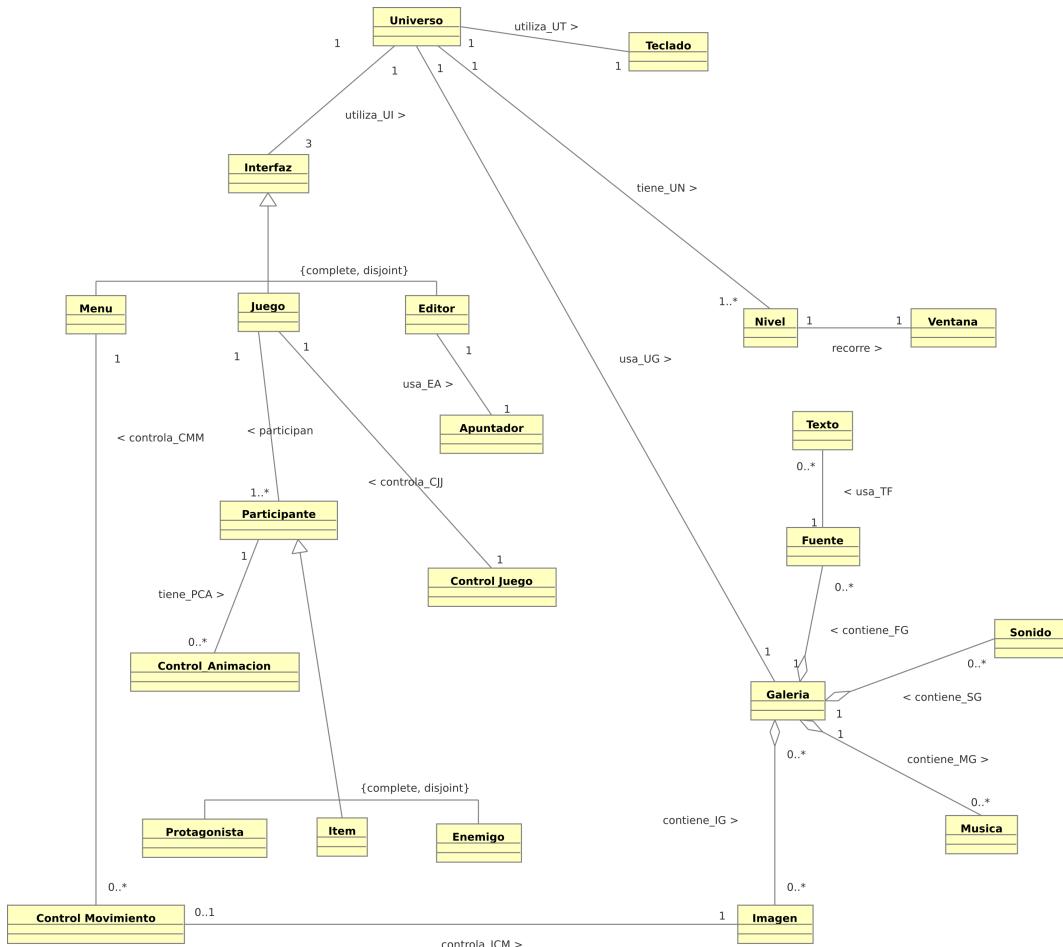


Figura 16.10: Diagrama de clases: Diagrama Global

16.11.4. Modelo de comportamiento del sistema

El modelo del comportamiento especifica cómo debe de actuar un sistema. El sistema a considerar es el que engloba a todos los objetos. Este modelo consta de dos partes. La primera es el diagrama de secuencia de sistema que nos muestra la secuencia de eventos entre los actores y el sistema. La segunda parte de este modelo está compuesta por los contratos de las operaciones del sistema que efecto que deben producir las operaciones del sistema.

16.11.4.1. Diagramas de secuencia del sistema y los Contratos de operaciones

Una vez descrito un caso de uso representamos mediante un diagrama de secuencia del sistema dicho caso de uso. Este diagrama nos servirá de aproximación visual a los casos de uso como complemento a la descripción anterior.

La principal utilidad y objetivo de este diagrama es permitirnos identificar las operaciones y eventos del sistema. El punto de partida de estos diagramas son los casos de uso que nos permiten identificar qué eventos son los que van de los actores hacia el sistema. Tendremos que definir un diagrama de secuencia para cada escenario relevante de un caso de uso. Para identificar estos escenarios en el sistema nos centraremos en los eventos que genera el usuario que hace uso de dicho sistema y que espera alguna operación como respuesta de dicha interacción.

La segunda parte del modelo es la realización de los contratos para las operaciones del sistema. Estos contratos describen el efecto que deben producir las operaciones del sistema. Las operaciones del sistema son operaciones internas que se ejecutan como respuestas a un evento producido, normalmente, por un actor o usuario.

Los contratos tienen unas partes bien diferenciadas: nombre de la operación, responsabilidades, precondiciones, postcondiciones... que utilizaremos en forma de plantilla. Construiremos contratos para operaciones complejas y para aquellas que no quedan claras en el proceso de especificación del sistema.

16.11.4.2. Diagramas de secuencia del sistema y Contratos de las operaciones del sistema

A continuación expondremos todos los diagramas de secuencia del sistema y los correspondientes contratos de las operaciones del sistema.

16. Un ejemplo del desarrollo software de un videojuego

DIAGRAMA DE SECUENCIA: Cargar Niveles

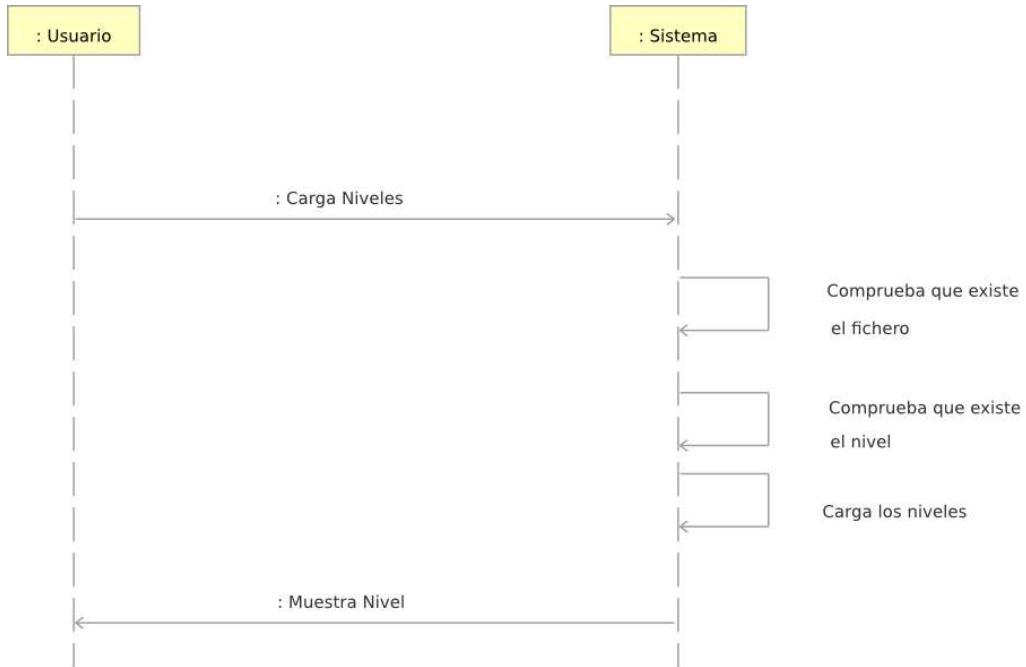


Figura 16.11: Diagrama de secuencia: Cargar Niveles

Contrato de las operaciones

Operación: Cargar Niveles

Responsabilidades: Carga los niveles del juego en memoria principal. Primero comprueba que existe el fichero de niveles y seguidamente que dicho fichero tenga algún nivel. De ser así carga el primero de ellos. Si no existe el fichero o está vacío se muestra el error y se cierra la aplicación.

Precondiciones: Debe de existir un fichero de niveles. Se debe de solicitar la carga de los niveles explícita o implícitamente asociado a alguna acción del usuario.

Postcondiciones:

- Carga en memoria principal el primer nivel de la colección almacenada en el fichero de niveles.
- Si no existe el fichero de niveles muestra un mensaje de error y termina la aplicación.

Operación: Muestra Nivel

Responsabilidades: Mostrar en pantalla el nivel actual de la aplicación. Si no existiese dicho nivel se muestra el error y se cierra la aplicación.

Precondiciones: El nivel a mostrar debe estar cargado en memoria principal.

Postcondiciones:

- Carga en memoria principal el primer nivel de la colección almacenada en el fichero de niveles.
- Si no hay nivel que mostrar se muestra un mensaje de error y se sale de la aplicación.

16. Un ejemplo del desarrollo software de un videojuego

DIAGRAMA DE SECUENCIA: Siguiente Nivel

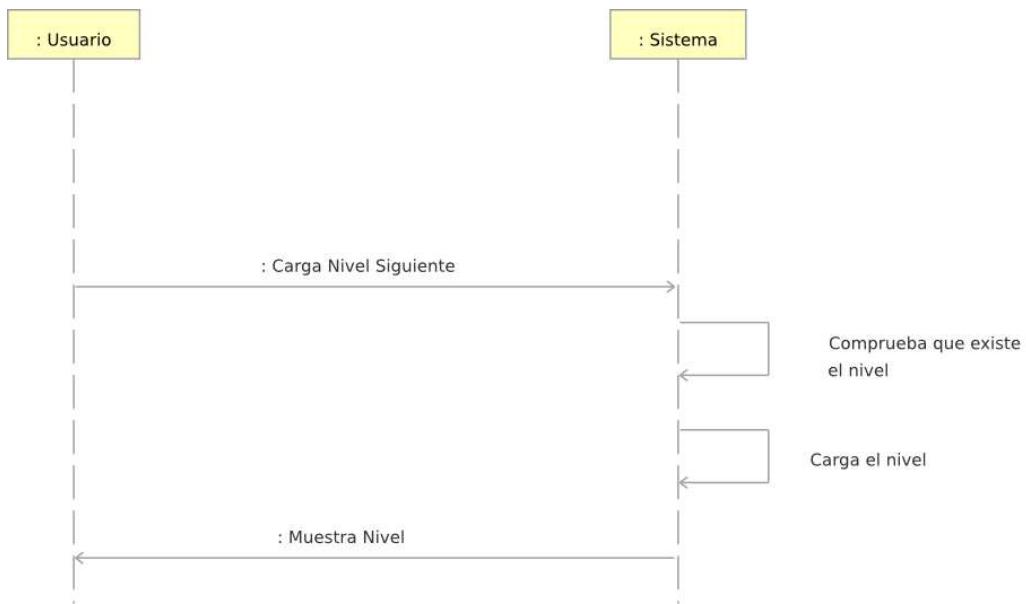


Figura 16.12: Diagrama de secuencia: Siguiente Nivel

Contrato de las operaciones

Operación: Carga Nivel Siguiente

Responsabilidades: Cargar el siguiente nivel al actual. En caso de no existir el siguiente nivel no realiza acción alguna.

Precondiciones: Debe de existir un nivel cargado.

Postcondiciones:

- Prepara el siguiente nivel disponible para ser mostrado.
- En caso no existir no realiza ninguna acción.

Operación: Muestra Nivel

Responsabilidades: Mostrar en pantalla el nivel actual de la aplicación. Si no existiese dicho nivel se muestra el error y se cierra la aplicación.

Precondiciones: El nivel a mostrar debe estar cargado en memoria principal.

Postcondiciones:

- Carga en memoria principal el primer nivel de la colección almacenada en el fichero de niveles.
- Si no hay nivel que mostrar se muestra un mensaje de error y se sale de la aplicación.

DIAGRAMA DE SECUENCIA: Nivel Anterior

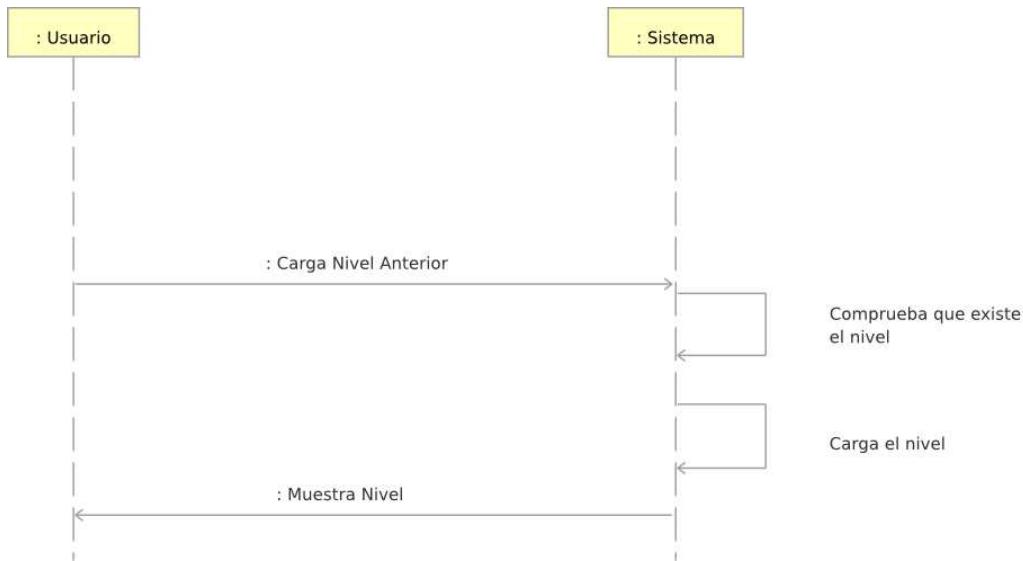


Figura 16.13: Diagrama de secuencia: Nivel Anterior

Contrato de las operaciones

Operación: Carga Nivel Anterior

Responsabilidades: Cargar el siguiente anterior al actual. En caso de no existir el siguiente nivel no realiza acción alguna.

Precondiciones: Debe de existir un nivel cargado.

Postcondiciones:

- Prepara el nivel anterior disponible para ser mostrado.
- En caso no existir no realiza ninguna acción.

Operación: Muestra Nivel

Responsabilidades: Mostrar en pantalla el nivel actual de la aplicación. Si no existiese dicho nivel se muestra el error y se cierra la aplicación.

Precondiciones: El nivel a mostrar debe estar cargado en memoria principal.

Postcondiciones:

- Carga en memoria principal el primer nivel de la colección almacenada en el fichero de niveles.
- Si no hay nivel que mostrar se muestra un mensaje de error y se sale de la aplicación.

16. Un ejemplo del desarrollo software de un videojuego

DIAGRAMA DE SECUENCIA: Nuevo Nivel

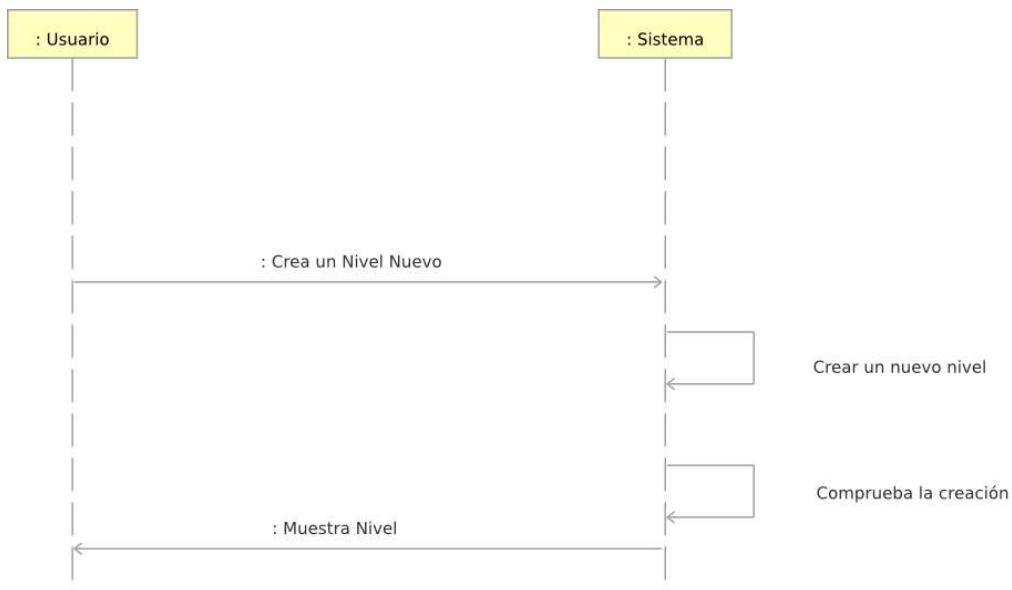


Figura 16.14: Diagrama de secuencia: Nuevo Nivel

Contrato de las operaciones

Operación: Crea un Nivel Nuevo

Responsabilidades: Crear un nivel nuevo al final de la lista de niveles. En caso de no poder alojar el nivel mostrar un mensaje de error.

Precondiciones: Debe de existir el fichero de niveles.

Postcondiciones:

- Prepara un nivel vacío para ser mostrado y editado.
- Si no se puede preparar dicho nivel se muestra un mensaje de error.

Operación: Muestra Nivel

Responsabilidades: Mostrar en pantalla el nivel actual de la aplicación. Si no existiese dicho nivel se muestra el error y se cierra la aplicación.

Precondiciones: El nivel a mostrar debe estar cargado en memoria principal.

Postcondiciones:

- Carga en memoria principal el primer nivel de la colección almacenada en el fichero de niveles.
- Si no hay nivel que mostrar se muestra un mensaje de error y se sale de la aplicación.

DIAGRAMA DE SECUENCIA: Editar Nivel

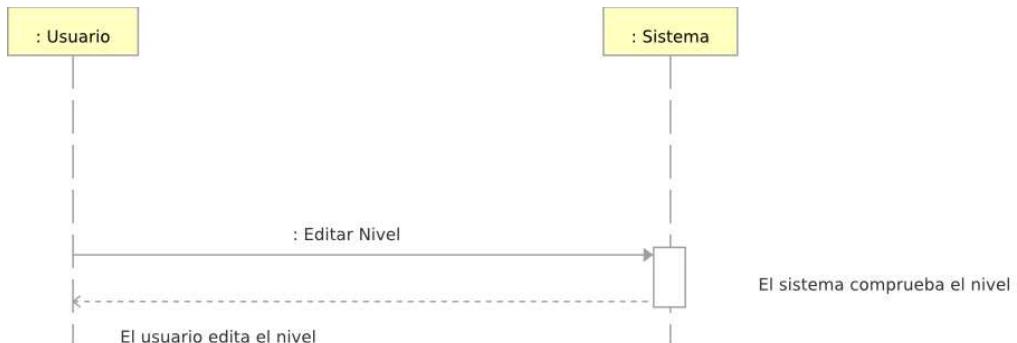


Figura 16.15: Diagrama de secuencia: Editar Nivel

Contrato de las operaciones

Operación: Editar Nivel

Responsabilidades: Prepara un nivel para ser editado y muestra el interfaz.

Precondiciones:

Postcondiciones: En caso de no existir el fichero de niveles se crea.

16. Un ejemplo del desarrollo software de un videojuego

DIAGRAMA DE SECUENCIA: Guardar Nivel

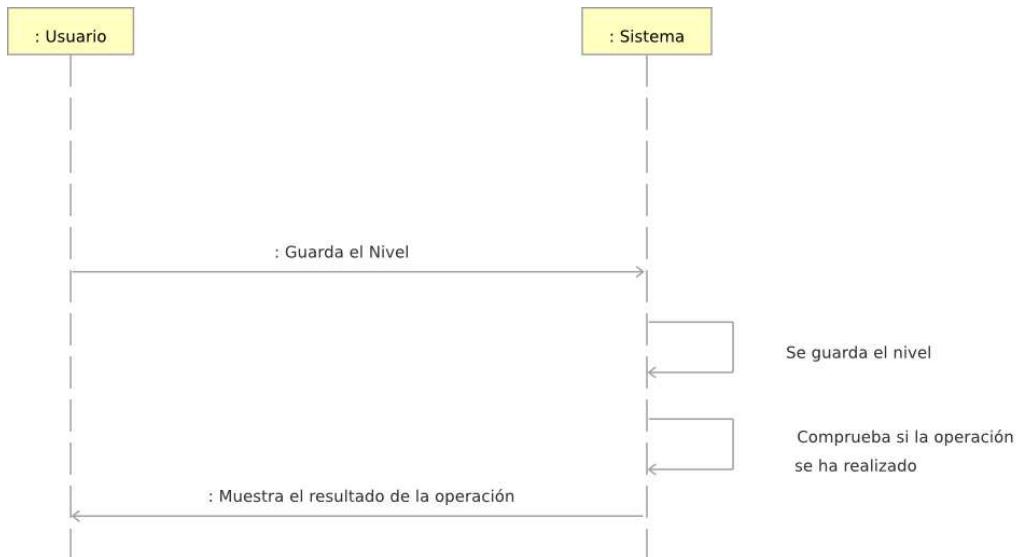


Figura 16.16: Diagrama de secuencia: Guardar Nivel

Contrato de las operaciones

Operación: Guarda el Nivel

Responsabilidades: Guardar el nivel actual en el fichero de niveles. En caso de no poder guardar lo muestra el error.

Precondiciones: Debe de existir un nivel cargado en memoria que guardar.

Postcondiciones:

- Guarda el nivel en el fichero de niveles.
- En caso de no poder realizar la acción el sistema muestra un mensaje.

DIAGRAMA DE SECUENCIA: Jugar

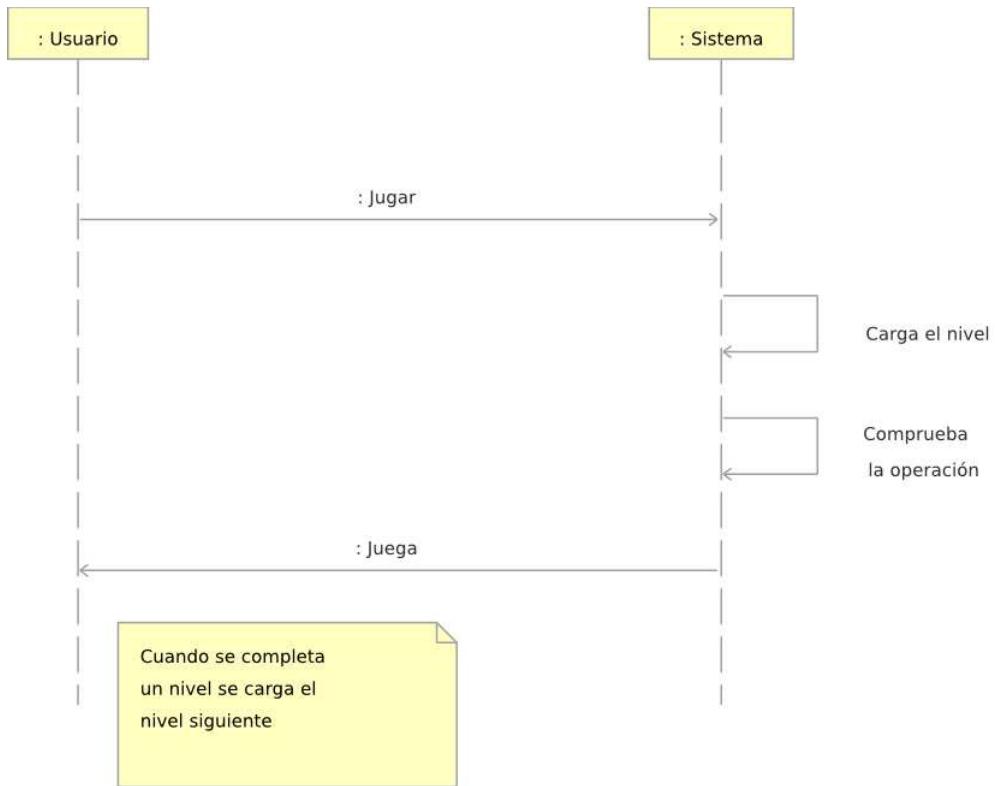


Figura 16.17: Diagrama de secuencia: Jugar

Contrato de las operaciones

Operación: Jugar

Responsabilidades: Prepara el sistema, los niveles y la lógica del juego para una partida. En caso de no poder realizar alguna de las operaciones muestra un mensaje de error y termina la aplicación.

Precondiciones: Deben de existir niveles en el fichero de niveles. Dicho fichero debe existir también. Se debe demandar la acción de jugar.

Postcondiciones:

- Devuelve el sistema preparado para jugar.
- En caso de error muestra un mensaje y cierra la aplicación.

Operación: Juega

Responsabilidades: Devolver el control al usuario para que pueda interactuar con la aplicación.

16. Un ejemplo del desarrollo software de un videojuego

Precondiciones: El sistema debe de haber sido preparado para que el usuario interactúe con la aplicación.

Postcondiciones: Pasa el control de la aplicación al usuario.

DIAGRAMA DE SECUENCIA: Gestionar Niveles

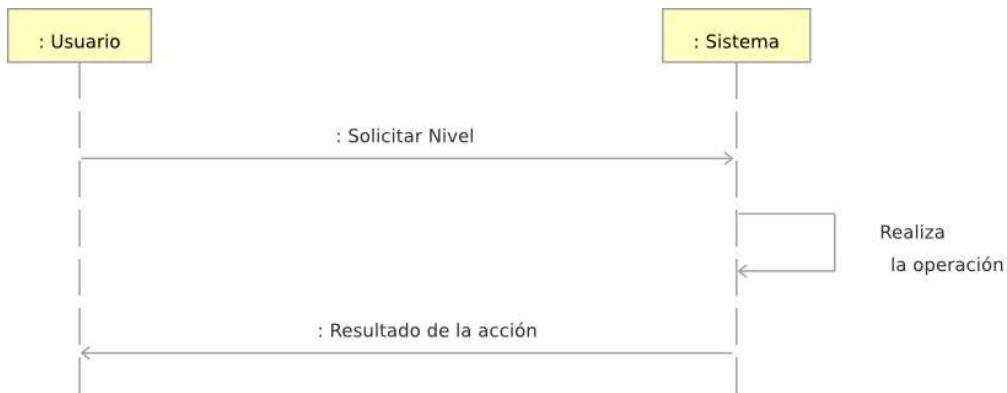


Figura 16.18: Diagrama de secuencia: Gestionar Niveles

Contrato de las operaciones

Operación: Solicitar Nivel

Responsabilidades: Solicitar al sistema una determinada acción sobre un nivel.

Precondiciones: El nivel solicitado debe de existir.

Postcondiciones: El sistema recibe la petición del usuario.

Operación: Resultado de la acción.

Responsabilidades: Debe de devolver al usuario el resultado de la acción previamente solicitada.

Precondiciones: Debe de existir una acción solicitada al sistema sobre un nivel.

Postcondiciones: Dependiendo del resultado de la acción

- Devuelve al usuario un mensaje con el resultado de la acción.
- Devuelve un nivel determinado a otro proceso de usuario.

16.12. Diseño del Sistema

Siguiendo con la metodología del análisis vamos a realizar un diseño orientado a objetos mediante UML. Hasta ahora habíamos especificado y analizado nuestra aplicación con lo que completamos el análisis de requisitos y la especificación del sistema. Ahora vamos a diseñar el sistema que no es más que describir fuera de toda duda razonable que va a hacer el sistema.

El diseño del sistema es la actividad de aplicar diferentes técnicas y principios con el propósito de definir un sistema con el suficiente detalle para que se pueda implementar. El resultado del diseño es un conjunto de diseños de diferentes partes del software.

Este proceso es mucho más sencillo una vez que hemos especificado qué debe hacer el sistema en los pasos anteriores. El proceso de diseño consiste en especificar el sistema con suficiente detalle para que se pueda implementar. El resultado del proceso de diseño es una arquitectura de software bien definida y los diseños de datos, interfaz y programas que nos permitan llevar a cabo la implementación del sistema software con garantías de éxito.

Para llevar a cabo el diseño vamos a usar patrones de diseño que nos permiten plantear los problemas concretos donde definiremos el contexto del problema, el problema en sí y la solución para dicho problema.

16.12.1. Arquitectura del sistema software

La arquitectura del sistema software es un esquema de organización estructural para estos sistemas. Con él especificaremos las responsabilidades de cada uno de los subsistemas definidos para organizar las relaciones entre ellos. Utilizaremos el patrón de arquitectura en capas.

El **contexto** del problema es que necesitamos descomponer nuestra aplicación en grupos de tareas con un mismo nivel de abstracción que sirvan de base a otras tareas más complejas.

El **problema** existente es que necesitamos que esta jerarquía esté bien definida para desarrollar nuestro software ya que las tareas de alto nivel no se pueden implementar utilizando los servicios de la plataforma ya que aumentaría la dependencia de nuestro sistema por lo que necesitamos servicios intermedios. Hay varias características que son deseables para nuestras aplicaciones. Deben ser:

Mantenibles: Un cambio en el código no puede propagarse a todo el sistema.

Reusable: Los componentes de nuestra aplicación deben de poder ser utilizados en otras aplicaciones por lo que deberemos de separar interfaz e implementación.

Cohesión: Responsabilidades similares deben agruparse para favorecer la mantenebilidad y la compresión del sistema.

Portabilidad: Esta es una característica deseable que no siempre puede satisfacerse.

En nuestro caso utilizamos SDL que nos proporciona un nivel más en la jerarquía de niveles con la que conseguimos la independencia del sistema software del sistema operativo y de cualquier hardware en el que queramos compilar nuestra aplicación.

La **solución** es estructurar al sistema en un número de capas suficiente para que cumpla todas las características deseables para nuestra aplicación. Los servicios que ofrece la capa n deben basarse en los servicios que ofrece la capa n-1 y la propia n. Los componentes de una capa han de estar al mismo nivel de abstracción.

Para nuestra aplicación SDL vamos a usar una arquitectura en tres capas. La primera de ellas será la capa de presentación que será la encargada de la interacción (interfaz) con el usuario. La segunda capa será la capa de dominio que es la responsable de la implementación de la funcionalidad del sistema. La tercera y última capa será la encargada de interactuar con el sistema para almacenar y obtener la información estática (o almacenada en disco) del sistema a la que llamaremos capa de gestión de datos.

Una vez determinado el patrón arquitectónico, el patrón de diseño nos proporciona una esquema que nos permite refinar los componentes y las relaciones de los subsistemas. Resuelven un problema de diseño general en un contexto determinado. Vamos a utilizar el patrón de diseño de UML.

No necesitamos realizar un diseño de la capa de gestión de datos ya que los únicos datos que vamos a almacenar en el disco es un fichero organizado secuencialmente donde estarán los distintos niveles de la aplicación. De la gestión del acceso a disco se encargará la clase que administrará los niveles consiguiendo una cohesión importante de la clase.

16.12.2. Diseño de la capa de dominio

En los siguientes apartados vamos a realizar el diseño de la capa de dominio, o lo que es lo mismo, la capa encargada de la implementación de la

16. Un ejemplo del desarrollo software de un videojuego

funcionalidad del sistema.

Una vez realizado los diagramas de las clases de diseño que describen las clases del software y sus operaciones debemos de realizar los contratos de cada una de las operaciones y los diagramas de secuencia de la respuesta a eventos externos. En mucha de las clases sustituiremos esta explicación formal por una más intuitiva e informal para no perder el rumbo didáctico de este temario.

16.12.3. Diagrama de clases de diseño

Este diagrama, como verás, es diferente al diagramas de clase conceptual ya que será especificará todos los detalles necesarios para la implementación del sistema mediante SDL y C++.

En este diagrama aparecen:

Clases: Que participan en las interacciones.

Relaciones: Entre las clases y nuevas clases que proporcionan la relaciones entre clases.

Atributos: Aparecen todos los atributos necesarios para la implementación de la aplicación.

Operaciones: Métodos y funciones necesarias para el desarrollo de la aplicación.

Para presentar estos diagramas vamos a seguir el mismo orden que utilizamos en cuando mostramos los diagramas en la fase de análisis.

16.12. Diseño del Sistema

El primer subdiagrama que presentamos incluye a las clases que gestionarán los elementos multimedia de la aplicación. Podemos decir que la clase principal de este grupo es la clase Galería que estará compuesta de las clases que están asociadas a ella. El diagrama es el de la figura 16.19.

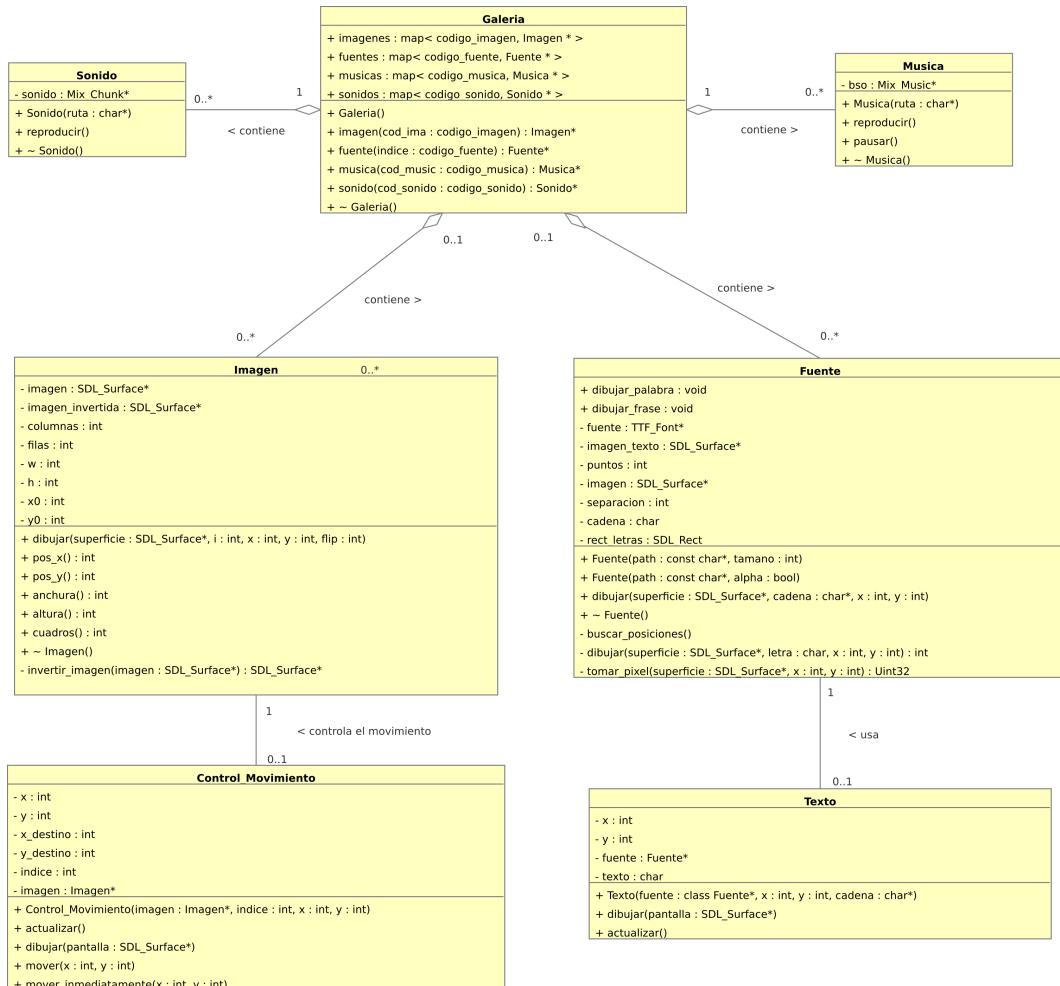


Figura 16.19: Diagrama de clases (diseño): Componentes multimedia

En este diagrama se muestran siete clases. Como hemos comentado Galería es la clase que las relaciona a todas. Como puedes observar hemos añadido todo lo necesario para la implementación de las clases. Será en dicho apartado donde estudiaremos todas las nuevas características de las clases. En este momento es importante que observes como han evolucionado las relaciones entre las clases y las propias clases frente a las del análisis.

16. Un ejemplo del desarrollo software de un videojuego

El segundo diagrama que presentamos relaciona a las clases que nos permiten crear y modificar niveles en la aplicación. En este caso la clase que permite relacionar a las demás componentes del diagrama es la clase Universo aunque la que contendrá toda la lógica para realizar esta tarea es la clase Editor.

