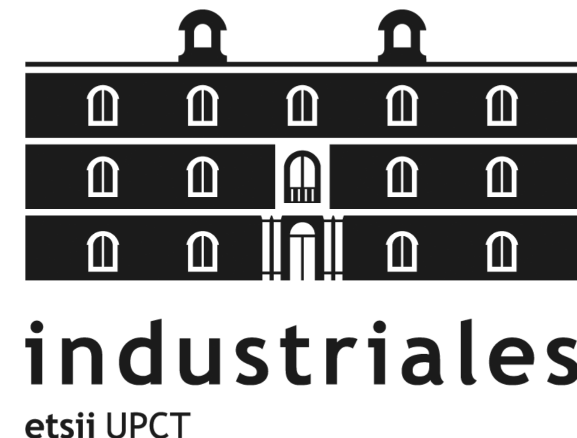


Estudio Técnico para la aplicación de primitivas SDL para el desarrollo de Interfaces Hombre-Máquina

Titulación: Ingeniería Industrial

Alumno/a: María Margallo Ros

Director/a/s: Joaquín Francisco Roca González

Cartagena, 1 de Marzo de 2016

Índice

Estudio Técnico para la aplicación de primitivas SDL para el desarrollo de Interfaces Hombre-Máquina 1

Índice de Figuras 1

1. Introducción 9
   - 1.1 Subsistemas y librerías adicionales 10
   - 1.2 Ventajas y desventajas 13
   - 1.3 Motivos para escoger SDL 13
2. Instalación 15
   - 2.1 Configuración de SDL 2.0 en Code::Blocks 16
3. Primeros pasos con SDL 20
   - 3.1 Diagrama árbol Funciones 20
   - 3.2 Primera Ventana Gráfica 24
   - 3.3 Mostrar una imagen en pantalla 27
   - 3.4 Programación Dirigida por Eventos (Event Driven Programming) 31
   - 3.5 Key Presss 34
   - 3.6 Alargar a ventana (Stretching to Window) 40
   - 3.7 Extensión de librerías y carga de otros formatos de imágenes 42
   - 3.8 Carga y Renderizado de Texturas 45
   - 3.9 Renderizado de Geometría 49
   - 3.10 Ventana de Visualización / Ventana Gráfica 53
   - 3.11 Color 55
   - 3.12 Clip Rendering - Sprite Sheets 62
   - 3.13 Modulación de Color 67
   - 3.14 Composición Alfa 76
   - 3.15 Animación de Sprites y Sincronización Vertical 87
   - 3.16 Rotación y Volteo 90
   - 3.17 Tipos de Fuentes 95
   - 3.18 Acciones con el Ratón. Mouse Events 100
   - 3.19 Key States 112
   - 3.20 Gamepads y Joysticks 116
   - 3.21 Música y Efectos de Sonido 121
   - 3.22 Control del Tiempo (Timing) 127
   - 3.23 Temporizadores Avanzados 131

3.24 Cálculo de Fotogramas Por Segundo ... 137
3.25 Limitación de la Velocidad de Fotogramas Por Segundo ... 139
3.26 Movimiento ... 142
3.27 Detección de la colisión ... 149
3.28 Detección de la colisión por pixel ... 158
3.29 Detección de colisión circular ... 166
3.30 Scrolling ... 175
3.31 Scrolling Backgrounds (Fondos de desplazamiento) ... 180
3.32 Archivos de lectura y escritura ... 185
3.33 Window Events ... 191
3.34 Ventanas mú4ltiples ... 198
3.35 Pantallas múltiples ... 206
3.36 Partículas en movimiento ... 213
3.37 Tiling (Embaldosado) ... 221
3.38 Manipulación de texturas ... 235
3.39 Fuentes de mapas de bits ... 242
3.40 Texture Streaming ... 254
3.41 Renderizar a una textura ... 258
3.42 Movimiento independiente de la velocidad de los fotogramas ... 264
3.43 Retrollamadas temporizadas ... 268
3.44 Multithreading ... 271
3.45 Semáforos ... 273
3.46 Atomic Operations ... 278
3.47 Exclusiones mutuas y Condiciones ... 281
3.48 SDL y OpenGL 2 ... 290
4. Estudio de la Comunicación UDP con Code::Blocks GCC ... 296
4.1 SIGNAL SENDER ... 297
4.2 SIGNAL RECEIVER ... 312
5. SDL y Raspberry PI ... 332
5.1 Raspberry PI ... 332
5.2 Configuración SDL2.0 para Linux ... 335
5.3 Ejemplos SDL para Linux ... 337
5.3.1 Vibración Joystick ... 337
5.3.2 SDL y Modern OpenGL ... 340

6. Alternativas a SDL 347
7. Conclusiones y Líneas futuras de trabajo 349
8. Bibliografía..350

Índice de Figuras

Figura 1: Logotipo librería SDL

Figura 2: Susbistemas SDL

Figura 3: Juego portado mediante SDL: Descent 2

Figura 4: Logos Sistemas Operativos

Figura 5: Logo Compiladores

Figura 6: Inicio de un Nuevo Proyecto

Figura 7: Propiedades de un Nuevo Proyecto

Figura 8: Build Options

Figura 9: Adición Directorio del Compilador

Figura 10: Search Directories: Linker

Figura 11: Search Directories: Other Linker Options

Figura 12: Build Targets. Selección Console Application

Figura 13: Resultado ejecutar aplicación Primera Ventana Gráfica una vez transcurridos 2 segundos

Figura 14: Resultado ejecutar aplicación Mostrar una imagen en pantalla

Figura 15: Cola de Eventos

Figura 16: Cola de eventos. SDL_PollEvent

Figura 17: Resultado ejecutar aplicación Programación Dirigida por Eventos

Figura 18: Resultado ejecutar aplicación Programación Dirigida por Eventos (II)

Figura 19: Resultado ejecutar aplicación Key Presses

Figura 20: Resultado ejecutar aplicación tras pulsar cursor arriba

Figura 21: Resultado ejecutar aplicación tras pulsar cursor abajo

Figura 22: Resultado ejecutar aplicación tras pulsar cursor izquierda

Figura 23: Resultado ejecutar aplicación tras pulsar cursor derecha

Figura 24: Resultado ejecutar aplicación Stretching to window

Figura 25: Resultado ejecutar aplicación Extensión de librerías y carga de otros formatos de imágenes

Figura 26: Resultado ejecutar aplicación Extensión de librerías y carga de otros formatos de imágenes (II)

Figura 27: Resultado ejecutar aplicación Carga y Renderizado de Texturas

Figura 28: Sistema de Coordenadas Natural

Figura 29: Sistema de Coordenadas de SDL

Figura 30: Origen Sistema de Coordenadas SDL

Figura 31: Resultado ejecutar aplicación Renderizado de geometrías

Figura 32: Secciones Ventana Gráfica

Figura 33: Resultado ejecutar aplicación Ventana Gráfica

Figura 34: Imagen parte anterior

Figura 35: fondo

Figura 36: Resultado ejecutar aplicación Color

Figura 37: Resultado ejecutar aplicación Color (II)

Figura 38: Sprite sheet

Figura 39: Sprite sheet (II)

Figura 40: Resultado ejecutar aplicación Clip Rendering y Sprite Sheets

Figura 41: Resultado ejecutar aplicación Clip Rendering y Sprite Sheets (II)

Figura 42: Modulación de Color (I)

Figura 43: Modulación de Color (II)

Figura 44: Resultado ejecutar aplicación Modulación de Color

Figura 45: Resultado ejecutar aplicación Modulación de Color (II)

Figura 46: Resultado ejecutar aplicación Modulación de Color (III)

Figura 47: Resultado ejecutar aplicación Modulación de Color (IV)

Figura 48: Resultado ejecutar aplicación Modulación de Color (V)

Figura 49: Resultado ejecutar aplicación Modulación de Color (VI)

Figura 50: Resultado ejecutar aplicación Modulación de Color (VII)

Figura 51: Resultado ejecutar aplicación Modulación de Color (VIII)

Figura 52: Resultado ejecutar aplicación Modulación de Color (IX)

Figura 53: Resultado ejecutar aplicación Modulación de Color (X)

Figura 54: Resultado ejecutar aplicación Composición Alfa

Figura 55: Resultado ejecutar aplicación Composición Alfa (II)

Figura 56: Resultado ejecutar aplicación Composición Alfa (III)

Figura 57: Resultado ejecutar aplicación Composición Alfa (IV)

Figura 58: Resultado ejecutar aplicación Composición Alfa (V)

Figura 59: Resultado ejecutar aplicación Composición Alfa (VI)

Figura 60: Resultado ejecutar aplicación Composición Alfa (VII)

Figura 61: Resultado ejecutar aplicación Composición Alfa (VIII)

Figura 62: Resultado ejecutar aplicación Composición Alfa (IX)

Figura 63: Resultado ejecutar aplicación Composición Alfa (X)

Figura 64: Resultado ejecutar aplicación Composición Alfa (XI)

Figura 65: Resultado ejecutar aplicación Composición Alfa (XII)

Figura 66: Resultado ejecutar aplicación Composición Alfa (XIII)

Figura 67: Resultado ejecutar aplicación Composición Alfa (XIV)

Figura 68: Sprite para crear animación

Figura 69: Fotograma de animación

Figura 70: Rotación

Figura 71: Volteo

Figura 72: Resultado ejecutar aplicación Tipos de Fuentes

Figura 73: Resultado ejecutar aplicación Acciones con el Ratón

Figura 74: Resultado ejecutar aplicación Acciones con el Ratón (II)

Figura 75: Resultado ejecutar aplicación Acciones con el Ratón (III)

Figura 76: Resultado ejecutar aplicación Acciones con el Ratón (IV)

Figura 77: Resultado ejecutar aplicación Acciones con el Ratón (V)

Figura 78: Resultado ejecutar aplicación Acciones con el Ratón (VI)

Figura 79: Resultado ejecutar aplicación Acciones con el Ratón (VII)

Figura 80: Resultado ejecutar aplicación Acciones con el Ratón (VIII)

Figura 81: Resultado ejecutar aplicación Acciones con el Ratón (IX)

Figura 82: Resultado ejecutar aplicación Acciones con el Ratón (X)

Figura 83: Resultado ejecutar aplicación Acciones con el Ratón (XI)

Figura 84: Resultado ejecutar aplicación Acciones con el Ratón (XII)

Figura 85: Resultado ejecutar aplicación Key States

Figura 86: Resultado ejecutar aplicación Key States (II)

Figura 87: Resultado ejecutar aplicación Key States (III)

Figura 88: Resultado ejecutar aplicación Key States (IV)

Figura 89: Resultado ejecutar aplicación Key States (V)

Figura 90: Resultado ejecutar aplicación Key States (VI)

Figura 91: Resultado ejecutar aplicación Gamepads y Joysticks

Figura 92: Resultado ejecutar aplicación Música y Efectos de Sonido

Figura 93: Resultado ejecutar aplicación Control del Tiempo

Figura 94: Resultado ejecutar aplicación Control del Tiempo (II)

Figura 95: Resultado ejecutar aplicación Control del Tiempo (III)

Figura 96: Temporizadores Avanzados

Figura 97: Temporizadores Avanzados (II)

Figura 98: Resultado ejecutar aplicación Cálculo de Fotogramas Por Segundo

Figura 99: Resultado ejecutar aplicación Cálculo de Fotogramas Por Segundo (II)

Figura 100: Resultado ejecutar aplicación Cálculo de Fotogramas Por Segundo (III)

Figura 101: Resultado ejecutar aplicación Limitación de la Velocidad de Fotogramas Por Segundo

Figura 102: Resultado ejecutar aplicación Limitación de la Velocidad de Fotogramas Por Segundo (II)

Figura 103: Resultado ejecutar aplicación Limitación de la Velocidad de Fotogramas Por Segundo (III)

Figura 104: Resultado ejecutar aplicación Movimiento

Figura 105: Resultado ejecutar aplicación Movimiento (II)

Figura 106: Resultado ejecutar aplicación Movimiento (III)

Figura 107: Resultado ejecutar aplicación Movimiento (IV)

Figura 108: Resultado ejecutar aplicación Movimiento (V)

Figura 109: Resultado ejecutar aplicación Movimiento (VI)

Figura 110: Resultado ejecutar aplicación Movimiento (VII)

Figura 111: Resultado ejecutar aplicación Movimiento (VIII)

Figura 112: Resultado ejecutar aplicación Movimiento (IX)

Figura 113: Resultado ejecutar aplicación Movimiento (X)

Figura 114: Resultado ejecutar aplicación Movimiento (XI)

Figura 115: Resultado ejecutar aplicación Movimiento (XII)