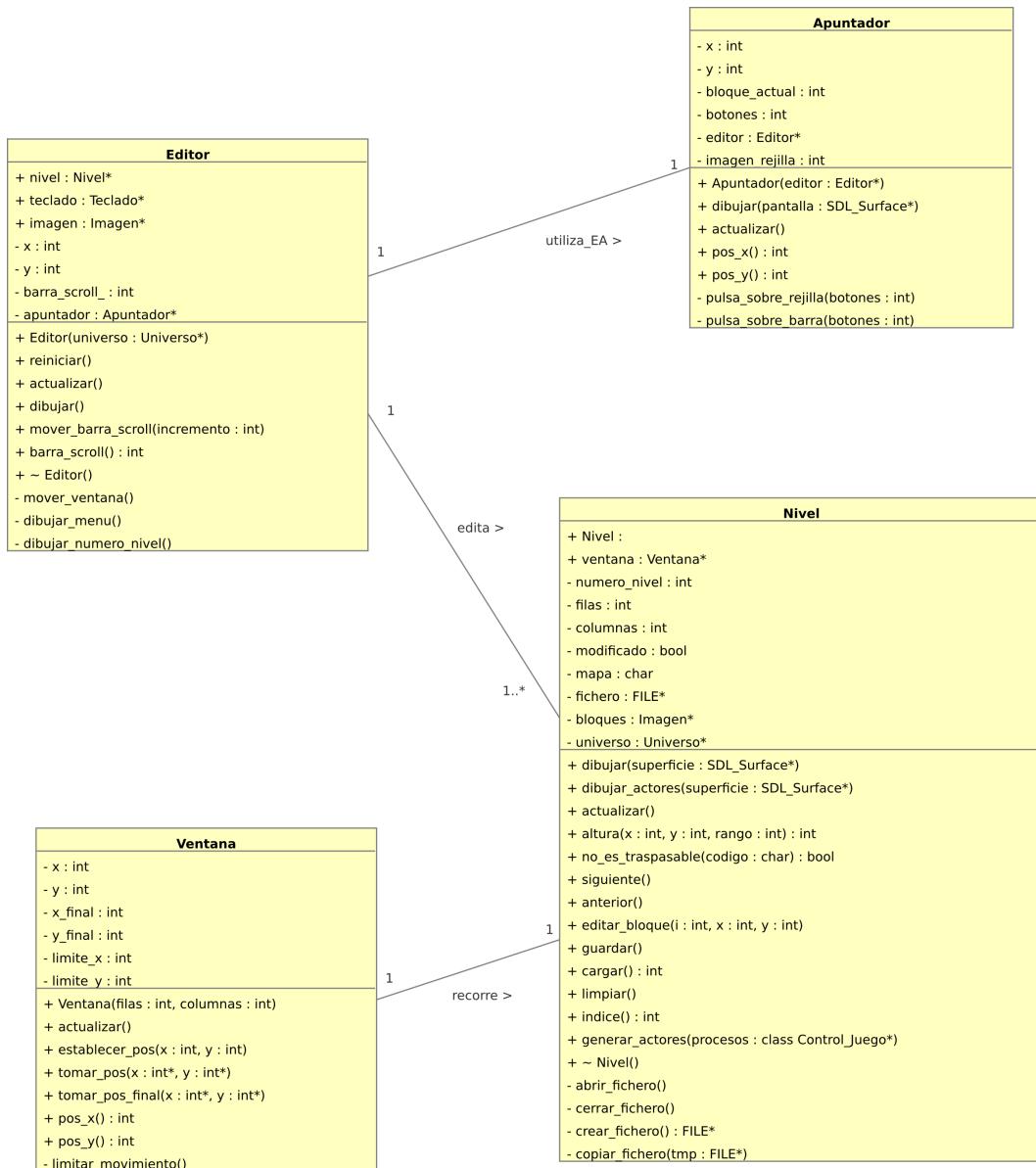


Figura 16.20: Diagrama de clases (diseño): Editando niveles

Como puedes observar hemos añadido todo lo necesario para la implementación de las clases. Será en dicho apartado donde estudiaremos todas las nuevas características de las clases. En este momento es importante que observes como han evolucionado las relaciones entre las clases y las propias clases

16.12. Diseño del Sistema

frente a las del análisis.

16. Un ejemplo del desarrollo software de un videojuego

El tercer diagrama que vamos a estudiar engloba las clases que están asociadas directamente con la clase Participante que recoge todos los elementos activos que participan en los niveles del juego.

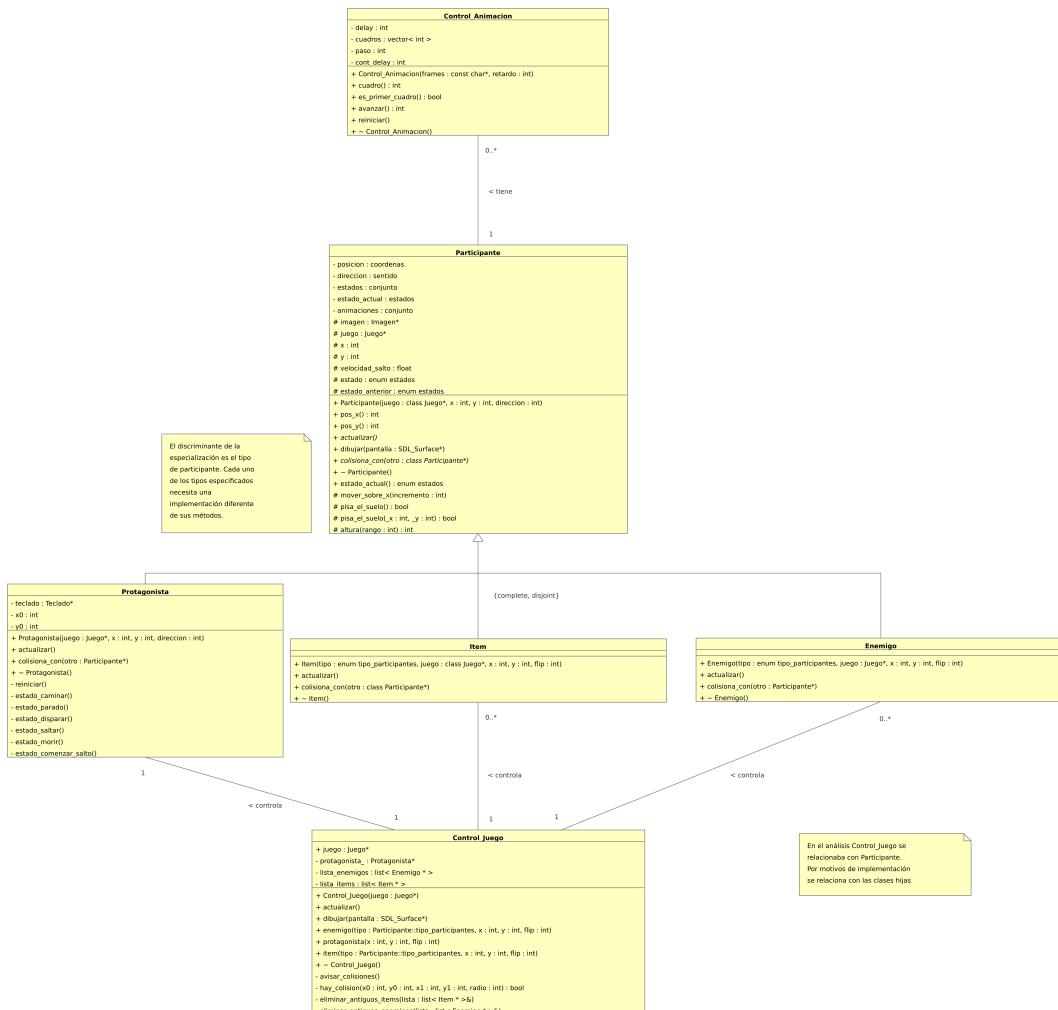


Figura 16.21: Diagrama de clases (diseño): Participantes

Como puedes observar hemos añadido todo lo necesario para la implementación de las clases. Será en dicho apartado donde estudiaremos todas las nuevas características de las clases. En este momento es importante que observes como han evolucionado las relaciones entre las clases y las propias clases frente a las del análisis.

16.12. Diseño del Sistema

Ahora vamos a estudiar uno de los diagramas de clases más importantes del análisis. Se trata de la clase Universo encargada de relacionar las clases del videojuego, los elementos de control y los elementos multimedia que representa a cada una de estas partes.

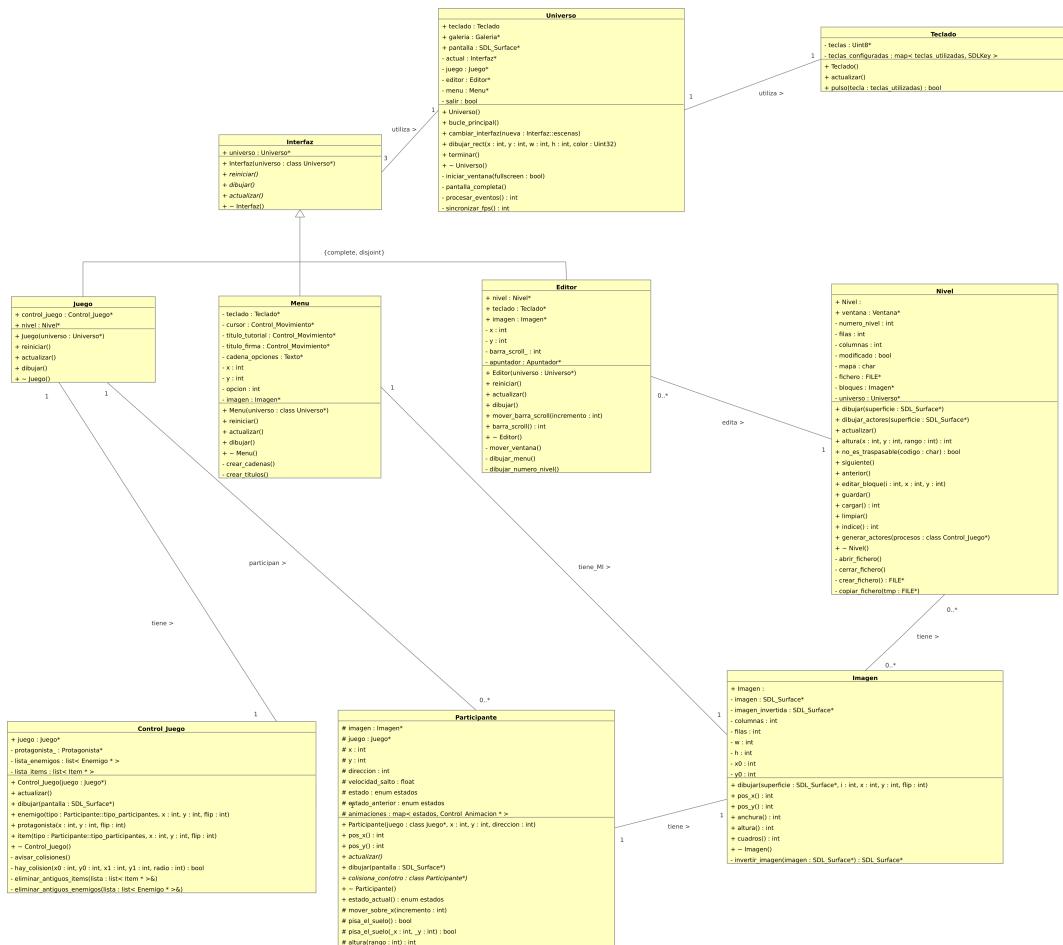


Figura 16.22: Diagrama de clases (diseño): Universo

Como puedes observar hemos añadido todo lo necesario para la implementación de las clases. Será en dicho apartado donde estudiaremos todas las nuevas características de las clases. En este momento es importante que observes como han evolucionado las relaciones entre las clases y las propias clases frente a las del análisis.

Con esta información puedes obtener como es el diagrama general de todas las clases. No se incluye en este temario por problemas de espacio pero puedes encontrarlo en el material del curso.

16. Un ejemplo del desarrollo software de un videojuego

16.12.4. Diagrama de secuencia

Estos diagramas van a definir la interacción entre las distintas clases. Vamos a dividir la presentación de estos diagramas por funcionalidad. Debemos de añadir un contrato por cada una de las operaciones que aparecen en los diagramas. Vamos omitir este paso ya que vamos a detallar minuciosamente los detalles del contrato en el apartado anterior.

El diseño del videojuego va tomando consistencia. Pronto codificaremos y entenderás mejor todo este proceso.

16.12. Diseño del Sistema

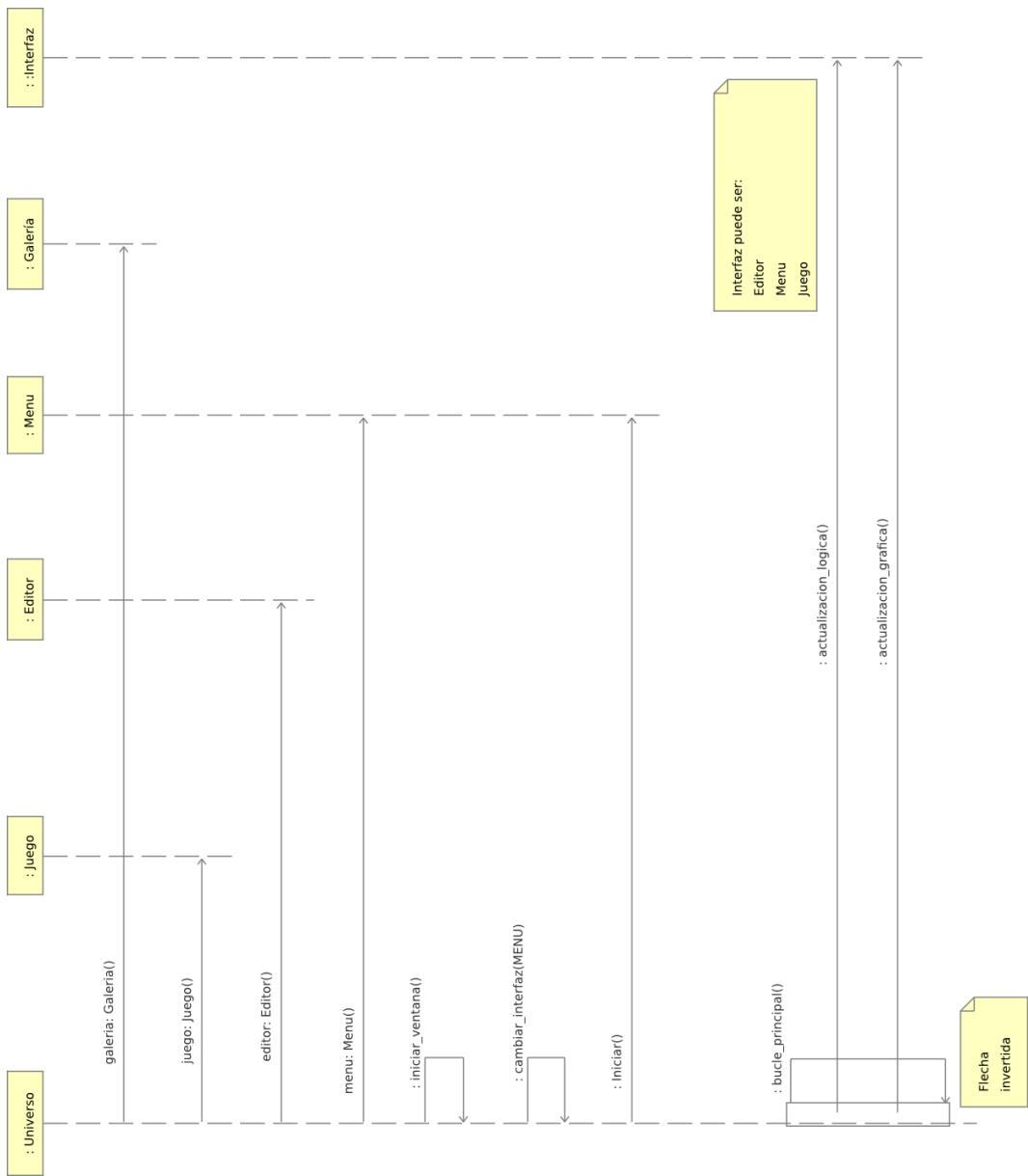


Figura 16.23: Diagrama de secuencia (diseño): Universo

En este diagrama mostramos las interacciones que deben darse entre las principales clases del sistema.

16. Un ejemplo del desarrollo software de un videojuego

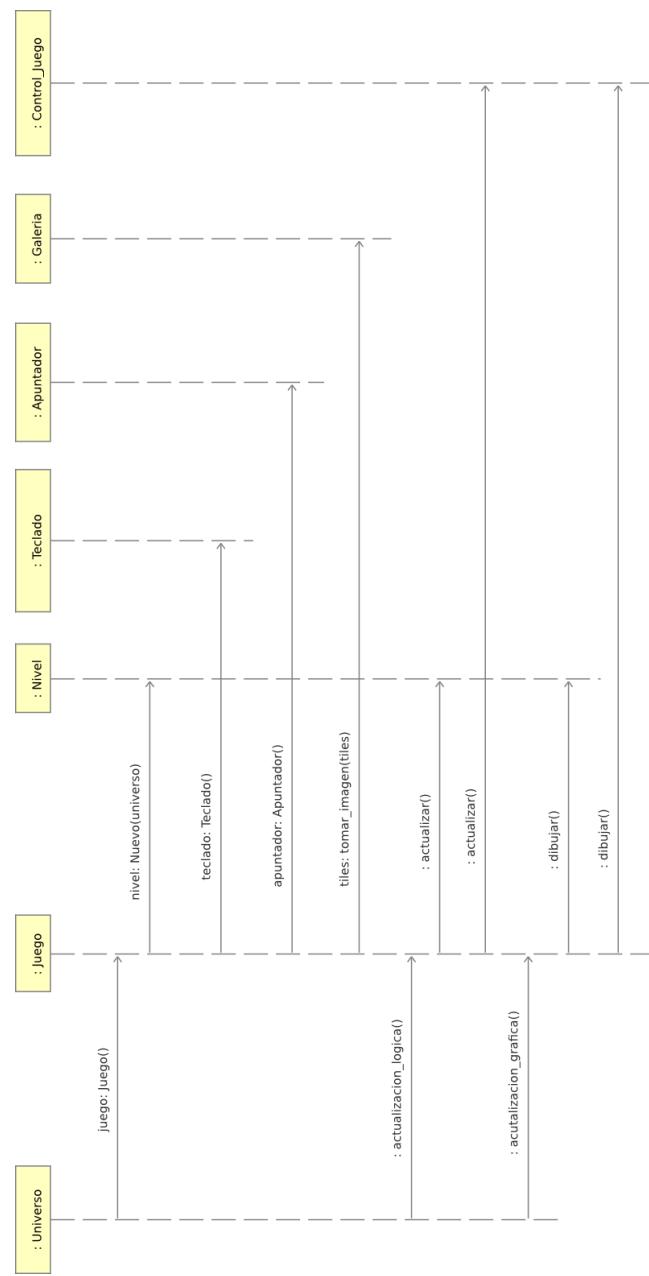


Figura 16.24: Diagrama de secuencia (diseño): Editor

En este diagrama presentamos las relaciones que mantiene el Editor con las demás clases del sistema.

16.12. Diseño del Sistema

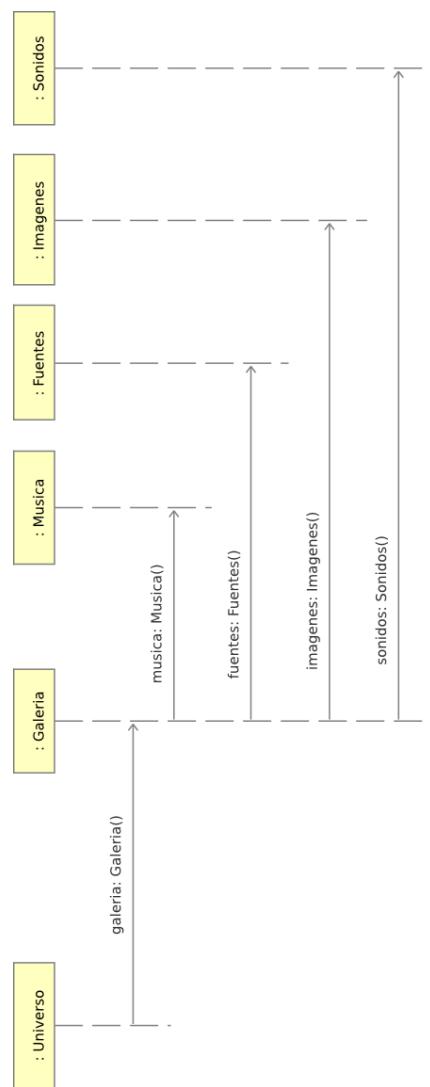


Figura 16.25: Diagrama de secuencia (diseño): Galería

En este diagrama presentamos las relaciones que mantiene la Galería con las demás clases del sistema.

16. Un ejemplo del desarrollo software de un videojuego

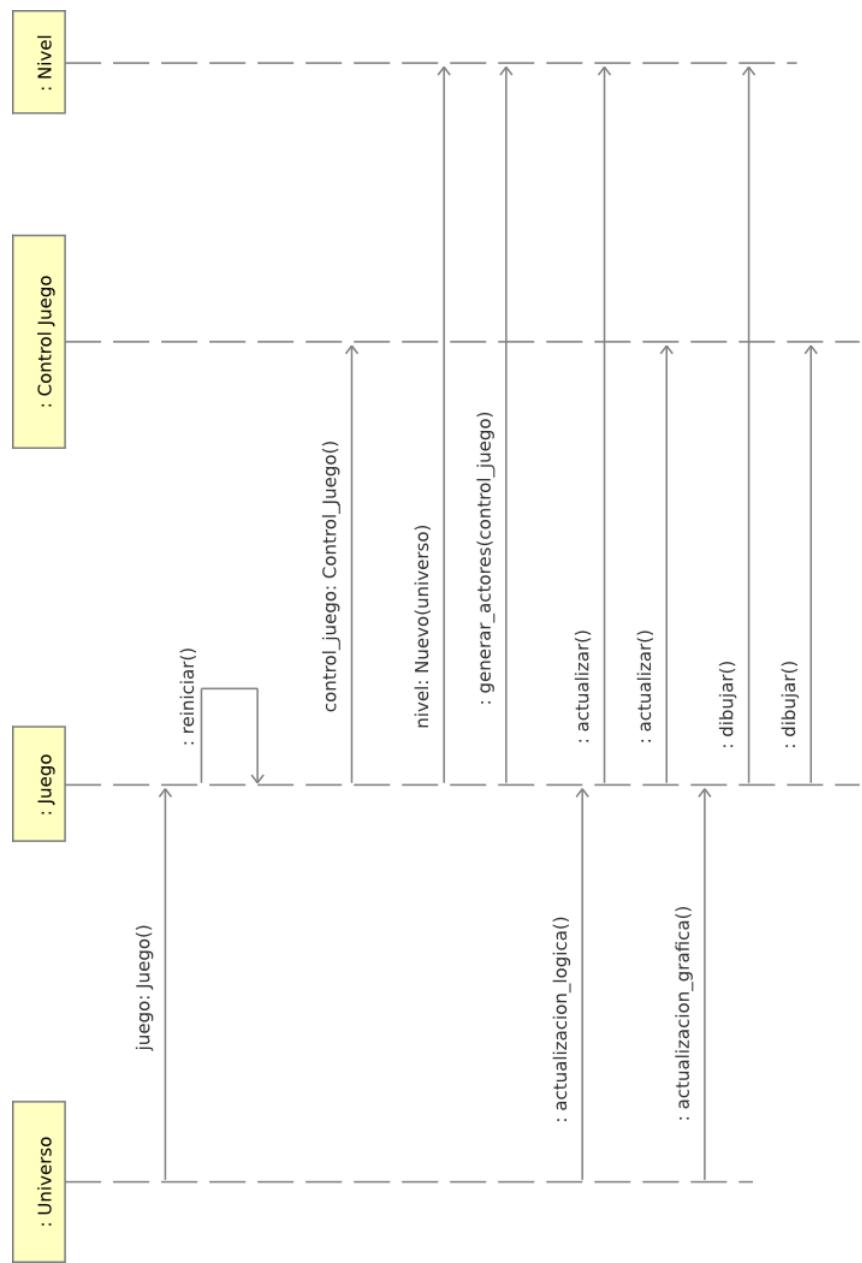


Figura 16.26: Diagrama de secuencia (diseño): Juego

En este diagrama presentamos las relaciones que mantiene el Juego con las demás clases del sistema.

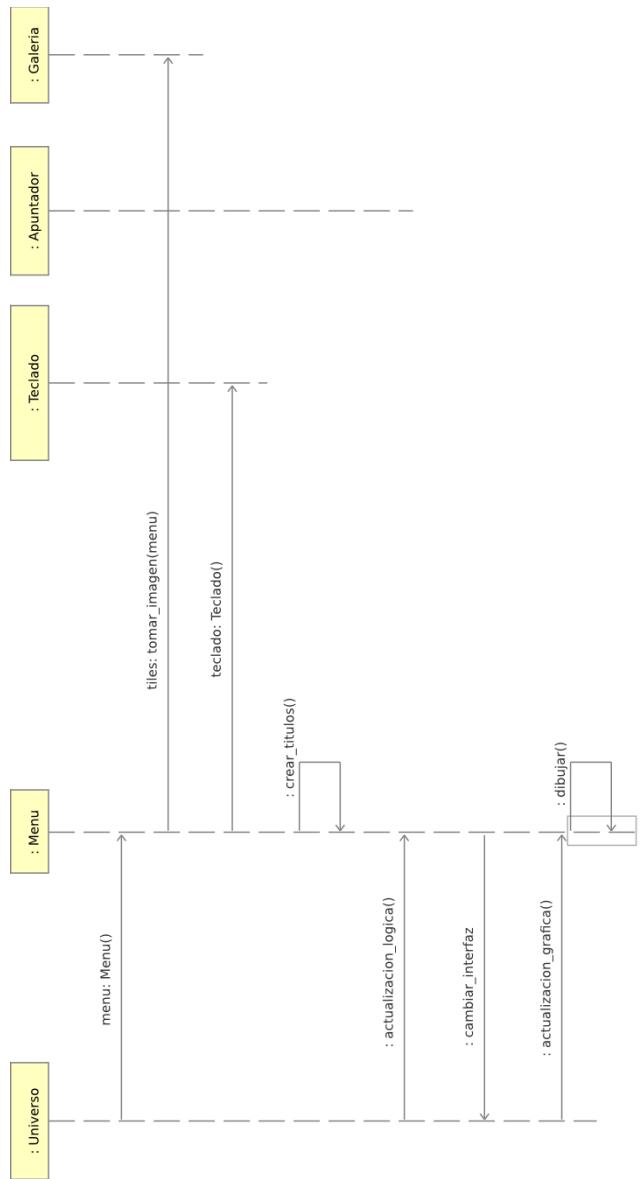


Figura 16.27: Diagrama de secuencia (diseño): Menu

En este diagrama presentamos las relaciones que mantiene el Menú con las demás clases del sistema.

16.12.5. Diseño de los personajes

En la etapa de codificación, junto a los aspectos más importantes de la misma, presentaremos el diseño lógico de los personajes como justificación a dicha codificación. Hay otra etapa que no hemos presentado todavía. Se trata del diseño gráfico que tenemos que aplicar a todas las creaciones y ejemplos que hemos realizado en el tutorial. En este apartado vamos a presentar las más importantes.

16. Un ejemplo del desarrollo software de un videojuego

El trabajo de creación de estas librerías gráficas es importante. Hay que invertir mucho tiempo para crear un personaje completo que sea válido para que sea utilizado en una animación. Estos personajes han sido creados con *The Gimp*. Ésta es una herramienta de edición de gráficos raster por lo que las modificaciones las realizamos píxel a píxel.

Vamos a presentar el diseño de los gráficos más relevantes que hemos creado en exclusividad para este tutorial:

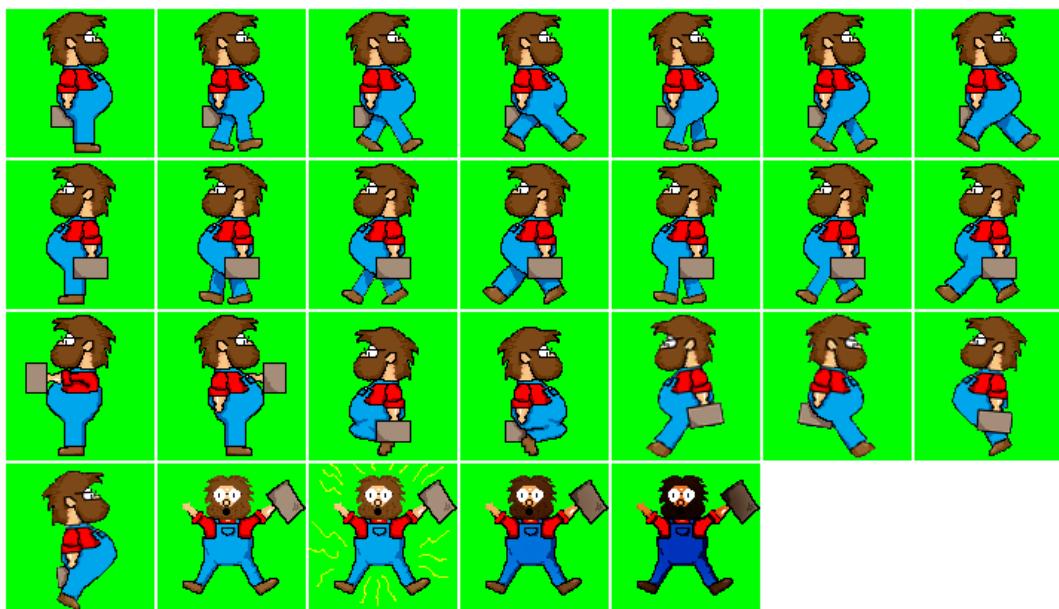


Figura 16.28: Diseño gráfico: Personaje Principal

El gráfico de la figura 16.28 contiene todas los fotogramas que componen la animación del personaje principal de nuestro videojuego final. Puedes comprobar el gran esfuerzo que hay que realizar para crear un personaje con todos estos cuadros. La fluidez de muchas de las animaciones del videojuego dependen de un diseño gráfico correcto.



Figura 16.29: Diseño gráfico: Enemigo polvo

El gráfico de la figura 16.29 contiene los fotogramas que componen la

16.12. Diseño del Sistema

animación de uno de los enemigos. Como puedes comprobar esta animación es mucho más simple ya que no influye tanto como la anterior en el desarrollo del videojuego.

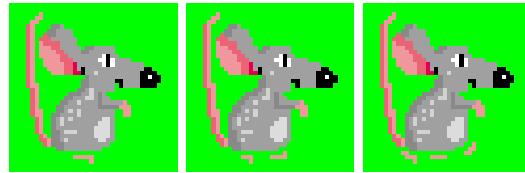


Figura 16.30: Diseño gráfico: Enemigo rata

El gráfico de la figura 16.30 contiene los fotogramas que componen la animación de uno de los enemigos. Estamos en el mismo caso que el enemigo polvo por las mismas razones.

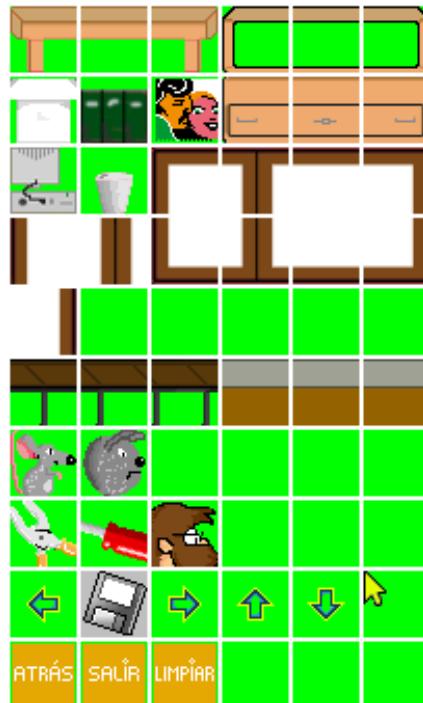


Figura 16.31: Diseño gráfico: Tiles o bloques

El gráfico de la figura 16.31 es uno de los más importantes en la creación del videojuego. Contiene todos los elementos que nos permitirá configurar nuestros niveles y las posiciones de todos los elementos del juego.

En las figuras 16.32 y 16.33 tenemos dos casos análogos. Se tratan de objetos que nos encontraremos en los niveles del tutorial y su recopilación

16. Un ejemplo del desarrollo software de un videojuego

será uno de los objetivos del mismo.



Figura 16.32: Diseño gráfico: Ítem destornillador



Figura 16.33: Diseño gráfico: Ítem alicate

Hay otros aspectos como el diseño de niveles que no se ha seguido una metodología establecida ya que no existen. Uno de los objetivos del juego desarrollado es que sea un borrador libre donde poder probar cosas más que sea un videojuego completo del mismo. El editor de niveles nos permite esta capacidad. No tiene razón de ser un esfuerzo relevante en el diseño de estos niveles ya que debe ser una capacidad que tienes que trabajar desde tus posibilidades.

Una buena forma de trabajar este aspecto es observar detalladamente como están diseñados los niveles otros juegos y llevar a cabo un aprendizaje por imitación.

16.13. Implementación

Una vez realizado el diseño de las clases estamos en condiciones de realizar la implementación de la aplicación. Vamos a presentar cada una de las clases detallando su diseño y mostrando el código resultante del proceso que hemos llevado a cabo. Esta explicación es algo extensa. Recurre a ella si no entiendes alguna de las partes del código de la aplicación.

Hemos realizado una implementación lo más simple posible sin renunciar al compromiso claridad/eficiencia que queremos poner de relieve antes de comenzar con el estudio del videojuego.

16.13.1. La clase Apuntador

La clase apuntador nace con el objetivo de mostrar el puntero del ratón en el editor de niveles así como para controlar la posición de dicho indicador

16.13. Implementación

en todo momento. Además nos va a permitir dar respuesta a las acciones producidas en el ratón sobre las diferentes partes de la pantalla.

El diseño de la clase es::

Apuntador	
- x : int	
- y : int	
- bloque_actual : int	
- botones : int	
- editor : Editor*	
- imagen rejilla : int	
+ Apuntador(editor : Editor*)	
+ dibujar(pantalla : SDL_Surface*)	
+ actualizar()	
+ pos_x() : int	
+ pos_y() : int	
- pulsa_sobre_rejilla(botones : int)	
- pulsa_sobre_barra(botones : int)	

Figura 16.34: Clase Apuntador

A partir de este diseño implementamos la clase. La definición de esta clase es:

```
1 ;// Listado: Apuntador.h
2 ;//
3 ;// La clase Apuntador controla el estado
4 ;// y la posición del cursor del ratón en la pantalla.
5 ;//
6 ;// Haremos uso del cursor del ratón en el Editor de niveles
7 ;
8 ;
9 ;#ifndef _APUNTADOR_H_
10 ;#define _APUNTADOR_H_
11 ;
12 ;#include <SDL/SDL.h>
13 ;
14 ;
15 ;class Editor; // Declaración adelantada
16 ;
17 ;
18 ;class Apuntador
19 ;{
20 ;
21 ; public:
22 ;
23 ;     // Constructor
24 ;     Apuntador(Editor *editor);
```

16. Un ejemplo del desarrollo software de un videojuego

```
25 ;
26 ;     void dibujar(SDL_Surface *pantalla);
27 ;     void actualizar(void);
28 ;
29 ;     // Consultoras
30 ;
31 ;     int pos_x(void);
32 ;     int pos_y(void);
33 ;
34 ; private:
35 ;
36 ;     // Posición del cursor
37 ;
38 ;     int x, y;
39 ;
40 ;     int bloque_actual;
41 ;
42 ;     // Controla la pulsación
43 ;     // de los botones del ratón
44 ;
45 ;     int botones;
46 ;
47 ;     // Nos permite asociar el cursor
48 ;     // con el editor de niveles
49 ;
50 ;     Editor *editor;
51 ;
52 ;     // Discrimina de la zona de pulsación
53 ;
54 ;     void pulsa_sobre_rejilla(int botones);
55 ;     void pulsa_sobre_barra(int botones);
56 ;
57 ;     // Indica en que imagen de la rejilla
58 ;     // de imágenes auxiliares se posiciona
59 ;     // la que representará al cursor
60 ;
61 ;     int imagen_rejilla;
62 ;};
63 ;
64 ;#endif
;-----
```

En la parte privada de la clase tenemos varias variables que nos van a permitir llevar el control sobre el apuntador con el que vamos a trabajar. En las variables *x* e *y* controlaremos la posición en la que está el ratón en un momento dado. La variable *bloque_actual* nos permite conocer en qué bloque, dentro de los recuadros en los que hemos dividido el nivel, ha sido pulsado. Esto es útil para alguno de los métodos de la clase que estudiaremos a continuación. La variable *botones* nos permite controlar el estado de los botones del ratón para poder reaccionar a las acciones que llevemos a cabo mediante dicho dispositivo. Mediante la variable *editor* asociamos el puntero con las acciones de un editor determinado. Nos queda por describir una variable y dos

16.13. Implementación

métodos. Los métodos veremos cual es su función ahora, cuando estudiemos la implementación de la clase. La variable *imagen_rejilla* localiza dentro de la rejilla de imágenes auxiliares cual utilizamos para representar el cursor del ratón.