Figura 116: Resultado ejecutar aplicación Movimiento (XII)

Figura 117: Ejemplo de no colisión entre dos polígonos

Figura 118: Ejemplo de colisión entre dos polígonos sólo en el eje Y

Figura 119: Ejemplo de colisión entre dos polígonos sólo en el eje X

Figura 120: Ejemplo de colisión entre dos polígonos en el eje X e Y

Figura 121: Resultado ejecutar aplicación Detección de la Colisión

Figura 122: Resultado ejecutar aplicación Detección de la Colisión (II)

Figura 123: Resultado ejecutar aplicación Detección de la Colisión (III)

Figura 124: Resultado ejecutar aplicación Detección de la Colisión (IV)

Figura 125: Resultado ejecutar aplicación Detección de la Colisión (V)

Figura 126: Resultado ejecutar aplicación Detección de la Colisión (VI)

Figura 127: Punto

Figura 128: Ejemplo puntos formados por rectángulos

Figura 129: Comprobación colisión por pixel

Figura 130: Resultado ejecutar aplicación Detección de la colisión por pixel

Figura 131: Resultado ejecutar aplicación Detección de la colisión por pixel (II)

Figura 132: Resultado ejecutar aplicación Detección de la colisión por pixel (III)

Figura 133: Resultado ejecutar aplicación Detección de la colisión por pixel (IV)

Figura 134: Ejemplo de colisión entre círculo y rectángulo

Figura 135: Ejemplo de colisión entre círculo y rectángulo (II)

Figura 136: Ejemplo de colisión entre círculo y rectángulo (III)

Figura 137: Resultado ejecutar aplicación Detección de colisión circular

Figura 138: Resultado ejecutar aplicación Detección de colisión circular (II)

Figura 139: Resultado ejecutar aplicación Detección de colisión circular (III)

Figura 140: Resultado ejecutar aplicación Detección de colisión circular (IV)

Figura 141: Scrolling

Figura 142: Scrolling (II)

Figura 143: Scrolling (III)

Figura 144: Scrolling (IV)

Figura 145: Scrolling (V)

Figura 146: Scrolling Backgrounds

Figura 147: Scrolling Backgrounds (II)

Figura 148: Scrolling Backgrounds (III)

Figura 149: Scrolling Backgrounds (IV)

Figura 150: Scrolling Backgrounds (V)

Figura 151: Scrolling Backgrounds (VI)

Figura 152: Scrolling Backgrounds (VII)

Figura 153: Resultado ejecutar aplicación Archivos de lectura y escritura

Figura 154: Resultado ejecutar aplicación Window Events

Figura 155: Resultado ejecutar aplicación Window Events (II)

Figura 156: Resultado ejecutar aplicación Ventanas múltiples

Figura 157: Resultado ejecutar aplicación Pantallas múltiples

Figura 158: Resultado ejecutar aplicación Partículas en movimiento

Figura 159: Resultado ejecutar aplicación Partículas en movimiento (II)

Figura 160: Resultado ejecutar aplicación Partículas en movimiento (III)

Figura 161: Resultado ejecutar aplicación Partículas en movimiento (IV)

Figura 162: Resultado ejecutar aplicación Partículas en movimiento (V)

Figura 163: Tiling

Figura 164: Tiling (II)

Figura 165: Tiling (III)

Figura 166: Resultado ejecutar aplicación Tiling

Figura 167: Resultado ejecutar aplicación Tiling (II)

Figura 168: Resultado ejecutar aplicación Tiling (III)

Figura 169: Resultado ejecutar aplicación Tiling (IV)

Figura 170: Resultado ejecutar aplicación Tiling (V)

Figura 171: Imagen previa aplicación Manipulación de texturas

Figura 172: Resultado ejecutar aplicación Manipulación de texturas

Figura 173: Caracteres previos

Figura 174: Palabra formada a partir de los caracteres previos

Figura 175: Imagen de textura bidimensional

Figura 176: Imagen de textura unidimensional

Figura 177: Detección de pixel de distinto color al fondo

Figura 178: Resultado ejecutar aplicación Fuentes de mapas de bits

Figura 179: Resultado ejecutar aplicación Texture Streaming

Figura 180: Resultado ejecutar aplicación Renderizar a textura

Figura 181: Resultado ejecutar aplicación Renderizar a textura (II)

Figura 182: Resultado ejecutar aplicación Renderizar a textura (III)

Figura 183: Resultado ejecutar aplicación Renderizar a textura (IV)

Figura 184: Resultado ejecutar aplicación Renderizar a textura (V)

Figura 184: Resultado ejecutar aplicación Movimiento independiente de la velocidad de los fotogramas

Figura 185: Resultado ejecutar aplicación Movimiento independiente de la velocidad de los fotogramas (II)

Figura 186: Resultado ejecutar aplicación Retrollamadas temporizadas

Figura 187: Resultado ejecutar aplicación Retrollamadas temporizadas (II)

Figura 188: Ejecución Multihilo

Figura 189: Resultado ejecutar aplicación Semáforos

Figura 190: Resultado ejecutar aplicación Semáforos (II)

Figura 191: Resultado ejecutar Atomic operations

Figura 192: Resultado ejecutar Atomic operations (II)

Figura 193: Ejemplo exclusiones mutuas y condiciones

Figura 194: Ejemplo exclusiones mutuas y condiciones (II)

Figura 195: Ejemplo exclusiones mutuas y condiciones (III)

Figura 196: Ejemplo exclusiones mutuas y condiciones (IV)

Figura 197: Ejemplo exclusiones mutuas y condiciones (V)

Figura 198: Ejemplo exclusiones mutuas y condiciones (VI)

Figura 199: Ejemplo exclusiones mutuas y condiciones (VII)

Figura 200: Ejemplo exclusiones mutuas y condiciones (VIII)

Figura 201: Resultado ejecutar aplicación exclusiones mutuas y condiciones

Figura 202: Sistema de coordenadas OpenGL

Figura 203: Resultado tras actualizar la pantalla

Figura 204: Esquema visual programa

Figura 205: Aspecto visual

Figura 206: Ejecución SignalSender

Figura 207: Ejecución SignalSender (II)

Figura 208: Ejecución SignalSender (III)

Figura 209: Ejecución SignalSender (IV)

Figura 210: Ejecución SignalReceiver

Figura 211: Ejecución SignalReceiver (II)

Figura 212: Ejecución SignalReceiver (III)

Figura 213: Resultado mostrar imagen por pantalla

Figura 214: Raspberry Pi 2

Figura 215: Raspberry Pi 2 empleada

Figura 216: Encendido Raspberry Pi 2

Figura 217: Encendido Raspberry Pi 2 (II)

Figura 218: Encendido Raspberry Pi 2 (III)

Figura 219: Configuración SDL Linux

Figura 220: Configuración SDL Linux (II)

Figura 221: Configuración SDL Linux (III)

Figura 222: Configuración SDL Linux (IV)

Figura 223: Vibración joystick

Figura 224: Shader program

Figura 225: Logo DirectX y Logo OpenGL

Figura 226: Logo pygame

Figura 227: Logo Box2D

1. Introducción

En 1982, tras la aparición de MS-DOS, programar videojuegos era una tarea ardua y bastante compleja que requería una gran cantidad de conocimientos del lenguaje ensamblador, así como de la estructura interna de la máquina, las tarjetas de video y audio. Al realizarse toda la programación sobre el hardware, cada juego se preparaba para los diferentes modelos de tarjetas para que funcionase en cada ordenador.

La programación de videojuegos no mejoró tampoco con la llegada de Windows al no permitir un acceso directo al hardware, además que su rendimiento gráfico no era suficiente. Puesto que el acceso a los recursos del sistema varía de un sistema operativo a otro, es imprescindible la creación de una interfaz común con el fin de estandarizar el acceso a los recursos del sistema. Entre diferentes apuestas aparecen OpenGL y DirectX.

OpenGL, orientado a gráficos en 3D, ofrece una interfaz multiplataforma, aunque también con posibilidades de 2D. No obstante, no permite el acceso al teclado, ratón, sonido…

Por su parte, DirectX está formado por un conjunto de APIs creadas para facilitar las tareas multimedia, como Direct3D, DirectSound o DirectPlay. Presenta como desventaja su uso complicado, ya que sólo utiliza las APIs de Windows.

Es en ese instante cuando, en torno al año 1984, Sam Lantinga, programador de la empresa Loki Games, crea una librería multiplataforma llamada SDL (acrónimo de Simple DirectMedia Layer) con el fin de portabilizar los juegos de Windows a otros Sistemas Operativos.

Figura 1: *Logotipo librería SDL*

Dicha librería, SDL, ofrece acceso al hardware del ordenador mediante una interfaz independiente del Sistema Operativo que, además, permite emplear la plataforma OpenGL para gráficos 3D.

Desarrollada en el lenguaje de programación C, SDL está diseñada para proporcionar un acceso a bajo nivel para realizar operaciones de dibujo en dos dimensiones, gestión de efectos de sonido y música, así como carga y gestión de imágenes. Además, proporciona herramientas para el desarrollo de videojuegos y aplicaciones multimedia.

Al tratarse de una biblioteca multiplataforma, una de sus grandes virtudes es su compatibilidad oficial con los sistemas Microsoft Windows, GNU/Linux, Mac OS X, Linux y Android, además de otras arquitecturas y sistemas como Sega Dreamcast, GP32, GP2X, etc. Además, con ella se asegura que toda aplicación realizada con esta librería podrá compilarse n distintas plataformas sin cambiar ninguna línea de código.

Pese a estar escrito en C y funcionar de forma nativa con C ++, existen enlaces disponibles para varios lenguajes, incluyendo C # y Python. La nueva versión SDL 2.0 está disponible en C, C++, C#, Genie, Pascal, Python y Vala.

Se distribuye bajo la licencia LGPL, que es la que ha provocado el gran avance y evolución de SDL, aunque a partir de la versión 2.0 esta librería se encuentra bajo la Licencia ZLib. Dicha licencia permite utilizar SDL libremente en cualquier software.

1.1 Subsistemas y librerías adicionales

La librería SDL está compuesta por diferentes subsistemas. Sostenidos por diferentes APIs, proporcionan acceso a las diferentes partes hardware. Algunos de sus aspectos más destacables son:

- **Vídeo y Gráficos**

Mediante la API proporcionada por SDL, es posible obrar de un modo muy versátil con los gráficos. Así, se puede trabajar con píxeles en crudo, directamente, construyendo líneas y polígonos y o bien cargar imágenes que rellenen píxeles facilitando así el trabajo de crear figuras o personajes.

También es posible la creación de superficies con canales alpha y colores clavo, así como el volcado de superficies convirtiendo los formatos al de destino de manera automática.

Además, el modo de video, de acceso a la memoria de video es configurado por SDL proporcionando modos en ventana o pantalla completa.

- **Eventos**

La entrada por teclado, ratón y joystick es proporcionada por SDL mediante el empleo de un modelo de entrada basado en eventos similar al utilizado en el desarrollo de aplicaciones X11, Windows o Mac OS.

Existe una ventaja y es que SDL proporciona una abstracción de éstos, es decir, se pueden usar sin la preocupación del sistema operativo en el que se esté desarrollando la aplicación. De este modo, el usuario puede olvidarse 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.

Además, SDL también da la posibilidad de conocer el estado de un dispositivo de entrada en un momento dado sin que éste tenga que haber realizado ninguna acción, es decir, es posible consultar el estado de una determinada tecla o de cierto botón del ratón cuando se considere conveniente.

- **Sonido**

SDL proporciona una sencilla API para explorar las capacidades proporcionadas por la tarjeta de sonido y lo que se necesita saber pera reproducir un sonido. El subsistema de sonido puede iniciarse a 8 o 16 bits, mono o estéreo.

Dado que el soporte para reproducir un sonido real es mínimo, para reproducir sonidos hay que servirse de librerías auxiliares que proporcionen una manera más cómoda de trabajar (si no se quiere que este aspecto ocupe la mayor parte del tiempo de desarrollo de la aplicación).

- **Empleo del CD-ROM**

SDL proporciona una API completa con la que leer y reproducir pistas en un CD-ROM.

- **Timers**

En función de la máquina en la que se ejecute la aplicación, ésta irá más rápida o más lenta.

Cuando se trata de aplicaciones de gestión, no es un aspecto que deba controlarse explícitamente, ya que, en los casos más comunes, cuanto más rápido se ejecute, mejor respuesta obtiene el usuario. Sin embargo, cuando se programan videojuegos el tiempo es un aspecto crítico que debe ser controlado en todo momento.