En la parte pública de la clase tenemos varias funciones. Los métodos que ofrecemos con esta clase son bastante básicos e intuitivos. Vamos a ver la implementación de dichas funciones:

```
1 ;// Listado: Apuntador.cpp
2 ;//
3 ;// Implementación de la clase Apuntador
4 ;
5 ;#include "CommonConstants.h"
6 ;#include "Apuntador.h"
7 ;#include "Nivel.h"
8 ;#include "Imagen.h"
9 ;#include "Editor.h"
10 ;#include "Universo.h"
11 ;
12 ;
13 ;
14 ;// Constructor
15 ;
16 ;Apuntador::Apuntador(Editor * editor):
17 ;    x(0), y(0), bloque_actual(0), imagen_rejilla(IMAGEN_PUNTERO){
18 ;
19 ;
20 ;    // Asociamos el apuntador con el editor
21 ;
22 ;    this->editor = editor;
23 ;}
24 ;
25 ;
26 ;
27 ;void Apuntador::dibujar(SDL_Surface *pantalla){
28 ;
29 ;    // Dibujamos el apuntador en su posición actual
30 ;
31 ;    editor->imagen->dibujar(pantalla, imagen_rejilla, x, y, 1);
32 ;}
33 ;
34 ;
35 ;
36 ;int Apuntador::pos_x(void) {
37 ;
38 ;    return x;
39 ;}
40 ;
41 ;
42 ;int Apuntador::pos_y(void) {
43 ;}
```

16. Un ejemplo del desarrollo software de un videojuego

```
44 ;     return y;
45 ;}
46 ;
47 ;void Apuntador::actualizar(void)
48 ;{
49 ;    // Para mantener pulsado el botón del ratón
50 ;
51 ;    static bool pulsado = false;
52 ;
53 ;    // Consultamos el estado del ratón
54 ;
55 ;    botones = SDL_GetMouseState(&x, &y);
56 ;
57 ;    // Pulsar sobre la rejilla y algún botón
58 ;
59 ;    if(x < BARRA_HERRAMIENTAS)
60 ;
61 ;        // Palsa sobre la superficie que define el nivel
62 ;
63 ;        pulsa_sobre_rejilla(botones);
64 ;
65 ;    else {
66 ;
67 ;        // Palsa en la barra de herramientas
68 ;
69 ;        if(botones == SDL_BUTTON(1)) {
70 ;            if(!pulsado) {
71 ;
72 ;                pulsa_sobre_barra(botones);
73 ;                pulsado = true;
74 ;
75 ;            }
76 ;        }
77 ;        else
78 ;            pulsado = false;
79 ;    }
80 ;}
81 ;
82 ;
83 ;void Apuntador::pulsa_sobre_rejilla(int botones)
84 ;{
85 ;    // Con el botón derecho eliminamos el elemento (-1)
86 ;
87 ;    if(botones == SDL_BUTTON(3))
88 ;        editor->nivel->editar_bloque(-1, x, y);
89 ;
90 ;
91 ;    // Con el izquierdo colocamos el nuevo elemento
92 ;
93 ;    if(botones == SDL_BUTTON(1))
94 ;        editor->nivel->editar_bloque(bloque_actual, x, y);
95 ;}
96 ;
```

16.13. Implementación

```
97 ;
98 ;void Apuntador::pulsa_sobre_barra(int botones)
99 ;{
100 ;    int fila = y / TAM_TITLE;
101 ;    int columna = (x / TAM_TITLE) - 17; // Varía la escala
102 ;
103 ;
104 ;    // Acciones segúna la pulsación en pantalla
105 ;
106 ;    if(fila == 0) {
107 ;
108 ;        switch(columna) {
109 ;
110 ;            // Primera fila
111 ;
112 ;            case 0:
113 ;
114 ;                // Flecha anterior
115 ;
116 ;                editor->nivel->anterior();
117 ;                break;
118 ;
119 ;            case 1:
120 ;
121 ;                // Icono de guardar
122 ;
123 ;                editor->nivel->guardar();
124 ;                break;
125 ;
126 ;
127 ;            case 2:
128 ;
129 ;                // Flecha de siguiente
130 ;
131 ;                editor->nivel->siguiente();
132 ;                break;
133 ;
134 ;        }
135 ;
136 ;    if(fila == 1) {
137 ;
138 ;        switch(columna) {
139 ;
140 ;            case 0:
141 ;
142 ;                // Recarga el nivel
143 ;
144 ;                editor->nivel->cargar();
145 ;                break;
146 ;
147 ;            case 1:
148 ;
149 ;                // Sale del editor
```

16. Un ejemplo del desarrollo software de un videojuego

```
150 ;
151 ;         editor->universo->cambiar_interfaz(Interfaz::ESCENA_MENU) ;
152 ;         break;
153 ;
154 ;     case 2:
155 ;
156 ;         // Limpia la escena
157 ;
158 ;         editor->nivel->limpiar();
159 ;         break;
160 ;     }
161 ; }
162 ;
163 ; // Desplazamiento del scroll
164 ;
165 ; if(fila == 3 && columna == 1) {
166 ;
167 ;     editor->mover_barra_scroll(-1);
168 ;     return;
169 ;
170 ; }
171 ;
172 ; if(fila == 14 && columna == 1) {
173 ;
174 ;     editor->mover_barra_scroll(+1);
175 ;     return;
176 ; }
177 ;
178 ; if(fila > 3 && fila < 14)
179 ;
180 ;     // Bloque según el icono que pulsemos
181 ;     // en el scroll
182 ;
183 ;     bloque_actual = (fila - 4) * 3 +
184 ;         editor->barra_scroll() * 3 +
185 ;         columna;
186 ;
187 ; }
```

El constructor inicializa las variables propias de la clase relacionando al puntero con el editor de niveles. Además inicializamos los valores de la posición e indicamos que recuadro de la rejilla auxiliar de imágenes vamos a usar para representar el puntero del ratón.

El método *dibujar* utiliza la relación de esta clase con la clase Editor para utilizar el método que nos permite dibujar en pantalla un bitmap. En este caso el método dibuja el apuntador del ratón en la posición a la que hace referencia las variables de la clase.

Los métodos *pos_x()* y *pos_y*, como es habitual, devuelven la posición del apuntador del ratón sobre la superficie principal de la aplicación. El método

actualizar() nos refresca el estado del ratón. Tomamos el estado del ratón y lo almacenamos en las tres variables de la clase que controlan el estado del ratón. La barra de herramientas del editor de niveles comienza en el punto 544 de ahí la estructura selectiva de este método que nos permite diferenciar el haber pulsado sobre un determinado botón de dicha barra y haber pulsado en la rejilla donde definiremos el nivel.

Vamos con la implementación de los métodos privados de la clase. El primero de ellos es *pulsa_sobre_barra()*. Lo primero que hacemos en este método es calcular en qué fila y columnam medidas en número de bloques, dentro de la barra de herramientas estamos situados. Una vez que hemos hecho este cambio de escala tenemos varias estructuras selectivas que van discriminando qué botón de dicha barra vamos pulsando. En el caso de la primera fila, segúna la columna, podemos pulsar sobre las flechas que nos permiten navegar por los niveles que hemos creados o en el botón que nos permite guardar las modificaciones hechas en el fichero de niveles.

En el caso de las filas sucesivas vamos actuando según el diseño de la barra de herramientas. Los casos, que podríamos llamar especiales, que nos permiten desplazar la barra donde tenemos almacenadas las iconos que nos permiten añadir elementos al nivel. Para realizar este desplazamiento utilizamos funciones implementadas en la clase editor. Si el elemento pulsado está entre los que componen dichos iconos calculamos el desplazamiento que produce que dicha tira de imágenes esté desplazada.

16.13.2. La clase Control Animacion

En capítulos anteriores hemos trabajado con esta clase. Se trata de una estructura que nos permite llevar un control de la animación de un personaje tanto para definir la secuencia de cuadros que debe de seguir como establecer un intervalo de tiempo entre que se reproduce un recuadro y el siguiente.

En este caso vamos a utilizar un poco más de la potencia que nos proporciona el lenguaje de programación C++. Vamos a cambiar el vector de bajo nivel que utilizábamos en nuestros ejemplos por un elemento *vector* de enteros que nos permitirá manejar mejor y más cómodamente los recursos del sistema en cuanto a memoria se refiere.

El diseño de la clase del que partimos para la implementación es el de la figura 16.35.

La definición de la clase es la siguiente:

16. Un ejemplo del desarrollo software de un videojuego

Control_Animacion
<pre>- delay : int - cuadros : vector< int > - paso : int - cont_delay : int + Control_Animacion(frames : const char*, retardo : int) + cuadro() : int + es_primer_cuadro() : bool + avanzar() : int + reiniciar() + ~ Control_Animacion()</pre>

Figura 16.35: Clase Control Animacion

```
;_____
1 ;// Listado: Control_Animacion.h
2 ;// Esta clase controla la secuencia de animación de los personajes de la
3 ;// aplicación
4 ;
5 ;#ifndef _CONTROL_ANIMACION_H_
6 ;#define _CONTROL_ANIMACION_H_
7 ;
8 ;#include <vector>
9 ;
10 ;using namespace std;
11 ;
12 ;class Control_Animacion {
13 ;    public:
14 ;
15 ;        // Constructor
16 ;        Control_Animacion(const char *frames, int retardo);
17 ;
18 ;        // Consultoras
19 ;        int cuadro(void);
20 ;        bool es_primer_cuadro(void);
21 ;
22 ;        // Modificadoras
23 ;        int avanzar(void);
24 ;        void reiniciar(void);
25 ;
26 ;        // Destructor
27 ;        ~Control_Animacion();
28 ;
29 ;    private:
30 ;
31 ;        // Establece el retardo entre
32 ;        // cuadros de la animacion
33 ;
34 ;        int delay;
35 ;
36 ;        // Almacena los cuadros que componen la animacion
37 ;
38 ;        vector<int> cuadros;
39 ;
40 ;        // Paso actual de la animación
41 ;        // que controlemos con esta clase
```

16.13. Implementación

```
42 ;
43 ;    int paso;
44 ;
45 ;    // Contador para llevar un control
46 ;    // del retraso
47 ;
48 ;    int cont_delay;
49 ;};
50 ;
51 ;#endif
;
```

En la parte privada de la clase definimos variables que nos servirán para controlar la animación. *cuadros* es un vector de alto nivel que almacena la secuencia de enteros que define el orden en el que se tienen que mostrar las imágenes con el fin de conseguir la animación deseada. En la variable *paso* almacenamos el número de la secuencia en el que nos encontramos mientras que las variables *delay* y *cont_delay* nos permiten llevar el control de la temporización de la animación.

La implementación de esta clase es la siguiente:

```
;_____
1 ;// Listado: Control_Animacion.cpp
2 ;// Implementación de la clase Control_Animacion
3 ;
4 ;#include <iostream>
5 ;#include "Control_Animacion.h"
6 ;#include "CommonConstants.h"
7 ;
8 ;
9 ;using namespace std;
10 ;
11 ;
12 ;Control_Animacion::Control_Animacion(const char *frames, int retardo):
13 ;    delay(retardo)
14 ;{
15 ;
16 ;    char frames_tmp[MAX_FRAMES];
17 ;    char *proximo;
18 ;
19 ;
20 ;    strcpy(frames_tmp, frames);
21 ;
22 ;    // Trabajamos con una copia de los cuadros
23 ;    // pasados como parámetro
24 ;
25 ;    // Extraemos de la cadena cada uno de los elementos
26 ;
27 ;    for(proximo = strtok(frames_tmp, ","); proximo; ){
28 ;
29 ;        this->cuadros.push_back(atoi(proximo));
30 ;        proximo = strtok(NULL, ",\0");
;
```

16. Un ejemplo del desarrollo software de un videojuego

```
31 ;    }
32 ;
33 ;    // Inicializamos las variables de la clase
34 ;
35 ;    this->cuadros.push_back(-1);
36 ;    this->paso = 0;
37 ;    this->cont_delay = 0;
38 ;
39 ;#ifdef DEBUG
40 ;    cout << "Control_Animacion::Control_Animacion()" << endl;
41 ;#endif
42 ;}
43 ;
44 ;
45 ;int Control_Animacion::cuadro(void) {
46 ;
47 ;    // Devolvemos el paso actual
48 ;    // de la animación
49 ;
50 ;    return cuadros[paso];
51 ;}
52 ;
53 ;
54 ;int Control_Animacion::avanzar(void) {
55 ;
56 ;    // Si ha pasado el tiempo suficiente
57 ;    // entre cuadro y cuadro
58 ;
59 ;    if((++ cont_delay) >= delay) {
60 ;
61 ;        // Reestablecemos el tiempo a 0
62 ;
63 ;        cont_delay = 0;
64 ;
65 ;        // Incrementamos el paso siempre
66 ;        // que no sea el último elemento
67 ;        // lo que hará que volvamos al
68 ;        // primer elemento
69 ;
70 ;        if(cuadros[++paso] == -1) {
71 ;            paso = 0;
72 ;            return 1;
73 ;        }
74 ;    }
75 ;
76 ;    return 0;
77 ;}
78 ;
79 ;
80 ;void Control_Animacion::reiniciar(void) {
81 ;
82 ;    // Volvemos al inicio de la animación
83 ;}
```

```

84 ;    paso = 0;
85 ;    cont_delay = 0;
86 ;}
87 ;
88 ;
89 ;bool Control_Animacion::es_primer_cuadro(void) {
90 ;
91 ;    if(paso == 0)
92 ;        return true;
93 ;
94 ;    return false;
95 ;}
96 ;
97 ;Control_Animacion::~Control_Animacion() {
98 ;
99 ;#ifndef DEBUG
100 ;    cout << "Control_Animacion::~Control_Animacion()" << endl;
101 ;#endif
102 ;}
;
```

En el constructor de la clase inicializamos las variables propias de esta clase así como descomponemos la cadena que recibe como parámetro con el fin de almacenar la secuencia que recibimos como parámetro en el vector que controlará la animación. El método *cuadro()* devuelve el frame que corresponde con el momento actual de la animación. La función miembro *avanzar()* pasa al siguiente cuadro de la animación siempre que haya pasado el tiempo suficiente. Esta es una forma de controlar la temporalidad que nos permite dar un comportamiento exclusivo a cada animación de las que posea un determinado personaje.

El método *reiniciar()* nos permite poner la animación a su estado inicial sin tener que volver a crearla y el método *es_primer_cuadro()* nos permite consultar si la animación se encuentra en dicho estado. El destructor no realiza ninguna tarea especial.

16.13.3. La clase Control Juego

Esta es una de las clases más importantes del desarrollo del videojuego. Nos permite llevar un control de la lógica del juego y de todos los personajes que están involucrados en el mismo. El diseño de la clase del que partimos para la implementación es el de la figura 16.36.

Vamos a estudiar la definición de la clase:

```

1 ;// Listado: Control_Juego.h
2 ;//
3 ;// La clase Control Juego proporciona un mando lógico sobre el juego
4 ;//
```

16. Un ejemplo del desarrollo software de un videojuego

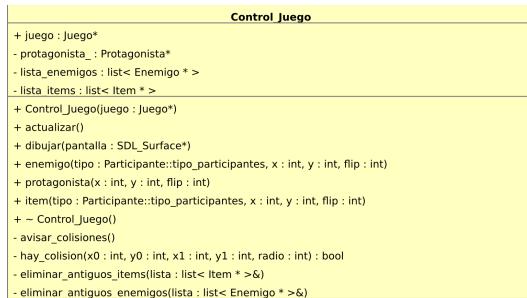


Figura 16.36: Clase Control Juego

```
5 ;
6 ;
7 ;#ifndef _CONTROL_JUEGO_H_
8 ;#define _CONTROL_JUEGO_H_
9 ;
10 ;#include <SDL/SDL.h>
11 ;#include <list>
12 ;#include "Participante.h"
13 ;
14 ;
15 ;// Declaración adelantada
16 ;
17 ;class Protagonista;
18 ;class Enemigo;
19 ;class Item;
20 ;class Juego;
21 ;
22 ;
23 ;using namespace std;
24 ;
25 ;
26 ;class Control_Juego {
27 ;
28 ; public:
29 ;
30 ;     // Constructor
31 ;
32 ;     Control_Juego(Juego *juego);
33 ;
34 ;     // Para realizar la actualización lógica
35 ;     // y poder mostrar el resultado en pantalla
36 ;
37 ;     void actualizar(void);
38 ;     void dibujar(SDL_Surface *pantalla);
39 ;
40 ;     Juego *juego;
41 ;
42 ;     // Nos permiten establecer a los participantes
43 ;     // del juego en el universo del juego
44 ;
```

16.13. Implementación

```
45 ; void enemigo(Participante::tipo_participantes tipo, int x, int y, int flip);
46 ; void protagonista(int x, int y, int flip);
47 ; void item(Participante::tipo_participantes tipo, int x, int y, int flip);
48 ;
49 ; // Destructor
50 ;
51 ; ~Control_Juego();
52 ;
53 ; private:
54 ;
55 ; void avisar_colisiones(void);
56 ;
57 ; // Galería de personajes y objetos del juego
58 ;
59 ; Protagonista *protagonista_;
60 ; list<Enemigo *> lista_enemigos;
61 ; list<Item *> lista_items;
62 ;
63 ; // Detección de colisiones
64 ;
65 ; bool hay_colision(int x0, int y0, int x1, int y1, int radio);
66 ;
67 ; void eliminar_antiguos_items(list<Item *>& lista);
68 ; void eliminar_antiguos_enemigos(list<Enemigo *>& lista);
69 ;};
70 ;
71 ;#endif
```

En la parte privada de la clase tenemos varios elementos. Los atributos *lista_enemigos* y *lista_items* son listas que nos permiten llevar un control de los objetos y enemigos presentes en el nivel mientras que con *protagonista* realizamos la misma tarea con el personaje principal del juego. Las listas han sido definidas sobre *list* de la STL que nos permitirán un manejo sencillo de este tipo de estructuras.

En esta parte de la clase definimos también varias funciones que vamos a necesitar para controlar la aplicación. *avisar_colisiones()* hace un recorrido por todos los items y adversarios para comprobar si existe colisión con el personaje principal. Con esto cubrimos todas las posibles colisiones a las que tenemos que reaccionar. La función *hay_colision()* es la encargada de la detección de colisiones en el juego por lo que es de vital importancia.

Las funciones *eliminar_antiguos()* se encargan de borrar aquellos elementos del juego que por un motivo o por otro estén en estado de eliminar. Esto suele ocurrir cuando un personaje muere o un ítem ha colisionado con el personaje principal del juego.

En la parte pública de la clase tenemos varios métodos, mucho de ellos comunes a la mayoría de las clases. El constructor de la clase, las funciones

16. Un ejemplo del desarrollo software de un videojuego

actualizar() y *dibujar()*. La primera actualiza el estado lógico de todos los componentes del juego mientras que la segunda los dibuja en determinados momentos. Tenemos tres método (*enemigo()*, *protagonista()* y *item()*) que nos permiten agregar cada uno de los elementos asociados a éstos a la aplicación. En enlace de dependencia con la clase *Juego* es público ya que necesitaremos tener acceso desde otras clases a través del Control del Juego.

Hemos realizado un recorrido rápido por los métodos y atributos clase, veamos ahora como se ha implementado dicha clase:

```
1 ;_____
2 // Listado: Control_Juego.cpp
3 // Implementación de la clase Control_Juego
4 ;
5 ;
6 ;
7 ;#include <iostream>
8 ;
9 ;#include "Control_Juego.h"
10 ;#include "Juego.h"
11 ;#include "Item.h"
12 ;#include "Enemigo.h"
13 ;#include "Protagonista.h"
14 ;
15 ;
16 ;using namespace std;
17 ;
18 ;
19 ;Control_Juego::Control_Juego(Juego *juego): protagonista_(NULL) {
20 ;
21 ;#ifndef DEBUG
22 ;    cout << "Control_Juego::Control_Juego()" << endl;
23 ;#endif
24 ;
25 ;    this->juego = juego;
26 ;
27 ;}
28 ;
29 ;
30 ;
31 ;void Control_Juego::actualizar(void) {
32 ;
33 ;
34 ;    // Actualizamos el protagonista
35 ;
36 ;    if(protagonista_)
37 ;        protagonista_->actualizar();
38 ;
39 ;    // Actualizamos todos los enemigos
40 ;
41 ;    for(list<Enemigo *>::iterator i = lista_enemigos.begin();
```

16.13. Implementación

```
42 ;         i != lista_enemigos.end(); ++i) {
43 ;
44 ;         (*i)->actualizar();
45 ;
46 ;     }
47 ;
48 ;
49 ;     // Actualizamos todos los items
50 ;
51 ;     for(list<Item *>::iterator i = lista_items.begin();
52 ;          i != lista_items.end(); ++i){
53 ;
54 ;         (*i)->actualizar();
55 ;     }
56 ;
57 ;
58 ;
59 ;     // Eliminamos los antiguos, marcados con ELIMINAR
60 ;
61 ;     eliminar_antiguos_enemigos(lista_enemigos);
62 ;     eliminar_antiguos_items(lista_items);
63 ;
64 ;     // Estudiamos las posibles colisiones
65 ;
66 ;     if(protagonista_ != NULL)
67 ;         avisar_colisiones();
68 ;
69 ;
70 ;}
71 ;
72 ;
73 ;
74 ;void Control_Juego::dibujar(SDL_Surface *pantalla) {
75 ;
76 ;
77 ;     // Dibujamos toda la lista de enemigos
78 ;
79 ;     for(list<Enemigo *>::iterator i = lista_enemigos.begin();
80 ;          i != lista_enemigos.end(); ++i){
81 ;
82 ;         (*i)->dibujar(juego->universo->pantalla);
83 ;
84 ;     }
85 ;
86 ;     // Dibujamos al protagonista
87 ;
88 ;     if(protagonista_)
89 ;         protagonista_->dibujar(juego->universo->pantalla);
90 ;
91 ;
92 ;     // Dibujamos toda la lista de items
93 ;
94 ;     for(list<Item *>::iterator i = lista_items.begin();
```

16. Un ejemplo del desarrollo software de un videojuego

```
95 ;         i != lista_items.end(); ++i){  
96 ;             (*i)->dibujar(juego->universo->pantalla);  
97 ;         }  
98 ;     }  
99 ; }  
100 ;  
101 ;  
102 ;}  
103 ;  
104 ;  
105 ;  
106 ;void Control_Juego::avisar_colisiones(void) {  
107 ;  
108 ;  
109 ;  
110 ;    static int radio = 20;  
111 ;  
112 ;    int x0, y0;  
113 ;    int x1, y1;  
114 ;  
115 ;    // Tomamos la posición del protagonista  
116 ;  
117 ;    x0 = protagonista_->pos_x();  
118 ;    y0 = protagonista_->pos_y();  
119 ;  
120 ;  
121 ;    // Comprobamos si hay colisión con los enemigos  
122 ;  
123 ;    for(list<Enemigo *>::iterator i = lista_enemigos.begin();  
124 ;        i != lista_enemigos.end(); ++i) {  
125 ;  
126 ;        // Posición del enemigo  
127 ;  
128 ;        x1 = (*i)->pos_x();  
129 ;        y1 = (*i)->pos_y();  
130 ;  
131 ;        // Comprobamos si hay colisión entre el elemento y el protagonista  
132 ;  
133 ;        if(hay_colision(x0, y0, x1, y1, radio)) {  
134 ;  
135 ;            if(protagonista_->estado_actual() != Participante::GOLPEAR) {  
136 ;  
137 ;                // Mata al protagonista  
138 ;  
139 ;                protagonista_->colisiona_con((*i));  
140 ;  
141 ;            } else {  
142 ;  
143 ;                // Muere el malo  
144 ;  
145 ;                (*i)->colisiona_con(protagonista_); // Elimina el enemigo  
146 ;  
147 ;            }  
148 ;        }  
149 ;    }  
150 ;}
```

16.13. Implementación

```
148 ;        }
149 ;
150 ;    }
151 ;
152 ;
153 ;
154 ;    // Comprobamos si hay colisión con los objetos
155 ;
156 ;    for(list<Item *>::iterator i = lista_items.begin();
157 ;        i != lista_items.end(); ++i) {
158 ;
159 ;        // Posición del item
160 ;
161 ;        x1 = (*i)->pos_x();
162 ;        y1 = (*i)->pos_y();
163 ;
164 ;
165 ;        // Comprobamos si hay colisión entre el elemento y el protagonista
166 ;
167 ;        if(hay_colision(x0, y0, x1, y1, radio)) {
168 ;
169 ;            (*i)->colisiona_con(protagonista_); // Elimina el item
170 ;
171 ;            return;
172 ;
173 ;        }
174 ;    }
175 ;
176 ;
177 ;}
178 ;
179 ;
180 ;// Creando los enemigos
181 ;
182 ;void Control_Juego::enemigo(Participante::tipo_participantes tipo,
183 ;                            int x, int y, int flip) {
184 ;
185 ;
186 ;#ifdef DEBUG
187 ;    cout << "Creando un enemigo tipo " << tipo << "en ( " << x << " - "
188 ;        << y << " )" << endl;
189 ;#endif
190 ;
191 ;
192 ;    // Almacenamos en las listas el tipo enemigo que queremos crear
193 ;
194 ;    switch(tipo) {
195 ;
196 ;        case Participante::TIPO_ENEMIGO_RATA:
197 ;
198 ;            lista_enemigos.push_back(new Enemigo(Participante::TIPO_ENEMIGO_RATA, juego, x, y, fl
```

16. Un ejemplo del desarrollo software de un videojuego

```
201 ;     case Participante::TIPO_ENEMIGO_MOTA:  
202 ;  
203 ;         lista_enemigos.push_back(new Enemigo(Participante::TIPO_ENEMIGO_MOTA, juego, x, y, fl  
204 ;         break;  
205 ;  
206 ;     default:  
207 ;         cerr << "Enemigo desconocido" << endl;  
208 ;     }  
209 ;  
210 ;  
211 ;}  
212 ;  
213 ;  
214 ;void Control_Juego::protagonista(int x, int y, int flip) {  
215 ;  
216 ;  
217 ;    if(protagonista_ != NULL)  
218 ;  
219 ;        cerr << "Ya existe un protagonista en el nivel" << endl;  
220 ;  
221 ;    else  
222 ;  
223 ;        protagonista_ = new Protagonista(juego, x, y, flip);  
224 ;  
225 ;  
226 ;}  
227 ;  
228 ;  
229 ;void Control_Juego::item(Participante::tipo_participantes tipo,  
230 ;                           int x, int y, int flip) {  
231 ;  
232 ;  
233 ;#ifdef DEBUG  
234 ;    cout << "Creando un item en ( " << x << " - "  
235 ;           << y << " )" << endl;  
236 ;#endif  
237 ;  
238 ;    // Almacenamos en las listas el tipo de item que queremos crear  
239 ;  
240 ;    switch(tipo) {  
241 ;  
242 ;        case Participante::TIPO_DESTORNILLADOR:  
243 ;  
244 ;            lista_items.push_back(new Item(Participante::TIPO_DESTORNILLADOR, juego, x, y, flip));  
245 ;            break;  
246 ;  
247 ;        case Participante::TIPO_ALICATE:  
248 ;  
249 ;            lista_items.push_back(new Item(Participante::TIPO_ALICATE, juego, x, y, flip));  
250 ;            break;  
251 ;  
252 ;        default:  
253 ;    }
```

16.13. Implementación

```
254 ;         cerr << "Item desconocido" << endl;
255 ;         return;
256 ;
257 ;
258 ;
259 ;}
260 ;
261 ;
262 ;Control_Juego::~Control_Juego() {
263 ;
264 ;    // Borramos las estructuras activas del juego
265 ;
266 ;    delete protagonista_;
267 ;
268 ;
269 ;#ifdef DEBUG
270 ;    cout << "Control_Juego::~Control_Juego()" << endl;
271 ;#endif
272 ;
273 ;}
274 ;
275 ;bool Control_Juego::hay_colision(int x0, int y0, int x1, int y1, int radio) {
276 ;
277 ;
278 ;    if((abs(x0 - x1) < (radio * 2)) &&(abs(y0 - y1) < (radio *2)))
279 ;        return true;
280 ;    else
281 ;        return false;
282 ;
283 ;
284 ;}
285 ;
286 ;void Control_Juego::eliminar_antiguos_items(list<Item *>& lista) {
287 ;
288 ;    list<Item *>::iterator i;
289 ;    bool eliminado = false;
290 ;
291 ;
292 ;    // Eliminamos los items marcados como eliminar
293 ;
294 ;    for(i = lista.begin(); i != lista.end() && eliminado == false; ++i) {
295 ;
296 ;        if((*i)->estado_actual() == Participante::ELIMINAR) {
297 ;
298 ;            lista.erase(i); // más eficiente que remove(*i)
299 ;            eliminado = true;
300 ;
301 ;        }
302 ;
303 ;    }
304 ;
305 ;}
306 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
307 ;void Control_Juego::eliminar_antiguos_enemigos(list<Enemigo *>& lista) {
308 ;
309 ;    list<Enemigo *>::iterator i;
310 ;    bool eliminado = false;
311 ;
312 ;    // Eliminamos los items cuyo estado sea eliminar
313 ;
314 ;    for(i = lista.begin(); i != lista.end() && eliminado == false; ++i) {
315 ;
316 ;        if((*i)->estado_actual() == Participante::ELIMINAR) {
317 ;
318 ;            lista.erase(i);
319 ;            eliminado = true;
320 ;
321 ;        }
322 ;
323 ;    }
324 ;}
```

Vamos a estudiar aquellos métodos que necesiten un mayor detalle. El método *actualizar()* hace una llamada a los métodos *actualizar()* de cada uno de los componentes que integran el nivel en el que nos encontramos. Esta operación es de $O(n)$ ya que tenemos que recorrer todos los elementos de las listas donde están almacenados dichos elementos. Para realizar estos recorridos hemos hecho uso de los iteradores propios de cada tipo de datos. Este es un tipo de dato que no habíamos utilizado hasta ahora por lo que damos un paso más en la utilización de C++ en este tutorial. Una de las utilidades de los iteradores es recorrer tipos de datos que no podemos indexar como un vector. Para ello definimos un iterador del tipo de la estructura a recorrer y definimos dicho recorrido desde el comienzo de la secuencia (*lista.begin()*), hasta el final de la misma (*lista.end()*) incrementando en una unidad dicho iterador por cada vuelta del bucle. Como puedes ver no es un concepto complicado pero si es muy útil para manejar las secuencias que nos proporciona la STL. Al final de este método hacemos una llamada a la función *eliminar_antiguos()* que eliminará aquellos elementos con tenga el estado ELIMINAR debido a alguna acción del juego.

16.13.4. La clase Control Movimiento

La clase *Control Movimiento* mueve determinados elementos de la aplicación dotándolos de un efecto especial. En nuestro caso los utilizamos en el menú de la pantalla principal para que los textos aparezcan mediante un efecto de movimiento progresivo.

El diseño de la clase del que partimos para la implementación es el de la figura 16.37.

16.13. Implementación

Control Movimiento
<pre>- x : int - y : int - x_destino : int - y_destino : int - indice : int - imagen : Imagen* + Control_Movimiento(imagen : Imagen*, indice : int, x : int, y : int) + actualizar() + dibujar(pantalla : SDL_Surface*) + mover(x : int, y : int) + mover_inmediatamente(x : int, y : int)</pre>

Figura 16.37: Clase Control Movimiento

Vamos a estudiar la definición de la clase:

```
;_____
1 ;// Listado: Control_Movimiento.h
2 ;//
3 ;// La clase Control Movimiento controla la animación externa de los personajes
4 ;// lo que es lo mismo, el movimiento sobre el mapa del nivel
5 ;
6 ;
7 ;#ifndef _CONTROL_MOVIMIENTO_H_
8 ;#define _CONTROL_MOVIMIENTO_H_
9 ;
10 ;#include <SDL/SDL.h>
11 ;
12 ;
13 ;class Imagen; // Declaración adelantada
14 ;
15 ;
16 ;class Control_Movimiento {
17 ;
18 ; public:
19 ;     //Constructor
20 ;
21 ;     Control_Movimiento(Imagen *imagen, int indice = 0, int x = 0, int y = 0);
22 ;
23 ;     // Modificadoras
24 ;
25 ;     void actualizar(void);
26 ;     void dibujar(SDL_Surface *pantalla);
27 ;
28 ;     void mover(int x, int y);
29 ;     void mover_inmediatamente(int x, int y);
30 ;
31 ; private:
32 ;
33 ;     // Posición
34 ;     int x, y;
35 ;
36 ;     // Destino
37 ;     int x_destino, y_destino;
```

16. Un ejemplo del desarrollo software de un videojuego

```
38 ;
39 ;    // Dentro de la rejilla especificamos qué imagen
40 ;    int indice;
41 ;
42 ;    // Rejilla de imágenes
43 ;    Imagen *imagen;
44 ;
45 ;};
46 ;
47 ;#endif
;
```

En la parte privada de la clase tenemos dos variables *x* e *y* que nos permiten almacenar la posición actual del elemento a mover así como *x_destino* e *y_destino* nos permiten especificar a dónde queremos realizar el movimiento lógico de la imagen que controla esta clase. Para saber qué imagen vamos a mover la clase realiza una asociación mediante una variable *imagen* de tipo *Imagen* y utiliza un indicador almacenado en la variable *indice* que nos permite seleccionar un determinado cuadro de una rejilla.

En la parte pública tenemos varios métodos que vamos a estudiar con la implementación de la clase que vemos a continuación.

```
;_____
1 ;// Listado: Control_Movimiento.cpp
2 ;//
3 ;// Implementación de la clase Control Movimiento
4 ;
5 ;
6 ;#include <iostream>
7 ;
8 ;#include "Control_Movimiento.h"
9 ;#include "Imagen.h"
10 ;
11 ;using namespace std;
12 ;
13 ;Control_Movimiento::Control_Movimiento(Imagen *imagen, int indice, int x, int y) {
14 ;
15 ;    // Inicializamos los atributos de la clase
16 ;
17 ;    this->imagen = imagen;
18 ;    this->x = x;
19 ;    this->y = y;
20 ;    x_destino = x;
21 ;    y_destino = y;
22 ;    this->indice = indice;
23 ;
24 ;#ifdef DEBUG
25 ;    cout << "Control_Movimiento::Control_Movimiento()" << endl;
26 ;#endif
27 ;}
28 ;
29 ;
```

16.13. Implementación

```
30 ;void Control_Movimiento::actualizar(void) {
31 ;
32 ;    // Actualización de la posición del elemento
33 ;
34 ;    int dx = x_destino - x;
35 ;    int dy = y_destino - y;
36 ;
37 ;    // Si hay movimiento
38 ;
39 ;    if(dx != 0) {
40 ;
41 ;        // Controlamos la precisión del movimiento
42 ;
43 ;        if(abs(dx) >= 4) // Si es mayor o igual que 4
44 ;            x += dx / 4; // la reducimos
45 ;        else
46 ;            x += dx / abs(dx); // Si es menor, reducimos a 1
47 ;    }
48 ;
49 ;    if(dy != 0) {
50 ;
51 ;        // Ídem para el movimiento vertical
52 ;
53 ;        if(abs(dy) >= 4)
54 ;            y += dy / 4;
55 ;        else
56 ;            y += dy / abs(dy);
57 ;    }
58 ;}
59 ;
60 ;
61 ;
62 ;void Control_Movimiento::dibujar(SDL_Surface *pantalla) {
63 ;
64 ;    imagen->dibujar(pantalla, indice, x, y, 1);
65 ;}
66 ;
67 ;
68 ;
69 ;void Control_Movimiento::mover(int x, int y) {
70 ;
71 ;    // Este movimiento es actualizado por la función actualizar()
72 ;    // de esta clase con la precisión que tenemos implementada
73 ;    // en ella
74 ;
75 ;    x_destino = x;
76 ;    y_destino = y;
77 ;}
78 ;
79 ;
80 ;void Control_Movimiento::mover_inmediatamente(int x, int y) {
81 ;
82 ;    // Esta función nos permite hacer un movimiento inmediato
```

16. Un ejemplo del desarrollo software de un videojuego

```
83 ; // a una determinada posición  
84 ;  
85 ;     mover(x,y);  
86 ;     this->x = x;  
87 ;     this->y = y;  
88 ;}
```

El primero que vemos es el constructor de la clase que se encarga de inicializar los atributos propios de la clase. El método *actualizar()* desplaza la imagen que controla reduciendo la velocidad de aproximación si se encuentra cerca de la posición de destino. Para conseguir este efecto vamos comprobando si el diferencial de distancia entre la posición actual y la de destino es menor que cuatro se reduce a la unidad mientras que si es mayor reducimos dicha distancia en una cuarta parte. Este movimiento es aplicado tanto en el eje horizontal como en el vertical.

El método *dibujar()* muestra por pantalla el cuadro específico de la imagen asociada mediante un método de la clase *Imagen*. El método *mover()* establece las variables para que el método *actualizar* se encargue de realizar el movimiento lógico de la imagen mientras que el método *mover_inmediatamente* fija mueve dicha imagen sin realizar el efecto, simplemente colocándola en el lugar correspondiente.

16.13.5. La clase Editor

La clase *Editor* controla todo lo referente al interfaz del editor de niveles que nos permitirá construir los distintos niveles para nuestro videojuego. Este editor dispone de una superficie editable donde poder realizar las modificaciones y una barra de herramientas con los elementos que podemos insertar en dicha área. Además todas las modificaciones las realizaremos con el ratón por lo que dispone de un dispositivo apuntador asociado a él.

El diseño de la clase del que partimos para la implementación es el de la figura 16.38.

Vamos a estudiar la definición de la clase:

```
;  
1 ;// Listado: Editor.h  
2 ;//  
3 ;// Controla las características del editor de niveles  
4 ;  
5 ;#ifndef _EDITOR_H_  
6 ;#define _EDITOR_H_  
7 ;  
8 ;#include <SDL/SDL.h>
```

16.13. Implementación



Figura 16.38: Clase Editor

```
9 ;#include "Interfaz.h"
10 ;
11 ;// Declaraciones adelantadas
12 ;
13 ;class Universo;
14 ;class Nivel;
15 ;class Teclado;
16 ;class Imagen;
17 ;class Apuntador;
18 ;
19 ;// Definición de la clase
20 ;
21 ;class Editor: public Interfaz {
22 ;
23 ;    public:
24 ;
25 ;        // Constructor
26 ;
27 ;        Editor(Universo *universo);
28 ;
29 ;        // Acciones de la clase
30 ;
31 ;        void reiniciar(void);
32 ;        void actualizar(void);
33 ;        void dibujar(void);
34 ;
35 ;        // Barra de menú
36 ;
37 ;        void mover_barra_scroll(int incremento);
38 ;        int barra_scroll(void);
```

16. Un ejemplo del desarrollo software de un videojuego

```
39 ;
40 ;    // Destructor
41 ;
42 ;    ~Editor();
43 ;
44 ;    Nivel *nivel;
45 ;    Teclado *teclado;
46 ;    Imagen *imagen;
47 ;
48 ; private:
49 ;
50 ;    int x, y;
51 ;    int barra_scroll_;
52 ;
53 ;    Apuntador *apuntador;
54 ;
55 ;    void mover_ventana(void);
56 ;    void dibujar_menu(void);
57 ;    void dibujar_numero_nivel(void);
58 ;};
59 ;
60 ;#endif
;-----
```

En la parte privada de la clase tenemos tres atributos. El atributo *apuntador* asocia un dispositivo apuntador al editor para poder realizar las modificaciones en el área editable. El atributo *barra_scroll_* nos permite controlar si el área donde se ha realizado un click forma parte de la barra de herramientas o de la superficie editable.

Además de estos atributos en la parte privada de la clase tenemos tres métodos que definen parte del comportamiento del editor. El primero de ellos, *mover_ventana()*, posiciona la ventana segúna el lugar (*x, y*) donde se encuentre el apuntador del ratón. El método *dibujar_menu()* se encarga de mostrar en pantalla la barra de herramientas que nos permite seleccionar las herramientas para editar el nivel. El último método de esta parte, *dibujar_numero_nivel()* se encarga de mostrarnos en pantalla el número de nivel que estamos editando.

En la parte pública de la clase tenemos varios métodos que vamos a estudiar con la implementación de la clase:

```
;-----
1 ;// Listado: Editor.cpp
2 ;//
3 ;// Implementación de la clase esditor
4 ;
5 ;#include <iostream>
6 ;
7 ;#include "Editor.h"
8 ;#include "Universo.h"
9 ;#include "Nivel.h"
```

16.13. Implementación

```
10 ;#include "Galeria.h"
11 ;#include "Imagen.h"
12 ;#include "Teclado.h"
13 ;#include "Apuntador.h"
14 ;#include "Fuente.h"
15 ;#include "Musica.h"
16 ;#include "CommonConstants.h"
17 ;
18 ;using namespace std;
19 ;
20 ;
21 ;// Constructor
22 ;
23 ;Editor::Editor(Universo *universo) : Interfaz(universo) {
24 ;
25 ;#ifdef DEBUG
26 ;    cout << "Editor::Editor()" << endl;
27 ;#endif
28 ;
29 ;    // Inicializamos los atributos de la clase
30 ;
31 ;    nivel = new Nivel(universo, FILAS_ZONA_EDITABLE, COLUMNAS_ZONA_EDITABLE);
32 ;
33 ;    // Acciones de teclado en el editor
34 ;
35 ;    teclado = &(universo->teclado);
36 ;
37 ;    // Apuntador del ratón para el editor
38 ;
39 ;    apuntador = new Apuntador(this);
40 ;
41 ;    barra_scroll_ = 0;
42 ;    x = 0;
43 ;    y = 0;
44 ;
45 ;    // Imagen con los tiles para el editor
46 ;
47 ;    imagen = universo->galeria->imagen(Galeria::TILES);
48 ;}
49 ;
50 ;void Editor::reiniciar(void) {
51 ;
52 ;    // Hacemos sonar la música
53 ;
54 ;    universo->galeria->musica(Galeria::MUSICA_EDITOR)->pausar();
55 ;    universo->galeria->musica(Galeria::MUSICA_EDITOR)->reproducir();
56 ;
57 ;}
58 ;
59 ;
60 ;
61 ;void Editor::actualizar(void)
62 ;{
```

16. Un ejemplo del desarrollo software de un videojuego

```
63 ;
64 ;    // Mueve la ventana a la posición correspondiente
65 ;
66 ;    mover_ventana();
67 ;
68 ;    // Actualiza los elementos del Editor
69 ;
70 ;    nivel->actualizar();
71 ;
72 ;    // Actualiza la posición del puntero del ratón
73 ;
74 ;    apuntador->actualizar();
75 ;
76 ;    // Si se pulsa la tecla de salida se vuelve al menú
77 ;
78 ;    if(teclado->pulso(Teclado::TECLA_SALIR))
79 ;        universo->cambiar_interfaz(ESCENA_MENU);
80 ;}
81 ;
82 ;
83 ;void Editor::mover_barra_scroll(int desplazamiento) {
84 ;
85 ;    // Incrementamos la posición de la barra
86 ;
87 ;    barra_scroll_ = barra_scroll_ + desplazamiento;
88 ;
89 ;    // Dentro de los parámetros permitidos
90 ;
91 ;    if(barra_scroll_ < 0)
92 ;        barra_scroll_ = 0;
93 ;
94 ;    if(barra_scroll_ > 5)
95 ;        barra_scroll_ = 5;
96 ;}
97 ;
98 ;
99 ;int Editor::barra_scroll(void) {
100 ;
101 ;    return (barra_scroll_);
102 ;}
103 ;
104 ;
105 ;Editor::~Editor() {
106 ;
107 ;    delete nivel;
108 ;    delete apuntador;
109 ;
110 ;#ifndef DEBUG
111 ;    cout << "Editor::~Editor() " << endl;
112 ;#endif
113 ;
114 ;}
115 ;
```

16.13. Implementación

```
116 ;
117 ;void Editor::mover_ventana(void) {
118 ;
119 ;    int x0 = x;
120 ;    int y0 = y;
121 ;
122 ;    // Según la pulsación de la tecla movemos la ventana de la aplicación
123 ;    // en un sentido u otro
124 ;
125 ;    if(teclado->pulso(Teclado::TECLA_IZQUIERDA))
126 ;        x -= 20;
127 ;
128 ;    else if(teclado->pulso(Teclado::TECLA_DERECHA))
129 ;        x += 20;
130 ;
131 ;    else if(teclado->pulso(Teclado::TECLA_BAJAR))
132 ;        y += 20;
133 ;
134 ;    else if(teclado->pulso(Teclado::TECLA_SUBIR))
135 ;        y -= 20;
136 ;
137 ;    // Si existe movimiento
138 ;
139 ;    if(x != x0 || y != y0) {
140 ;
141 ;        nivel->ventana->establecer_pos(x, y);
142 ;        nivel->ventana->tomar_pos_final(&x, &y);
143 ;
144 ;    }
145 ;}
146 ;
147 ;
148 ;
149 ;void Editor::dibujar(void)
150 ;{
151 ;    // Dibuja todos los elementos del editor
152 ;
153 ;    nivel->dibujar(universo->pantalla);
154 ;    nivel->dibujar_actores(universo->pantalla);
155 ;
156 ;    // Mostramos el menú
157 ;
158 ;    dibujar_menu();
159 ;
160 ;    apuntador->dibujar(universo->pantalla); // Mostramos el puntero del ratón
161 ;    dibujar_numero_nivel(); // Mostramos el número del nivel actual
162 ;
163 ;
164 ;    // Actualizamos la pantalla
165 ;
166 ;    SDL_Flip(universo->pantalla);
167 ;}
168 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
169 ;
170 ;
171 ;void Editor::dibujar_menu(void) {
172 ;
173 ;    int i, j;
174 ;
175 ;    // Fondo
176 ;
177 ;    universo->dibujar_rect(544, 0, 96, 480,\n                                SDL_MapRGB(universo->pantalla->format, 25, 10, 50));
178 ;
179 ;
180 ;    imagen->dibujar(universo->pantalla, 48, 544, 0);
181 ;    imagen->dibujar(universo->pantalla, 49, 576, 0);
182 ;    imagen->dibujar(universo->pantalla, 50, 608, 0);
183 ;
184 ;    imagen->dibujar(universo->pantalla, 54, 544, 32);
185 ;    imagen->dibujar(universo->pantalla, 55, 576, 32);
186 ;    imagen->dibujar(universo->pantalla, 56, 608, 32);
187 ;
188 ;    // Flechas para el scrolling
189 ;
190 ;    imagen->dibujar(universo->pantalla, 51, 576, 92);
191 ;    imagen->dibujar(universo->pantalla, 52, 576, 448);
192 ;
193 ;    // Elementos
194 ;
195 ;    for(i = 0; i < 10; i++) {
196 ;        for(j = 0; j < 3 ; j++) {
197 ;
198 ;            imagen->dibujar(universo->pantalla,
199 ;                            (i + barra_scroll_) * 3 + j,
200 ;                            544 + j * 32, 96 + 32 + i * 32);
201 ;
202 ;        }
203 ;    }
204 ;
205 ;
206 ;void Editor::dibujar_numero_nivel(void) {
207 ;
208 ;    char numero_aux[20];
209 ;
210 ;    Fuente *fuente = universo->galeria->fuente(Galeria::FUENTE_MENU);
211 ;
212 ;    sprintf(numero_aux, "Nivel - %d", nivel->indice());
213 ;
214 ;    fuente->dibujar(universo->pantalla, numero_aux, 10, 10);
215 ;}
```

El primero de los métodos es el constructor de la clase. En este método inicializamos el nivel que vamos a editar definiendo la zona que va a ser editable de la pantalla. Asociamos un teclado y un ratón al editor. Inicializamos los atributos de la clase y cargamos la imagen que contienen los reacuadros o tiles que nos van a servir para llenar las partes del nivel que queramos dotar

de un cierto comportamiento.

El método *reiniciar()* reestablece la música del editor. El método *actualizar()* se encarga de que la posición de la ventana que nos permite navegar por el nivel sea la correcta así como de actualizar la posición del apuntador del ratón y del estado del nivel que estamos editando. Si pulsamos la tecla de salida este método se encarga de que salgamos del editor al menú principal.

El método *mover_barra_scroll()* mueve dicha barra siempre que el desplazamiento sea mayor que 5 píxeles. *mover_ventana()* como puedes observar mueve la ventana veinte píxeles hacia el lugar que hayamos indicado por medio del teclado y los métodos *dibujar...()* muestran por pantalla todas las imágenes que componen el editor.

16.13.6. La clase Enemigo

La clase enemigo nos permite crear adversarios en el videojuego. Esta clase es hija de la clase participante y tiene una implementación muy básica.

El diseño de la clase del que partimos para la implementación es el de la figura 16.39.

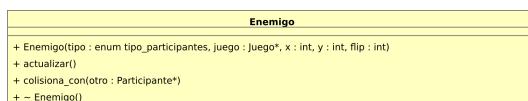


Figura 16.39: Clase Enemigo

Vamos a estudiar la definición de la clase:

```

1 ;_____
2 ;// Listado: Enemigo.h
3 ;//
4 ;// Clase Enemigo, heredada de Participante, para el control
5 ;// de los adversarios del juego
6 ;
7 ;#ifndef _ENEMIGO_H_
8 ;#define _ENEMIGO_H_
9 ;
10 ;
11 ;class Juego;
12 ;
13 ;class Enemigo : public Participante {
14 ;
15 ; public:
16 ;
17 ;     // Constructor

```

16. Un ejemplo del desarrollo software de un videojuego

```
18 ;
19 ;     Enemigo(enum tipo_participantes tipo, Juego *juego, int x, int y, int flip = 1);
20 ;
21 ;     // Modificadora
22 ;
23 ;     void actualizar(void);
24 ;
25 ;     // Consultora
26 ;
27 ;     void colisiona_con(Participante *otro);
28 ;
29 ;     virtual ~Enemigo();
30 ;
31 ;};
32 ;
33 ;
34 ;#endif
;
```

Todos los elementos que definimos en esta clase son públicos ya que los privados son heredados. Todos los elementos de los que disponemos son métodos que nos permiten interactuar con los enemigos. Vamos a ver la implementación de estos métodos y vamos a estudiar cada uno de ellos:

```
;_____
1 ;// Listado: Enemigo.cpp
2 ;//
3 ;// Implementación de la clase Enemigo
4 ;
5 ;#include <iostream>
6 ;
7 ;#include "Enemigo.h"
8 ;#include "Juego.h"
9 ;#include "Universo.h"
10 ;#include "Galeria.h"
11 ;#include "Sonido.h"
12 ;#include "Imagen.h"
13 ;#include "Control_Animacion.h"
14 ;#include "Nivel.h"
15 ;
16 ;
17 ;using namespace std;
18 ;
19 ;
20 ;Enemigo::Enemigo(enum tipo_participantes tipo, Juego *juego, int x, int y, int direccion):
21 ;    Participante(juego, x, y, direccion) {
22 ;
23 ;#ifndef DEBUG
24 ;    cout << "Enemigo::Enemigo()" << endl;
25 ;#endif
26 ;
27 ;    // Creamos las animaciones para el personaje
28 ;
29 ;    animaciones[PARADO] = new Control_Animacion("0", 5);
```

16.13. Implementación

```
30 ;     animaciones[CAMINAR] = new Control_Animacion("0,1,2", 5);
31 ;     animaciones[MORIR] = new Control_Animacion("1,2,1,2,1,2", 1);
32 ;
33 ;     // Según el tipo de enemigo que estemos creando
34 ;
35 ;     switch(tipo) {
36 ;
37 ;         case TIPO_ENEMIGO_RATA:
38 ;
39 ;             imagen = juego->universo->galeria->imagen(Galeria::ENEMIGO_RATA);
40 ;             break;
41 ;
42 ;         case TIPO_ENEMIGO_MOTA:
43 ;
44 ;             imagen = juego->universo->galeria->imagen(Galeria::ENEMIGO_MOTA);
45 ;             break;
46 ;
47 ;     default:
48 ;         cout << "Enemigo::Enemigo() -> Enemigo no contemplado" << endl;
49 ;
50 ;     }
51 ;
52 ;     // Estado inicial para los enemigos
53 ;
54 ;     estado = CAMINAR;
55 ;}
56 ;
57 ;
58 ;
59 ;void Enemigo::actualizar(void) {
60 ;
61 ;    if(estado == MORIR) {
62 ;
63 ;        if(animaciones[estado]->avanzar())
64 ;            estado = ELIMINAR;
65 ;    }
66 ;
67 ;    else
68 ;        // Hacemos avanzar la animación
69 ;        animaciones[estado]->avanzar();
70 ;
71 ;        // Tiene que llegar al suelo
72 ;
73 ;        velocidad_salto += 0.1;
74 ;        y += altura((int) velocidad_salto);
75 ;
76 ;        if(pisa_el_suelo()) {
77 ;
78 ;            // Hacemos que gire si se acaba el suelo
79 ;
80 ;            if(!pisa_el_suelo( x + direccion, y))
81 ;                direccion *= -1;
82 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
83 ;         x += direccion;
84 ;
85 ;
86 ;}
87 ;
88 ;void Enemigo::colisiona_con(Participante *otro) {
89 ;
90 ;    // Si colisiona y el personaje principal está golpeando
91 ;    // muere para desaparecer
92 ;
93 ;    if(estado != MORIR) {
94 ;
95 ;        juego->universo->\n
96 ;            galeria->sonidos[Galeria::MATA_MALO]->reproducir();
97 ;
98 ;        estado = MORIR;
99 ;    }
100 ;
101 ;
102 ;}
103 ;
104 ;
105 ;Enemigo::~Enemigo() {
106 ;
107 ;#ifndef DEBUG
108 ;    cout << "Enemigo::~Enemigo" << endl;
109 ;#endif
110 ;
111 ;}
```

El constructor de la clase se encarga de inicializar todos los métodos de la clase. Lo primero que realizamos es la inicialización de las animaciones que van a responder a los diferentes estados del personaje según transcurra el videojuego. Seguidamente, según sea el tipo de enemigo que queremos utilizar, cargamos de la galería una imágenes u otras. El estado inicial para los enemigos será el de caminar.

El siguiente método es el de *actualizar()*. Lo primero que hace este método es comprobar que el enemigo no ha muerto. De ser así se avanza en la animación y se marca su estado con la etiqueta ELIMINAR para que al actualizar la lógica del juego haga desaparecer este elemento de la pantalla. Si el enemigo no ha muerto todavía el método hace avanzar la animación del enemigo. En caso de no estar en el suelo se le hace caer con una velocidad que aumenta según vaya cayendo el personaje.

El método *colisiona_con()* define que acciones han de realizarse cuando se produce una colisión con el elemento que recibe como parámetro. En este caso las colisiones sólo pueden producirse con el personaje principal. En el caso de que se informe a un objeto de esta clase de una colisión esta provocará que se pase el elemento al estado MORIR para que se reproduzca la animación y sea

eliminado.

16.13.7. La clase Fuente

La clase *Fuente* nos permite dibujar en una superficie SDL cualquier una palabra o un texto que queramos mostrar a partir de una fuente TTF o de una rejilla de letras tipográficas.

El diseño de la clase del que partimos para la implementación es el de la figura 16.40.

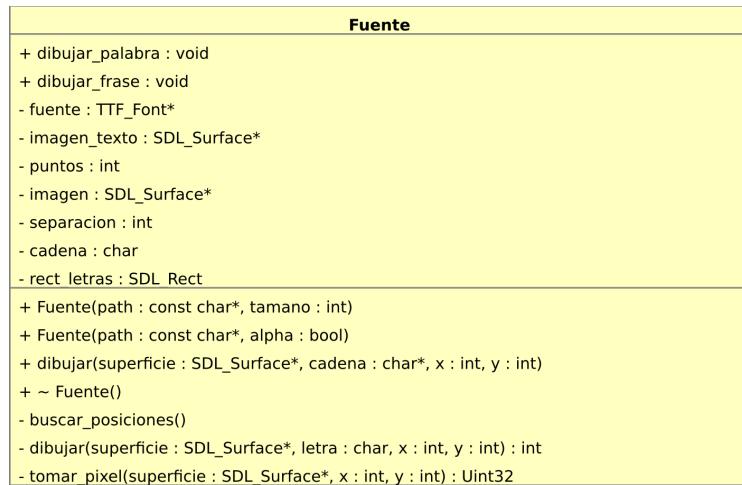


Figura 16.40: Clase Fuente

Vamos a estudiar la definición de la clase:

```

1 ;_____
2 ;// Listado: Fuente.h
3 ;//
4 ;// Esta clase controla la impresión de textos en pantalla
5 ;// mediante fuentes ttf o tira de imágenes
6 ;
7 ;#ifndef _FUENTE_H_
8 ;#define _FUENTE_H_
9 ;
10 ;#include <SDL/SDL.h>
11 ;#include <SDL/SDL_image.h>
12 ;#include <SDL/SDL_ttf.h>
13 ;
14 ;#include "CommonConstants.h"
15 ;
16 ;
17 ;class Fuente {
  
```

16. Un ejemplo del desarrollo software de un videojuego

```
18 ;
19 ; public:
20 ;
21 ; // Constructores
22 ;
23 ; Fuente(const char *path, int tamano = 20); // Fuente ttf
24 ; Fuente(const char *path, bool alpha = false); // Fuente ya renderizada
25 ;
26 ;
27 ; // Funciones para escribir sobre una superficie
28 ;
29 ; void dibujar(SDL_Surface *superficie, char *cadena, int x, int y);
30 ;
31 ;
32 ; // Funciones para escribir utilizando una fuente ttf
33 ;
34 ; void dibujar_palabra(SDL_Surface *superficie,
35 ;                         char *palabra, int x, int y, SDL_Color color);
36 ;
37 ; void dibujar_frase(SDL_Surface *superficie,
38 ;                         char *frase, int x, int y, SDL_Color color);
39 ;
40 ; // Destructor
41 ; ~Fuente();
42 ;
43 ; private:
44 ;
45 ; TTF_Font *fuente; // Para manejar la fuente TTF
46 ; SDL_Surface *imagen_texto; // Imagen obtenida a partir de la fuente
47 ;
48 ; // Tamaño de la fuente
49 ;
50 ; int puntos;
51 ;
52 ;
53 ; // Dónde almacenar la imagen resultante para mostrar
54 ;
55 ; SDL_Surface *imagen;
56 ;
57 ; int separacion; // entre las letras
58 ;
59 ;
60 ; // Define el orden de las letras en la rejilla-imagen
61 ;
62 ; char cadena[113 + 1];
63 ;
64 ;
65 ; // Mantiene la posición de cada letra en la rejilla-imagen
66 ;
67 ; SDL_Rect rect_letras[113];
68 ;
69 ; void buscar_posiciones(void);
70 ;
```

16.13. Implementación

```
71 ;
72 ;    // Dibuja una letra única
73 ;
74 ;    int dibujar(SDL_Surface *superficie, char letra, int x, int y);
75 ;
76 ;
77 ;    // Toma un pixel de una superficie
78 ;
79 ;    Uint32 tomar_pixel(SDL_Surface *superficie, int x, int y);
80 ;
81 ;
82 ;};
83 ;
84 ;
85 ;#endif
```

En la parte privada de la clase tenemos varios elementos. El primero es una variable de tipo `TTF_Font` que nos servirá para hacer uso de la librería `SDL_ttf` y el segundo una variable de tipo `SDL_Surface` por si optamos por la utilización de la rejilla de letras tipográficas. El atributo *puntos* sirve para almacenar el tamaño de la fuente a utilizar. Los métodos privados de la clase los estudiaremos cuando veamos la implementación de la misma.

En la parte pública de la clase se ofrecen varios métodos. Disponemos de dos constructores según el método que vayamos a utilizar. El método *dibujar()* que nos permite escribir una cadena en una superficie SDL y dos métodos auxiliares de éste que son *dibujar_palabra()* y *dibujar_frase()* que son una especificación del método *dibujar()*.

Para saber cómo y qué hace cada método vamos a estudiar la implementación de la clase.

```
1 ;// Listado: Fuente.cpp
2 ;//
3 ;// Implementación de la clase Fuente
4 ;
5 ;#include <iostream>
6 ;#include "Fuente.h"
7 ;
8 ;
9 ;using namespace std;
10 ;
11 ;
12 ;// Mediante fuente ttf
13 ;
14 ;Fuente::Fuente(const char *path, int tamano) {
15 ;
16 ;    // Iniciamos la librería SDL_ttf
17 ;
18 ;    if(TTF_Init() < 0) {
```

16. Un ejemplo del desarrollo software de un videojuego

```
19 ;
20 ;         cerr << "No se puede iniciar SDL_ttf" << endl;
21 ;         exit(1);
22 ;
23 ;     }
24 ;
25 ;     // Al terminar cerramos SDL_ttf
26 ;
27 ;     atexit(TTF_Quit);
28 ;
29 ;
30 ;     // Tamaño en puntos de la fuente
31 ;
32 ;     puntos = tamano;
33 ;
34 ;
35 ;     // Cargamos la fuente que queremos utilizar con un determinado tamaño
36 ;
37 ;     fuente = TTF_OpenFont(path, tamano);
38 ;
39 ;     if(fuente == NULL) {
40 ;
41 ;         cerr << "No se puede abrir la fuente deseada" << endl;
42 ;         exit(1);
43 ;
44 ;     }
45 ;
46 ;
47 ;
48 ;
49 ;// Mediante fuentes en una rejilla-imagen
50 ;
51 ;Fuente::Fuente(const char *path, bool alpha) {
52 ;
53 ;#ifdef DEBUG
54 ;    cout << "Fuente::Fuente(): " << path << endl;
55 ;#endif
56 ;
57 ;
58 ;    // Cargamos la imagen que contiene las letras
59 ;    // renderizadas a utilizar
60 ;
61 ;    imagen = IMG_Load(path);
62 ;
63 ;    if(imagen == NULL) {
64 ;
65 ;        cerr << "Fuente::Fuente(): " << SDL_GetError() << endl;
66 ;        exit(1);
67 ;
68 ;    }
69 ;
70 ;    if(!alpha) {
71 ;
```

16.13. Implementación

```
72 ;         // A formato de imagen
73 ;
74 ;     SDL_Surface *tmp = imagen;
75 ;
76 ;     imagen = SDL_DisplayFormat(tmp);
77 ;     SDL_FreeSurface(tmp);
78 ;
79 ;     if(imagen == NULL) {
80 ;
81 ;         cerr << "Fuente::Fuente(): " << SDL_GetError() << endl;
82 ;         exit(1);
83 ;
84 ;     }
85 ;
86 ;     // Establecemos el color key
87 ;
88 ;     Uint32 colorkey = SDL_MapRGB(imagen->format, 255, 0, 255);
89 ;     SDL_SetColorKey(imagen, SDL_SRCCOLORKEY, colorkey);
90 ;
91 ; }
92 ;
93 ; buscar_posiciones();
94 ;}
95 ;
96 ;
97 ;
98 ;
99 ;void Fuente::dibujar_palabra(SDL_Surface *superficie, char *palabra, \
100 ;                                int x, int y, SDL_Color color) {
101 ;
102 ;     // Renderizamos y almacenamos en una superficie
103 ;
104 ;     imagen = TTF_RenderText_Solid(fuente, palabra, color);
105 ;
106 ;
107 ;     // Convertimos la imagen obtenida a formato de pantalla
108 ;
109 ;     SDL_Surface *tmp = imagen;
110 ;     imagen = SDL_DisplayFormat(tmp);
111 ;     SDL_FreeSurface(tmp);
112 ;
113 ;
114 ;     // Colocamos en la pantalla principal en(x, y)
115 ;
116 ;     SDL_Rect destino;
117 ;
118 ;     destino.x = x;
119 ;     destino.y = y;
120 ;     destino.w = imagen->w;
121 ;     destino.h = imagen->h;
122 ;
123 ;     SDL_BlitSurface(imagen, NULL, superficie, &destino);
124 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
125 ;}
126 ;
127 ;
128 ;// Construimos frases con fuentes ttf cuyo espacio no sea compatible
129 ;// con SDL_ttf
130 ;
131 ;void Fuente::dibujar_frase(SDL_Surface *superficie, char *frase, \
132 ;                                int x, int y, SDL_Color color) {
133 ;
134 ;    int offset_x = 0; // Distancia entre una letra y la siguiente
135 ;
136 ;    // Separamos la frase en palabras
137 ;
138 ;    int k = 0;
139 ;
140 ;    // Mientras no termine la cadena
141 ;
142 ;    while(frase[k] != '\0') {
143 ;
144 ;        int i = 0;
145 ;        char palabra[MAX_LONG_PAL];
146 ;
147 ;        for(int j = 0; j < MAX_LONG_PAL; j++)
148 ;            palabra[j] = '\0';
149 ;
150 ;        // Por si empieza por espacio
151 ;
152 ;        while(frase[k] == ' ')
153 ;            k++;
154 ;
155 ;        // Hasta encontrar un espacio o el final de la cadena
156 ;
157 ;        while(frase[k] != ' ' && frase[k] != '\0') {
158 ;            palabra[i] = frase[k];
159 ;            i++; k++;
160 ;        }
161 ;
162 ;        if(frase[k] != '\0')
163 ;            k++;
164 ;
165 ;
166 ;        // No utilizamos la función anterior para facilitar el
167 ;        // cálculo del offset
168 ;
169 ;        // Renderizamos y almacenamos en una superficie
170 ;
171 ;        imagen = TTF_RenderText_Solid(fuente, palabra, color);
172 ;
173 ;
174 ;        // Convertimos a formato de pantalla
175 ;
176 ;        SDL_Surface *tmp = imagen;
177 ;        imagen = SDL_DisplayFormat(tmp);
```

16.13. Implementación

```
178 ;     SDL_FreeSurface(tmp);
179 ;
180 ;
181 ;     // Colocamos en la pantalla principal en (x, y)
182 ;
183 ;     SDL_Rect destino;
184 ;
185 ;     destino.x = x + offset_x;
186 ;     destino.y = y;
187 ;     destino.w = imagen->w;
188 ;     destino.h = imagen->h;
189 ;
190 ;     // Calculamos el offset entre letras
191 ;
192 ;     offset_x = offset_x + imagen->w + puntos / 3;
193 ;
194 ;     SDL_BlitSurface(imagen, NULL, superficie, &destino);
195 ; }
196 ;}
197 ;
198 ;
199 ;
200 ;Fuente::~Fuente() {
201 ;
202 ;     // Liberamos la superficie
203 ;
204 ;     SDL_FreeSurface(imagen);
205 ;}
206 ;
207 ;
208 ;
209 ;void Fuente::dibujar(SDL_Surface *superficie, char *texto, int x, int y)
210 ;{
211 ;     int i;
212 ;     int aux = 0;
213 ;
214 ;     for(i = 0; texto[i] != '\0'; i++) {
215 ;
216 ;         // Dibujamos carácter a carácter la frase
217 ;
218 ;         aux = dibujar(superficie, texto[i], x, y);
219 ;         x = x + aux + 2;
220 ;     }
221 ;}
222 ;
223 ;
224 ;
225 ;int Fuente::dibujar(SDL_Surface *superficie, char letra, int x, int y)
226 ;{
227 ;
228 ;     static int dep = 0;
229 ;
230 ;     if(letra == ' ') {
```

16. Un ejemplo del desarrollo software de un videojuego

```
231 ;
232 ;      // Espacio en la rejilla
233 ;
234 ;      x += 16;
235 ;      return 16;
236 ;
237 ; }
238 ;
239 ; bool encuentra = false;
240 ;
241 ; int i;
242 ;
243 ; // Buscamos si disponemos de representación gráfica para el carácter
244 ;
245 ; for(i = 0; cadena[i] != '\0' && encuentra == false; i++) {
246 ;
247 ;     if(cadena[i] == letra)
248 ;         encuentra = true;
249 ;
250 ;     // Realiza un incremento más de i
251 ; }
252 ;
253 ; i--; // i era el '\0'
254 ;
255 ;
256 ; // Comprobamos si el carácter ha sido encontrado
257 ;
258 ; if(encuentra == false && dep == 0) {
259 ;
260 ;     cerr << "No se encuentra el carácter: " << letra << endl;
261 ;     dep = 1;
262 ;
263 ;     return 0;
264 ; }
265 ;
266 ; SDL_Rect destino; // Lugar donde vamos a posicionar la letra
267 ;
268 ; destino.x = x;
269 ; destino.y = y;
270 ; destino.w = rect_letras[i].w;
271 ; destino.h = rect_letras[i].h;
272 ;
273 ; // Hacemos el blitting sobre la superficie
274 ;
275 ; SDL_BlitSurface(imagen, &rect_letras[i], superficie, &destino);
276 ;
277 ; // Devolvemos el ancho de la letra
278 ;
279 ; return rect_letras[i].w;
280 ;}
281 ;
282 ;
283 ;
```

16.13. Implementación

```
284 ;void Fuente::buscar_posiciones(void) {
285 ;
286 ;    const int LEYENDO = 0;
287 ;    const int DIVISION = 1;
288 ;
289 ;    int indice = 0;
290 ;    Uint32 color = 0;
291 ;    Uint32 negro = SDL_MapRGB(imagen->format, 0, 0, 0);
292 ;    int estado = DIVISION;
293 ;
294 ;
295 ;    strcpy(cadena, "abcdefghijklmnoprstuvwxyz" \
296 ;           "ABCDEFGHIJKLMNPQRSTUVWXYZ" \
297 ;           "1234567890" \
298 ;           "ñÑáéíóúÁÉÍÓÚäëíöü" \
299 ;           "!;?:@#$%&'+=-><*/,.:;-_( )[]{}|^~`~\\\" );
300 ;
301 ;    if(SDL_MUSTLOCK(imagen)) {
302 ;
303 ;        if(SDL_LockSurface(imagen) < 0) {
304 ;
305 ;            cerr << "No se puede bloquear " << imagen << " con " \
306 ;                << "SDL_LockSurface" << endl;
307 ;            return;
308 ;
309 ;        }
310 ;    }
311 ;
312 ;    indice = -1;
313 ;
314 ;    for(int x = 0; x < imagen->w; x++) {
315 ;
316 ;        color = tomar_pixel(imagen, x, 0);
317 ;
318 ;        if(estado == DIVISION && color == negro) {
319 ;
320 ;            estado = LEYENDO;
321 ;
322 ;            indice++;
323 ;
324 ;            rect_letras[indice].x = x;
325 ;            rect_letras[indice].y = 2;
326 ;            rect_letras[indice].h = imagen->h - 2;
327 ;            rect_letras[indice].w = 0;
328 ;
329 ;        }
330 ;
331 ;        if(color == negro)
332 ;            rect_letras[indice].w++;
333 ;        else
334 ;            estado = DIVISION;
335 ;    }
336 ;}
```

16. Un ejemplo del desarrollo software de un videojuego

```
337 ;     if(SDL_MUSTLOCK(imagen))
338 ;             SDL_UnlockSurface(imagen);
339 ;
340 ;
341 ;
342 ;// Fuente: SDL_Doc
343 ;
344 ;Uint32 Fuente::tomar_pixel(SDL_Surface *superficie, int x, int y)
345 ;{
346 ;    int bpp = superficie->format->BytesPerPixel;
347 ;    Uint8 *p =(Uint8 *)superficie->pixels + y * superficie->pitch + x * bpp;
348 ;
349 ;    switch(bpp) {
350 ;        case 1:
351 ;            return *p;
352 ;
353 ;        case 2:
354 ;            return *(Uint16 *)p;
355 ;
356 ;        case 3:
357 ;            if(SDL_BYTEORDER == SDL_BIG_ENDIAN)
358 ;                return p[0] << 16 | p[1] << 8 | p[2];
359 ;            else
360 ;                return p[0] | p[1] << 8 | p[2] << 16;
361 ;
362 ;        case 4:
363 ;            return *(Uint32 *)p;
364 ;
365 ;        default:
366 ;            return 0;
367 ;    }
368 ;}
```

El primer constructor utiliza una fuente ttf y `SDL_ttf` para proporcionar un texto en una superficie. Lo primero que hace el constructor es iniciar la citada librería ya que depende de ella. Una vez realizada esta tarea inicializa los atributos de la clase referentes a esta alternativa de trabajo.

El segundo constructor utiliza una rejilla de caracteres para mostrar un texto en pantalla. La idea es simple. Se prepara un BMP con el alfabeto listo para escoger una posición de la rejilla y mostrar una letra. Repitiendo esta acción podemos construir cualquier texto con unas fuentes personalizadas. Lo primero que hace este constructor es cargar la imagen donde está almacenada dichar rejilla. Seguidamente preparamos la imagen para que esté en el mismo formato que la superficie principal y establecemos el color key que utilizamos para toda la aplicación. Para terminar el constructor hace una llamada a `buscar_posiciones()`. Esta función es privada a esta clase y se encarga de establecer una relación entre la rejilla de imágenes y un vector de caracteres que nos permitirá seleccionar más ágilmente las letras que queremos colocar sobre una superficie.

La clase ofrece dos métodos para dibujar texto. El primero dibuja un texto en una superficie haciendo uso del otro método que nos permite dibujar una sola letra en dicha superficie. Dibujando letra tras letra podemos dibujar un texto, como no podía ser de otra forma.

Para terminar la clase implementa un método de la librería de SDL para comprobar que la letra que está tomando de la rejilla es una imagen y no un píxel de división de letras.