Los Timers son un tipo de variable que permiten conocer una marca de tiempo dentro de una aplicación y, por tanto, sirven de ayuda a la hora de realizar la tarea de control. Con ellos se verifican los tiempos en los videojuegos para, de esta manera, poder independizar el comportamiento de la aplicación de la máquina en la que se esté ejecutando. Los timers de SDL trabajan a una resolución de 10 milisegundos, resolución más que suficiente para controlar los tiempos de un videojuego.

SDL provee una API sencilla, limpia y confiable independiente de la máquina y del sistema operativo.

- **Gestión de dispositivos**

Habitualmente los sistemas operativos requieren que el programador configure determinados aspectos sobre los dispositivos que se van a emplear e inicializar dichos dispositivos antes de su uso.

En primer lugar, se debe consultar al sistema sobre qué puede hacer con el hardware, en qué limita al sistema operativo y qué permite el hardware en cuestión. Una vez utilizado el dispositivo, hay que comunicarle al sistema operativo que se ha terminado de usar para dejarlo libre en caso de que otra aplicación quiera utilizarlo.

SDL ofrece una manera simple de descubrir qué puede hacer el hardware y a continuación, mediante un par de líneas de código, se puede dejar todo listo para utilizar el dispositivo. Las funciones que proporciona SDL para liberar o cerrar el dispositivo son aún más simples.

Como conclusión, SDL facilita la tarea de tomar un dispositivo para poder utilizarlo a la vez que simplifica la tarea de liberarlo.

- **Red**

SDL proporciona una API no nativa para trabajar con redes a bajo nivel. Esta librería adicional permite enviar paquetes de distinto tipo sobre el protocolo IP, además de 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. Presenta la ventaja de que SDL se ocupa de muchas tareas, consideradas molestas, de bajo nivel que hacen que preparar y manejar las conexiones sea tedioso.

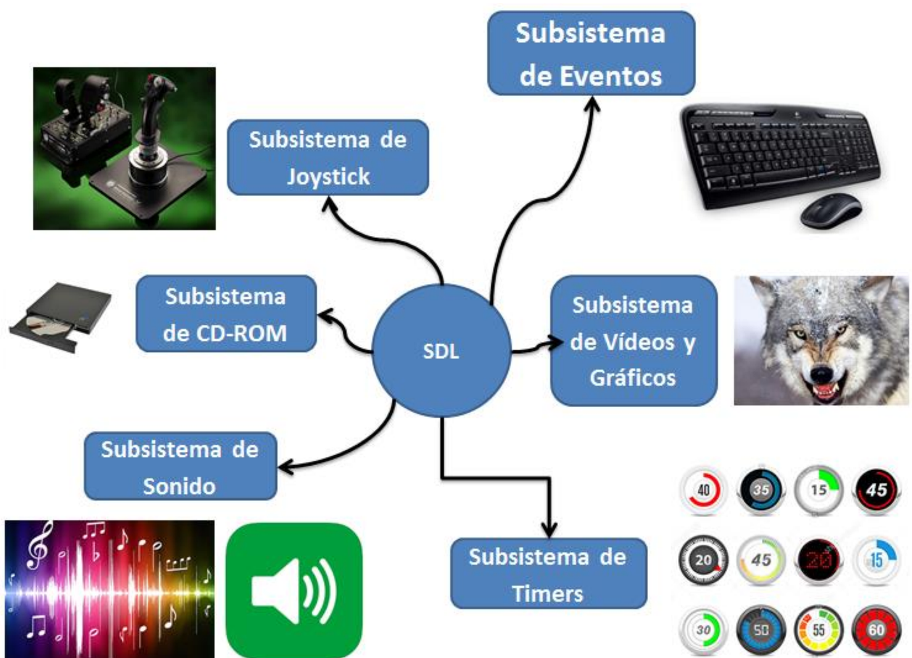


Figura 2: *Susbistemas SDL*

Con el fin de complementar la funcionalidad y capacidad de la biblioteca base, se ha desarrollado un conjunto de bibliotecas adicionales. Entre ellas:

- **SDL Mixer:** Extiende las capacidades de SDL para la gestión y uso de sonido y música en aplicaciones y juegos. Es compatible con formatos de sonido como Wave, MP3 y OGG, y formatos de música como MOD, S3M, IT y XM.

- **SDL Image**: Extiende las capacidades de SDL para trabajar con diferentes formatos de imagen. Los formatos compatibles son los siguientes: BMP, JPEG, TIFF, PNG, PNM, PCX, XPM, LBM, GIF, y TGA.

- **SDL Net:** Proporciona funciones y tipos de datos multiplataforma para programar aplicaciones que trabajen con redes.

- **SDL RTF:** Posibilita el abrir para leer en aplicaciones SDL archivos de texto usando el formato Rich Text Format RTF.

- **SDL TTF:** Permite usar tipografías TrueType en aplicaciones SDL.

1.2 Ventajas y desventajas

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. Así, 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 se quiere hacer y el resultado final.

- **Flexibilidad**: La librería es multiplataforma. Las aplicaciones creadas se ejecutan 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 se puede consultar cualquier aspecto de la implementación por si puede aportar alguna idea o se quiere mejorar algo. Al no haber nada bloqueado, se puede ver cómo está implementada cada una de las partes de SDL.

En resumen, SDL es una librería portable, intuitiva y estable como una roca (así la define su creador) en la que las restricciones son mínimas. Su potencial ha conseguido que hoy en día puedan ejecutarse juegos como *Civilization* o *Descent 2* en máquinas Linux.

Sin embargo, también presenta desventajas:

- Uno de los principales problemas de esta librería, es la escasa documentación existente. Es necesaria cierta habilidad para leer cientos de líneas de código en programas de ejemplo con el fin de encontrar aquello que nos hace falta. Actualmente existen proyectos de ampliación y traducción de la documentación.
- Esta librería tampoco permite realizar juegos en tres dimensiones, aunque esta afirmación no es del todo cierta ya que se complementa con OpenGL para crear este tipo de videojuegos.

1.3 Motivos para escoger SDL

Uno de principales motivos para escoger esta librería es su licencia, que permite desarrollar aplicaciones sin tener que estar pendientes de los aspectos legales que envuelven a la biblioteca. Otro aspecto a tener en cuenta es que se trata de una librería libre en la que es posible conocer cómo se implementan los distintos aspectos de la misma.

Además, al ser libre, los usuarios de la misma pueden informar de los fallos y proponer soluciones a estos fallos. De esta manera, favorece un crecimiento del conocimiento del programador.

El hecho de 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.

Como se ha comentado anteriormente, la librería SDL es multiplataforma. El fabricante garantiza que las aplicaciones desarrolladas con ella funcionarán perfectamente bajo Linux, Microsoft Windows (C). Para que esto sea cierto se han de 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 se esté implementando dicha aplicación. De nada vale que SDL sea multiplataforma si a la hora de implementar la aplicación se utilizan técnicas, funciones o constantes que dependen del sistema.

Figura 3: *Juego portado mediante SDL: Descent 2*

En función del sistema operativo que se ejecute, SDL se sustentará sobre una parte del mismo u otro. En Microsoft Windows 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.

Una cualidad importante de SDL es su, relativamente, poca complejidad. Proporciona las herramientas básicas sobre todo lo que tiene que hacer la librería para funcionar en cualquier máquina para programar un videojuego.

2. Instalación

Para poder usar gráficos, sonidos, teclados, joysticks… se necesita una API que incluya todas estas características de hardware y las convierta en elementos con los que C++ pueda interactuar. Ésta es la función de SDL: toma las herramientas Linux/ Mac / Android/ iOS… y las envuelve de forma que pueda programarse en SDL y compilarse en cualquier plataforma que soporta.

Para su instalación:

- En primer lugar, se escoge el sistema operativo:

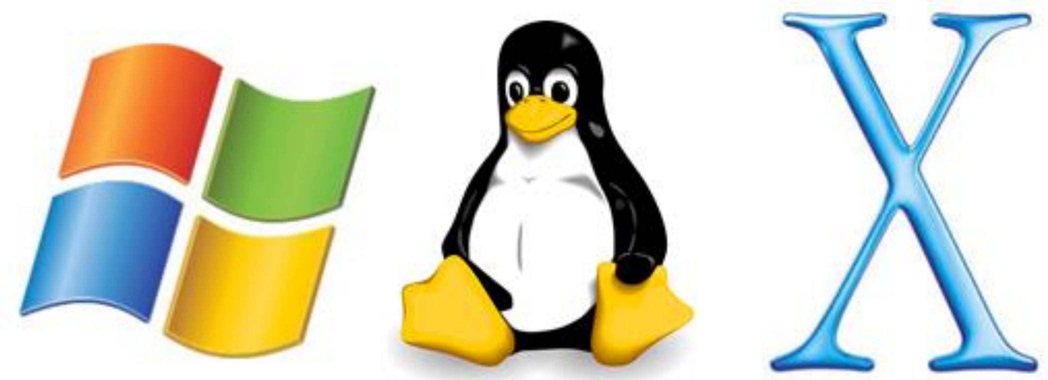


Figura 4: *Logos Sistemas Operativos*

En este caso se elegirá Windows.

- En segundo lugar, se selecciona IDE/Compilador:

Figura 5: *Logo Compiladores*

En este caso se trabajará con Code::Blocks (13.12)

2.1 Configuración de SDL 2.0 en Code::Blocks

- En primer lugar, una vez descargadas las cabeceras, librerías, así como archivos binarios de SDL, se inicia Code::Blocks y se crea un nuevo proyecto vacío:

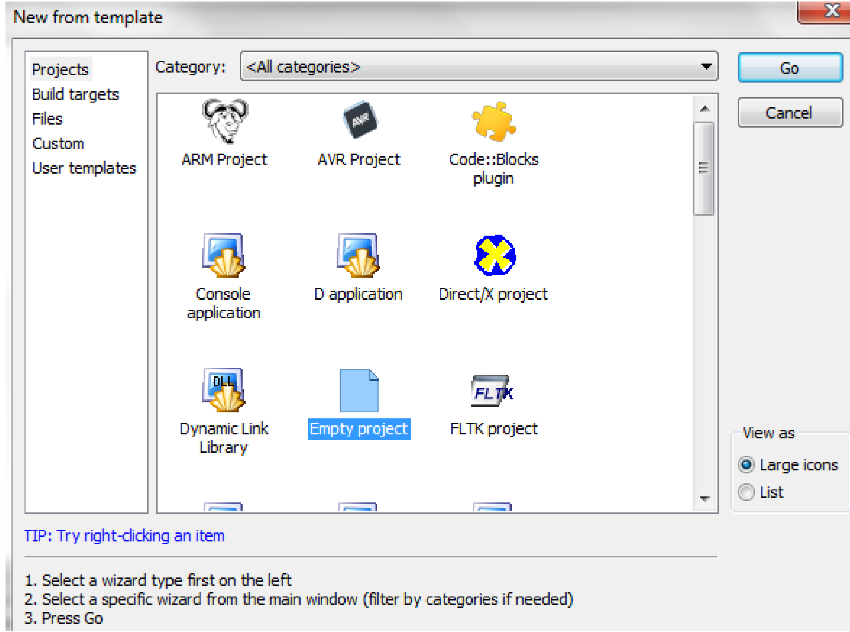


Figura 6: *Inicio de un Nuevo Proyecto*

- En segundo lugar, se accede a sus propiedades:

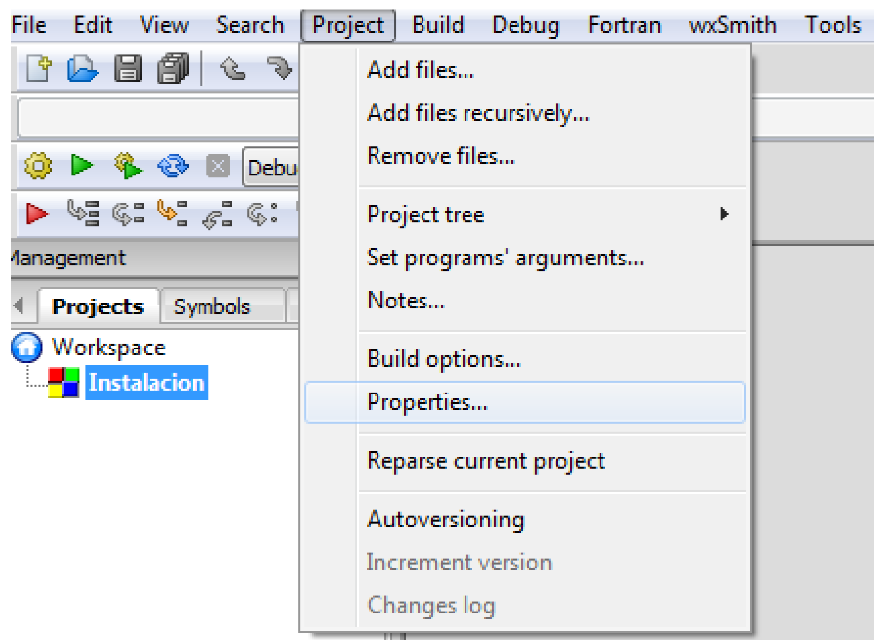


Figura 7: *Propiedades de un Nuevo Proyecto*

- En tercer lugar, hay que indicarle a Code::Blocks dónde buscar los archivos. Para ello, hacer click en Build Options:

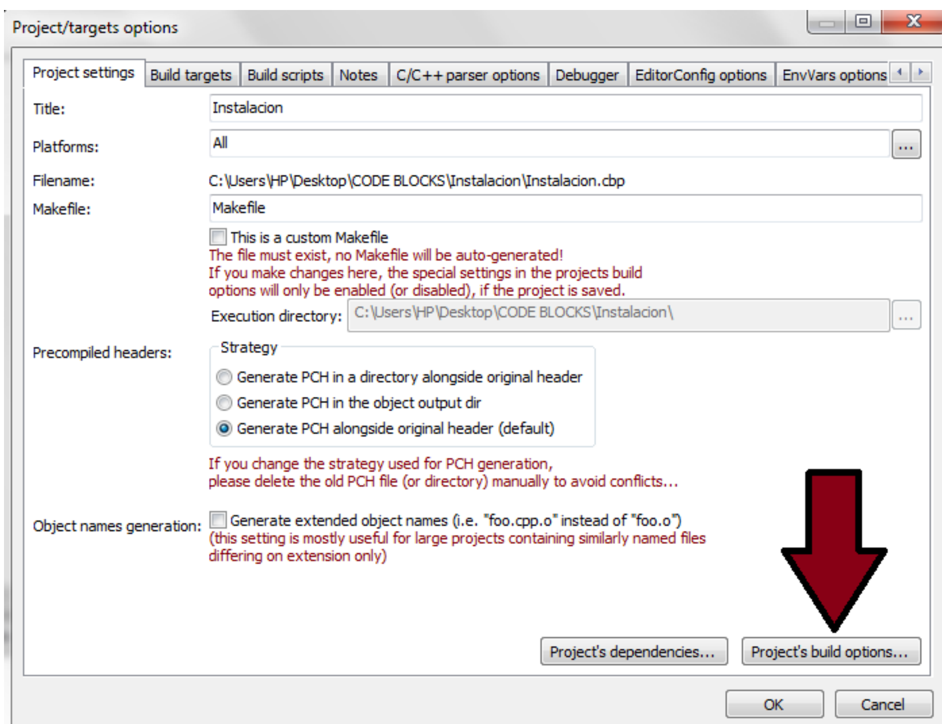


Figura 8: *Build Options*

En Search Directories, se añade un nuevo directorio del compilador. Haciendo click en add, se selecciona el directorio include de SDL2. (Cuando se pregunte si se quiere que sea una ruta alternativa, decir no).

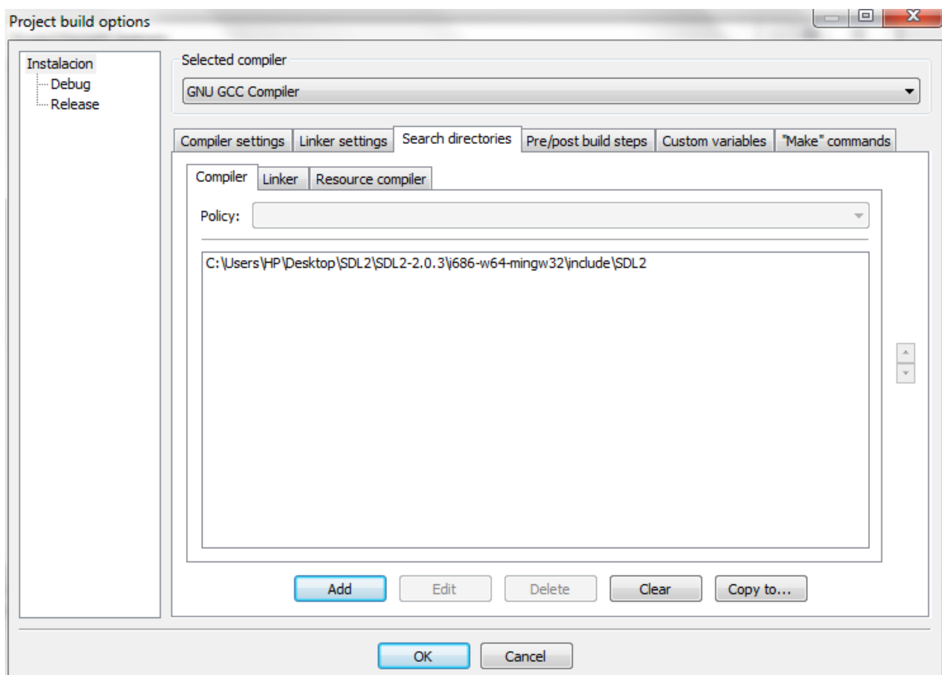


Figura 9: *Adición Directorio del Compilador*

- En cuarto lugar, se indica a Code::Block dónde buscar los archivos de librería. Para ello:

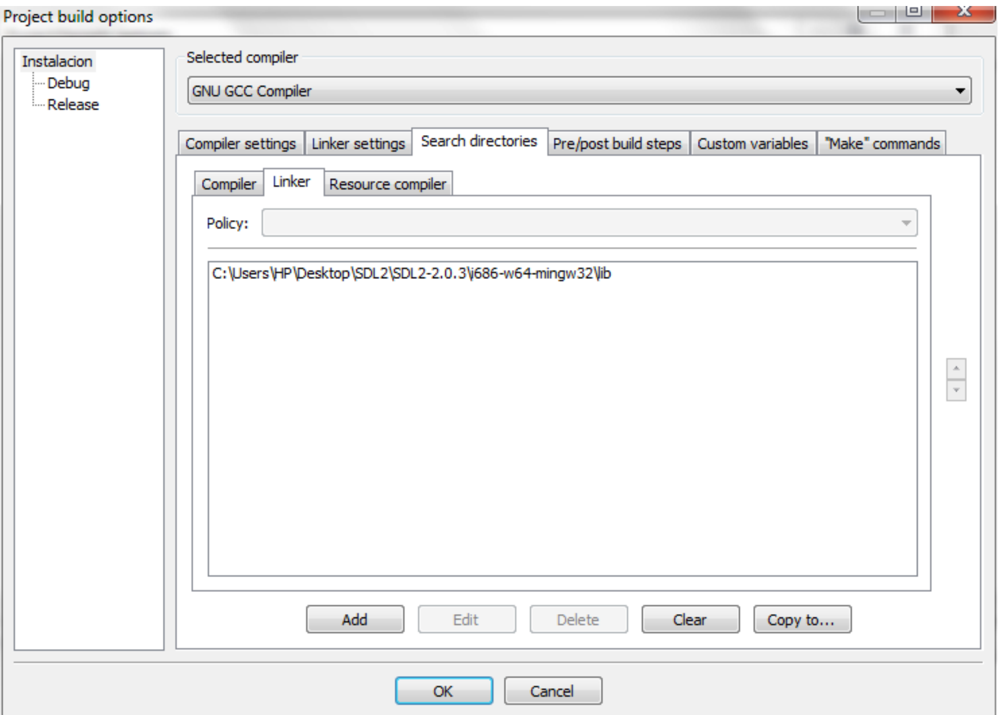


***Figura 10**: Search Directories: Linker*

- En quinto lugar, para poder compilar códigos de SDL 2, hay que indicar al compilador qué bibliotecas enlazar. En Linker Settings hay que copiar la frase <*-lmingw32 -lSDL2main -lSDL2*> en Other Linker Options:

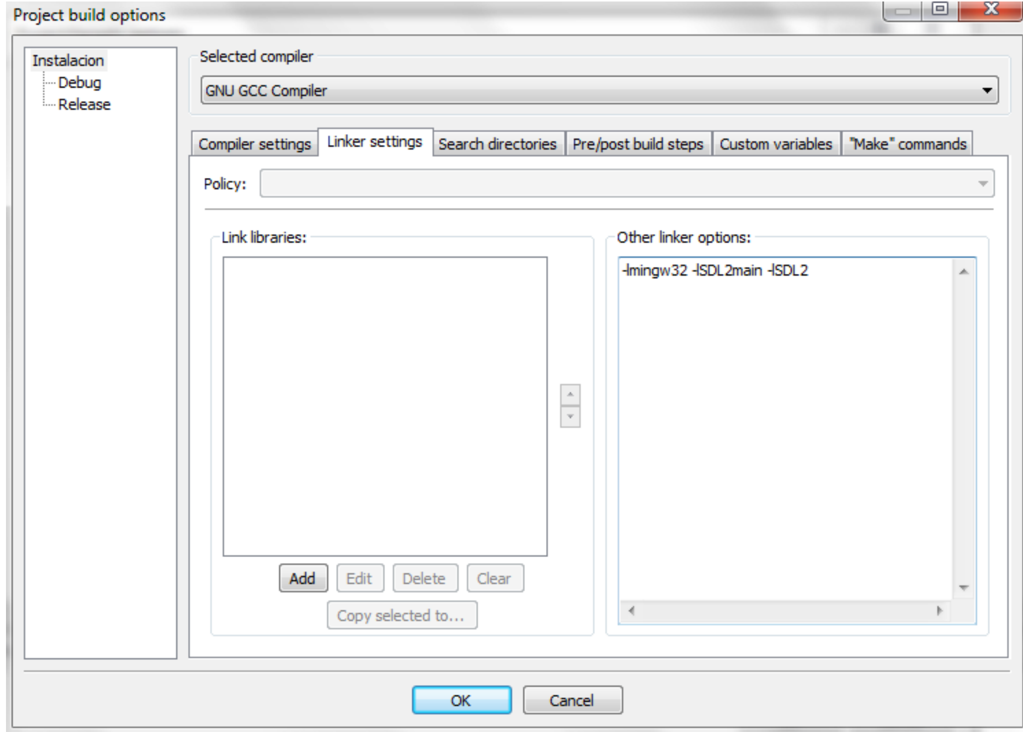


***Figura 11:** Search Directories: Other Linker Options*

- En sexto lugar, se vuelve a las propiedades y en Build targerts se selecciona el tipo (console application)