16.13.8. La clase Galeria

La clase *Galería* es una de las más importantes de nuestra aplicación. Gestiona todos los elementos multimedia que vamos a utilizar en el videojuego por lo que su implementación es crucial. Esta galería almacena cuatro tipo de elementos: imágenes, sonidos, fuentes y música. El fin de esta galería es tener una gestión controlada de dichos elementos que pueden utilizarse para muy diferentes fines.

Para cada uno de los elementos de la clase tenemos implementada una clase que nos permite envolver a cada elemento ofreciendo una mayor potencia que si almacenásemos el elemento en crudo.

El diseño de la clase del que partimos para la implementación es el de la figura 16.41.

Galeria
+ imagenes : map< codigo_imagen, Imagen * >
+ fuentes : map< codigo_fuente, Fuente * >
+ musicas : map< codigo_musica, Musica * >
+ sonidos : map< codigo_sonido, Sonido * >
+ Galeria()
+ imagen(cod_ima : codigo_imagen) : Imagen*
+ fuente(indice : codigo_fuente) : Fuente*
+ musica(cod_music : codigo_musica) : Musica*
+ sonido(cod_sonido : codigo_sonido) : Sonido*
+ ~ Galeria()

Figura 16.41: Clase Galeria

Vamos a estudiar la definición de la clase:

```
;  
1 // Listado: Galeria.h  
2 //
```

16. Un ejemplo del desarrollo software de un videojuego

```
3 ;// Controla los elementos multimedia de la aplicación
4 ;
5 ;#ifndef _GALERIA_H_
6 ;#define _GALERIA_H_
7 ;
8 ;#include <map>
9 ;#include "CommonConstants.h"
10 ;
11 ;
12 ;// Declaración adelantada
13 ;
14 ;class Imagen;
15 ;class Fuente;
16 ;class Musica;
17 ;class Sonido;
18 ;
19 ;
20 ;using namespace std;
21 ;
22 ;class Galeria {
23 ;
24 ; public:
25 ;
26 ;     // Tipos de imágenes contenidas en la galería
27 ;
28 ;     enum codigo_imagen {
29 ;
30 ;         TILES,
31 ;         PERSONAJE_PPAL,
32 ;         ENEMIGO_RATA,
33 ;         ENEMIGO_MOTA,
34 ;         MENU,
35 ;         TITULO_TUTORIAL,
36 ;         TITULO_FIRMA,
37 ;         ITEM_ALICATE,
38 ;         ITEM_DESTORNILLADOR
39 ;     };
40 ;
41 ;     // Fuentes almacenadas en la galería
42 ;
43 ;     enum codigo_fuente {
44 ;
45 ;         FUENTE_MENU
46 ;     };
47 ;
48 ;     enum codigo_musica {
49 ;
50 ;         MUSICA_MENU,
51 ;         MUSICA_EDITOR,
52 ;         MUSICA_JUEGO
53 ;
54 ;     };
55 ;
```

```

56 ; enum codigo_sonido {
57 ;
58 ;     COGE_ITEM,
59 ;     MATA_MALO,
60 ;     PASA_NIVEL,
61 ;     EFECTO_MENU,
62 ;     MUERE_BUENO
63 ;
64 ; };
65 ;
66 ;
67 ;
68 ; // Constructor
69 ;
70 ; Galeria ();
71 ;
72 ; // Consultoras
73 ;
74 ; Imagen *imagen(codigo_imagen cod_ima);
75 ; Fuente *fuente(codigo_fuente indice);
76 ; Musica *musica(codigo_musica cod_music);
77 ; Sonido *sonido(codigo_sonido cod_sonido);
78 ;
79 ; ~Galeria();
80 ;
81 ; // Conjunto de imágenes y de fuentes
82 ; // que vamos a utilizar en la aplicación
83 ;
84 ; map<codigo_imagen, Imagen *> imagenes;
85 ; map<codigo_fuente, Fuente *> fuentes;
86 ; map<codigo_musica, Musica *> musicas;
87 ; map<codigo_sonido, Sonido *> sonidos;
88 ;};
89 ;
90 ;#endif
;
```

Como puedes ver la definición de la clase es bastante sencilla. Tenemos unos enumerados para cada tipo de dato que vamos a almacenar que nos permiten trabajar más cómodamente a la hora de implementar el código del videojuego.

La clase ofrece varios métodos que nos permiten obtener cada uno de los componentes sólo con pasarle como parámetro el valor del enumerado correspondiente al tipo de elemento que queremos obtener. Así tenemos un método *imagen()*, *fuente()*, *musica()*, *sonido()* y *fuente()* que nos permite extraer un elemento de cada tipo de la galería.

Los elementos serán almacenados en elementos aplicación por lo que utilizaremos el tipo *map* definido en la STL. El elemento clave estará definido sobre los enumerados de la clase y el valor será un puntero al elemento que queremos gestionar.

16. Un ejemplo del desarrollo software de un videojuego

Veamos la implementación de la clase:

```
;_____
1 ;// Listado: Galeria.cpp
2 ;//
3 ;// Implementación de la clase galería del videojuego
4 ;
5 ;#include <iostream>
6 ;
7 ;#include "Galeria.h"
8 ;#include "Imagen.h"
9 ;#include "Fuente.h"
10 ;#include "Musica.h"
11 ;#include "Sonido.h"
12 ;
13 ;using namespace std;
14 ;
15 ;
16 ;Galeria::Galeria() {
17 ;
18 ;#ifdef DEBUG
19 ;    cout << "Galeria::Galeria()" << endl;
20 ;#endif
21 ;
22 ;    // Cargamos las rejillas en la galería para las animaciones
23 ;    // y las imágenes fijas
24 ;
25 ;    imagenes[PERSONAJE_PPAL] = new Imagen("Imagenes/personaje_principal.bmp", \
26 ;                                            4, 7, 45, 90, false);
27 ;    imagenes[TILES] = new Imagen("Imagenes/bloques.bmp", 10, 6);
28 ;    imagenes[ENEMIGO_RATA] = new Imagen("Imagenes/enemigo_rata.bmp", 1, 3, 45, 72, false);
29 ;    imagenes[ENEMIGO_MOTA] = new Imagen("Imagenes/enemigo_mota.bmp", 1, 3, 45, 72, false);
30 ;    imagenes[TITULO_TUTORIAL] = new Imagen("Imagenes/titulo_tutorial.bmp", 1, 1);
31 ;    imagenes[TITULO_FIRMA] = new Imagen("Imagenes/titulo_libsdl.bmp", 1, 1);
32 ;    imagenes[ITEM_ALICATE] = new Imagen("Imagenes/alicate.bmp", 1, 4, 25, 57);
33 ;    imagenes[ITEM_DESTORNILLADOR] = new Imagen("Imagenes/destornillador.bmp", 1, 4, 40, 18);
34 ;
35 ;    // Cargamos las fuentes en la galería
36 ;
37 ;    fuentes[FUENTE_MENU] = new Fuente("Imagenes/arial_black.png", true);
38 ;
39 ;    // Cargamos la música de la galería
40 ;
41 ;    musicas[MUSICA_MENU] = new Musica("Musica/menu.wav");
42 ;    musicas[MUSICA_EDITOR] = new Musica("Musica/editor.wav");
43 ;    musicas[MUSICA_JUEGO] = new Musica("Musica/juego.wav");
44 ;
45 ;    // Cargamos los sonidos de efectos
46 ;
47 ;    sonidos[COGE_ITEM] = new Sonido("Sonidos/item.wav");
48 ;    sonidos[MATA_MALO] = new Sonido("Sonidos/malo.wav");
49 ;    sonidos[PASA_NIVEL] = new Sonido("Sonidos/nivel.wav");
50 ;    sonidos[EFFECTO_MENU] = new Sonido("Sonidos/menu.wav");
```

16.13. Implementación

```
51 ;     sonidos[MUERE_BUENO] = new Sonido("Sonidos/muere.wav");
52 ;}
53 ;
54 ;
55 ;Imagen *Galeria::imagen(codigo_imagen cod_ima) {
56 ;
57 ;    // Devolvemos la imagen solicitada
58 ;
59 ;    return imagenes[cod_ima];
60 ;}
61 ;
62 ;
63 ;Fuente *Galeria::fuente(codigo_fuente indice) {
64 ;
65 ;    // Devolvemos la fuente solicitada
66 ;
67 ;    return fuentes[indice];
68 ;}
69 ;
70 ;Musica *Galeria::musica(codigo_musica cod_music) {
71 ;
72 ;    // Devolvemos la música solicitada
73 ;
74 ;    return musicas[cod_music];
75 ;
76 ;}
77 ;
78 ;Sonido *Galeria::sonido(codigo_sonido cod_sonido) {
79 ;
80 ;    // Devolvemos el sonido solicitado
81 ;
82 ;    return sonidos[cod_sonido];
83 ;
84 ;}
85 ;
86 ;
87 ;Galeria::~Galeria() {
88 ;
89 ;    // Descargamos la galería
90 ;
91 ;    delete imagenes[PERSONAJE_PPAL];
92 ;    delete imagenes[TILES];
93 ;    delete imagenes[ENEMIGO_RATA];
94 ;    delete imagenes[ENEMIGO_MOTA];
95 ;    delete imagenes[TITULO_TUTORIAL];
96 ;    delete imagenes[TITULO_FIRMA];
97 ;    delete imagenes[ITEM_ALICATE];
98 ;    delete imagenes[ITEM_DESTORNILLADOR];
99 ;
100 ;    delete fuentes[FUENTE_MENU];
101 ;
102 ;    delete musicas[MUSICA_MENU];
103 ;    delete musicas[MUSICA_JUEGO];
```

16. Un ejemplo del desarrollo software de un videojuego

```
104 ;     delete musicas[MUSICA_EDITOR];
105 ;
106 ;     delete sonidos[COGE_ITEM];
107 ;     delete sonidos[MATA_MALO];
108 ;     delete sonidos[PASA_NIVEL];
109 ;     delete sonidos[EFECTO_MENU];
110 ;     delete sonidos[MUERE_BUENO];
111 ;
112 ;#ifdef DEBUG
113 ;     cout << "Galeria::~Galeria()" << endl;
114 ;#endif
115 ;
116 ;
```

Como puedes ver la implementación de la clase no es más que un pequeño ejercicio de programación. El constructor inicializa todos los elementos de la galería que serán utilizados en el videojuego. El destructor libera los recursos utilizados por la galería.

Los métodos observadores son muy simples. Acceden a los *maps* y devuelven el valor consultado que ha sido pasado como parámetros

16.13.9. La clase Imagen

La clase imagen le da soporte a las rejillas de imágenes que vamos a utilizar en el videojuego. Se encarga de cargar una imagen que esté en cualquier formato compatible con `SDL_image` e indexar cada uno de los cuadros que la componen. En caso de ser una imagen única tendría un sólo cuadro. Esta forma de actuar es muy práctica a la hora de cargar animaciones ya que nos permite reducir el acceso a disco considerablemente.

Imagínate que para crear una animación hacen falta 20 imágenes. Podemos tener una rejilla con esas 20 imágenes y cargarlas solamente una vez en memoria principal. Con esta clase la tendríamos perfectamente indexada y el hecho de animar la imagen sólo supondría recorrer las imágenes de la rejilla.

El diseño de la clase del que partimos para la implementación es el de la figura 16.42.

Vamos a estudiar la definición de la clase:

```
1 ;_____
2 ;// Listado: Imagen.h
3 ;//
4 ;// Clase para facilitar el trabajo con imágenes
5 ;#ifndef _IMAGEN_H_
6 ;#define _IMAGEN_H_
```

16.13. Implementación

Imagen
+ Imagen : - imagen : SDL_Surface* - imagen_invertida : SDL_Surface* - columnas : int - filas : int - w : int - h : int - x0 : int - y0 : int + dibujar(superficie : SDL_Surface*, i : int, x : int, y : int, flip : int) + pos_x() : int + pos_y() : int + anchura() : int + altura() : int + cuadros() : int + ~Imagen() - invertir_imagen(imagen : SDL_Surface*) : SDL_Surface*

Figura 16.42: Clase Imagen

```
7 ;  
8 ;#include <SDL/SDL.h>  
9 ;  
10 ;  
11 ;class Imagen {  
12 ;  
13 ; public:  
14 ;  
15 ;     // Constructor  
16 ;  
17 ;     Imagen(char *ruta, int filas, int columnas,\  
18 ;             int x = 0, int y = 0, bool alpha = false);  
19 ;  
20 ;     void dibujar(SDL_Surface *superficie, int i, int x, int y, int flip = 1);  
21 ;  
22 ;     // Consultoras  
23 ;  
24 ;     int pos_x();  
25 ;     int pos_y();  
26 ;  
27 ;     int anchura();  
28 ;     int altura();  
29 ;     int cuadros();  
30 ;  
31 ;     // Destructor  
32 ;  
33 ;     ~Imagen();  
34 ;  
35 ; private:  
36 ;  
37 ;     SDL_Surface *imagen;  
38 ;     SDL_Surface *imagen_invertida;  
39 ;  
40 ;     // Propiedades de la rejilla de la imagen  
41 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
42 ;     int columnas, filas;
43 ;
44 ;     // Ancho y alto por frame o recuerdo de la animación
45 ;
46 ;     int w, h;
47 ;
48 ;     // Coordenadas origen
49 ;
50 ;     int x0, y0;
51 ;
52 ;     // Rota 180 grado en horizontal
53 ;
54 ;     SDL_Surface * invertir_imagen(SDL_Surface *imagen);
55 ;};
56 ;
57 ;#endif
```

La definición de la clase dota de cierta potencia al concepto de imagen. En la parte privada de la clase tenemos la superficie SDL donde almacenaremos la imagen. La superficie donde almacenaremos la invertida de esta imagen. Esta invertida se utiliza para reproducir un movimiento en dos sentidos con una sola imagen. Tanto de derecha a izquierdas como de izquierdas a derecha.

Dos atributos, *columnas* y *filas*, nos permiten almacenar el número de filas y columnas de nuestra imagen. El método que utilizamos para invertir la imagen lo estudiaremos con la implementación de la clase que veremos a continuación.

En la parte pública de la clase observamos los diferentes métodos que nos ofrece esta clase. Aparte del constructor y el destructor todos los demás métodos son observadores. Podemos obtener el tamaño de un cuadro de la rejilla de imágenes con los métodos *anchura()* y *altura()*, el número de cuadros de los que consta la rejilla con *caudors()*, la posición de una imagen en un determinado momento con *pos_x()* y *pos_y()*...

Veamos la implementación de la clase:

```
;_____
1 ;// Listado: Imagen.cpp
2 ;//
3 ;// Implementación de la clase imagen
4 ;
5 ;#include <iostream>
6 ;#include <SDL/SDL_image.h>
7 ;
8 ;#include "Imagen.h"
9 ;
10 ;using namespace std;
11 ;
12 ;Imagen::Imagen(char *ruta, int filas, int columnas, int x, int y, bool alpha) {
```

16.13. Implementación

```
13 ;
14 ;    // Inicializamos los atributos de la clase
15 ;
16 ;    this->filas = filas;
17 ;    this->columnas = columnas;
18 ;    x0 = x;
19 ;    y0 = y;
20 ;
21 ;
22 ;#ifdef DEBUG
23 ;    cout << "-> Cargando" << ruta << endl;
24 ;#endif
25 ;
26 ;    // Cargamos la imagen
27 ;
28 ;    imagen = IMG_Load(ruta);
29 ;
30 ;    if(imagen == NULL) {
31 ;
32 ;        cerr << "Error: " << SDL_GetError() << endl;;
33 ;        exit(1);
34 ;    }
35 ;
36 ;    if(!alpha) {
37 ;
38 ;        SDL_Surface *tmp = imagen;
39 ;
40 ;        imagen = SDL_DisplayFormat(tmp);
41 ;        SDL_FreeSurface(tmp);
42 ;
43 ;        if(imagen == NULL) {
44 ;            printf("Error: %s\n", SDL_GetError());
45 ;            exit(1);
46 ;        }
47 ;
48 ;        // Calculamos el color transparente, en nuestro caso el verde
49 ;
50 ;        Uint32 colorkey = SDL_MapRGB(imagen->format, 0, 255, 0);
51 ;
52 ;        // Lo establecemos como color transparente
53 ;
54 ;        SDL_SetColorKey(imagen, SDL_SRCCOLORKEY, colorkey);
55 ;    }
56 ;
57 ;    // Hallamos la imagen invertida utilizada en el mayor de los casos
58 ;
59 ;    imagen_invertida = invertir_imagen(imagen);
60 ;
61 ;    if(imagen_invertida == NULL) {
62 ;
63 ;        cerr << "No se pudo invertir la imagen: " << SDL_GetError() << endl;
64 ;        exit(1);
65 ;    }
```

16. Un ejemplo del desarrollo software de un videojuego

```
66 ;    }
67 ;
68 ;    // El ancho de una imagen de la rejilla
69 ;    // es el total entre el número de columnas
70 ;
71 ;    w = imagen->w / columnas;
72 ;
73 ;    // El alto de una imagen de la rejilla
74 ;    // es el total entre el número de filas
75 ;
76 ;    h = imagen->h / filas;
77 ;
78 ;}
79 ;
80 ;
81 ;
82 ;void Imagen::dibujar(SDL_Surface *superficie, int i, int x, int y, int flip)
83 ;{
84 ;
85 ;    SDL_Rect destino;
86 ;
87 ;    destino.x = x - x0;
88 ;    destino.y = y - y0;
89 ;
90 ;    // No se usan
91 ;
92 ;    destino.h = 0;
93 ;    destino.w = 0;
94 ;
95 ;    // Comprobamos que el número de imagen indicado sea el correcto
96 ;
97 ;    if(i < 0 || i > (filas * columnas)) {
98 ;
99 ;        cerr << "Imagen::Dibujar = No existe el cuadro" << i << endl;
100 ;       return;
101 ;
102 ;    }
103 ;
104 ;    SDL_Rect origen;
105 ;
106 ;    // Separaciones de 2 píxeles dentro de las rejillas para observar
107 ;    // bien donde empieza una imagen y donde termina la otra
108 ;
109 ;    origen.w = w - 2;
110 ;    origen.h = h - 2;
111 ;
112 ;    // Seleccionamos cual de las imágenes es la que vamos a dibujar
113 ;
114 ;    switch(flip) {
115 ;
116 ;        case 1:
117 ;
118 ;            origen.x = ((i % columnas) * w) + 2;
```

16.13. Implementación

```
119 ;         origen.y = ((i / columnas) * h) + 2;
120 ;
121 ;         // Realizamos el blit
122 ;
123 ;         SDL_BlitSurface(imagen, &origen, superficie, &destino);
124 ;         break;
125 ;
126 ;     case -1:
127 ;
128 ;         origen.x = ((columnas-1) - (i % columnas)) * w + 1;
129 ;         origen.y = (i / columnas) * h + 2;
130 ;
131 ;         // Copiamos la imagen en la superficie
132 ;
133 ;         SDL_BlitSurface(imagen_invertida, &origen, superficie, &destino);
134 ;         break;
135 ;
136 ;     default:
137 ;         cerr << "Caso no válido: Imagen invertida o no" << endl;
138 ;         break;
139 ;
140 ;     }
141 ;}
142 ;
143 ;
144 ;// Devuelve la posición con respecto a la horizontal de la imagen
145 ;
146 ;int Imagen::pos_x() {
147 ;
148 ;    return x0;
149 ;
150 ;}
151 ;
152 ;
153 ;// Devuelve la posición con respecto a la vertical de la imagen
154 ;
155 ;int Imagen::pos_y() {
156 ;
157 ;    return y0;
158 ;
159 ;}
160 ;
161 ;// Devuelve la anchura de un cuadro de la rejilla
162 ;
163 ;int Imagen::anchura() {
164 ;
165 ;    return w;
166 ;
167 ;}
168 ;
169 ;
170 ;// Devuelve la altura de un cuadro de la rejilla
171 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
172 ;int Imagen::altura() {
173 ;
174 ;    return h;
175 ;
176 ;}
177 ;
178 ;// Devuelve el número de cuadros de la rejilla de la imagen
179 ;
180 ;int Imagen::cuadros() {
181 ;
182 ;    return columnas * filas;
183 ;
184 ;}
185 ;
186 ;Imagen::~Imagen()
187 ;{
188 ;    SDL_FreeSurface(imagen);
189 ;    SDL_FreeSurface(imagen_invertida);
190 ;
191 ;#ifdef DEBUG
192 ;    cout << "<- Liberando imagen" << endl;
193 ;#endif
194 ;}
195 ;
196 ;
197 ;SDL_Surface * Imagen::invertir_imagen(SDL_Surface *imagen) {
198 ;
199 ;    SDL_Rect origen;
200 ;    SDL_Rect destino;
201 ;
202 ;    // Origen y destino preparados para copiar linea a linea
203 ;
204 ;    origen.x = 0;
205 ;    origen.y = 0;
206 ;    origen.w = 1;
207 ;    origen.h = imagen->h;
208 ;
209 ;    destino.x = imagen->w;
210 ;    destino.y = 0;
211 ;    destino.w = 1;
212 ;    destino.h = imagen->h;
213 ;
214 ;    SDL_Surface *invertida;
215 ;
216 ;    // Pasamos imagen a formato de pantalla
217 ;
218 ;    invertida = SDL_DisplayFormat(imagen);
219 ;    if(invertida == NULL) {
220 ;
221 ;        cerr << "No podemos convertir la imagen al formato de pantalla" << endl;
222 ;        return NULL;
223 ;
224 ;    }
```

```

225 ;
226 ;    // Preparamos el rectángulo nuevo vacío del color transparente
227 ;
228 ;    SDL_FillRect(invertida, NULL, SDL_MapRGB(invertida->format, 0, 255, 0));
229 ;
230 ;    // Copiamos linea vertical a linea vertical, inversamente
231 ;
232 ;    for(int i = 0; i < imagen->w; i++) {
233 ;
234 ;        SDL_BlitSurface(imagen, &origen, invertida, &destino);
235 ;        origen.x = origen.x + 1;
236 ;        destino.x = destino.x - 1;
237 ;
238 ;    }
239 ;
240 ;    return invertida;
241 ;
242 ;}
;
```

El constructor de la clase inicializa los atributos de dicha clase. Utiliza la librería `SDL_image` para cargar la imagen que almacenará dicha clase ofreciendo así una mayor potencia que si sólo pudiese cargar el formato nativo de `SDL`.

Una vez cargada la imagen establecemos el color de la transparencia y la invertimos ya que en nuestra aplicación vamos a necesitar de dichas imágenes invertidas. Un vez realizado todo el proceso calculamos cuanto ocupa un cuadro de la rejilla de imágenes y lo almacenamos en los atributos de la clase.

El método `dibujar()` nos permite mostrar un recuadro de la rejilla en una posición (x, y) de una superficie pasada como parámetro. El parámetro `flip` estará establecido a 1 si queremos la imagen original o a -1 si queremos dibujar la invertida. El método comprueba que los parámetros sean correctos y, seguidamente, realiza el blit sobre la superficie destino. Para distinguir si utilizar la imagen original o la invertida utiliza una estructura selectiva. Esta estructura es muy importante ya que la posición de origen de la superficie a copiar varía si la imagen a copiar es la normal o la invertida.

Los demás métodos observadores se limitan a devolver atributos de la clase asociados a la definición del método y no merecen mayor explicación.

16.13.10. La clase Interfaz

La clase `Interfaz` es una clase virtual que nos permite definir un interfaz abstracto entre los que movernos según necesite la clase `Universo`. Esta clase unifica las características de los distintos aspectos del juegos que son el Menú, el Juego y el Editor de niveles.

16. Un ejemplo del desarrollo software de un videojuego

El diseño de la clase del que partimos para la implementación es el de la figura 16.43.



Figura 16.43: Clase Interfaz

Vamos a estudiar la definición de la clase:

```
1 ;// Listado: Interfaz.h
2 ;//
3 ;// Superclase de las diferentes escenas disponibles en la aplicación
4 ;
5 ;#ifndef _INTERFAZ_H_
6 ;#define _INTERFAZ_H_
7 ;
8 ;
9 ;// Declaración adelantada
10 ;
11 ;class Universo;
12 ;
13 ;
14 ;
15 ;class Interfaz
16 ;{
17 ; public:
18 ;
19 ;     // Tres son las escenas que encontramos en la aplicación
20 ;
21 ;     enum escenas {
22 ;
23 ;         ESCENA_MENU,
24 ;         ESCENA_JUEGO,
25 ;         ESCENA_EDITOR
26 ;
27 ;     };
28 ;
29 ;     // Constructor
30 ;
31 ;     Interfaz(Universo *universo);
32 ;
33 ;     // Funciones virtuales puras comunes a todas las escenas
34 ;
```

16.13. Implementación

```
35 ;     virtual void reiniciar(void) = 0;
36 ;     virtual void dibujar(void) = 0;
37 ;     virtual void actualizar(void) = 0;
38 ;
39 ;     Universo *universo;
40 ;
41 ;     // Destructor
42 ;
43 ;     virtual ~Interfaz();
44;};
45 ;
46 ;#endif
;
```

Como puedes ver la definición de la clase es bastante breve. Define un tipo enumerado para poder identificar el tipo de escena en el que nos encontramos. Además del constructor y el destructor tenemos tres métodos virtuales que serán implementados en las clases hijas y que dotan de un comportamiento común a las escenas de la aplicación.

Además tenemos un atributo *universo* que nos permite asociar esta clase a la clase Universo que integra la aplicación. Veamos la implementación de la clase:

```
;_____
1 ;// Listado: Interfaz.cpp
2 ;//
3 ;// Implementación de la clase interfaz
4 ;
5 ;#include <iostream>
6 ;
7 ;#include "Interfaz.h"
8 ;#include "Universo.h"
9 ;
10 ;using namespace std;
11 ;
12 ;
13 ;Interfaz::Interfaz(Universo *universo) {
14 ;
15 ;#ifdef DEBUG
16 ;    cout << "Interfaz::Interfaz()" << endl;
17 ;#endif
18 ;
19 ;    this->universo = universo;
20 ;}
21 ;
22 ;
23 ;Interfaz::~Interfaz()
24 ;{
25 ;    // No realiza ninguna acción particular
26 ;}
```

Como puedes observar sólo está implementado el constructor de la clase

16. Un ejemplo del desarrollo software de un videojuego

que inicializa el único atributo de la clase.

16.13.11. La clase Item

La clase *Item* es una subclase de la clase *Participante* que nos permite definir elementos u objetos del juego con los que nuestro personaje principal puede interactuar. Estos objetos son elementos muy importantes del juego ya que nos permiten ganar puntos o recuperar vidas dotando a la aplicación de toda su esencia.

El diseño de la clase del que partimos para la implementación es el de la figura 16.44.

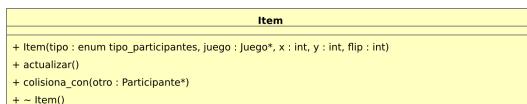


Figura 16.44: Clase Item

Vamos a estudiar la definición de la clase:

```
1 ;// Listado: Item.h
2 ;//
3 ;// Esta clase controla todo lo relativo a los items u objetos del juego
4 ;
5 ;#ifndef _ITEM_H_
6 ;#define _ITEM_H_
7 ;
8 ;#include "Participante.h"
9 ;
10 ;class Item: public Participante {
11 ;
12 ;    public:
13 ;
14 ;        //Constructor
15 ;
16 ;        Item(enum tipo_participantes tipo, Juego *juego, int x, int y, int flip = 1);
17 ;
18 ;        void actualizar(void);
19 ;        void colisiona_con(Participante *otro);
20 ;
21 ;        // Destructor
22 ;
23 ;        virtual ~Item();
24 ;
25 ;};
```

16.13. Implementación

```
27 ;#endif  
;
```

La definición de esta clase responde a la definición de su clase madre. Tenemos el constructor que nos permite crear un ítem de distinto tipo. El método *actualizar()* que nos permite avanzar en el estado del objeto en un momento dado. Sólo hay otro método en esta clase. Se trata del método que informa de una colisión con otro participante para poder realizar las acciones oportunas que veremos en la implementación de la clase.

La implementación de esta clase es:

```
;  
1 ;// Listado: Item.cpp  
2 ;//  
3 ;// Implementación de la clase Item  
4 ;  
5 ;#include <iostream>  
6 ;  
7 ;#include "Item.h"  
8 ;#include "Juego.h"  
9 ;#include "Universo.h"  
10 ;#include "Galeria.h"  
11 ;#include "Sonido.h"  
12 ;#include "Imagen.h"  
13 ;#include "Control_Animacion.h"  
14 ;#include "Nivel.h"  
15 ;  
16 ;  
17 ;using namespace std;  
18 ;  
19 ;  
20 ;Item::Item(enum tipo_participantes tipo, Juego *juego, int x, int y, int flip):  
21 ;    Participante(juego, x, y, flip) {  
22 ;  
23 ;#ifdef DEBUG  
24 ;    cout << "Item::Item()" << endl;  
25 ;#endif  
26 ;  
27 ;    // Animaciones de los estados de los objetos  
28 ;  
29 ;    animaciones[PARADO] = new Control_Animacion("0,0,0,1,2,2,1", 10);  
30 ;    animaciones[MORIR] = new Control_Animacion("1,2,3,3", 7);  
31 ;  
32 ;    // Imagen según el tipo de item creado  
33 ;  
34 ;    switch(tipo) {  
35 ;  
36 ;        case TIPO_ALICATE:  
37 ;  
38 ;            imagen = juego->universo->galeria->imagen(Galeria::ITEM_ALICATE);  
39 ;            break;  
40 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
41 ;     case TIPO_DESTORNILLADOR:
42 ;
43 ;         imagen = juego->universo->galeria->imagen(Galeria::ITEM_DESTORNILLADOR);
44 ;         break;
45 ;
46 ;     default:
47 ;         cerr << "Item::Item(): Caso no contemplado" << endl;
48 ;         break;
49 ;
50 ;
51 ;     estado = PARADO;
52 ;
53 ;
54 ;
55 ;void Item::actualizar(void)
56 ;{
57 ;
58 ;    // Si el item "muere" lo marcamos para eliminar
59 ;    // si no avanzamos la animación
60 ;
61 ;    if(estado == MORIR) {
62 ;
63 ;        if(animaciones[estado]->avanzar())
64 ;            estado = ELIMINAR;
65 ;
66 ;    }
67 ;    else
68 ;        animaciones[estado]->avanzar();
69 ;
70 ;
71 ;    // Por si está colocado en las alturas
72 ;    // y tiene que caer en alguna superficie
73 ;    // necesita una velocidad de caida
74 ;
75 ;    velocidad_salto += 0.1;
76 ;    y += altura((int) velocidad_salto);
77 ;
78 ;
79 ;
80 ;
81 ;void Item::colisiona_con(Participante *otro) {
82 ;
83 ;    // Si colisiona, muere para desaparecer
84 ;
85 ;    if(estado != MORIR) {
86 ;
87 ;        juego->universo->\ 
88 ;                    galeria->sonidos[Galeria::COGE_ITEM]->reproducir();
89 ;
90 ;        estado = MORIR;
91 ;    }
92 ;
93 ;}
```

```

94 ;
95 ;
96 ;Item::~Item() {
97 ;
98 ;#ifdef DEBUG
99 ;    cout << "Item::~Item()" << endl;
100 ;#endif
101 ;
102 ;}
;
```

El constructor de esta clase lo primero que hace es inicializar las animaciones de las que va a hacer uso dicho objeto. Dependiendo del tipo de ítem cargamos una imagen u otra de la galería para mostrar las animaciones. El estado inicial de los objetos es parado.

En el método *actualizar* se comprueba si el ítem en cuestión ha “muerto” por lo que debe ser eliminado. En cualquier caso se avanza la animación. En el caso de que el objeto no esté sobre un suelo sólido éste caerá hasta encontrarlo.

El método *colisiona_con()* sirve para informar a un ítem que se ha producido una colisión. En este caso si se produce una colisión marcaremos al objeto con el estado morir así será eliminado en las siguientes iteraciones del game loop.

16.13.12. La clase Juego

La clase *Juego* es una clase hija de Interfaz. Esta clase define el comportamiento de la escena Juego que es en la que jugamos propiamente a los niveles que hemos diseñado.

El diseño de la clase del que partimos para la implementación es el de la figura 16.45.

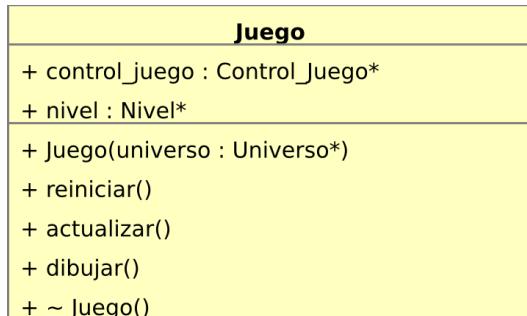


Figura 16.45: Clase CJuego

Vamos a estudiar la definición de la clase:

16. Un ejemplo del desarrollo software de un videojuego

```
1 ;_____
2 // Listado: Juego.h
3 ;///
4 ;// Esta clase hija de interfaz nos permite relacionar el control del juego
5 ;// con los niveles para cohesionar los módulos
6 ;
7 ;#ifndef _JUEGO_H_
8 ;#define _JUEGO_H_
9 ;
10 ;#include <SDL/SDL.h>
11 ;#include "Interfaz.h"
12 ;
13 ;// Declaración adelantada
14 ;
15 ;class Universo;
16 ;class Control_Juego;
17 ;class Nivel;
18 ;
19 ;class Juego : public Interfaz {
20 ;
21 ;    public:
22 ;        // Constructor
23 ;
24 ;        Juego(Universo *universo);
25 ;
26 ;        // Funciones heradadas
27 ;
28 ;        void reiniciar(void);
29 ;        void actualizar(void);
30 ;        void dibujar(void);
31 ;
32 ;        Control_Juego *control_juego;
33 ;        Nivel *nivel;
34 ;
35 ;        // Destructor
36 ;        ~Juego();
37 ;};
38 ;
39 ;#endif
;
```

La clase implementa los métodos que hereda de Interfaz y añade dos variables que nos van a permitir asociar dicha clase con las clases Control_Juego y Nivel que son necesarias para el transcurso de una partida. Control_Juego, como su nombre indica, nos provee de control sobre el transcurso del juego y necesitamos la clase que gestiona los niveles para poder interactuar con ellos.

Veamos la implementación de la clase:

```
1 ;_____
2 // Listado: Juego.cpp
3 ;///
4 ;// Implementación de la clase juego
```

16.13. Implementación

```
4 ;
5 ;#include <iostream>
6 ;
7 ;#include "Participante.h"
8 ;#include "Juego.h"
9 ;#include "Universo.h"
10 ;#include "Nivel.h"
11 ;#include "Control_Juego.h"
12 ;#include "Galeria.h"
13 ;#include "Imagen.h"
14 ;#include "Musica.h"
15 ;
16 ;
17 ;using namespace std;
18 ;
19 ;
20 ;Juego::Juego(Universo *universo): Interfaz(universo) {
21 ;
22 ;#ifdef DEBUG
23 ;    cout << "Juego::Juego()" << endl;
24 ;#endif
25 ;
26 ;    nivel = NULL;
27 ;    control_juego = NULL;
28 ;
29 ;    // Iniciamos el juego
30 ;
31 ;    reiniciar();
32 ;}
33 ;
34 ;
35 ;void Juego::reiniciar(void) {
36 ;
37 ;    // Hacemos sonar la música
38 ;
39 ;    universo->galeria->musica(Galeria::MUSICA_JUEGO)->pausar();
40 ;    universo->galeria->musica(Galeria::MUSICA_JUEGO)->reproducir();
41 ;
42 ;    // Si se ha iniciado el juego
43 ;    // Eliminamos los datos anteriores (restauramos)
44 ;
45 ;    if(control_juego != NULL)
46 ;        delete control_juego;
47 ;
48 ;    if(nivel != NULL)
49 ;        delete nivel;
50 ;
51 ;    // Generamos de nuevo el entorno del juego
52 ;
53 ;    control_juego = new Control_Juego(this);
54 ;    nivel = new Nivel(universo);
55 ;
56 ;    nivel->generar_actores(control_juego);
```

16. Un ejemplo del desarrollo software de un videojuego

```
57 ;
58 ;#ifndef DEBUG
59 ;    cout << "Juego::reiniciado" << endl;
60 ;#endif
61 ;
62 ;
63 ;
64 ;void Juego::actualizar(void) {
65 ;
66 ;    // Si pulsamos la tecla de salir, salimos al menu
67 ;
68 ;    if(universo->teclado.pulso(Teclado::TECLA_SALIR))
69 ;        universo->cambiar_interfaz(ESCENA_MENU);
70 ;
71 ;    // Actualizamos la posición de la ventana
72 ;
73 ;    nivel->actualizar();
74 ;
75 ;    // Actualizamos el estado de los items,
76 ;    // enemigos y el personaje principal
77 ;
78 ;    control_juego->actualizar();
79 ;}
80 ;
81 ;
82 ;
83 ;void Juego::dibujar(void) {
84 ;
85 ;    // Dibujamos en la superficie principal
86 ;
87 ;    // El nivel
88 ;
89 ;    nivel->dibujar(universo->pantalla);
90 ;
91 ;    // Los personajes, items y enemigos
92 ;
93 ;    control_juego->dibujar(universo->pantalla);
94 ;
95 ;    // Actualizamos la pantalla
96 ;
97 ;    SDL_Flip(universo->pantalla);
98 ;}
99 ;
100 ;
101 ;Juego::~Juego() {
102 ;
103 ;    // Liberamos la memoria
104 ;    delete control_juego;
105 ;    delete nivel;
106 ;
107 ;#ifdef DEBUG
108 ;    cout << "Juego::~Juego()" << endl;
109 ;#endif
```

```
110 ;}
;
```

El constructor de la clase utiliza el constructor de Interfaz así como inicializa los atributos definidos en la clase hija. Una vez construido un elemento de este tipo realiza una llamada al método reiniciar para comenzar el transcurso del juego.

El método *reiniciar()* comienza a reproducir la música del juego, si existían los elementos nivel o control_juego los elimina así como crea nuevas instancias para estos dos atributos. Una vez realizadas todas estas tareas el método genera los actores y les proporciona el correspondiente control.

El método *actualizar()* comprueba si se ha pulsado la tecla de salida con la que se volvería al menú principal del juego y llama a los métodos actualizar de nivel y de control_juego para que renueven sus estados.

El método *dibujar()* se encarga de que nivel y control_juego dibujen mediante sus métodos *dibujar()* las nuevas posiciones y elementos del nivel. Una vez que hayan realizado el blitting nuestro método dibujar se encarga de hacer el flip de los búfferes con los que mostrar toda esta nueva información.

El destructor de la clase se encarga de liberar la memoria ocupada por los atributos propios de esta clase. Como puedes ver no tiene mayor complicación.

16.13.13. La clase Menu

Como la clase Juego, esta clase es la encargada de implementar el interfaz de menú en nuestra aplicación. El menú tiene una serie de opciones y animaciones que lo hacen ser un caso concreto de la clase Interfaz de la que es hija. Veamos los aspectos de esta clase.

El diseño de la clase del que partimos para la implementación es el de la figura 16.46.

Vamos a estudiar la definición de la clase:

```
1 ;// Listado: Menu.h
2 ;//
3 ;// Esta clase controla los aspectos relevantes al Menú de la aplicación
4 ;
5 ;#ifndef _MENU_H_
6 ;#define _MENU_H_
7 ;
8 ;#include <SDL/SDL.h>
9 ;#include "Interfaz.h"
10 ;#include "CommonConstants.h"
```

16. Un ejemplo del desarrollo software de un videojuego

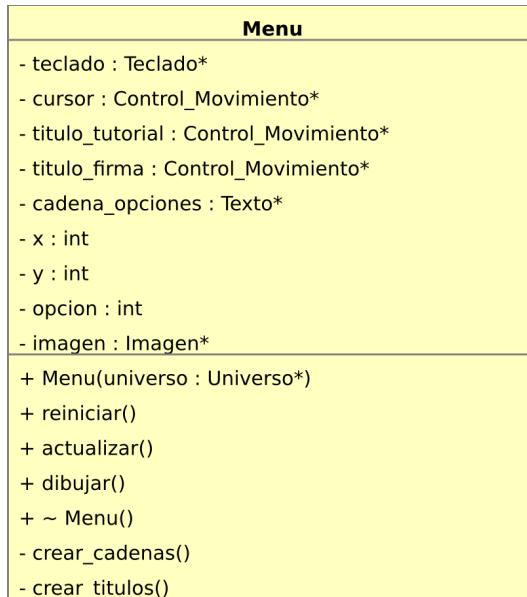


Figura 16.46: Clase Menu

```
11 ;
12 ;// Declaraciones adelantadas
13 ;
14 ;class Teclado;
15 ;class Control_Movimiento;
16 ;class Texto;
17 ;class Imagen;
18 ;
19 ;
20 ;class Menu: public Interfaz {
21 ;
22 ; public:
23 ;
24 ;     // Constructor
25 ;
26 ;     Menu(Universo *universo);
27 ;
28 ;     // Funciones heradas de Interfaz
29 ;     // Operativa de la clase
30 ;
31 ;     void reiniciar (void);
32 ;     void actualizar (void);
33 ;     void dibujar (void);
34 ;
35 ;     // Destructor
36 ;
37 ;     ~Menu();
38 ;
39 ; private:
40 ;
41 ;     // Controla el dispositivo de entrada
```

16.13. Implementación

```
42 ;
43 ;     Teclado *teclado;
44 ;
45 ;     // Controla los movimientos y posición de los elementos del menú
46 ;
47 ;     Control_Movimiento *cursor;
48 ;     Control_Movimiento *titulo_tutorial;
49 ;     Control_Movimiento *titulo_firma;
50 ;
51 ;     // Aquí almacenamos las frases que componen el menú
52 ;
53 ;     Texto *cadena_opciones[NUM_OPCIONES];
54 ;
55 ;     int x, y;
56 ;     int opcion;
57 ;     Imagen *imagen;
58 ;
59 ;     // Crea las cadenas que se almacenarán en cadena_opciones
60 ;     void crear_cadenas(void);
61 ;
62 ;     // Crea los títulos del juego
63 ;     void crear_titulos(void);
64 };
65 ;
66 ;#endif
```

En la parte privada de esta clase tenemos varios atributos que nos permiten dotar de comportamiento al menú. El atributo *teclado* nos permite asociar el dispositivo de entrada con el menú. Los atributos *cursor*, *titulo_tutorial* y *titulo_firma* al ser del tipo *Control_Movimiento* nos permiten dotar movimiento a estos elementos dentro del menú. Tenemos un atributo *cadena_opciones* donde almacenamos las cadenas de texto que corresponden con las diferentes alternativas dentro del menú.

Para controlar la posiciones del cursor está los atributos (*x*, *y*) mientras que el atributo *opcion* almacena la elección que hayamos realizado. Mediante el atributo *imagen* asociaremos la imagen de los elementos del menú. Los métodos privados *crear_cadenas()* y *crear_titulos()* los estudiaremos a continuación con la implementación de la clase.

En la parte pública de la clase encontramos los métodos comunes a todas las interfaces: *reniciar()*, *actualizar()* y *dibujar()*. Vamos a presentar la implementación de esta clase para analizar cada uno de estos métodos:

```
1 ;// Listado: Menu.cpp
2 ;//
3 ;// Implementación de la clase Menu
4 ;
5 ;#include <iostream>
6 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
7 ;#include "Menu.h"
8 ;#include "Universo.h"
9 ;#include "Galeria.h"
10 ;#include "Imagen.h"
11 ;#include "Sonido.h"
12 ;#include "Teclado.h"
13 ;#include "Control_Movimiento.h"
14 ;#include "Fuente.h"
15 ;#include "Texto.h"
16 ;#include "Musica.h"
17 ;
18 ;
19 ;using namespace std;
20 ;
21 ;
22 ;Menu::Menu(Universo *universo) : Interfaz(universo) {
23 ;
24 ;#ifdef DEBUG
25 ;    cout << "Menu::Menu()" << endl;
26 ;#endif
27 ;
28 ;    // El dispositivo de entrada nos lo da el entorno del juego
29 ;
30 ;    teclado = &(universo->teclado);
31 ;
32 ;
33 ;    // Cargamos el título del juego
34 ;
35 ;    imagen = universo->galeria->imagen(Galeria::MENU);
36 ;
37 ;    x = y = opcion = 0;
38 ;
39 ;
40 ;    // Cargamos el cursor de selección
41 ;
42 ;    cursor = new Control_Movimiento(universo->galeria->imagen(Galeria::TILES), 50, 100, 300);
43 ;
44 ;
45 ;    // Creamos los títulos
46 ;
47 ;    crear_titulos();
48 ;
49 ;    // Creo las cadenas de las opciones
50 ;
51 ;    crear_cadenas();
52 ;
53 ;    // Inicio el Menu
54 ;
55 ;    reiniciar();
56 ;}
57 ;
58 ;
59 ;void Menu::reiniciar(void) {
```

16.13. Implementación

```
60 ;
61 ;    // Hacemos sonar la música
62 ;
63 ;    universo->galeria->musica(Galeria::MUSICA_MENU)->pausar();
64 ;    universo->galeria->musica(Galeria::MUSICA_MENU)->reproducir();
65 ;
66 ;
67 ;    // Colocamos cada parte del título en su lugar
68 ;    // Desde una posición de origen hasta la posición final
69 ;
70 ;    titulo_tutorial->mover_inmediatamente(100, -250);
71 ;    titulo_tutorial->mover(100,110);
72 ;
73 ;    titulo_firma->mover_inmediatamente(640, 200);
74 ;    titulo_firma->mover(400, 200);
75 ;
76 ;
77 ;
78 ;void Menu::actualizar(void) {
79 ;
80 ;    static int delay = 0; // Variable con "memoria"
81 ;
82 ;    // Actualizamos el cursor seleccionador
83 ;
84 ;    cursor->actualizar();
85 ;
86 ;    // Actualizamos los títulos
87 ;
88 ;    titulo_tutorial->actualizar();
89 ;    titulo_firma->actualizar();
90 ;
91 ;    // Si pulsamos abajo y no estamos en la última
92 ;    // (controlamos no ir excesivamente rápido)
93 ;
94 ;    if(teclado->pulso(Teclado::TECLA_BAJAR) && opcion < 2 && delay == 0) {
95 ;
96 ;        delay = 30; // Retardo de 30 ms
97 ;        opcion++; // Bajamos -> opcion = opcion + 1
98 ;        cursor->mover(100, 300 + 50 * opcion); // Movemos el cursor
99 ;
100 ;        // Reproducimos un efecto de sonido
101 ;
102 ;        universo->galeria->sonidos[Galeria::EFFECTO_MENU]->reproducir();
103 ;
104 ;    }
105 ;
106 ;    // Si pulsamos arriba y no estamos en la primera opción
107 ;
108 ;    if(teclado->pulso(Teclado::TECLA_SUBIR) && opcion > 0 && delay == 0) {
109 ;
110 ;        delay = 30; // Retardo de 30 ms
111 ;        opcion--; // Subimos -> opcion = opcion - 1
112 ;        cursor->mover(100, 300 + 50 * opcion); // Movemos el cursor
```

16. Un ejemplo del desarrollo software de un videojuego

```
113 ;
114 ;      // Reproducimos un efecto de sonido
115 ;
116 ;      universo->galeria->sonidos[Galeria::EFECTO_MENU]->reproducir();
117 ;
118 ; }
119 ;
120 ;
121 ; if(delay) // Reducimos el retardo
122 ;     delay--;
123 ;
124 ;
125 ; // Si aceptamos
126 ;
127 ; if(teclado->pulso(Teclado::TECLA_ACEPTAR)) {
128 ;
129 ;     // Entramos en una opción determinada
130 ;
131 ;     switch(opcion) {
132 ;
133 ;         case 0: // Jugar
134 ;
135 ;             universo->cambiar_interfaz(ESCENA_JUEGO);
136 ;             break;
137 ;
138 ;         case 1: // Editar niveles
139 ;
140 ;             universo->cambiar_interfaz(ESCENA_EDITOR);
141 ;             break;
142 ;
143 ;         case 2: // Salir de la aplicación
144 ;
145 ;             universo->terminar();
146 ;             break;
147 ;     }
148 ; }
149 ;}
150 ;
151 ;
152 ;
153 ;void Menu::dibujar(void) {
154 ;
155 ;     // Dibujamos un rectángulo de fondo con el color anaranjado
156 ;
157 ;     SDL_FillRect(universo->pantalla, NULL,\n
158 ;                  SDL_MapRGB(universo->pantalla->format, 161, 151, 240));
159 ;
160 ;     // Dibujamos el cursor que selecciona una opción u otra
161 ;
162 ;     cursor->dibujar(universo->pantalla);
163 ;
164 ;     // Dibujamos los títulos
165 ;
```

16.13. Implementación

```
166 ;     titulo_tutorial->dibujar(universo->pantalla);
167 ;     titulo_firma->dibujar(universo->pantalla);
168 ;
169 ;     // Dibujamos las 3 cadenas de opciones
170 ;
171 ;     for(int i = 0; i < NUM_OPCIONES; i++)
172 ;         cadena_opciones[i]->dibujar(universo->pantalla);
173 ;
174 ;     // Actualizamos la pantalla del entorno
175 ;
176 ;     SDL_Flip(universo->pantalla);
177 ;
178 ;
179 ;
180 ;Menu::~Menu() {
181 ;
182 ;#ifdef DEBUG
183 ;    cout << "Menu::~Menu()" << endl;
184 ;#endif
185 ;
186 ;    // Liberamos memoria
187 ;
188 ;    delete cursor;
189 ;    delete titulo_tutorial;
190 ;    delete titulo_firma;
191 ;
192 ;
193 ;    // También de las cadenas
194 ;
195 ;    for(int i = 0; i < NUM_OPCIONES; i++)
196 ;        delete cadena_opciones[i];
197 ;
198 ;
199 ;
200 ;
201 ;void Menu::crear_cadenas(void) {
202 ;
203 ;    // Crea las cadenas que mostraremos como opciones
204 ;
205 ;    char textos[][20] = { {"Jugar"}, {"Editar niveles"}, {"Salir"} };
206 ;
207 ;
208 ;    // Cargamos la fuente
209 ;
210 ;    Fuente *fuente = universo->galeria->fuente(Galeria::FUENTE_MENU);
211 ;
212 ;
213 ;    // La almacenamos en el vector de tipo Texto
214 ;
215 ;    for(int i = 0; i < NUM_OPCIONES; i++) {
216 ;
217 ;        cadena_opciones[i] = new Texto(fuente, 150, 300 + i * 50, textos[i]);
218 ;    }
```

16. Un ejemplo del desarrollo software de un videojuego

```
219 ;}
220 ;
221 ;
222 ;
223 ;void Menu::crear_titulos(void) {
224 ;
225 ;    // Creamos los títulos animados
226 ;
227 ;    titulo_tutorial =
228 ;        new Control_Movimiento(universo->galeria->imagen(Galeria::TITULO_TUTORIAL),\
229 ;                                0, 300, 200);
230 ;
231 ;    titulo_firma =
232 ;        new Control_Movimiento(universo->galeria->imagen(Galeria::TITULO_FIRMA),\
233 ;                                0, 300, 400);
234 ;}
```

El constructor de la clase inicializa los atributos de la misma creando un control de movimiento para el cursor, creando los títulos, las cadenas que los componen así como reiniciando el estado para que todos los elementos que componen el menú estén en un estado inicial.

El método *reiniciar()* se encarga de mostrar las animaciones iniciales del menú así como de reestablecer la música correspondiente a esta escena del juego. Los títulos del juego se mueven hacia la posición final desde una posición exterior para producir un efecto de movimiento

En el método *actualizar()* renovamos el estado del cursor y de los dos títulos que componen la escena. Comprobamos si se ha pulsado alguna de las teclas que producen movimiento entre las opciones del menú. Si es así marcamos un retardo y mostramos un efecto de movimiento así como reproducimos un sonido asociado a dicho evento. Si se pulsa la tecla definida como ACEPTAR cambiamos de escena a la que tuviésemos seleccionada como opción.

El método *dibujar()* rellenamos el fondo de un color liso para seguidamente “pintar” encima de él todos los elementos del menú. Empezamos por el cursor, seguidamente mostrámos los títulos y para terminar hacemos blit sobre la superficie principal con las cadenas que nos muestran las opciones. Para mostrar todos estos cambios hacemos una llamada a la función *SDL_Flip()* alternando los búfferes.

El destructor libera la memoria utilizada por los atributos de esta clase. Ahora vamos a describir los métodos privados de la clase. El primero de ellos es *crear_cadenas()*. Con este método creamos las cadenas que luego serán las opciones del menú con la fuente inicializada en el mismo método. Los textos de estas cadenas son introducidos en un vector que será recorrido a la hora de

mostrar dichas opciones.

El método privado *crear_titulos()* crea los elementos decorativos del menú, que como bien indica el nombre del método, es el título de la aplicación que estamos desarrollando. Además al ser del tipo *Control_Movimiento* los dotamos del movimiento que ya vimos en el método *actualizar()*.

16.13.14. La clase Musica

La clase *Musica* nos permite dotar de este elemento a nuestro videojuego. Hemos decidido no complicar la definición de esta clase porque no es necesario para que nos ofrezca toda la potencia que necesitamos.

El diseño de la clase del que partimos para la implementación es el de la figura 16.47.

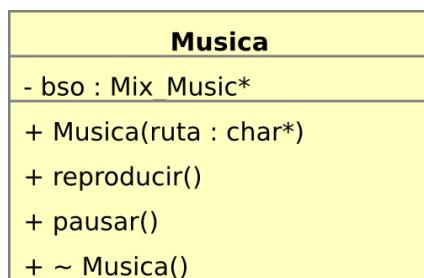


Figura 16.47: Clase Musica

Vamos a estudiar la definición de la clase:

```

1 ;// Listado: Musica.h
2 ;//
3 ;// Clase para facilitar el trabajo con la musica
4 ;
5 ;#ifndef _MUSICA_H_
6 ;#define _MUSICA_H_
7 ;
8 ;#include <SDL/SDL.h>
9 ;#include <SDL/SDL_mixer.h>
10 ;
11 ;class Musica {
12 ;
13 ; public:
14 ;
15 ;     // Constructor
16 ;
17 ;     Musica(char *ruta);
18 ;
  
```

16. Un ejemplo del desarrollo software de un videojuego

```
19 ;     void reproducir();
20 ;     void pausar();
21 ;
22 ;     ~Musica();
23 ;
24 ; private:
25 ;
26 ;     Mix_Music *bso;
27 ;
28;};
29 ;
30 ;#endif
```

Para implementar el control de la música vamos a utilizar la librería auxiliar *SDL_mixer*. De ahí que en la parte privada de la clase tengamos un atributo *bso* de un tipo de esta librería que es el que asociará la música con la clase.

En cuanto a la parte pública podemos ver que, además del constructor y el destructor, tenemos dos métodos que nos permitirán reproducir y detener la música. Vamos a ver la implementación de la clase.

```
1 ;
2 ;// Listado: Musica.cpp
3 ;//
4 ;// Implementación de la clase Música
5 ;
6 ;#include <iostream>
7 ;
8 ;#include "Musica.h"
9 ;#include "CommonConstants.h"
10 ;
11 ;using namespace std;
12 ;
13 ;Musica::Musica(char *ruta) {
14 ;
15 ;    // Cargamos la música
16 ;
17 ;    bso = Mix_LoadMUS(ruta);
18 ;
19 ;    if(bso == NULL) {
20 ;
21 ;        cerr << "Música no disponible" << endl;
22 ;        exit(1);
23 ;
24 ;    }
25 ;
26 ;    // Establecemos un volumen predeterminado
27 ;
28 ;    Mix_VolumeMusic(VOLUMEN_MUSICA);
29 ;
30 ;#ifdef DEBUG
```

16.13. Implementación

```
31 ;     cout << "Música cargada" << endl;
32 ;#endif
33 ;
34 ;}
35 ;
36 ;void Musica::reproducir() {
37 ;
38 ;    Mix_PlayMusic(bso, -1);
39 ;
40 ;}
41 ;
42 ;
43 ;void Musica::pausar() {
44 ;
45 ;    Mix_PauseMusic();
46 ;
47 ;}
48 ;
49 ;Musica::~Musica() {
50 ;
51 ;    Mix_FreeMusic(bso);
52 ;
53 ;}
```

El constructor de la clase carga el sonido en el atributo destinado para ello y establece un volumen predeterminado para la música. El método *reproducir()* utiliza una función de la librería auxiliar para hacer sonar la música del juego y el método *pausar()* hace una llamada a *Mix_PauseMusic()* con el fin de detener la música.

El destructor libera los recursos asociados al atributo de la clase mediante la función *Mix_FreeMusic()*.

16.13.15. La clase Nivel

La clase nivel es una de las más importantes del desarrollo de nuestra aplicación. Esta clase encarga de controlar todo lo referente a la construcción, carga y almacenamiento de los niveles así como de los elementos que componen dicho nivel.

El diseño de la clase del que partimos para la implementación es el de la figura 16.48.

Vamos a estudiar la definición de la clase:

```
1 ;// Listado: Nivel.h
2 ;//
3 ;// Esta clase controla todo lo relevante a la construcción
4 ;// de los niveles del juego
```

16. Un ejemplo del desarrollo software de un videojuego

Nivel
+ Nivel : + ventana : Ventana* - numero_nivel : int - filas : int - columnas : int - modificado : bool - mapa : char - fichero : FILE* - bloques : Imagen* - universo : Universo* + dibujar(superficie : SDL_Surface*) + dibujar_actores(superficie : SDL_Surface*) + actualizar() + altura(x : int, y : int, rango : int) : int + no_es_traspasable(codigo : char) : bool + siguiente() + anterior() + editar_bloque(i : int, x : int, y : int) + guardar() + cargar() : int + limpiar() + indice() : int + generar_actores(procesos : Control_Juego*) + ~ Nivel() - abrir_fichero() - cerrar_fichero() - crear_fichero() : FILE* - copiar_fichero(tmp : FILE*)

Figura 16.48: Clase Nivel

```
5 ;  
6 ;#ifndef _NIVEL_H_  
7 ;#define _NIVEL_H_  
8 ;  
9 ;#include <SDL/SDL.h>  
10 ;#include "Ventana.h"  
11 ;#include "CommonConstants.h"  
12 ;  
13 ;class Ventana;  
14 ;class Imagen;  
15 ;class Universo;  
16 ;class Control_Juego;  
17 ;  
18 ;class Nivel {  
19 ;  
20 ; public:  
21 ;  
22 ;    // Constructor (al menos el tamaño de una ventana)  
23 ;
```

16.13. Implementación

```
24 ;     Nivel(Universo *universo,\n25 ;             int filas = FILAS_VENTANA, int columnas = COLUMNAS_VENTANA);\n26 ;\n27 ;\n28 ;     // Funciones de dibujo\n29 ;\n30 ;     void dibujar(SDL_Surface *superficie);\n31 ;     void dibujar_actores(SDL_Surface *superficie);\n32 ;\n33 ;     // Actualizando el nivel\n34 ;\n35 ;     void actualizar(void);\n36 ;     int altura(int x, int y, int rango);\n37 ;\n38 ;     // Comprueba si el elemento es traspasable\n39 ;\n40 ;     bool no_es_traspasable(char codigo);\n41 ;\n42 ;     // Pasa al nivel siguiente o al anterior\n43 ;\n44 ;     void siguiente(void);\n45 ;     void anterior(void);\n46 ;\n47 ;     // Funciones para la edición del nivel\n48 ;\n49 ;     void editar_bloque(int i, int x, int y);\n50 ;     void guardar(void);\n51 ;     int cargar(void);\n52 ;     void limpiar(void);\n53 ;\n54 ;     // Devuelve el número de nivel\n55 ;\n56 ;     int indice(void);\n57 ;\n58 ;     // Genera los actores que participan en el nivel\n59 ;\n60 ;     void generar_actores(Control_Juego *procesos);\n61 ;\n62 ;     // Foco de la acción\n63 ;\n64 ;     Ventana *ventana;\n65 ;\n66 ;     // Destructor\n67 ;\n68 ;     ~Nivel();\n69 ;\n70 ; private:\n71 ;\n72 ;     int numero_nivel;\n73 ;     int filas, columnas;\n74 ;\n75 ;     // indica si este nivel ha sido editado\n76 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
77 ;     bool modificado;
78 ;
79 ;     // Almacena las características del nivel
80 ;
81 ;     char mapa[FILAS_NIVEL][COLUMNAS_NIVEL];
82 ;
83 ;     // Para guardar el nivel
84 ;
85 ;     FILE *fichero;
86 ;
87 ;     Imagen *bloques;
88 ;     Universo *universo;
89 ;
90 ;     // Funciones para el manejo de ficheros
91 ;
92 ;     void abrir_fichero(void);
93 ;     void cerrar_fichero(void);
94 ;     FILE *crear_fichero(void);
95 ;     void copiar_fichero(FILE *tmp);
96 ;};
97 ;
98 ;#endif
```

En la parte privada de la clase tenemos varios atributos y un varios métodos para la gestión de ficheros. El atributo *numero_nivel* almacena en qué nivel nos estamos moviendo actualmente, ya sea para editarla o durante la partida. Los atributos *filas* y *columnas* almacenan el número de filas y columnas en *tiles* o recuadros que componen el nivel.

Utilizamos un atributo booleano *modificado* para saber si existen cambios en el nivel. Esto es útil cuando estamos editando el nivel para guardarla sólo en el caso de que existan cambios o bien, si se quiere salir del editor y no se han guardado los cambios, para avisar al usuario de este incidente.

El siguiente atributo *mapa* es el que tiene toda la información del nivel. Es del tipo matriz de **char** ya que es suficiente con 8 bits para detallar la información de cada *tile*. En cada posición de la matriz indicaremos la información que debe almacenarse/mostrarse del nivel.

El atributo *fichero* asocia un fichero de niveles a la clase mientras que los métodos *abrir_fichero()*, *cerrar_fichero*, *crear_fichero()* y *copiar_fichero()* nos permiten trabajar con este fichero cómodamente.

Los atributos *bloques* y *universo* asocian un elemento de la clase **Imagen** y al nivel con el **Universo** que lo engloba.

En la parte pública de la clase tenemos todos métodos necesarios para interactuar con los niveles. El método *dibujar()* se encarga de mostrar la

16.13. Implementación

escena del nivel por pantalla mientras que el método *dibujar_actores()* hace lo propio con los participantes del nivel.

El siguiente método que definimos es un método común a muchas de nuestras clases. Se trata de *actualizar()* y la única tarea que realiza es la de posicionar la ventana que se nos muestra del nivel en el lugar correcto. El método *altura()* calcula la diferencia entre la posición del participante y el elemento no traspasable más próximo en el eje *y*. Esta función toma como parámetro un rango como comprobación máxima de altura que devolverá en caso de que no exista elemento de referencia para calcular dicha altura.

El método *no_es_traspasable()* nos permite conocer si un objeto del mapa se puede traspasar o no. La utilidad reside en poder comprobar si elementos que, por ejemplo, pueden ser decorativos vamos a usarlos como plataformas para componer el nivel.

Los métodos *siguiente* y *anterior* nos permiten navegar por los distintos niveles que tenemos almacenados en el fichero de niveles. Los siguientes métodos son útiles para gestionar las operaciones que ponemos realizar con el editor de niveles. *editar_bloque()* nos permite modificar el bloque de la posición (x, y) con el elemento *i* de la rejilla de utilidades. El método *guardar()* lo utilizamos para almacenar el nivel en el fichero de niveles. Como podrás intuir el método *cargar()* nos permite traer el nivel a memoria principal ya sea para editar o para jugar sobre él. El método *limpiar()* deja el nivel actual en blanco sin ningún elemento para que podamos comenzar a construir nuestra aplicación.

El método *indice()* devuelve el número de nivel actual en el que nos encontramos y *generar_actores()* crea todos los participantes necesarios sobre el nivel sobre el que vamos a jugar para que podamos interactuar con ellos.

El atributo *ventana* asocia una clase de este tipo con la clase Nivel para poder movernos por el mismo durante el juego o la edición del nivel. Vamos a estudiar la definición de la clase:

```
1 ;// Listado: Nivel.cpp
2 ;//
3 ;// Implementación de la clase Nivel
4 ;
5 ;#include <iostream>
6 ;
7 ;#include "Nivel.h"
8 ;#include "Control_Juego.h"
9 ;#include "Universo.h"
10 ;#include "Protagonista.h"
11 ;#include "Juego.h"
```

16. Un ejemplo del desarrollo software de un videojuego

```
12 ;#include "Imagen.h"
13 ;#include "Galeria.h"
14 ;
15 ;using namespace std;
16 ;
17 ;
18 ;Nivel::Nivel(Universo *universo, int filas, int columnas) {
19 ;
20 ;#ifdef DEBUG
21 ;    cout << "Nivel::Nivel()" << endl;
22 ;#endif
23 ;
24 ;    // Inicializamos las variables
25 ;
26 ;    this->universo = universo;
27 ;    this->bloques = universo->galeria->imagen(Galeria::TILES);
28 ;
29 ;    this->filas = filas;
30 ;    this->columnas = columnas;
31 ;
32 ;    numero_nivel = 0;
33 ;    fichero = NULL;
34 ;    modificado = false;
35 ;
36 ;    ventana = new Ventana(filas, columnas);
37 ;
38 ;    // Cargamos el fichero
39 ;
40 ;    abrir_fichero();
41 ;    cargar();
42 ;
43 ;}
44 ;
45 ;// Dibuja los bloques del nivel
46 ;
47 ;void Nivel::dibujar(SDL_Surface *superficie)
48 ;{
49 ;    // Columna y fila que se lee del mapa
50 ;
51 ;    int lx, ly;
52 ;
53 ;    // Zona que se pierde al dibujar el primer bloque si no es múltiplo de 32
54 ;
55 ;    int margen_x, margen_y;
56 ;
57 ;    // Número de bloques a dibujar sobre x e y
58 ;
59 ;    int num_bloques_x, num_bloques_y;
60 ;
61 ;    // 1 bloque - 8 bits
62 ;
63 ;    char bloque;
64 ;
```

16.13. Implementación

```
65 ; // Posicionamos
66 ;
67 ; ly = ventana->pos_y() / TAMANO_BLOQUE;
68 ; lx = ventana->pos_x() / TAMANO_BLOQUE;
69 ;
70 ; // Cálculo del sobrante
71 ;
72 ; margen_y = ventana->pos_y() % TAMANO_BLOQUE;
73 ; margen_x = ventana->pos_x() % TAMANO_BLOQUE;
74 ;
75 ;
76 ;
77 ; // Si hay sobrante necesitamos un bloque más
78 ;
79 ; if(margen_x == 0)
80 ;     num_bloques_x = columnas;
81 ; else
82 ;     num_bloques_x = columnas + 1;
83 ;
84 ; if(margen_y == 0)
85 ;     num_bloques_y = filas;
86 ; else
87 ;     num_bloques_y = filas + 1;
88 ;
89 ;
90 ; // Dibujamos los bloques
91 ;
92 ; for(int col = 0; col < num_bloques_x; col++) {
93 ;     for(int fil = 0; fil < num_bloques_y; fil++) {
94 ;
95 ;         bloque = mapa[fil + ly][col + lx];
96 ;
97 ;         if(bloque != -1 && bloque < 36) {
98 ;             bloques->dibujar(superficie, bloque,\n99 ;                             col * TAMANO_BLOQUE - margen_x,\n100 ;                            fil * TAMANO_BLOQUE - margen_y, 1);
101 ;         }
102 ;     }
103 ; }
104 ;}
105 ;
106 ;
107 ;
108 ;void Nivel::dibujar_actores(SDL_Surface *superficie) {
109 ;
110 ;    // Columna y fila que se lee del mapa
111 ;
112 ;    int lx, ly;
113 ;
114 ;    // Zona que se pierde al dibujar
115 ;    // el primer bloque si no es múltiplo de 32
116 ;
117 ;    int margen_x, margen_y;
```

16. Un ejemplo del desarrollo software de un videojuego

```
118 ;
119 ;    // Número de bloques a dibujar sobre x e y
120 ;
121 ;    int num_bloques_x, num_bloques_y;
122 ;
123 ;    // 1 bloque - 8 bits
124 ;
125 ;    char bloque;
126 ;
127 ;
128 ;    // Posición según bloque
129 ;
130 ;    ly = ventana->pos_y() / TAMANO_BLOQUE;
131 ;    lx = ventana->pos_x() / TAMANO_BLOQUE;
132 ;
133 ;    // Calculamos el sobrante
134 ;
135 ;    margen_y = ventana->pos_y() % TAMANO_BLOQUE;
136 ;    margen_x = ventana->pos_x() % TAMANO_BLOQUE;
137 ;
138 ;
139 ;
140 ;    // Si hay sobrante necesitamos un bloque más
141 ;
142 ;    if(margen_x == 0)
143 ;        num_bloques_x = columnas;
144 ;    else
145 ;        num_bloques_x = columnas + 1;
146 ;
147 ;    if(margen_y == 0)
148 ;        num_bloques_y = filas;
149 ;    else
150 ;        num_bloques_y = filas + 1;
151 ;
152 ;
153 ;    Imagen *imagen_tmp;
154 ;    Galeria::codigo_imagen codigo_tmp;
155 ;    int x0, y0;
156 ;
157 ;
158 ;    for(int col = 0; col < num_bloques_x; col++) {
159 ;
160 ;        for(int fil = 0; fil < num_bloques_y; fil++) {
161 ;
162 ;            bloque = mapa[fil + ly][col + lx];
163 ;
164 ;            if(bloque > 35 && bloque < 45) {
165 ;
166 ;                switch(bloque) {
167 ;
168 ;                    case 36:
169 ;
170 ;                        x0 = 16;
```

16.13. Implementación

```
171 ;           y0 = TAMANO_BLOQUE;
172 ;           codigo_tmp = Galeria::ENEMIGO_RATA;
173 ;           break;
174 ;
175 ;       case 37:
176 ;
177 ;           x0 = 16;
178 ;           y0 = TAMANO_BLOQUE;
179 ;           codigo_tmp = Galeria::ENEMIGO_MOTA;
180 ;           break;
181 ;
182 ;       case 42:
183 ;
184 ;           x0 = 16;
185 ;           y0 = TAMANO_BLOQUE;
186 ;           codigo_tmp = Galeria::ITEM_ALICATE;
187 ;           break;
188 ;
189 ;       case 43:
190 ;
191 ;           x0 = 16;
192 ;           y0 = TAMANO_BLOQUE;
193 ;           codigo_tmp = Galeria::ITEM_DESTORNILLADOR;
194 ;           break;
195 ;
196 ;
197 ;       case 44:
198 ;
199 ;           x0 = 16;
200 ;           y0 = TAMANO_BLOQUE;
201 ;           codigo_tmp = Galeria::PERSONAJE_PPAL;
202 ;           break;
203 ;
204 ;       default:
205 ;
206 ;           codigo_tmp = Galeria::TILES; // que componen la escena
207 ;           break;
208 ;
209 ;
210 ;       if(codigo_tmp != Galeria::TILES) {
211 ;
212 ;           imagen_tmp = universo->galeria->imagen(codigo_tmp);
213 ;           imagen_tmp->dibujar(superficie, 0,\n
214 ;                               col * TAMANO_BLOQUE - margen_x + x0,\n
215 ;                               fil * TAMANO_BLOQUE - margen_y + y0, 1);
216 ;
217 ;       }
218 ;
219 ;   }
220 ;
221 ;}
222 ;
223 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
224 ;// Actualiza la ventana en la que nos encontramos
225 ;
226 ;void Nivel::actualizar(void) {
227 ;
228 ;    ventana->actualizar();
229 ;}
230 ;
231 ;
232 ;// Calcula la altura de un elemento
233 ;
234 ;int Nivel::altura(int x, int y, int rango) {
235 ;
236 ;    // Indicamos si estamos fuera de la ventana
237 ;
238 ;    if(x < 0 || x >= ANCHO_VENTANA * 2 || y < 0 || y >= ALTO_VENTANA * 2)
239 ;        return rango;
240 ;
241 ;    int fila, columna;
242 ;
243 ;
244 ;
245 ;
246 ;    for(int h = 0; h < rango; h++) {
247 ;
248 ;        columna = x / TAMANO_BLOQUE;
249 ;        fila = (y + h) / TAMANO_BLOQUE;
250 ;
251 ;        if((y + h) % TAMANO_BLOQUE == 0 &&
252 ;            no_es_traspasable(mapa[fila][columna]))
253 ;            return h;
254 ;    }
255 ;
256 ;    return rango;
257 ;}
258 ;
259 ;
260 ;// Devuelve true si un elemento no es traspasable
261 ;
262 ;bool Nivel::no_es_traspasable(char codigo)
263 ;{
264 ;
265 ;    // Codigo de la rejilla de los bloques
266 ;    // En la imagen que almacena los tiles
267 ;    // estos elementos son los que están definidos
268 ;    // como no transpasables
269 ;
270 ;    if(codigo >= 0 && codigo <= 5 || codigo >= 30 && codigo <= 35)
271 ;        return true;
272 ;    else
273 ;        return false;
274 ;
275 ;}
276 ;
```