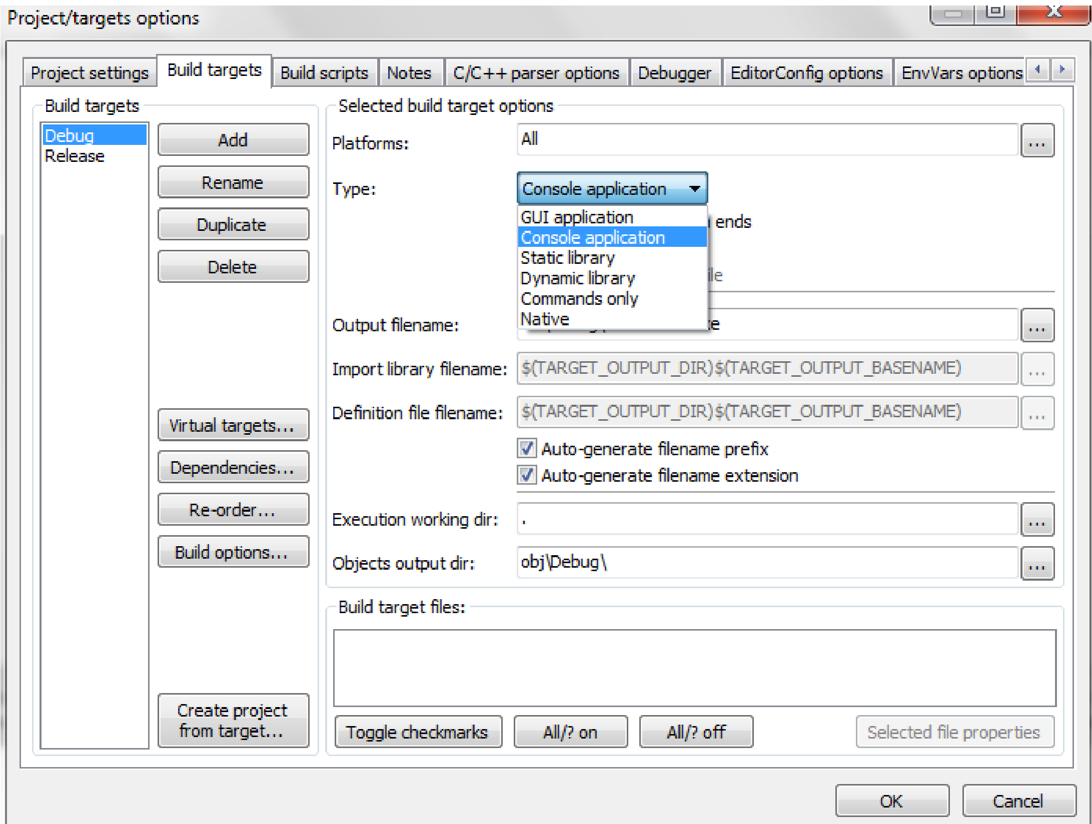
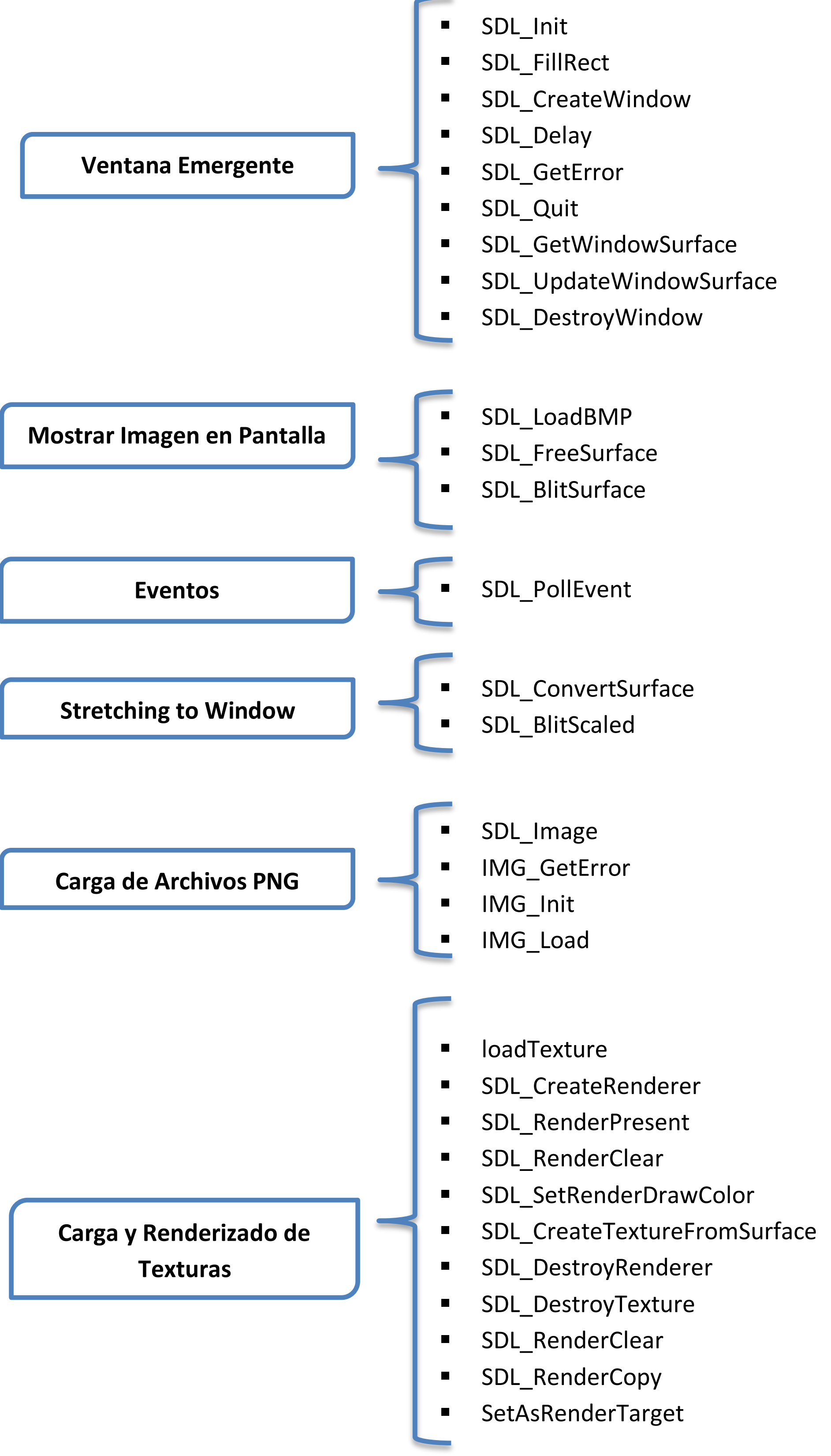


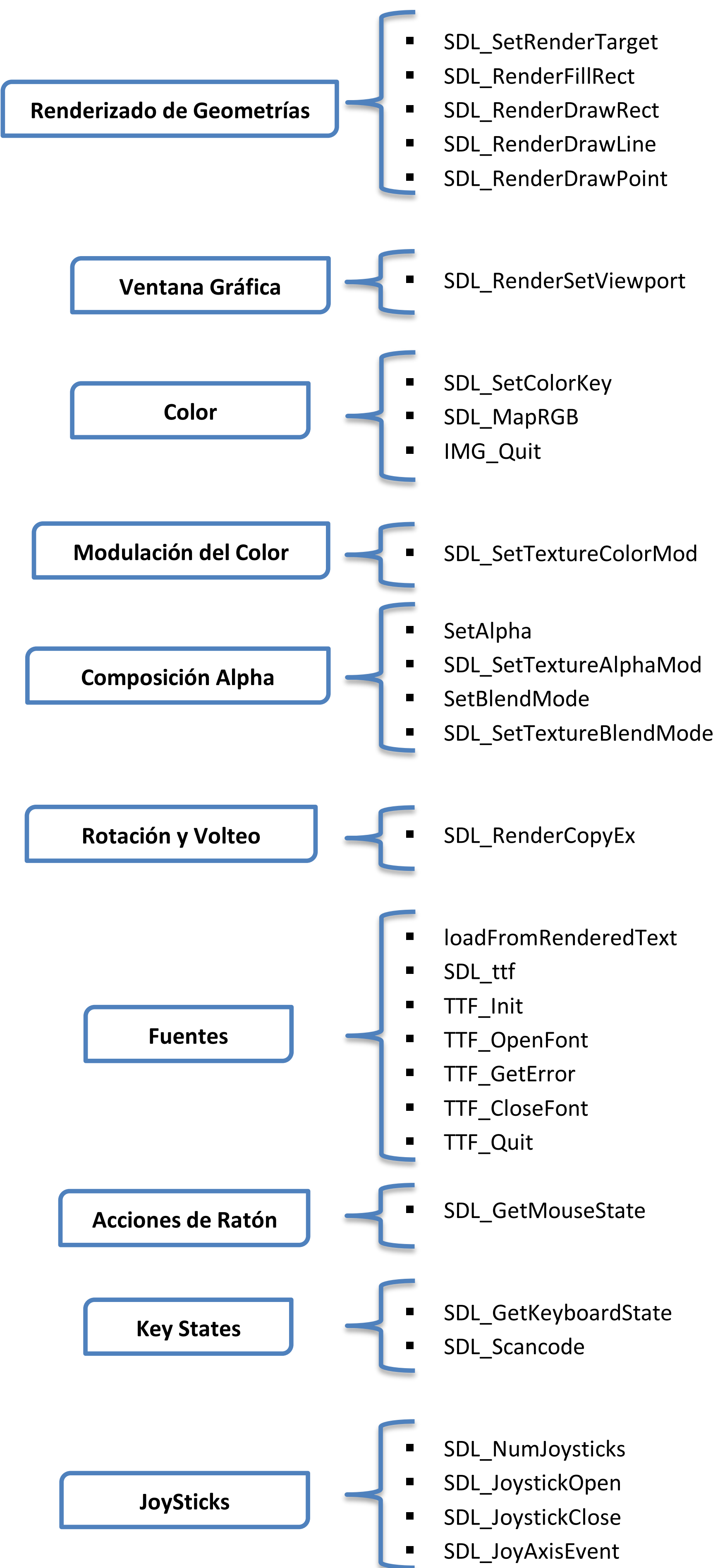
Figura 12: *Build Targets. Selección Console Application*

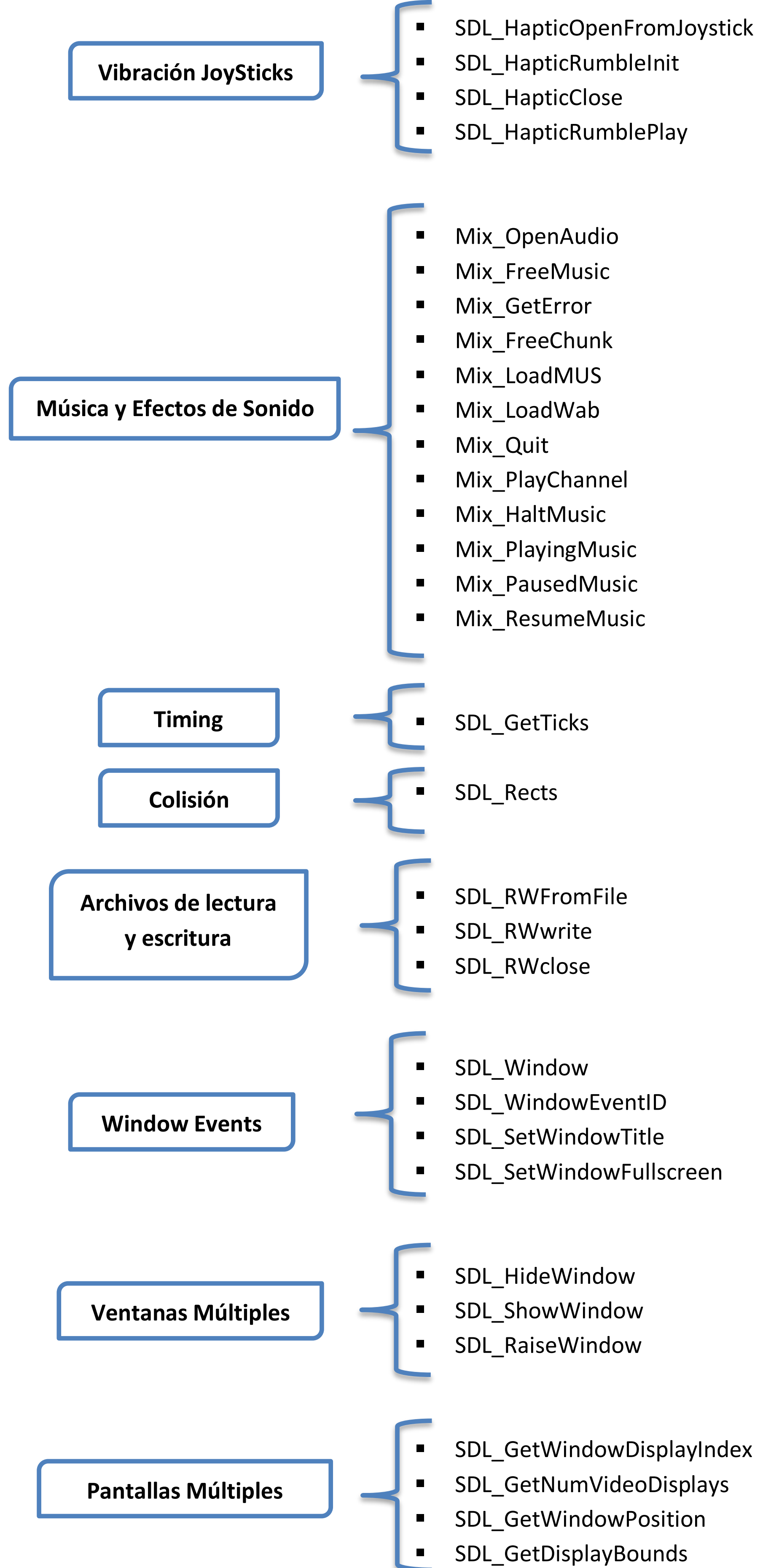
- Para finalizar, cuando se ejecuta la aplicación, el sistema operativo debe ser capaz de encontrar el archivo dll. Para ello, se copia el archivo SDL2.dll bien en la carpeta del ejecutable o bien en el directorio del sistema (system directory)