16.13. Implementación

```
277 ;
278 ;// Esta función edita el bloque actual
279 ;// Si se le pasa -1 en i, limpia el bloque
280 ;// Las posiciones x e y son relativas
281 ;
282 ;void Nivel::editar_bloque(int i, int x, int y)
283 ;{
284 ;    // Calculamos la posición absoluta
285 ;
286 ;    int fila_destino = (y + ventana->pos_y()) / TAMANO_BLOQUE;
287 ;    int columna_destino = (x + ventana->pos_x()) / TAMANO_BLOQUE;
288 ;
289 ;    // Marcamos el bloque
290 ;
291 ;    mapa[fila_destino][columna_destino] = i;
292 ;
293 ;    // Si se edita un bloque se activa
294 ;    // la opción de guardar el nivel modificado
295 ;
296 ;    modificado = true;
297 ;}
298 ;
299 ;
300 ;// Esta función almacena el mapa en un fichero
301 ;
302 ;void Nivel::guardar(void)
303 ;{
304 ;    FILE * salida;
305 ;
306 ;    if(modificado == false) {
307 ;        cerr << "Nivel::guardar() ->" 
308 ;            << " Nivel no modificado, no se almacenará" << endl;
309 ;        return;
310 ;    }
311 ;
312 ;    salida = fopen("niveles.dat", "rb+");
313 ;
314 ;    // Si no existe, intentamos crear un fichero
315 ;
316 ;    if(salida == NULL) {
317 ;
318 ;        salida = crear_fichero();
319 ;
320 ;        if(salida == NULL) {
321 ;
322 ;            // No podemos crearlo
323 ;
324 ;            cerr << "Nivel::guardar() -> Sin acceso de "
325 ;                << "escritura al sistema de ficheros" << endl;
326 ;            return;
327 ;        }
328 ;
329 ;    }
```

16. Un ejemplo del desarrollo software de un videojuego

```
330 ;
331 ;
332 ;    if(fseek(salida, BLOQUES_NIVEL * numero_nivel, SEEK_SET)) {
333 ;
334 ;        cerr << "Nivel::guardar() -> Error en el fichero" << endl;
335 ;        fclose(salida);
336 ;        return;
337 ;
338 ;    }
339 ;    else {
340 ;
341 ;        if(fwrite(&mapa, sizeof(char), BLOQUES_NIVEL, salida) < BLOQUES_NIVEL) {
342 ;
343 ;            cerr << "Nivel::guardar() -> Error de "
344 ;                << "escritura en el fichero" << endl;
345 ;
346 ;            fclose(salida);
347 ;            return;
348 ;        }
349 ;    }
350 ;
351 ;    modificado = false; // Una vez guardado, ya no está modificado
352 ;
353 ;    fclose(fichero);
354 ;
355 ;    fflush(salida);
356 ;    fichero = salida;
357 ;}
358 ;
359 ;
360 ;
361 ;// Carga el fichero de niveles
362 ;
363 ;int Nivel::cargar(void) {
364 ;
365 ;    if(fseek(fichero, BLOQUES_NIVEL * numero_nivel, SEEK_SET)) {
366 ;
367 ;        cerr << "Sin acceso al fichero de niveles" << endl;
368 ;        return 1;
369 ;    }
370 ;    else {
371 ;
372 ;        if(fread(&mapa, sizeof(char), BLOQUES_NIVEL, fichero) < BLOQUES_NIVEL) {
373 ;            cerr << "No se puede cargar el fichero de niveles" << endl;
374 ;            return 1;
375 ;
376 ;        }
377 ;    }
378 ;
379 ;    return 0;
380 ;}
381 ;
382 ;
```

16.13. Implementación

```
383 ;// Limpia el mapa del nivel
384 ;
385 ;void Nivel::limpiar(void) {
386 ;
387 ;    // Rellena a -1 todas las posiciones
388 ;
389 ;    memset(mapa, -1, BLOQUES_NIVEL);
390 ;
391 ;}
392 ;
393 ;
394 ;// Pasa de nivel
395 ;
396 ;void Nivel::siguiente(void)
397 ;{
398 ;    // Un nivel más
399 ;
400 ;    numero_nivel++;
401 ;
402 ;
403 ;    if(cargar())
404 ;
405 ;        // Si no podemos acceder a un nivel más (no existe, o fallo)
406 ;        numero_nivel--;
407 ;}
408 ;
409 ;
410 ;// Pasa al nivel anterior
411 ;
412 ;void Nivel::anterior(void)
413 ;{
414 ;    // Nivel inicial es el 0
415 ;
416 ;    if(numero_nivel > 0) {
417 ;
418 ;        numero_nivel--;
419 ;        cargar(); // el nivel
420 ;
421 ;    }
422 ;}
423 ;
424 ;
425 ;int Nivel::indice(void) {
426 ;
427 ;    return numero_nivel;
428 ;
429 ;}
430 ;
431 ;
432 ;void Nivel::generar_actores(Control_Juego * control_juego)
433 ;{
434 ;    int bloque;
435 ;    int x, y;
```

16. Un ejemplo del desarrollo software de un videojuego

```
436 ;
437 ;    for(int col = 0; col < COLUMNAS_NIVEL; col++) {
438 ;        for(int fil = 0; fil < FILAS_NIVEL; fil++) {
439 ;
440 ;            // Según la información del bloque
441 ;
442 ;            bloque = mapa[fil][col];
443 ;
444 ;            x = (col * TAMANO_BLOQUE) + 16;
445 ;            y = (fil * TAMANO_BLOQUE) + TAMANO_BLOQUE;
446 ;
447 ;            // Cargamos el actor correspondiente
448 ;
449 ;            switch(bloque) {
450 ;
451 ;                case 36:
452 ;
453 ;                    control_juego->enemigo(Participante::TIPO_ENEMIGO_RATA,\n
454 ;                                         x, y, 1);
455 ;                    break;
456 ;
457 ;                case 37:
458 ;                    control_juego->enemigo(Participante::TIPO_ENEMIGO_MOTA,\n
459 ;                                         x, y, 1);
460 ;                    break;
461 ;
462 ;                case 42:
463 ;                    control_juego->item(Participante::TIPO_ALICATE,\n
464 ;                                         x, y, 1);
465 ;                    break;
466 ;
467 ;                case 43:
468 ;                    control_juego->item(Participante::TIPO_DESTORNILLADOR,\n
469 ;                                         x, y, 1);
470 ;                    break;
471 ;
472 ;                case 44:
473 ;                    control_juego->protagonista(x, y, 1);
474 ;                    break;
475 ;
476 ;                default:
477 ;                    break;
478 ;
479 ;            }
480 ;        }
481 ;    }
482 ;}
483 ;
484 ;
485 ;Nivel::~Nivel(void) {
486 ;
487 ;#ifndef DEBUG
488 ;    cout << "Nivel::~Nivel()" << endl;
```

16.13. Implementación

```
489 ;#endif
490 ;
491 ;    delete ventana;
492 ;    cerrar_fichero();
493 ;}
494 ;
495 ;
496 ;void Nivel::abrir_fichero(void) {
497 ;
498 ;    fichero = fopen("niveles.dat", "rb");
499 ;
500 ;    if(fichero == NULL)
501 ;        cerr << "No se encuentra el fichero niveles.dat" << endl;
502 ;
503 ;}
504 ;
505 ;
506 ;void Nivel::cerrar_fichero(void) {
507 ;
508 ;    // Si existe el fichero
509 ;
510 ;    if(fichero) {
511 ;
512 ;        fclose(fichero); // Lo cerramos
513 ;        fichero = NULL;
514 ;
515 ;    }
516 ;    else {
517 ;
518 ;        cerr << "El fichero de niveles no estaba abierto" << endl;
519 ;    }
520 ;}
521 ;
522 ;
523 ;FILE * Nivel::crear_fichero(void) {
524 ;
525 ;    FILE * tmp;
526 ;
527 ;    tmp = fopen("niveles.dat", "wb+");
528 ;
529 ;    if(tmp == NULL)
530 ;
531 ;        cerr << "No se puede crear el fichero niveles.dat" << endl;
532 ;    else {
533 ;
534 ;        // Lo copiamos en el nuestro
535 ;
536 ;        copiar_fichero(tmp);
537 ;        cout << "Fichero de niveles creado" << endl;
538 ;    }
539 ;
540 ;    return tmp;
541 ;}
```

16. Un ejemplo del desarrollo software de un videojuego

```
542 ;
543 ;
544 ;// Copiamos el fichero abierto a nuestro fichero de niveles
545 ;// para facilitar la edición
546 ;
547 ;void Nivel::copiar_fichero(FILE * tmp) {
548 ;
549 ;    char mapa_tmp[FILAS_NIVEL] [COLUMNAS_NIVEL];
550 ;
551 ;    for(int i = 0; i < COLUMNAS_NIVEL; i++) {
552 ;
553 ;        // Copiamos el fichero, nivel a nivel
554 ;
555 ;        fseek(fichero, i * BLOQUES_NIVEL, SEEK_SET);
556 ;        fseek(tmp, i * BLOQUES_NIVEL, SEEK_SET);
557 ;
558 ;        fread(mapa_tmp, BLOQUES_NIVEL, 1, fichero);
559 ;        fwrite(mapa_tmp, BLOQUES_NIVEL, 1, tmp);
560 ;
561 ;    }
562 ;
563 ;    cout << "Almacenado fichero de niveles" << endl;
564 ;}
```

Vamos a empezar analizando el constructor de la clase. El constructor inicializa todos los atributos de la clase y crea una nueva ventana que nos permita navegar por el nivel. Una vez realizada esta tarea abre el fichero de niveles y carga el primer nivel en el editor o en la propia escena de juego.

El método *dibujar()* realiza un cálculo de los bloques que tiene que dibujar en la ventana teniendo en cuenta que es posible que haya bloques que no tengan que dibujarse enteros. Una vez calculado se recorre el contenido de la variable *mapa* que contiene la información de los elementos del nivel dibujando en cada bloque el elemento que corresponda.

El método *dibujar_actores()* es análogo al anterior pero en vez de dibujar los elementos correspondientes en cada uno de los tiles lo que hace es convertir los tiles marcados con el código de algún personaje en el propio personaje. Así conseguimos generar todos los actores reales del videojuego.

La implementación del método *altura()* comprueba que el elemento esté dentro del nivel. Si es así calcula su altura dentro de un rango máximo. Lo que hace es comprobar los bloques que existen desde la posición en el eje *y* del elemento a comprobar hasta que se acaba el nivel de los elementos hasta comprobar si existe alguno que no sea traspasable para así identificar la altura.

El método *no_es_traspasable()* devuelve un valor verdadero si el elemento que recibe como parámetro no es traspasable, como no podía ser de otra

forma. Utiliza una estructura selectiva para comprobar el código del elemento. Hemos definido los primeros 6 y los últimos 6 elementos de la rejilla de tiles como elementos no traspasables o sólidos.

La edición de un bloque del nivel es bastante sencilla. Se calcula sobre qué bloque está situado el cursor del ratón y seguidamente establecemos en dicha posición el código *i* del elemento que queremos colocar en dicho lugar. Para terminar marcamos el nivel a modificado para que sea tenido en cuenta a la hora de grabar el nivel.

Para guardar el nivel hacemos uso del método *guardar()*. Este método nos permite guardar el nivel en la posición correspondiente al número de nivel que queramos guardar. La primera comprobación que realiza es que el nivel haya sido modificado para luego abrir el fichero de datos, buscar la posición donde debemos guardarlo almacenando el contenido de la variable *mapa* en dicho fichero. Una vez realizada esta tarea se marca la bandera *modificado* a false ya que no existen modificaciones sin guardar y se reestablece el fichero.

El método *cargar()* simplemente busca la posición del fichero donde está almacenado y carga la información del nivel en la variable *mapa*. El método limpiar rellena el contenido de mapa con valores -1 que indican que la posición de bloque está vacía. Los métodos *siguiente()* y *anterior()* cargan los respectivos niveles en la variable *mapa* en caso de no existir vuelven al nivel de partida.

El método *generar_actores()* establece en cada bloque marcado con un código referente a un participante un control sobre dicho personaje. Esto dotará a cada imagen de cierto comportamiento que dotará al juego de interactividad.

El resto de los métodos de la clase son simples ejercicios de programación que nos permiten manejar ficheros de disco. Utilizamos funciones C para este manejo para que te sea más familiar aunque el manejo de fichero en C++ de ficheros es muy potente y cómodo de utilizar.

16.13.16. La clase Participante

La clase participante es la clase madre de todos los actores que convergen en los niveles del videojuego. Nos permite establecer unas características generales para todos estos participantes y poderlos manejar así de una manera más o menos común.

El diseño de la clase del que partimos para la implementación es el de la figura 16.49.

16. Un ejemplo del desarrollo software de un videojuego

Participante
imagen : Imagen* # juego : Juego* # x : int # y : int # direccion : int # velocidad_salto : float # estado : enum estados # estado_anterior : enum estados # animaciones : map< estados, Control_Animacion * > + Participante(juego : Juego*, x : int, y : int, direccion : int) + pos_x() : int + pos_y() : int + actualizar() + dibujar(pantalla : SDL_Surface*) + colisiona_con(otro : Participante*) + ~ Participante() + estado_actual() : enum estados # mover_sobre_x(incremento : int) # pisa_el_suelo() : bool # pisa_el_suelo(_x : int, _y : int) : bool # altura(rango : int) : int

Figura 16.49: Clase Participante

Vamos a estudiar la definición de la clase:

```
;  
1 ;// Listado: Participante.h  
2 ;//  
3 ;// Clase madre que proporciona una interfaz para los participantes  
4 ;// que componen el juego  
5 ;  
6 ;#ifndef _PARTICIPANTE_H_  
7 ;#define _PARTICIPANTE_H_  
8 ;  
9 ;#include <iostream>  
10 ;#include <map>  
11 ;#include <SDL/SDL.h>  
12 ;  
13 ;class Control_Animacion;  
14 ;class Imagen;  
15 ;class Juego;  
16 ;  
17 ;using namespace std;  
18 ;  
19 ;class Participante {  
20 ;  
21 ; public:  
22 ;
```

16.13. Implementación

```
23 ; // Tipos de participantes
24 ;
25 ; enum tipo_participantes {
26 ;
27 ;     TIPO_PROTAGONISTA,
28 ;     TIPO_ENEMIGO_RATA,
29 ;     TIPO_ENEMIGO_MOTA,
30 ;     TIPO_ALICATE,
31 ;     TIPO_DESTORNILLADOR
32 ; };
33 ;
34 ; // Estados posibles de los participantes
35 ;
36 ; enum estados {
37 ;
38 ;     PARADO,
39 ;     CAMINAR,
40 ;     SALTAR,
41 ;     MORIR,
42 ;     ELIMINAR,
43 ;     GOLPEAR
44 ; };
45 ;
46 ; // Constructor
47 ;
48 ; Participante(Juego *juego, int x, int y, int direccion = 1);
49 ;
50 ; // Consultoras
51 ; int pos_x(void);
52 ; int pos_y(void);
53 ;
54 ;
55 ; virtual void actualizar(void) = 0;
56 ;
57 ; void dibujar(SDL_Surface *pantalla);
58 ;
59 ; //
60 ; virtual void colisiona_con(Participante *otro) = 0;
61 ;
62 ; virtual ~Participante();
63 ;
64 ; estados estado_actual(void) {return estado;};
65 ;
66 ; protected:
67 ;     void mover_sobre_x(int incremento);
68 ;     bool pisa_el_suelo(void);
69 ;     bool pisa_el_suelo(int _x, int _y);
70 ;     int altura(int rango);
71 ;
72 ;     Imagen *imagen;
73 ;     Juego *juego;
74 ;     int x, y;
75 ;     int direccion;
```

16. Un ejemplo del desarrollo software de un videojuego

```
76 ;     float velocidad_salto;
77 ;
78 ;     enum estados estado;
79 ;     enum estados estado_anterior;
80 ;
81 ;     map<estados, Control_Animacion*> animaciones;
82;};
83 ;
84 ;#endif
```

En la parte protegida de la clase tenemos varios atributos y métodos que al ser heredados se convertirán en la parte priva de los items, del personaje principal y de sus adversarios. Entre estos atributos tenemos dos que asocian esta clase con las clases Imagen y Juego. Cada personaje tiene una imagen o rejilla que lo representa gráficamente dentro de un juego de ahí estas variables.

El atributo *direccion* indica si el movimiento del personaje es hacia la derecha con un 1 y hacia la izquierda con un -1. Los atributos (x, y) sirven para establecer la posición de cualquier participante y la *velocidad_salto* se aplica cuando dicho participante no está sobre un elemento no traspasable. Además de todo esto el método define una variable *estado* para saber la situación en la que se encuentra y un *estado_anterior* para saber como llegamos a dicho estado.

Para terminar con los atributos de parte protegida la clase dispone de un *map* donde almacenaremos todas las animaciones asociadas a dicho elemento. Al utilizar esta implementación de la aplicación tendremos un acceso directo a dichas animaciones además de estar gestionadas de forma dinámica. En versiones anteriores utilizamos un vector con el consecuente desperdicio de memoria.

En la parte protegida también tenemos varios métodos auxiliares. El método *mover_sobre_x()* mueve y controla el movimiento sobre el eje horizontal mientras que *pisa_el_suelo()* tiene dos versiones para comprobar si un elemento está sobre una superficie no traspasable. La función *altura()* calcula la altura de la posición del participante dentro de un rango análogamente a como se hacía en la clase Nivel.

En la parte pública de la clase definimos dos enumerados que nos van a permitir definir los estados que pueden tomar los participantes así como el tipo de participantes que van a coexistir en el videojuego. La idea es utilizar estos enumerados con el fin de que la indexación de la aplicación donde almacenamos las animaciones de cada personaje sea más cómoda.

Los métodos que ofrece la parte pública ya son conocidos por nosotros. Se tratan de, aparte del constructor y el destructor, los métodos que nos

16.13. Implementación

permiten actualizar la lógica de los participantes (*actualizar()*) y mostrar los participantes en la pantalla (*dibujar()*). Además de estos incorpora un método virtual *colisiona_con()* que nos permite informar a un elemento del juego que ha colisionado.

Como métodos consultores podemos obtener la posición del participante así como el estado actual del mismo mediante los métodos *estado_actual()*, *pos_x()* y *pos_y()*. Vamos a estudiar la implementación de esta clase.

```
1 ;// Listado: Participante.cpp
2 ;//
3 ;// Implementación de la clase Participante
4 ;
5 ;
6 ;#include <iostream>
7 ;
8 ;#include "Participante.h"
9 ;#include "Juego.h"
10 ;#include "Nivel.h"
11 ;#include "Control_Animacion.h"
12 ;#include "Imagen.h"
13 ;
14 ;using namespace std;
15 ;
16 ;
17 ;Participante::Participante(Juego *juego, int x, int y, int direccion)
18 ;{
19 ;
20 ;#ifdef DEBUG
21 ;    cout << "Participante::Participante()" << endl;
22 ;#endif
23 ;
24 ;    // Inicializamos las variables
25 ;
26 ;    this->juego = juego;
27 ;    this->direccion = 1;
28 ;    this->x = x;
29 ;    this->y = y;
30 ;    velocidad_salto = 0.0;
31 ;}
32 ;
33 ;
34 ;int Participante::pos_x(void) {
35 ;
36 ;    return x;
37 ;};
38 ;
39 ;
40 ;int Participante::pos_y(void) {
41 ;    return y;
42 ;};
```

16. Un ejemplo del desarrollo software de un videojuego

```
43 ;
44 ;
45 ;Participante::~Participante() {
46 ;
47 ;#ifdef DEBUG
48 ;    cout << "Participante::~Participante()" << endl;
49 ;#endif
50 ;
51 ;}
52 ;
53 ;void Participante::mover_sobre_x(int incremento) {
54 ;
55 ;    if(incremento > 0) {
56 ;        if(x < ANCHO_VENTANA * 2 - 30)
57 ;            x += incremento;
58 ;    }
59 ;    else {
60 ;        if(x > 30)
61 ;            x += incremento;
62 ;    }
63 ;}
64 ;
65 ;
66 ;int Participante::altura(int rango) {
67 ;
68 ;    return juego->nivel->altura(x, y, rango);
69 ;}
70 ;
71 ;
72 ;
73 ;bool Participante::pisa_el_suelo(void) {
74 ;
75 ;    if(altura(1) == 0)
76 ;        return true;
77 ;    else
78 ;        return false;
79 ;}
80 ;
81 ;
82 ;bool Participante::pisa_el_suelo(int _x, int _y) {
83 ;
84 ;    if(juego->nivel->altura(_x, _y, 1) == 0)
85 ;        return true;
86 ;    else
87 ;        return false;
88 ;
89 ;}
90 ;
91 ;
92 ;void Participante::dibujar(SDL_Surface *pantalla) {
93 ;
94 ;    imagen->dibujar(pantalla, animaciones[estado]->cuadro(), \
95 ;                      x - juego->nivel->ventana->pos_x(), \
```

```
96 ; y - juego->nivel->ventana->pos_y(), direccion);  
97 ;}  
;
```

Como podía suponer la implementación de esta clase es muy ligera ya que la esencia de los métodos estará implementada en sus clases hijas. El constructor de la clase se limita a inicializar los valores de los atributos de la misma y de los demás métodos públicos sólo se implementan las funciones que devuelven la posición del participante en un momento dado ya que es una codificación común para todos los tipos de participantes del juego.

El método *mover_sobre_x()* controla el movimiento sobre el eje horizontal y el rango que tiene el mismo mientras que la implementación del método *altura()* utiliza el el método que presentamos en la clase Nivel para el cálculo de la misma.

Para comprobar si un participante *pisa_el_suelo()* se comprueba si en un rango de un píxel se puede calcular la altura. Si es así se estaría pisando el suelo.

El método *dibujar()* utiliza el que implementamos en la clase Imagen para mostrar una imagen en una superficie en una posición determinada utilizando esta vez el cuadro actual de la animación del participante del estado actual en el que se encuentre.

16.13.17. La clase Protagonista

La clase *Protagonista* es una especificación de la clase *Participante* por lo que es similar a esta última.

El diseño de la clase del que partimos para la implementación es el de la figura 16.50.

Vamos a estudiar la definición de la clase:

```
1 ;// Listado: Protagonista.h
2 ;//
3 ;// Esta clase controla los distintos aspectos del protagonista
4 ;
5 ;
6 ;#ifndef _PROTAGONISTA_H_
7 ;#define _PROTAGONISTA_H_
8 ;
9 ;#include <SDL/SDL.h>
10 ;#include "Participante.h"
11 ;
12 ;// Declaración adelantada
13 ;
14 ;class Juego;
```

16. Un ejemplo del desarrollo software de un videojuego

Protagonista
<pre>- teclado : Teclado* - x0 : int - y0 : int + Protagonista(juego : Juego*, x : int, y : int, direccion : int) + actualizar() + colisiona_con(otro : Participante*) + ~ Protagonista() - reiniciar() - estado_caminar() - estado_parado() - estado_disparar() - estado_saltar() - estado_morir() - estado_comenzar_salto()</pre>

Figura 16.50: Clase Protagonista

```
15 ;class Participante;
16 ;class Teclado;
17 ;
18 ;class Protagonista: public Participante {
19 ;
20 ;    public:
21 ;
22 ;        // Constructor
23 ;
24 ;        Protagonista(Juego *juego, int x, int y, int direccion = 1);
25 ;
26 ;        void actualizar(void);
27 ;        void colisiona_con(Participante *otro);
28 ;
29 ;        // Destructor
30 ;
31 ;        ~Protagonista();
32 ;
33 ;    private:
34 ;
35 ;        // Dispositivo que controla el personaje
36 ;
37 ;        Teclado *teclado;
38 ;
39 ;        int x0, y0;
40 ;
41 ;        void reiniciar(void);
42 ;
43 ;        // Estados del protagonista
44 ;        // Para la implementación del autómata
45 ;
46 ;        void estado_caminar(void);
47 ;        void estado_parado(void);
48 ;        void estado_disparar(void);
49 ;        void estado_saltar(void);
```

16.13. Implementación

```
50 ;     void estado_morir(void);
51 ;     void estado_comenzar_salto(void);
52 ;
53;};
54;
55;#endif
```

Añade a los métodos de su clase madre varios métodos privados auxiliares para implementar el autómata que hemos diseñado para conocer que estados y movimientos puede realizar el protagonista en un momento determinado.

Además incluye un atributo que permite asociar al personaje con el teclado que será el dispositivo de entrada. Las acciones sobre el teclado determinaran los cambios en el diagrama de transiciones o de estados. Cuando muera el protagonista (y si todavía le quedasen vidas) debe de reiniciar su estado y el de la animación que lo representa al estado inicial del juego. Para realizar esta tarea se ha incluido el método *reiniciar()*.

Existen dos atributos nuevos que nos permiten almacenar la posición original del protagonista por si tenemos que restaurarlo a ella después de que ocurre alguna acción del juego.

Pasemos a estudiar la implementación de la clase:

```
;_____
1 ;// Listado: Protagonista
2 ;//
3 ;// Implementación de la clase Protagonista
4 ;
5 ;#include <iostream>
6 ;
7 ;#include "Protagonista.h"
8 ;#include "Juego.h"
9 ;#include "Teclado.h"
10 ;#include "Galeria.h"
11 ;#include "Sonido.h"
12 ;#include "Universo.h"
13 ;#include "Nivel.h"
14 ;#include "Imagen.h"
15 ;#include "Control_Animacion.h"
16 ;
17 ;
18 ;using namespace std;
19 ;
20 ;
21 ;Protagonista::Protagonista(Juego *juego, int x, int y, int direccion):
22 ;    Participante(juego, x, y, direccion)
23 ;{
24 ;
25 ;#ifdef DEBUG
```

16. Un ejemplo del desarrollo software de un videojuego

```
26 ;     cout << "Protagonista::Protagonista()" << endl;
27 ;#endif
28 ;
29 ;    // Inicializamos los atributos de la clase
30 ;
31 ;    this->juego = juego;
32 ;    this->teclado = &(juego->universo->teclado);
33 ;
34 ;    imagen = juego->universo->galeria->imagen(Galeria::PERSONAJE_PPAL);
35 ;
36 ;    // Asociamos al personaje las animaciones
37 ;    // según la rejilla que cargamos para controlarlo
38 ;
39 ;    animaciones[PARADO] = new Control_Animacion("0", 5);
40 ;    animaciones[CAMINAR] = new Control_Animacion("1,2,3,2,1,0,4,5,6,5,4,0", 6);
41 ;    animaciones[SALTAR] = new Control_Animacion("21,19", 0);
42 ;    animaciones[GOLPEAR] = new Control_Animacion("15", 20);
43 ;    animaciones[MORIR] = new Control_Animacion("22, 23, 23, 24, 25, 23, 25", 10);
44 ;
45 ;    x0 = x;
46 ;    y0 = y;
47 ;
48 ;    reiniciar();
49 ;}
50 ;
51 ;
52 ;
53 ;void Protagonista::actualizar(void) {
54 ;
55 ;    // Establecemos la posición de la ventana
56 ;
57 ;    juego->nivel->ventana->establecer_pos(x - 320, y - 240);
58 ;
59 ;    // Si no estamos en el mismo estado
60 ;
61 ;    if(estado != estado_anterior) {
62 ;
63 ;        // Comenzamos la animación y guardamos el estado
64 ;
65 ;        animaciones[estado]->reiniciar();
66 ;        estado_anterior = estado;
67 ;    }
68 ;
69 ;    // Implementación del autómata
70 ;    // Según sea el estado
71 ;
72 ;    switch(estado) {
73 ;
74 ;        case PARADO:
75 ;
76 ;            estado_parado();
77 ;            break;
78 ;    }
```

16.13. Implementación

```
79 ;     case CAMINAR:
80 ;
81 ;         estado_caminar();
82 ;         break;
83 ;
84 ;     case GOLPEAR:
85 ;
86 ;         estado_disparar();
87 ;         break;
88 ;
89 ;     case SALTAR:
90 ;
91 ;         estado_saltar();
92 ;         break;
93 ;
94 ;     case MORIR:
95 ;
96 ;         estado_morir();
97 ;         break;
98 ;
99 ; default:
100 ;     cout << "Estado no contemplado" << endl;
101 ;     break;
102 ; }
103 ;
104 ;}
105 ;
106 ;void Protagonista::colisiona_con(Participante *otro) {
107 ;
108 ;    if(estado != MORIR) {
109 ;
110 ;        juego->universo-> \
111 ;            galeria->sonidos[Galeria::MUERE_BUENO]->reproducir();
112 ;
113 ;        estado = MORIR; // Muere el personaje
114 ;        velocidad_salto = -5; // Hace el efecto de morir
115 ;    }
116 ;}
117 ;
118 ;
119 ;void Protagonista::reiniciar(void) {
120 ;
121 ;    // Reestablecemos el personaje
122 ;
123 ;    x = x0;
124 ;    y = y0;
125 ;    direccion = 1;
126 ;
127 ;    estado = PARADO;
128 ;    velocidad_salto = 0;
129 ;    estado_anterior = estado;
130 ;}
131 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
132 ;
133 ;// Destructor
134 ;
135 ;Protagonista::~Protagonista()
136 ;{
137 ;
138 ;#ifndef DEBUG
139 ;    cout << "Protagonista::~Protagonista()" << endl;
140 ;#endif
141 ;
142 ;}
143 ;
144 ;// Funciones que implementan el diagrama de estados
145 ;
146 ;void Protagonista::estado_parado(void) {
147 ;
148 ;    // Estando parado podemos realizar las
149 ;    // siguientes acciones
150 ;
151 ;    if(teclado->pulso(Teclado::TECLA_IZQUIERDA)) {
152 ;
153 ;        direccion = -1;
154 ;        estado = CAMINAR;
155 ;
156 ;    }
157 ;
158 ;    if(teclado->pulso(Teclado::TECLA_DERECHA)) {
159 ;
160 ;        direccion = 1;
161 ;        estado = CAMINAR;
162 ;
163 ;    }
164 ;
165 ;    if(teclado->pulso(Teclado::TECLA_GOLPEAR))
166 ;        estado = GOLPEAR;
167 ;
168 ;    if(teclado->pulso(Teclado::TECLA_SALTAR)) {
169 ;
170 ;        velocidad_salto = -5;
171 ;        estado = SALTAR;
172 ;
173 ;    }
174 ;}
175 ;
176 ;
177 ;
178 ;void Protagonista::estado_caminar(void) {
179 ;
180 ;    // Acciones que podemos realizar
181 ;    // mientras caminamos
182 ;
183 ;    animaciones[estado]->avanzar();
184 ;}
```

16.13. Implementación

```
185 ;     mover_sobre_x(direccion * 2);
186 ;
187 ;     if(direccion == 1 && ! teclado->pulso(Teclado::TECLA_DERECHA))
188 ;         estado = PARADO;
189 ;
190 ;     if(direccion == -1 && ! teclado->pulso(Teclado::TECLA_IZQUIERDA))
191 ;         estado = PARADO;
192 ;
193 ;     if(teclado->pulso(Teclado::TECLA_GOLPEAR))
194 ;         estado = GOLPEAR;
195 ;
196 ;     if(teclado->pulso(Teclado::TECLA_SALTAR)) {
197 ;
198 ;         velocidad_salto = -5;
199 ;         estado = SALTAR;
200 ;
201 ;     }
202 ;
203 ;     if(!pisa_el_suelo()) {
204 ;
205 ;         velocidad_salto = 0;
206 ;         estado = SALTAR;
207 ;
208 ;     }
209 ;}
210 ;
211 ;
212 ;void Protagonista::estado_disparar(void) {
213 ;
214 ;    // Cuando golpeamos nos detenemos
215 ;
216 ;    if(animaciones[estado]->avanzar())
217 ;        estado = PARADO;
218 ;
219 ;}
220 ;
221 ;
222 ;
223 ;void Protagonista::estado_saltar(void) {
224 ;
225 ;    // Acciones que podemos realizar al saltar
226 ;
227 ;    velocidad_salto += 0.1;
228 ;
229 ;    if(teclado->pulso(Teclado::TECLA_IZQUIERDA)) {
230 ;
231 ;        direccion = -1;
232 ;        mover_sobre_x(direccion * 2);
233 ;
234 ;    }
235 ;
236 ;    if(teclado->pulso(Teclado::TECLA_DERECHA)) {
237 ;
```

16. Un ejemplo del desarrollo software de un videojuego

```
238 ;         direccion = 1;
239 ;         mover_sobre_x(direccion * 2);
240 ;
241 ;     }
242 ;
243 ;
244 ; // Cuando la velocidad cambia de signo empezamos a caer
245 ;
246 ; if(velocidad_salto > 0.0) {
247 ;
248 ;     y += altura((int) velocidad_salto);
249 ;
250 ;     if(animaciones[estado]->es_primer_cuadro())
251 ;         animaciones[estado]->avanzar();
252 ;
253 ;     if(velocidad_salto >= 1.0 && pisa_el_suelo())
254 ;         estado = PARADO;
255 ;
256 ;     if(y > ALTO_VENTANA * 2) {
257 ;         estado = MORIR;
258 ;         velocidad_salto = -5;
259 ;     }
260 ; }
261 ; else {
262 ;
263 ;     y += (int) velocidad_salto;
264 ;
265 ; }
266 ;}
267 ;
268 ;
269 ;void Protagonista::estado_morir(void) {
270 ;
271 ; // Morimos
272 ;
273 ; velocidad_salto += 0.1;
274 ; y += (int) velocidad_salto;
275 ;
276 ; mover_sobre_x(direccion * 2 * - 1);
277 ;
278 ; animaciones[estado]->avanzar();
279 ;
280 ; if(y > ALTO_VENTANA * 2 + 300) // Salimos de la pantalla
281 ;     reiniciar();
282 ;}
```

En el constructor de la clase hacemos uso del constructor de la clase base de ésta para inicializar los atributos heredados. En el resto de este método se inicializan los atributos de la clase y preparan las animaciones que vamos a utilizar según el estado del protagonista. Al terminar el constructor se reinicia el estado del protagonista para que esté en un estado inicial.

El método *actualizar()* tiene una implementación algo más extensa de lo que estamos acostumbrados. Lo primero que se hace en el método es establecer la posición de la ventana del nivel. Se toma la posición del protagonista como centro de dicha ventana. Seguidamente si el personaje ha cambiado de estado se reinicia la animación para comenzar con el nuevo tipo de ésta. Lo siguiente que nos encontramos es la implementación del autómata por el método de los *cases*. Según sea el estado en el que nos encontramos entramos en un caso u otro y realizamos unas determinadas acciones que están incluidas en cada uno de las funciones que hemos definido con tal fin y que estudiaremos a continuación.

El método *colisiona_con()* hace que el protagonista muera reproduciendo una animación que hemos creado con este fin mientras que el método *reiniciar()* de esta clase reestablece todos los atributos a su estado general: la posición, el estado, el salto, la dirección...

El resto de la implementación de la clase concierne al desarrollo de la codificación del autómata del personaje principal. El diagrama de estados o transiciones de este es el mismo que estudiamos en capítulos anteriores simplificando su implementación.

El método *estado_parado()* se corresponde con el diagrama cuando el personaje está parado. Cuando está en este estado el personaje puede caminar hacia la izquierda, hacia la derecha, golpear o saltar. Según sea la tecla que pulsemos entraremos en un caso u otro dentro la misma. Los demás métodos tienen un razonamiento análogo respondiendo al diagrama de transiciones.

En el caso de *estado_morir()* ya no podemos realizar ninguna acción más por lo que ponemos en marcha la animación del estado y cuando el personaje haya salido de la ventana el doble de su altura reiniciaremos el protagonista.

16.13.18. La clase Sonido

La clase sonido nos proporciona todo aquello que necesitamos para introducir sonidos de acción en nuestra aplicación. Con ayuda de la galería podemos gestionar todo lo referente a estos sonidos.

El diseño de la clase del que partimos para la implementación es el de la figura 16.51.

Vamos a estudiar la definición de la clase:

```
1 ;// Listado: Sonido.h
2 ;//
3 ;// Clase para facilitar el trabajo con sonidos
```

16. Un ejemplo del desarrollo software de un videojuego

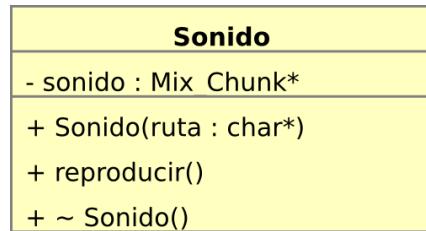


Figura 16.51: Clase Sonido

```
4 ;
5 ;#ifndef _SONIDO_H_
6 ;#define _SONIDO_H_
7 ;
8 ;#include <SDL/SDL.h>
9 ;#include <SDL/SDL_mixer.h>
10 ;
11 ;class Sonido {
12 ;
13 ; public:
14 ;
15 ;     // Constructor
16 ;
17 ;     Sonido(char *ruta);
18 ;
19 ;     void reproducir();
20 ;
21 ;
22 ;     // Destructor
23 ;
24 ;     ~Sonido();
25 ;
26 ; private:
27 ;
28 ;     Mix_Chunk *sonido;
29 ;
30 ;};
31 ;
32 ;#endif
;
```

Como puedes ver en la definición de la clase utilizamos la librería auxiliar *SDL_mixer* para poder manejar más cómodamente estos sonidos. La clase consta del destructor y el destructor pertinentes y un método que nos permite reproducir este sonido una vez en el juego. Los sonidos de acciones no suelen ser iterativos y como nosotros no vamos a necesitar que se produzcan repetidamente no vamos a necesitar en este método ningún parámetro que especifique dicho número de repeticiones.

Veamos la implementación de la clase:

16.13. Implementación

```
1 ;_____
2 // Listado: Sonido.cpp
3 //_____
4 ;_____
5 #include <iostream>
6 ;
7 #include "Sonido.h"
8 ;
9 using namespace std;
10 ;
11 ;
12 ;
13 ;Sonido::Sonido(char *ruta) {
14 ;
15 ;    // Cargamos el sonido
16 ;
17 ;    sonido = Mix_LoadWAV(ruta);
18 ;
19 ;    if(sonido == NULL) {
20 ;
21 ;        cerr << "Sonido " << ruta << " no disponible" << endl;
22 ;        exit(1);
23 ;
24 ;    }
25 ;
26 ;    Mix_AllocateChannels(1);
27 ;
28 ;#ifdef DEBUG
29 ;    cout << "Sonido cargado" << endl;
30 ;#endif
31 ;
32 ;}
33 ;
34 ;
35 ;void Sonido::reproducir() {
36 ;
37 ;    Mix_PlayChannel(-1, sonido, 0);
38 ;
39 ;}
40 ;
41 ;
42 ;Sonido::~Sonido() {
43 ;
44 ;    Mix_FreeChunk(sonido);
45 ;
46 ;}_____
```

El constructor de la clase carga el fichero de sonido especificado como parámetro y reserva un canal de audio. El método reproducir hace sonar este chunk por todos los canales una vez. El destructor de la clase libera los recursos reservados para dicho sonido.

16. Un ejemplo del desarrollo software de un videojuego

16.13.19. La clase Teclado

Para no complicar más la implementación del videojuego y como ya hicimos un repaso profundo sobre el manejo del joystick en el capítulo correspondiente a su estudio vamos a definir como elementos de entrada del videojuego al ratón y al joystick. Con esta clase vamos a dotar de toda la potencia que necesitamos para recibir la entrada de teclado.

El diseño de la clase del que partimos para la implementación es el de la figura 16.52.

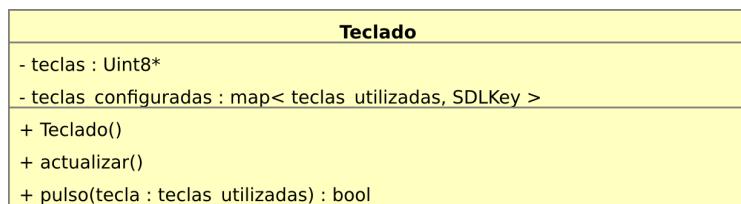


Figura 16.52: Clase Teclado

Vamos a estudiar la definición de la clase:

```
1 ;_____
2 ;// Listado: Teclado.h
3 ;// Control del dispositivo de entrada
4 ;
5 ;#ifndef _TECLADO_H_
6 ;#define _TECLADO_H_
7 ;
8 ;#include <SDL/SDL.h>
9 ;#include <map>
10 ;
11 ;using namespace std;
12 ;
13 ;
14 ;class Teclado {
15 ;
16 ;    public:
17 ;
18 ;        // Teclas a usar en la aplicación
19 ;
20 ;        enum teclas_utilizadas {
21 ;
22 ;            TECLA_SALIR,
23 ;            TECLA_SUBIR,
24 ;            TECLA_BAJAR,
25 ;            TECLA_ACEPTAR,
26 ;            TECLA_GOLPEAR,
27 ;            TECLA_IZQUIERDA,
```

16.13. Implementación

```
28 ;      TECLA_DERECHA,
29 ;      TECLA_SALTAR,
30 ;      TECLA_GUARDAR
31 ;
32 ;      };
33 ;
34 ;      // Constructor
35 ;
36 ;      Teclado();
37 ;
38 ;      // Consultoras
39 ;
40 ;      // Actualiza la información del teclado
41 ;
42 ;      void actualizar(void);
43 ;
44 ;      // Informa si una tecla ha sido pulsada
45 ;
46 ;      bool pulso(teclas_utilizadas tecla);
47 ;
48 ;
49 ; private:
50 ;
51 ;      // Para conocer el estado de la pulsación
52 ;      // de las teclas en todo momento
53 ;
54 ;      Uint8* teclas;
55 ;
56 ;
57 ;      // Asocia las teclas que necesitamos a la
58 ;      // constante SDL que la representa
59 ;
60 ;      map<teclas_utilizadas, SDLKey> teclas_configuradas;
61 ;};
62 ;
63 ;#endif
```

Lo primero que nos encontramos en la definición de la clase es un enumerado que nos permitirá trabajar más cómodamente con las constantes de teclado. Además del constructor la clase tiene dos métodos. El primero es común a muchas de nuestras clases. Se trata *actualizar()* y se encarga de renovar la información que tenemos del teclado.

El segundo, el método *pulso()*, nos permite conocer el estado de una tecla que le pasamos como parámetro. Este método consultivo será el que más utilicemos ya que tendremos que reaccionar a la pulsación de determinadas teclas en determinados momentos.

En la parte privada de la clase tenemos una variable *teclas* con el estado del teclado y una aplicación implementada mediante un *map* que nos permite tener almacenada la relación entre las constantes de SDL y las teclas que

16. Un ejemplo del desarrollo software de un videojuego

tenemos configuradas en nuestro enumerado.

Vamos a ver la implementación de la clase:

```
1 ;// Listado: Teclado.cpp
2 ;//
3 ;// Implementación de la clase teclado
4 ;
5 ;#include <iostream>
6 ;#include "Teclado.h"
7 ;
8 ;
9 ;using namespace std;
10 ;
11 ;
12 ;Teclado::Teclado() {
13 ;
14 ;#ifndef DEBUG
15 ;    cout << "Teclado::Teclado()" << endl;
16 ;#endif
17 ;
18 ;
19 ;    // Configuramos la teclas que usaremos en la aplicación
20 ;
21 ;    teclas_configuradas[TECLA_SALIR] = SDLK_ESCAPE;
22 ;    teclas_configuradas[TECLA_SUBIR] = SDLK_UP;
23 ;    teclas_configuradas[TECLA_BAJAR] = SDLK_DOWN;
24 ;    teclas_configuradas[TECLA_ACEPTAR] = SDLK_RETURN;
25 ;    teclas_configuradas[TECLA_GOLPEAR] = SDLK_SPACE;
26 ;    teclas_configuradas[TECLA_IZQUIERDA] = SDLK_LEFT;
27 ;    teclas_configuradas[TECLA_DERECHA] = SDLK_RIGHT;
28 ;    teclas_configuradas[TECLA_SALTAR] = SDLK_UP;
29 ;    teclas_configuradas[TECLA_GUARDAR] = SDLK_s;
30 ;
31 ;}
32 ;
33 ;
34 ;
35 ;void Teclado::actualizar(void) {
36 ;
37 ;    // Actualizamos el estado del teclado mediante mapeo
38 ;
39 ;    teclas = SDL_GetKeyState(NULL);
40 ;}
41 ;
42 ;
43 ;bool Teclado::pulso(teclas_utilizadas tecla) {
44 ;
45 ;    // Comprobamos si una tecla está pulsada
46 ;
47 ;    if(teclas[teclas_configuradas[tecla]])
48 ;        return true;
```

16.13. Implementación

```
49 ;     else
50 ;         return false;
51 ;}
```

El constructor de la clase relaciona las constantes SDL mediante el *map* con el enumerado que hemos definido para manejar el teclado. El método *actualizar()* utiliza la función *SDL_GetKeyState()* para renovar el estado del teclado. El método *pulso()* simplemente consulta el estado de la tecla que recibe como parámetro que es un elemento del enumerado.

16.13.20. La clase Texto

La clase texto nos permite construir frases con las fuentes almacenadas en una imagen que nos permiten hacer rótulos para nuestro juego.

El diseño de la clase del que partimos para la implementación es el de la figura 16.53.

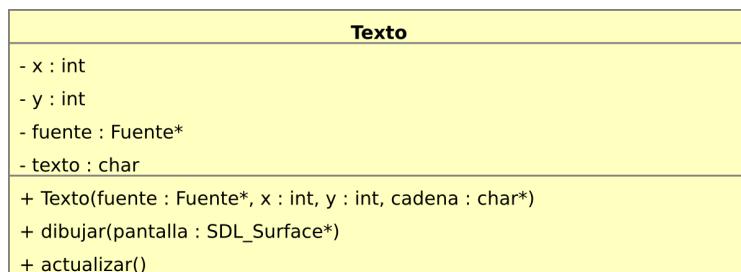


Figura 16.53: Clase Texto

Vamos a estudiar la definición de la clase:

```
1 ;// Listado: Texto.h
2 ;//
3 ;// Con esta clase controlamos los textos con los que va a trabajar la
4 ;// aplicación
5 ;
6 ;
7 ;#ifndef _TEXTO_H_
8 ;#define _TEXTO_H_
9 ;
10 ;#include <SDL/SDL.h>
11 ;#include "CommonConstants.h"
12 ;
13 ;class Fuente;
14 ;
15 ;
16 ;class Texto {
```

16. Un ejemplo del desarrollo software de un videojuego

```
17 ; public:  
18 ;  
19 ;     // Constructor  
20 ;  
21 ;     Texto(Fuente *fuente, int x, int y, char *cadena);  
22 ;  
23 ;     void dibujar(SDL_Surface *pantalla);  
24 ;     void actualizar(void);  
25 ;  
26 ; private:  
27 ;  
28 ;     int x, y;  
29 ;  
30 ;     Fuente *fuente;  
31 ;     char texto[MAX_TAM_TEXTO];  
32 ;  
33 ;};  
34 ;  
35 ;#endif  
;
```

En la parte privada de la clase tenemos varios elementos. Los atributos *x* e *y* nos permiten almacenar la posición de la imagen. La variable *texto* nos permite almacenar el texto a construir mientras que el atributo *fuente* asocia el texto con la fuente a utilizar.

En la parte pública tenemos elementos cuya funcionalidad es común a la mayoría de las clases de nuestra aplicación. Se trata de *dibujar()* que muestra (copia) el texto generado en una superficie de SDL y el método *actualizar()* que renueva el estado de dicho texto cuando se llama a dicho método.

Vamos a ver la implementación de la clase:

```
;  
1 ;// Listado:Texto.cpp  
2 ;//  
3 ;// Implementación de la clase Texto  
4 ;  
5 ;#include <iostream>  
6 ;  
7 ;#include "Fuente.h"  
8 ;#include "Control_Movimiento.h"  
9 ;#include "Texto.h"  
10 ;  
11 ;using namespace std;  
12 ;  
13 ;  
14 ;Texto::Texto(Fuente *fuente, int x, int y, char *cadena) {  
15 ;  
16 ;#ifdef DEBUG  
17 ;    cout << "Texto::Texto()" << endl;  
18 ;#endif
```

```

19 ;
20 ;    // Iniciamos las variables
21 ;
22 ;    this->fuente = fuente;
23 ;    this->x = x;
24 ;    this->y = y;
25 ;
26 ;    // Copiamos la cadena a nuestro texto
27 ;
28 ;    strcpy(texto, cadena);
29 ;
30 ;}
31 ;
32 ;
33 ;void Texto::dibujar(SDL_Surface * pantalla) {
34 ;
35 ;    // Dibujamos el texto en pantalla
36 ;
37 ;    fuente->dibujar(pantalla, texto, x, y);
38 ;}
39 ;
40 ;
41 ;void Texto::actualizar(void) {
42 ;
43 ;    // No realiza ninguna acción en particular
44 ;}
;
```

El constructor de la clase inicializa los atributos de la misma y copia la cadena que recibe como parámetro en el atributo *texto* de la clase. El método dibujar utiliza el implementado en la clase fuente para dibujar el texto creado en pantalla.

En cuanto al método *actualizar()* no realizamos ninguna acción ya que lo hemos dejado preparado por si el lector quiere agregar algún tipo de efecto al texto de los menús.

16.13.21. La clase Universo

La clase *Universo* establece el lazo de único entre todas las capacidades implementadas en las diferentes clases. Su principal función es ejecutar el bucle del juego y controlar aspectos como la sincronización del videojuego, la salida del juego o la presentación en modo ventana o pantalla completa.

El diseño de la clase del que partimos para la implementación es el de la figura 16.54.

Vamos a estudiar la definición de la clase:

16. Un ejemplo del desarrollo software de un videojuego

Universo
+ teclado : Teclado + galeria : Galería* + pantalla : SDL_Surface* - actual : Interfaz* - juego : Juego* - editor : Editor* - menu : Menu* - salir : bool + Universo() + bucle_principal() + cambiar_interfaz(nueva : Interfaz::escenas) + dibujar_rect(x : int, y : int, w : int, h : int, color : Uint32) + terminar() + ~Universo() - iniciar_ventana(fullscreen : bool) - pantalla_completa() - procesar_eventos() : int - sincronizar_fps() : int

Figura 16.54: Clase Universo

```
;  
1 ;// Listado: Universo.h  
2 ;//  
3 ;// Nos permite cohesionar todas las clases de la aplicación y  
4 ;// nos permite llevar un control global sobre la misma  
5 ;  
6 ;  
7 ;#ifndef _UNIVERSO_H_  
8 ;#define _UNIVERSO_H_  
9 ;  
10 ;#include <SDL/SDL.h>  
11 ;#include <SDL/SDL_mixer.h>  
12 ;  
13 ;#include "Teclado.h"  
14 ;#include "Interfaz.h"  
15 ;  
16 ;  
17 ;// Declaraciones adelantadas  
18 ;  
19 ;class Galeria;  
20 ;class Juego;  
21 ;class Editor;  
22 ;class Menu;  
23 ;  
24 ;  
25 ;class Universo {  
26 ;  
27 ; public:  
28 ;  
29 ;     // Constructor  
30 ;  
31 ;     Universo();  
32 ;  
33 ;  
34 ;     // Proporciona la "reproducción continua"  
35 ;
```

16.13. Implementación

```
36 ; void bucle_principal(void);
37 ;
38 ;
39 ; // Nos permite el cambio entre "opciones" del juego
40 ;
41 ; void cambiar_interfaz(Interfaz::escenas nueva);
42 ;
43 ;
44 ; // Dibuja un rectángulo en pantalla
45 ;
46 ; void dibujar_rect(int x, int y, int w, int h, Uint32 color = 0);
47 ;
48 ;
49 ; // Finaliza el juego
50 ;
51 ; void terminar(void);
52 ;
53 ; ~Universo(); // Destructor
54 ;
55 ;
56 ; Teclado teclado; // Controla el dispositivo de entrada
57 ; Galeria *galeria; // Almacena todas las imágenes necesarias
58 ; SDL_Surface *pantalla; // Superficie principal del videojuego
59 ;
60 ;
61 ; private:
62 ;
63 ; // Escena en la que estamos
64 ;
65 ; Interfaz *actual;
66 ;
67 ;
68 ; // Pantallas del juego
69 ;
70 ; Juego *juego;
71 ; Editor *editor;
72 ; Menu *menu;
73 ;
74 ; // Variable que modificamos cuando queremos salir de la aplicación
75 ;
76 ; bool salir;
77 ;
78 ; void iniciar_ventana(bool fullscreen);
79 ; void pantalla_completa(void);
80 ;
81 ;
82 ; // Estudia los eventos en un momento dado
83 ;
84 ; int procesar_eventos(void);
85 ;
86 ; // Lleva el control del tiempo
87 ;
88 ; int sincronizar_fps(void);
```

16. Un ejemplo del desarrollo software de un videojuego

```
89 ;};  
90 ;  
91 ;#endif  
;
```

En la parte privada de la clase tenemos varios atributos que nos permiten asociar a la clase directamente con las clases Interfaz, Juego, Editor y Menu. Estas son las clases principales, después de Universo, en la jerarquía de la aplicación. Además de estos atributos posee otro, *salir*, que será marcado a verdadero cuando el usuario quiera terminar con la aplicación.

Además de estos atributos la clase tiene una serie de métodos privados que estudiaremos con la implementación de la clase.

En la parte pública de la clase tenemos varios atributos que nos permiten asociar la clase con el teclado, la galería multimedia y la pantalla o superficie principal del juego. Además tenemos una serie de métodos para llevar a cabo las tareas básicas de la aplicación que pasamos a detallar con la implementación:

```
;_____  
1 ;// Listado: Universo.cpp  
2 ;//  
3 ;// Implementación de la clase Universo  
4 ;  
5 ;#include <iostream>  
6 ;  
7 ;#include "Universo.h"  
8 ;#include "Juego.h"  
9 ;#include "Editor.h"  
10 ;#include "Menu.h"  
11 ;#include "Galeria.h"  
12 ;#include "CommonConstants.h"  
13 ;  
14 ;  
15 ;using namespace std;  
16 ;  
17 ;  
18 ;Universo::Universo() {  
19 ;  
20 ;#ifdef DEBUG  
21 ;    cout << "Universo::Universo()" << endl;  
22 ;#endif  
23 ;  
24 ;    iniciar_ventana(false); // Iniciamos en modo ventana  
25 ;  
26 ;    galeria = new Galeria; // Creamos todo lo necesario  
27 ;  
28 ;    juego = new Juego(this);  
29 ;    editor = new Editor(this);  
30 ;    menu = new Menu(this);  
31 ;
```

16.13. Implementación

```
32 ;     salir = false; // No queremos salir (todavía)
33 ;
34 ;     actual = juego; // Establecemos escena actual
35 ;
36 ;     cambiar_interfaz(Interfaz::ESCENA_MENU); // Cambiamos a menu
37 ;
38 ;}
39 ;
40 ;
41 ;
42 ;void Universo::bucle_principal(void) {
43 ;
44 ;    int rep;
45 ;
46 ;
47 ;    // Bucle que realiza el polling
48 ;
49 ;    while(salir == false && procesar_eventos()) {
50 ;
51 ;        teclado.actualizar(); // Tomamos el estado del teclado
52 ;
53 ;        rep = sincronizar_fps(); // Sincronizamos el tiempo
54 ;
55 ;        // Actualizamos lógicamente
56 ;
57 ;        for(int i = 0; i < rep; i++) {
58 ;
59 ;            actual->actualizar(); // Actualizamos la escena actual
60 ;
61 ;        }
62 ;
63 ;        // Limpiamos la pantalla
64 ;
65 ;        SDL_FillRect(pantalla, NULL, \
66 ;                      SDL_MapRGB(pantalla->format, 200, 200, 200));
67 ;
68 ;        // Actualizamos las imágenes
69 ;
70 ;        actual->dibujar();
71 ;    }
72 ;}
73 ;
74 ;
75 ;void Universo::iniciar_ventana(bool fullscreen) {
76 ;
77 ;    int banderas = 0;
78 ;
79 ;    // Iniciamos todos los subsistemas
80 ;
81 ;    if(SDL_Init(0) < 0) {
82 ;
83 ;        cerr << "Universo::iniciar_ventana:" << SDL_GetError() << endl;
84 ;        exit(1);
```

16. Un ejemplo del desarrollo software de un videojuego

```
85 ;
86 ;
87 ;
88 ;    // Al salir, cerramos libSDL
89 ;
90 ;    atexit(SDL_Quit);
91 ;
92 ;    if(fullscreen)
93 ;        banderas |= SDL_FULLSCREEN;
94 ;
95 ;    banderas |= SDL_HWSURFACE | SDL_DOUBLEBUF;
96 ;
97 ;
98 ;    // Establecemos el modo de video
99 ;
100 ;   pantalla = SDL_SetVideoMode(ANCHO_VENTANA, ALTO_VENTANA, BPP, banderas);
101 ;
102 ;   if(pantalla == NULL) {
103 ;
104 ;       cerr << "Universo::iniciar_ventana:" << SDL_GetError() << endl;
105 ;       exit(1);
106 ;
107 ;   }
108 ;
109 ;   SDL_WM_SetCaption("Wiki libSDL", NULL);
110 ;
111 ;   // Ocultamos el cursor
112 ;
113 ;   SDL_ShowCursor(SDL_DISABLE);
114 ;
115 ;   // Inicializamos la librería SDL_Mixer
116 ;
117 ;   if(Mix_OpenAudio(22050, MIX_DEFAULT_FORMAT,\n
118 ;                   1, 2048) < 0) {
119 ;
120 ;       cerr << "Subsistema de Audio no disponible" << endl;
121 ;       exit(1);
122 ;   }
123 ;
124 ;   // Al salir cierra el subsistema de audio
125 ;
126 ;   atexit(Mix_CloseAudio);
127 ;
128 ;}
129 ;
130 ;
131 ;
132 ;int Universo::procesar_eventos(void) {
133 ;
134 ;    static SDL_Event event; // Con "memoria"
135 ;
136 ;
137 ;    // Hacemos el polling de eventos
```

16.13. Implementación

```
138 ;
139 ;     while(SDL_PollEvent(&event)) {
140 ;
141 ;         // Se estudia
142 ;
143 ;         switch(event.type) {
144 ;
145 ;             case SDL_QUIT:
146 ;                 return 0;
147 ;
148 ;             case SDL_KEYDOWN:
149 ;
150 ;                 // Tecla de salida
151 ;
152 ;                 if(event.key.keysym.sym == SDLK_q)
153 ;                     return 0;
154 ;
155 ;                 // Tecla paso a pantalla completa
156 ;
157 ;                 if(event.key.keysym.sym == SDLK_f)
158 ;
159 ;                     pantalla_completa();
160 ;
161 ;                 break;
162 ;
163 ;             default:
164 ;
165 ;                 // No hacemos nada
166 ;
167 ;                 break;
168 ;
169 ;             }
170 ;         }
171 ;
172 ;     return 1;
173 ;}
174 ;
175 ;
176 ;void Universo::pantalla_completa(void) {
177 ;
178 ;    // Alterna entre pantalla completa y ventana
179 ;
180 ;    SDL_WM_ToggleFullScreen(pantalla);
181 ;}
182 ;
183 ;
184 ;int Universo::sincronizar_fps(void) {
185 ;
186 ;    static int t0;
187 ;    static int tl = SDL_GetTicks();
188 ;    static int frecuencia = 1000 / 100;
189 ;    static int tmp;
190 ;}
```

16. Un ejemplo del desarrollo software de un videojuego

```
191 ;#ifdef FPS
192 ;    static int fps = 0;
193 ;    static int t_fps = 0;
194 ;#endif
195 ;
196 ;    // Tiempo de referencia
197 ;
198 ;    t0 = SDL_GetTicks();
199 ;
200 ;#ifdef FPS
201 ;
202 ;    // Actualizamos información cada segundo
203 ;    if((t0 - t_fps) >= 1000) {
204 ;        cout << "FPS = " << fps << endl;
205 ;        fps = 0;
206 ;        t_fps += 1000 + 1;
207 ;    }
208 ;
209 ;    fps++;
210 ;
211 ;#endif
212 ;
213 ;    // Estudio del tiempo
214 ;
215 ;    if((t0 - tl) >= frecuencia) {
216 ;
217 ;        tmp = (t0 - tl) / frecuencia;
218 ;        tl += tmp * frecuencia;
219 ;        return tmp;
220 ;
221 ;    } else {
222 ;
223 ;        // Tenemos que esperar para cumplir con la frecuencia
224 ;
225 ;        SDL_Delay(frecuencia - (t0 - tl));
226 ;        tl += frecuencia;
227 ;
228 ;        return 1;
229 ;    }
230 ;}
231 ;
232 ;
233 ;void Universo::cambiar_interfaz(Interfaz::escenas nueva) {
234 ;
235 ;    Interfaz *anterior = actual;
236 ;
237 ;    // Segundo sea la escena a la que queremos cambiar
238 ;
239 ;    switch(nueva) {
240 ;
241 ;        case Interfaz::ESCENA_MENU:
242 ;
243 ;            actual = menu;
```

16.13. Implementación

```
244 ;         break;
245 ;
246 ;     case Interfaz::ESCENA_JUEGO:
247 ;
248 ;         actual = juego;
249 ;         break;
250 ;
251 ;     case Interfaz::ESCENA_EDITOR:
252 ;
253 ;         actual = editor;
254 ;         break;
255 ;     }
256 ;
257 ;     if(anterior == actual) {
258 ;         cout << "Universo: Cambia a la misma escena" << endl;
259 ;     }
260 ;
261 ;     // Una vez cambiada a la nueva escena, la reiniciamos
262 ;
263 ;     actual->reiniciar();
264 ;}
265 ;
266 ;
267 ;
268 ;void Universo::dibujar_rect(int x, int y, int w, int h, Uint32 color) {
269 ;
270 ;    SDL_Rect rect;
271 ;
272 ;    // Almacenamos la variable en un rectángulo
273 ;
274 ;    rect.x = x;
275 ;    rect.y = y;
276 ;    rect.h = h;
277 ;    rect.w = w;
278 ;
279 ;    // Dibujamos el rectángulo
280 ;
281 ;    SDL_FillRect(pantalla, &rect, color);
282 ;}
283 ;
284 ;
285 ;// Queremos salir de la aplicación
286 ;
287 ;void Universo::terminar(void) {
288 ;
289 ;    salir = true; // Afecta al polling
290 ;}
291 ;
292 ;// Destructor
293 ;
294 ;Universo::~Universo() {
295 ;
296 ;    delete juego;
```

16. Un ejemplo del desarrollo software de un videojuego

```
297 ;     delete galeria;
298 ;     delete editor;
299 ;     delete menu;
300 ;
301 ;#ifdef DEBUG
302 ;     cout << "Universo::~Universo()" << endl;
303 ;     cout << "Gracias por jugar" << endl;
304 ;#endif
305 ;
306 ;}
```

El constructor de la clase inicializa todos los atributos de la misma creando las diferentes escenas y estableciendo la principal para comenzar con la aplicación. El método *bucle_principal()* es un bucle “infinito” que va actualizando la escena en la que se encuentre la aplicación haciendo una llamada a su método actualizar. Por cada vuelta del bucle se limpia la superficie principal para que sea repintada.

Para sincronizar los fps (cuadros por segundo) se utiliza una función que nos devuelve el número de veces que tenemos que actualizar la escena actual para establecer un margen máximo de fps.

Necesitamos una función que nos permita establecer las propiedades que vamos a utilizar para ejecutar nuestra aplicación. El método *iniciar_ventana()* establece el modo de video que vamos a utilizar en la aplicación e inicializa el subsistema de audio para que pueda ser utilizado.

Es necesario manejar ciertos eventos que nos permitan interrumpir la aplicación o cambiar, por ejemplo, el formato de la ventana. En *procesar_eventos()* manejamos aquellos eventos que sean especiales. Nos referimos a eventos especiales cuando hablamos de salir del juego o de cambiar el modo de juego a pantalla completa. La implementación de la gestión del teclado la hemos visto hace un par de clases.

El método *sincronizar_fps()* es uno de los más interesantes de esta clase. Toma dos marcas de tiempo y realiza una espera activa hasta que se agote el intervalo de tiempo que queremos utilizar para que pueda realizarse alguna acción más. Este método es el encargado de la temporización del juego marcando la barrera de 100 fps como la aceptable para ejecutar el videojuego con buena fluidez.

En el juego podremos cambiar de escenario en distintos momentos. Por ejemplo si nos encontramos en el menú principal podremos entrar en el editor de niveles o a jugar un partida. Si estamos jugando podremos salir al menú pulsando una tecla. De este cambio de escenario se encarga el método *cambiar_interfaz()* que hace un cambio de escena a la que recibe como

parámetro. Una vez realizado el cambio reinicia la escena para partir de su estado inicial.

Necesitamos dibujar rectángulos sobre una superficie que nos permita crear fondos y limpiar la pantalla principal por cada vuelta del bucle. El método que se encarga de esta tarea es *dibujar_rect()*. La implementación de este método no tiene ningún secreto, hace uso de la función *SDL_FillRect()* para llenar un cuadro en una superficie.

Los dos últimos métodos son triviales. El método *terminar()* nos permite indicar a la clase Universo que queremos acabar con la aplicación marcando la bandera *salir* a **true** mientras que el destructor de la clase acaba con los recursos acaparados por dicha clase en el constructor.

16.13.22. La clase Ventana

La clase ventana proporciona una funcionalidad fundamental dentro de la aplicación. Nos permite establecer una cámara en forma de ventana que va a ir siguiendo nuestro personaje durante el transcurso del juego y que nos permitirá movernos por todo el nivel cuando estemos usando el editor para crear o modificar niveles.

El diseño de la clase del que partimos para la implementación es el de la figura 16.55.

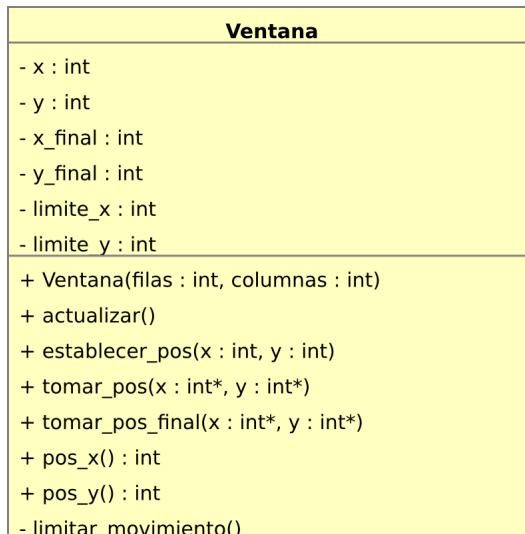


Figura 16.55: Clase Ventana

Vamos a estudiar la definición de la clase:

16. Un ejemplo del desarrollo software de un videojuego

```
;_____
1 ;// Listado: Ventana.h
2 ;//
3 ;// Esta clase nos permite navegar por las superficies de los niveles
4 ;
5 ;#ifndef _VENTANA_H_
6 ;#define _VENTANA_H_
7 ;
8 ;#include <SDL/SDL.h>
9 ;
10 ;class Ventana {
11 ;
12 ; public:
13 ;
14 ;     // Constructor
15 ;     Ventana(int filas, int columnas);
16 ;
17 ;     void actualizar(void);
18 ;
19 ;     void establecer_pos(int x, int y);
20 ;     void tomar_pos(int * x, int * y);
21 ;     void tomar_pos_final(int * x, int * y);
22 ;
23 ;     int pos_x(void);
24 ;     int pos_y(void);
25 ;
26 ; private:
27 ;
28 ;     int x, y;
29 ;     int x_final, y_final;
30 ;     int limite_x, limite_y;
31 ;
32 ;     void limitar_movimiento(void);
33 ;};
34 ;
35 ;#endif
;
```

En la parte privada de la clase tenemos varios atributos que nos permiten establecer un control sobre el movimiento de la ventana. (x, y) informan acerca de la posición actual de la ventana mientras que x_final e y_final hacen lo propio sobre la posición a la que queremos llegar con dicha ventana.

El método privado *limitar_movimiento()* nos permite que la ventana no se nos salga del tamaño del nivel para que no se nos muestren partes que no deben de ser cargadas en pantalla. Veamos la implementación de la clase.

```
;_____
1 ;// Listado: Ventana.cpp
2 ;//
3 ;// Implementación de la clase ventana
4 ;
```

```

5 ;#include <iostream>
6 ;
7 ; #include "Ventana.h"
8 ; #include "Nivel.h"
9 ;
10 ; using namespace std;
11 ;
12 ;
13 ; Ventana::Ventana(int filas, int columnas) {
14 ;
15 ;     // Inicializamos las variables
16 ;     x = y = 0;
17 ;     x_final = y_final = 0;
18 ;
19 ;     limite_x = ANCHO_VENTANA * 2 - columnas * TAMANO_BLOQUE;
20 ;     limite_y = ALTO_VENTANA * 2 - filas * TAMANO_BLOQUE;
21 ; }
22 ;
23 ;
24 ;
25 ; void Ventana::actualizar(void)
26 ; {
27 ;     int incremento_x = x_final - x;
28 ;     int incremento_y = y_final - y;
29 ;
30 ;     // Si existe variación
31 ;
32 ;     if(incremento_x != 0) {
33 ;
34 ;         // Controlamos el movimiento de la ventana
35 ;         if(abs(incremento_x) >= 10)
36 ;             x += incremento_x / 10; // Reducimos la cantidad de movimiento
37 ;
38 ;         else // Sobre todo en movimientos pequeños
39 ;             x += incremento_x / abs(incremento_x);
40 ;
41 ;     }
42 ;
43 ;     // Si existe variación
44 ;
45 ;     if(incremento_y != 0) {
46 ;
47 ;         // Animación de movimiento fluida
48 ;         if(abs(incremento_y) >= 10)
49 ;             y += incremento_y / 10;
50 ;         else
51 ;             y += incremento_y / abs(incremento_y);
52 ;     }
53 ; }
54 ;
55 ;
56 ;// Funciones referentes al posicionamiento
57 ;

```

16. Un ejemplo del desarrollo software de un videojuego

```
58 ; void Ventana::establecer_pos(int x, int y) {
59 ;
60 ;     x_final = x;
61 ;     y_final = y;
62 ;
63 ;     limitar_movimiento();
64 ; }
65 ;
66 ;
67 ; void Ventana::tomar_pos_final(int *x, int *y) {
68 ;
69 ;     *x = x_final;
70 ;     *y = y_final;
71 ;}
72 ;
73 ;
74 ;
75 ;void Ventana::tomar_pos(int *x, int *y) {
76 ;
77 ;     *x = this->x;
78 ;     *y = this->y;
79 ;}
80 ;
81 ;
82 ;int Ventana::pos_x(void) {
83 ;
84 ;     return x;
85 ;
86 ;}
87 ;
88 ;int Ventana::pos_y(void) {
89 ;
90 ;     return y;
91 ;
92 ;}
93 ;
94 ;
95 ;void Ventana::limitar_movimiento(void) {
96 ;
97 ;     // Comprobamos que se cumplen los límites lógicos de pantalla
98 ;
99 ;     if(x_final < 0)
100 ;         x_final = 0;
101 ;
102 ;     if(y_final < 0)
103 ;         y_final = 0;
104 ;
105 ;     if(x_final > limite_x)
106 ;         x_final = limite_x;
107 ;
108 ;     if(y_final > limite_y)
109 ;         y_final = limite_y;
110 ;}
```

16.14. Recopilando

El constructor de la clase establece la posición inicial de la ventana así como fija el límite de la misma en el eje vertical y horizontal. El método *actualizar()* realiza un movimiento progresivo hasta el destino final del a ventana. Si existe variación en alguno de los ejes y dependiendo de la distancia a la que se encuentre del objetivo realizará un movimiento más preciso o uno con mayor desplazamiento. Podemos ver como existe para cada uno de los ejes una comprobación sobre si la ventana está en la posición final y de no ser así, dependiendo del tamaño del incremento de posición que tenga que realizar avanzará más o menos.

El método *establecer_pos()* nos permite determinar la posición de destino hacia donde ha de moverse la ventana mientras que los métodos *tomar_pos()*, *pos_y()*, *pos_x()* nos permiten conocer la posición actual y final de la ventana.

El método *limitar_movimiento()* comprueba si la posición final de la ventana se sale del rango del tamaño del nivel para evitar movimientos no permitidos. La manera de realizar la comprobación es muy simple basta con comprobar que el destino final es mayor que la posición (0, 0) pero menor que el límite máximo de la ventana en los dos ejes.

16.14. Recopilando

Como puedes observar para realizar una pequeña aplicación hemos tenido que desarrollar un duro trabajo. Quizás este apartado ha sido demasiado formal pero era necesario que vieses que un proceso de desarrollo conlleva de la propia codificación.

En este capítulo hemos integrado todos los aspectos vistos durante el curso. Ahora te toca a tí crear tu aplicación

16. Un ejemplo del desarrollo software de un videojuego

Versión 1.0, terminada a día 30 de enero de 2008.

Copyright (C) 2007-2008 Antonio García Alba, Escuela Superior de Ingeniería (Universidad de Cádiz)

Este documento es libre. Se otorga permiso para copiarlo, distribuirlo y/o modificarlo bajo los términos de la licencia FDL (GNU Free Documentation License) versión 1.2 o posterior, publicada por la Fundación de Software Libre [1]. No contiene secciones invariantes, texto de portada ni de respaldo.

Puede encontrar la última versión de este documento en:

<http://www.uca.es/softwarelibre/wikiSDL>

[1] <http://www.gnu.org/licenses/fdl.html>