3. Primeros pasos con SDL

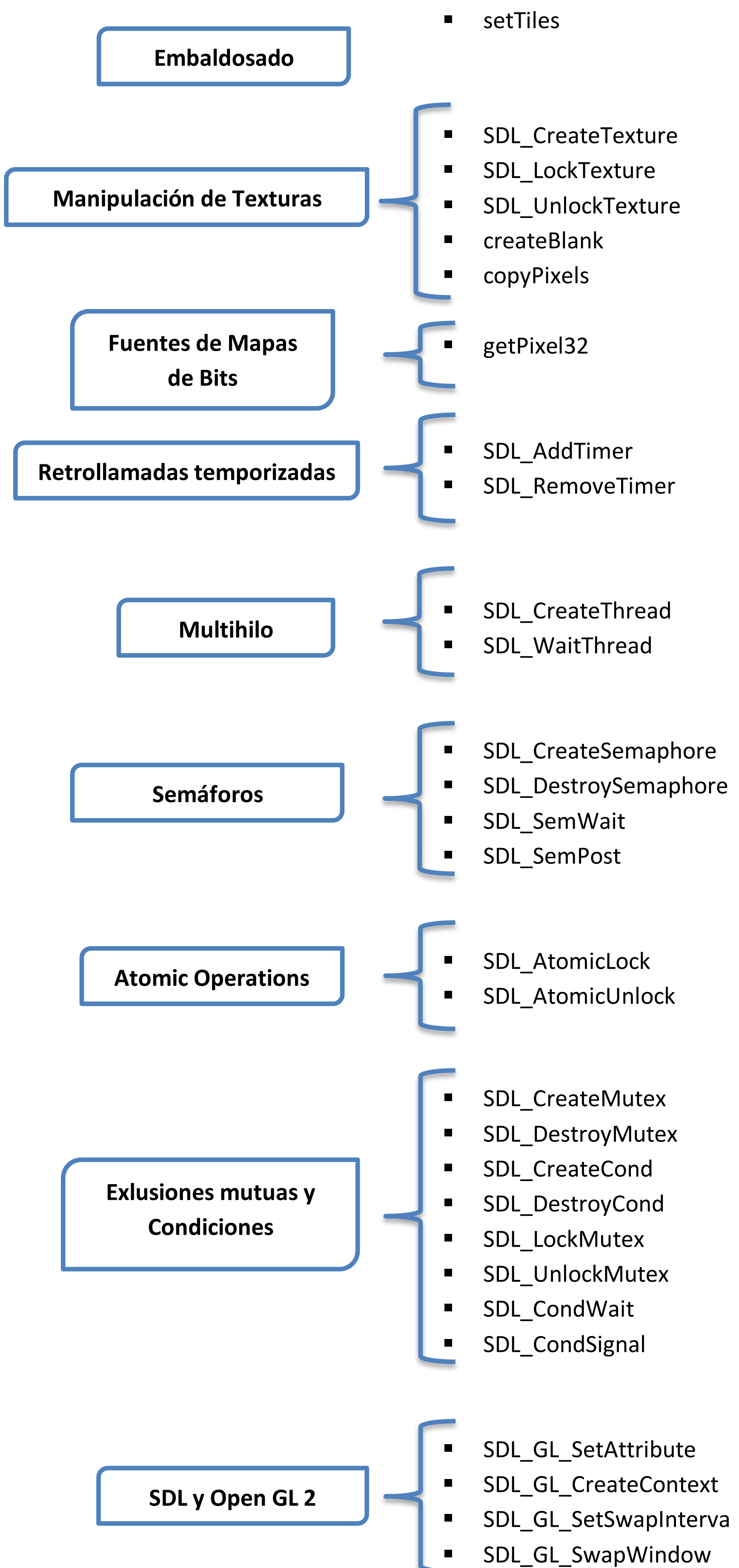
3.1 Diagrama árbol Funciones




Renderizado de Geometrías
SDL_SetRenderTarget
SDL_RenderFillRect
SDL_RenderDrawRect
SDL_RenderDrawLine
SDL_RenderDrawPoint
Ventana Gráfica
SDL_RenderSetViewport
Color
SDL_SetColorKey
SDL_MapRGB
IMG_Quit
Modulación del Color
SDL_SetTextureColorMod
Composición Alpha
SetAlpha
SDL_SetTextureAlphaMod
SetBlendMode
SDL_SetTextureBlendMode
Rotación y Volteo
SDL_RenderCopyEx
Fuentes
loadFromRenderedText
SDL_ttf
TTF_Init
TTF_OpenFont
TTF_GetError
TTF_CloseFont
TTF_Quit
Acciones de Ratón
SDL_GetMouseState
Key States
SDL_GetKeyboardState
SDL_Scancode
JoySticks
SDL_NumJoysticks
SDL_JoystickOpen
SDL_JoystickClose
SDL_JoyAxisEvent



Vibración JoySticks
SDL_HapticOpenFromJoystick
SDL_HapticRumbleInit
SDL_HapticClose
SDL_HapticRumblePlay
Música y Efectos de Sonido
Mix_OpenAudio
Mix_FreeMusic
Mix_GetError
Mix_FreeChunk
Mix_LoadMUS
Mix_LoadWab
Mix_Quit
Mix_PlayChannel
Mix_HaltMusic
Mix_PlayingMusic
Mix_PausedMusic
Mix_ResumeMusic
Timing
SDL_GetTicks
Colisión
SDL_Rects
Archivos de lectura y escritura
SDL_RWFromFile
SDL_RWwrite
SDL_RWclose
Window Events
SDL_Window
SDL_WindowEventID
SDL_SetWindowTitle
SDL_SetWindowFullscreen
Ventanas Múltiples
SDL_HideWindow
SDL_ShowWindow
SDL_RaiseWindow
Pantallas Múltiples
SDL_GetWindowDisplayIndex
SDL_GetNumVideoDisplays
SDL_GetWindowPosition
SDL_GetDisplayBounds



Embaldosado
setTiles
Manipulación de Texturas
SDL_CreateTexture
SDL_LockTexture
SDL_UnlockTexture
createBlank
copyPixels
Fuentes de Mapas de Bits
getPixel32
Retrollamadas temporizadas
SDL_AddTimer
SDL_RemoveTimer
Multihilo
SDL_CreateThread
SDL_WaitThread
Semáforos
SDL_CreateSemaphore
SDL_DestroySemaphore
SDL_SemWait
SDL_SemPost
Atomic Operations
SDL_AtomicLock
SDL_AtomicUnlock
Exlusiones mutuas y Condiciones
SDL_CreateMutex
SDL_DestroyMutex
SDL_CreateCond
SDL_DestroyCond
SDL_LockMutex
SDL_UnlockMutex
SDL_CondWait
SDL_CondSignal
SDL y Open GL 2
SDL_GL_SetAttribute
SDL_GL_CreateContext
SDL_GL_SetSwapInterva
SDL_GL_SwapWindow

3.2 Primera Ventana Gráfica

Una vez se ha configurado SDL, un paso importante es crear una ventana emergente (pop-up).

```
//Using SDL and standard IO
#include <SDL.h>
#include <stdio.h>

//Screen dimension constants
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
```

En la parte superior del archivo fuente se incluye SDL (ya que se necesitan sus funciones y tipos de datos para cualquier código de SDL) y el IO estándar de C para imprimir los errores en la consola. A continuación, se incluyen las cabeceras y se declara el ancho y el alto de la ventana.

```
int main( int argc, char* args[] )
{
        //The window we'll be rendering to
        SDL_Window* window = NULL;

        //The surface contained by the window
        SDL_Surface* screenSurface = NULL;

        //Initialize SDL
        if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
        {
printf( "SDL could not initialize! SDL_Error: %s\n", SDL_GetError() );

        }
```

Ésta es la parte superior de la función principal.

Los argumentos de la función son enteros seguidos por un array de caracteres y el valor devuelto es un entero. (SDL requiere este tipo de main, por lo que es compatible con múltiples plataformas)

A continuación se declara la ventana SDL que se creará. Tras declarar la ventana y la superficie de pantalla se inicializa SDL (ya que no puede llamarse a ninguna función de SDL sin haberla inicializado primero).

Si hay un error, SDL_Init devuelve -1. Dicho error se muestra en la consola con un mensaje indicando qué ocurrió. **SDL_GetError** devuelve el último error producido por una función de SDL con el mensaje: *"SDL could not initialize! SDL_Error: "*.

```
else

        {

//Create window
window = SDL_CreateWindow( "SDL Tutorial", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN
);
                if( window == NULL )

                {
printf( "Window could not be created! SDL_Error: %s\n", SDL_GetError()
);
                }
```

Si SDL se ha inicializado correctamente, se creará una ventana mediante **SDL_CreateWindow**.

- El primer argumento establece el título de la venta.
- Los dos siguientes argumentos definen la posición X e Y.
- El cuarto y quinto argumento definen la anchura y altura de la ventana.
- El último argumento son los flags de creación. **SDL_WINDOW_SHOWN** se asegura de que se muestre la ventana cuando está creada.

Si hay algún error, **SDL_CreateWindow** devuelve NULL y si no hay ninguna ventana, se imprime el error en la consola.

```
else
                {

//Get window surface
screenSurface = SDL_GetWindowSurface( window );

//Fill the surface white
SDL_FillRect( screenSurface, NULL, SDL_MapRGB( screenSurface->format,
0xFF, 0xFF, 0xFF ) );

//Update the surface
SDL_UpdateWindowSurface( window );

//Wait two seconds
SDL_Delay( 2000 );

                }
        }
```

SDL_GetWindowSurface: si la ventana se ha creado correctamente, obtiene la superficie de la ventana y podrá dibujarse en ella.

SDL_FillRect rellena la superficie de la ventana. En este caso, será simplemente blanca.

SDL_UpdateWindowSurface actualiza la ventana, de modo que muestra todo lo que se ha dibujado en ella.

Si tan sólo se crea la ventana, se rellena y se actualiza, todo lo que se verá será la ventana durante un segundo y se cerrará. Para evitar que desaparezca se llama a **SDL_Delay**, que esperará durante una cantidad determinada en milisegundos.

```
//Destroy window
SDL_DestroyWindow( window );

//Quit SDL subsystems
SDL_Quit();

return 0;
```

Tras mostrar la ventana durante 2 segundos, ésta se destruirá para liberar su memoria. Una vez se haya desasignado, se cierra SDL y se devuelve 0 para finalizar el programa.

```
Process returned 0 (0x0)   execution time : 2.037 s
Press any key to continue.
```

Figura 13: *Resultado ejecutar aplicación Primera Ventana Gráfica una vez han transcurrido 2 segundos*

3.3 Mostrar una imagen en pantalla

```
//Starts up SDL and creates window
bool init();

//Loads media
bool loadMedia();

//Frees media and shuts down SDL
void close();
```

En el código anterior se muestran las funciones para controlar la inicialización, carga y cierre de la aplicación SDL. Han de ser declaradas en la parte superior del archivo fuente.

```
//The window we'll be rendering to
SDL_Window* gWindow = NULL;

//The surface contained by the window
SDL_Surface* gScreenSurface = NULL;

//The image we will load and show on the screen
SDL_Surface* gHelloWorld = NULL;
```

En estas líneas se muestra la declaración de variables globales para que el código sea lo más simple posible.

SDL Surface es un tipo de datos de imágenes que contiene los píxeles de la imagen junto con los datos necesarios para renderizarla.

Se tratará la imagen en la pantalla (lo que se ve dentro de la ventana) y la imagen que se carga desde un archivo. Ambas son punteros superficies SDL debido a que se estará asignando dinámicamente memoria para cargar las imágenes y a que es mejor hacer referencia a una imagen por su posición de memoria.

```
bool init()
{
//Initialization flag
bool success = true;

//Initialize SDL
if( SDL_Init( SDL_INIT_VIDEO ) < 0 )

      {
printf( "SDL could not initialize! SDL_Error: %s\n", SDL_GetError() );

success = false;
      }

      else
```

```
	{

    //Create window

gWindow = SDL_CreateWindow( "SDL Tutorial", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT,
SDL_WINDOW_SHOWN);

		if( gWindow == NULL )
		{

printf("Window could not be created! SDL_Error: %s\n",SDL_GetError());
success = false;
		}

		else
		{

    //Get window surface
    gScreenSurface = SDL_GetWindowSurface( gWindow );

		}
	}

	return success;
}
```

Se toma el código de inicialización y creación de ventana y se coloca en su propia función.

Para mostrar imágenes en el interior de la venta se llama a **SDL_GetWindowSurface** con el fin de obtener la superficie contenida por la ventana.

```
bool loadMedia()
{
	//Loading success flag
	bool success = true;

	//Load splash image
gHelloWorld =
SDL_LoadBMP("02_getting_an_image_on_the_screen/hello_world.bmp" );

if( gHelloWorld == NULL )
	{

printf( "Unable to load image %s! SDL Error: %s\n",
"02_getting_an_image_on_the_screen/hello_world.bmp", SDL_GetError() );

success = false;

}

	return success;
}
```

La imagen se carga empleando la función de carga **SDL_LoadBMP**, que toma la ruta de un archivo BMP y devuelve la superficie cargada.

Si la función devuelve NULL quiere decir que no se ha cargado, de modo que se imprime el error en la consola mediante **SDL_GetError**.

```
void close()
{
	//Deallocate surface
	SDL_FreeSurface( gHelloWorld );
	gHelloWorld = NULL;

	//Destroy window
	SDL_DestroyWindow( gWindow );
	gWindow = NULL;

	//Quit SDL subsystems
	SDL_Quit();
}
```

Así, se destruye la ventana.

Al salir de SDL hay que tener cuidado con la superficie de cargada, que se libera mediante **SDL_FreeSurface.**

SDL_DestroyWindow se ocupa de la superficie de pantalla.

```
int main( int argc, char* args[] )

{
	//Start up SDL and create window
	if( !init() )

	{
		printf( "Failed to initialize!\n" );
	}

	else
	{
		//Load media
		if( !loadMedia() )

		{
			printf( "Failed to load media!\n" );
		}

		else

		{
    //Apply the image

    SDL_BlitSurface( gHelloWorld, NULL, gScreenSurface, NULL );
```

Se inicializa SDL y se carga la imagen en la función principal.

Si se realiza correctamente, se combina la superficie cargada sobre la superficie de la pantalla mediante **SDL_BlitSurface**.

El primer argumento de **SDL_BlitSurface** es la imagen de origen y el tercero es el destino.

```
//Update the surface
SDL_UpdateWindowSurface( gWindow );
```

SDL_UpdateWindowSurface actualiza la pantalla para mostrar lo dibujado en ella.

```
//Wait two seconds
SDL_Delay( 2000 );
            }
        }

        //Free resources and close SDL
        close();

        return 0;
}
```

Una vez se ha renderizado todo a la ventana, transcurrirán dos segundos hasta que la imagen desaparezca. Tras la espera, se cierra el programa.

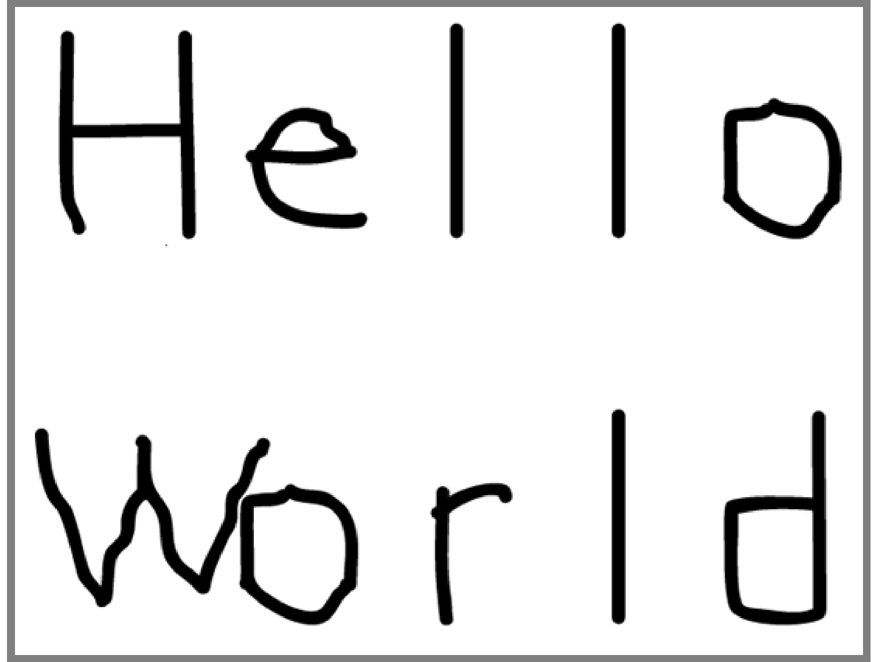


Figura 14: *Resultado ejecutar aplicación Mostrar una imagen en pantalla*

3.4 Programación Dirigida por Eventos (Event Driven Programming)

Valiéndose del sistema de gestión de eventos (event handling system), además de colocar imágenes en la pantalla, puede manipularse la entrada del usuario.

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;
```

Tras inicializar SDL, se declara un flag que comprueba si el usuario ha salido o no de la aplicación. Desde que comienza la aplicación hasta llegar a este punto, dicho flag debe inicializarse como false. Además, se declara una unión **SDL_Event.**

```
//While application is running
while( !quit )

{
```

En esta aplicación el programa esperará hasta que el usuario disponga cerrarlo. De esta manera, se tendrá la aplicación en bucle hasta que el usuario decida. Este bucle, que continúa trabajando mientras la aplicación está activa, se denomina bucle principal.

```
//Handle events on queue
while( SDL_PollEvent( &e ) != 0 )

      {

      //User requests quit
      if( e.type == SDL_QUIT )

            {
                  quit = true;
            }
      }
```

El bucle de eventos se encuentra en la parte superior del bucle principal y su función es procesar la cola de eventos hasta que está vacía. Al pulsar una tecla, mover el ratón o tocar una pantalla táctil se añaden eventos a la anteriormente denominada cola de eventos.

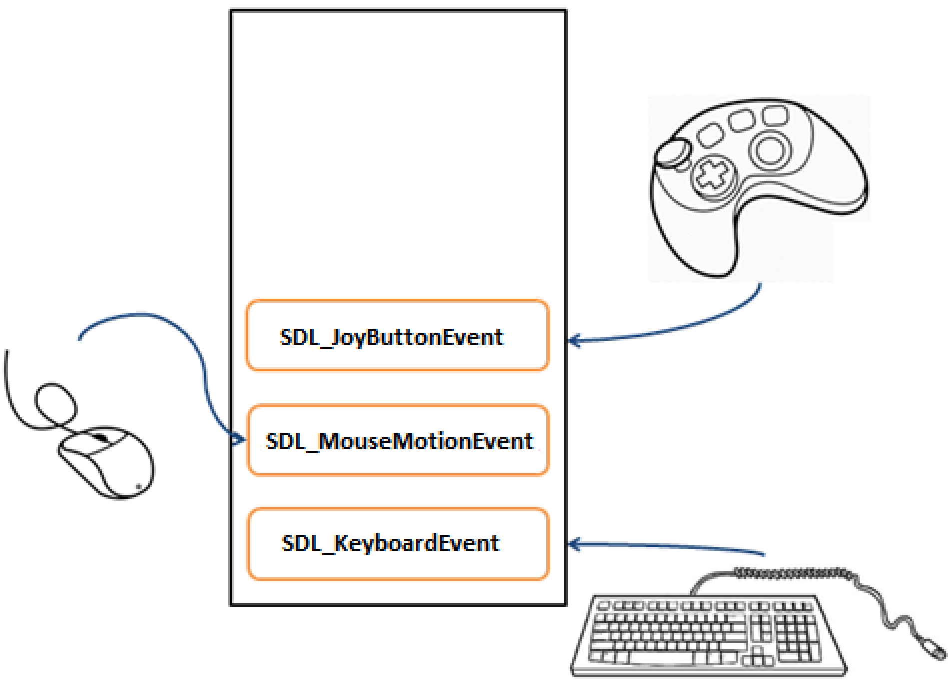


Figura 15: *Cola de Eventos*

La cola de eventos irá almacenando los eventos en el orden en el que ocurran y esperará hasta que sean procesados.

SDL_PollEvent explora la cola de eventos obteniendo de esta manera el evento más reciente.

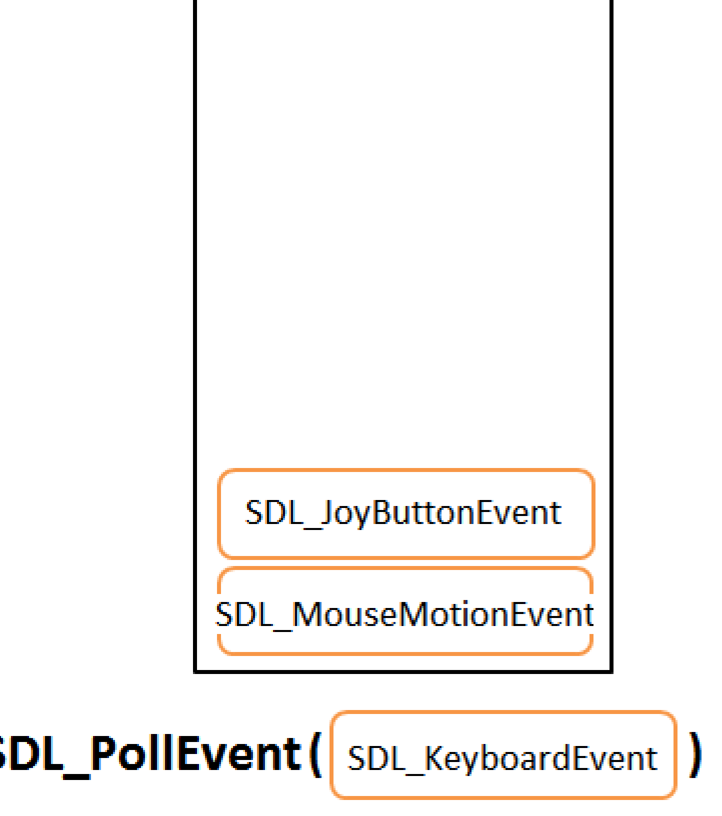


Figura 16: *Cola de eventos. SDL_PollEvent*

Una vez la cola de eventos esté vacía **SDL_PollEvent** devolverá 0.

En el caso de existir en la cola de eventos un evento tipo **SDL_Quit** (establece el indicador de abandono en true para que se pueda salir de la aplicación), el flag toma el valor true y puede cerrarse la aplicación.

```
            //Apply the image
            SDL_BlitSurface( gXOut, NULL, gScreenSurface, NULL );

            //Update the surface
            SDL_UpdateWindowSurface( gWindow );
                  }
```

Tras haber terminado de procesar los eventos, se dibuja la pantalla y se actualiza. Si el flag tiene el valor true, la aplicación saldrá al final del bucle; si aún tiene el valor false, continuará hasta que el usuario cierre la ventana.

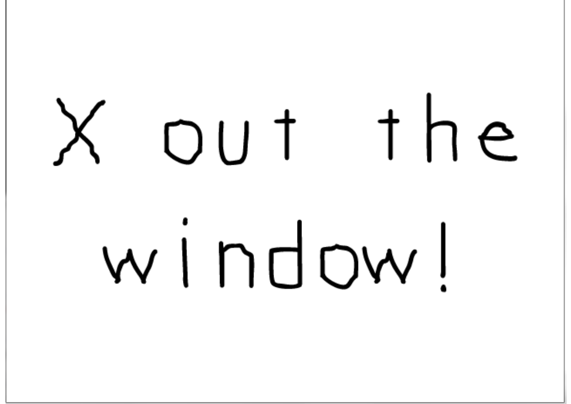


Figura 17*: Resultado ejecutar aplicación Programación Dirigida por Eventos*

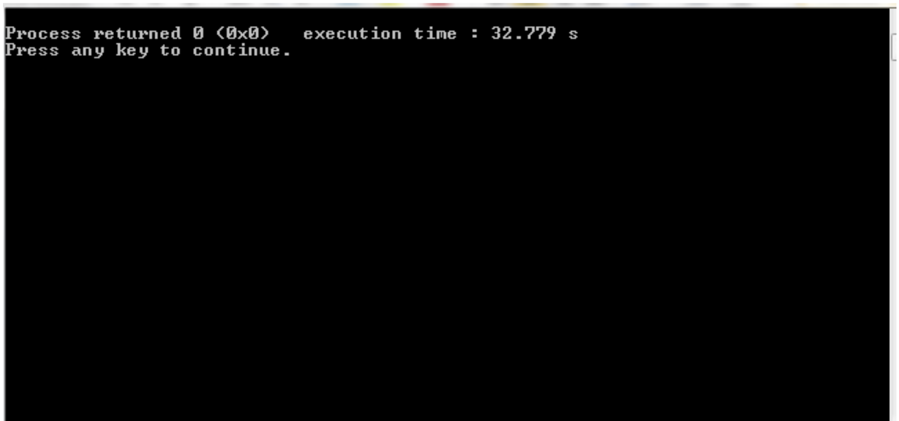


Figura 18*: Resultado ejecutar aplicación Programación Dirigida por Eventos (II)*

Hasta que el usuario no ha decidido cerrar la ventana, ésta ha seguido activa. En este caso, como se puede observar, han transcurrido casi 33 segundos.

3.5 Key Presss

Uno de los eventos que SDL es capaz de realizar es el de hacer click en X para salir de las ventanas. Otro tipo de entrada comúnmente utilizada es la del teclado. El objetivo es conseguir que aparezcan distintas imágenes dependiendo de la tecla que se haya pulsado.

```
//Key press surfaces constants
enum KeyPressSurfaces

{
        KEY_PRESS_SURFACE_DEFAULT,
        KEY_PRESS_SURFACE_UP,
        KEY_PRESS_SURFACE_DOWN,
        KEY_PRESS_SURFACE_LEFT,
        KEY_PRESS_SURFACE_RIGHT,
        KEY_PRESS_SURFACE_TOTAL
};
```

En la parte superior del código se declara una enumeración de las distintas superficies que se tienen.

Por defecto, se empieza a contar desde 0 y se continúa incrementando de uno en uno por cada enumeración declarada. De esta manera:

- KEY_PRESS_SURFACE_DEFAULT es 0.
- KEY_PRESS_SURFACE_UP es 1.
- KEY_PRESS_SURFACE_DOWN es 2.
- KEY_PRESS_SURFACE_LEFT es 3.
- KEY_PRESS_SURFACE_RIGHT es 4.
- KEY_PRESS_SURFACE_TOTAL es 5.

```
//Starts up SDL and creates window
bool init();

//Loads media
bool loadMedia();

//Frees media and shuts down SDL
void close();

//Loads individual image
SDL_Surface* loadSurface( std::string path );

//The window we'll be rendering to
SDL_Window* gWindow = NULL;

//The surface contained by the window
SDL_Surface* gScreenSurface = NULL;

//The images that correspond to a keypress
SDL_Surface* gKeyPressSurfaces[ KEY_PRESS_SURFACE_TOTAL ];
```

```
//Current displayed image
SDL_Surface* gCurrentSurface = NULL;
```

En lugar de copiar/pegar código cada vez que se necesite, se usará una función que realice esta operación. Será necesario contar con un array de punteros a las superficies SDL denominado **gKeyPressSurfaces** para contener todas las imágenes que se usen. Dependiendo del tipo de tecla que el usuario presione, se aplicará **gCurrentSurface** a una de estas superficies.

```
SDL_Surface* loadSurface( std::string path )
{
	//Load image at specified path
	SDL_Surface* loadedSurface = SDL_LoadBMP( path.c_str() );
	if( loadedSurface == NULL )

	{
printf( "Unable to load image %s! SDL Error: %s\n", path.c_str(),
SDL_GetError() );
	}

	return loadedSurface;

}
```

LoadSurface carga imágenes e informa de los errores ocurridos en el caso de que sucedan.

Al tener la carga de imágenes y el informe de errores ocurridos contenidos en una misma función es más sencillo añadir y depurar la carga de la imagen.
La finalidad de esta función es cargar la superficie y devolver la recién cargada superficie. En este programa la superficie cargada se cancela en la función de cierre.

```
bool loadMedia()

{
	//Loading success flag
	bool success = true;

	//Load default surface
gKeyPressSurfaces[ KEY_PRESS_SURFACE_DEFAULT ] = loadSurface(
"04_key_presses/press.bmp" );

	if( gKeyPressSurfaces[ KEY_PRESS_SURFACE_DEFAULT ] == NULL )

	{
		printf( "Failed to load default image!\n" );
		success = false;
	}


	//Load up surface
gKeyPressSurfaces[ KEY_PRESS_SURFACE_UP ] = loadSurface(
"04_key_presses/up.bmp" );
```

```
	if( gKeyPressSurfaces[ KEY_PRESS_SURFACE_UP ] == NULL )

	{
		printf( "Failed to load up image!\n" );
		success = false;
	}

	//Load down surface
gKeyPressSurfaces[ KEY_PRESS_SURFACE_DOWN ] = loadSurface(
"04_key_presses/down.bmp" );

	if( gKeyPressSurfaces[ KEY_PRESS_SURFACE_DOWN ] == NULL )

	{
		printf( "Failed to load down image!\n" );
		success = false;
	}

	//Load left surface
gKeyPressSurfaces[ KEY_PRESS_SURFACE_LEFT ] = loadSurface(
"04_key_presses/left.bmp" );
	if( gKeyPressSurfaces[ KEY_PRESS_SURFACE_LEFT ] == NULL )
	{
		printf( "Failed to load left image!\n" );
		success = false;
	}

	//Load right surface
gKeyPressSurfaces[ KEY_PRESS_SURFACE_RIGHT ] = loadSurface(
"04_key_presses/right.bmp" );

	if( gKeyPressSurfaces[ KEY_PRESS_SURFACE_RIGHT ] == NULL )

	{
		printf( "Failed to load right image!\n" );
		success = false;
	}

	return success;
}
```

En la función **loadMedia** se cargan todas las imágenes que se usarán para renderizar en la pantalla.

```
			//Main loop flag
			bool quit = false;

			//Event handler
			SDL_Event e;

			//Set default current surface
gCurrentSurface = gKeyPressSurfaces[ KEY_PRESS_SURFACE_DEFAULT ];
```

En la función principal, antes de entrar en el bucle principal, se fija la superficie que se mostrará por defecto.

```
//Handle events on queue
while( SDL_PollEvent( &e ) != 0 )
       {
//User requests quit
if( e.type == SDL_QUIT )
{
       quit = true;
}
//User presses a key
else if( e.type == SDL_KEYDOWN )
{
//Select surfaces based on key press
switch( e.key.keysym.sym )
{
       case SDLK_UP: gCurrentSurface =
gKeyPressSurfaces[KEY_PRESS_SURFACE_UP ];
       break;

       case SDLK_DOWN:gCurrentSurface = gKeyPressSurfaces[
KEY_PRESS_SURFACE_DOWN ];
       break;

       case SDLK_LEFT:gCurrentSurface = gKeyPressSurfaces[
KEY_PRESS_SURFACE_LEFT ];
       break;

       case SDLK_RIGHT: gCurrentSurface = gKeyPressSurfaces[
KEY_PRESS_SURFACE_RIGHT ];
       break;

       default:gCurrentSurface = gKeyPressSurfaces[
KEY_PRESS_SURFACE_DEFAULT ];
       break;

}
```

Éste es el bucle de eventos. Se controlará el evento **SDL_KEYDOWN**, que ocurre siempre que se presione una tecla del teclado.

Como se ha comentado, la finalidad de este código es establecer las superficies basándose en qué tecla ha sido pulsada.

```
//Apply the current image
SDL_BlitSurface( gCurrentSurface, NULL, gScreenSurface, NULL );
```

```
//Update the surface
SDL_UpdateWindowSurface( gWindow );
```

Después de que las teclas se hayan pulsado y la superficie se haya establecido, se combina la superficie seleccionada con la pantalla.

Al ejecutar la aplicación aparece la siguiente ventana indicando que se presionne los cursores arriba, abajo, izquierda o derecha:

Press: up, down,
left, or right!

***Figura 19**: Resultado ejecutar aplicación Key Presses*

Caso 1: Cursor arriba

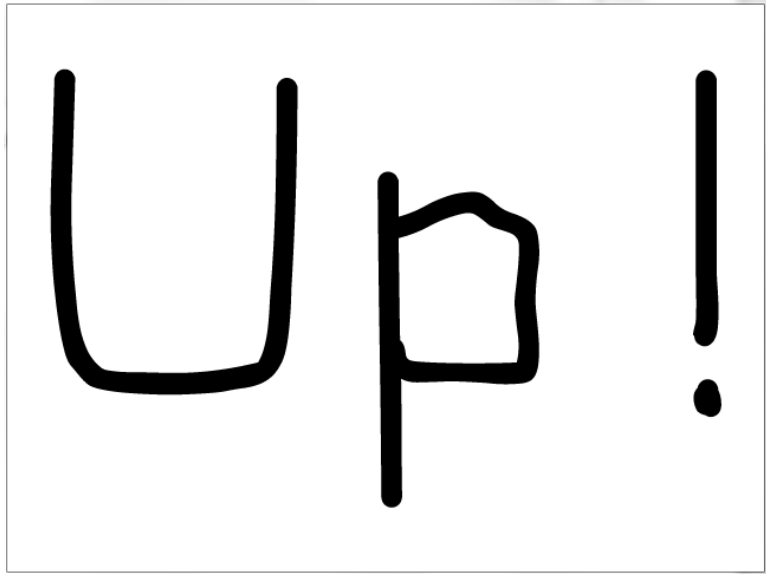


***Figura 20:** Resultado ejecutar aplicación tras pulsar cursor arriba*

- **Caso 2:** Cursor abajo

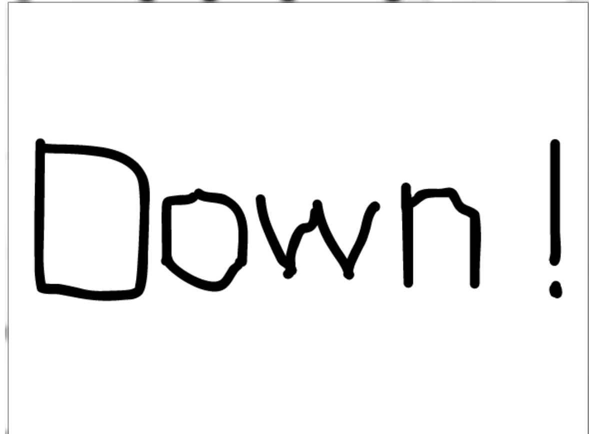


Figura 21: *Resultado ejecutar aplicación tras pulsar cursor abajo*

- **Caso 3:** Cursor izquierda

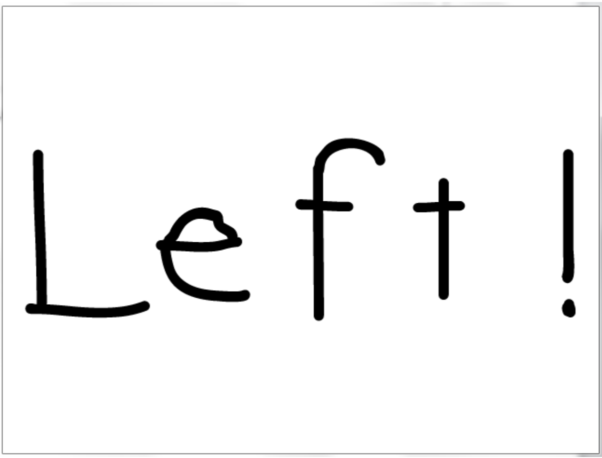


Figura 22: *Resultado ejecutar aplicación tras pulsar cursor izquierda*

- **Caso 4:** Cursor derecha

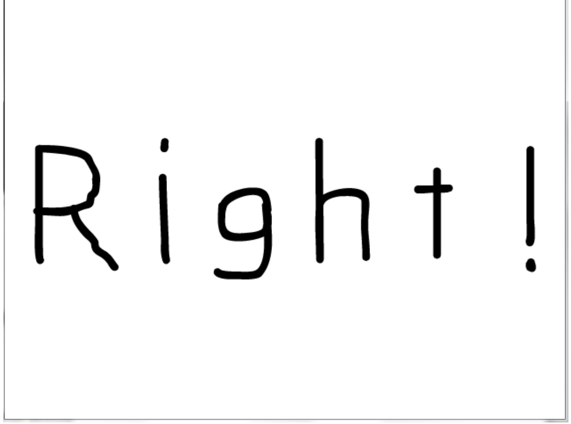


Figura 23: *Resultado ejecutar aplicación tras pulsar cursor derecha*

3.6 Alargar a ventana (Stretching to Window)

SDL 2.0 posee una nueva característica para superficies denominada <*alargamiento suave*>, que permite combinar una imagen a escala a un tamaño diferente. La finalidad de este código es escoger una imagen de la mitad del tamaño de la pantalla y alargarla hasta alcanzar el tamaño completo.

```
SDL_Surface* loadSurface( std::string path )

{
        //The final optimized image
        SDL_Surface* optimizedSurface = NULL;

        //Load image at specified path
        SDL_Surface* loadedSurface = SDL_LoadBMP( path.c_str() );

        if( loadedSurface == NULL )

        {
printf( "Unable to load image %s! SDL Error: %s\n", path.c_str(),
SDL_GetError() );

        }
```

Las imágenes se cargarán al inicio de la función. Además, se declarará un puntero a la imagen final optimizada.

```
else
        {
                //Convert surface to screen format
optimizedSurface = SDL_ConvertSurface( loadedSurface, gScreenSurface-
>format, NULL );

                if( optimizedSurface == NULL )
                {

printf( "Unable to optimize image %s! SDL Error: %s\n", path.c_str(),
SDL_GetError() );

                }

                //Get rid of old loaded surface
                SDL_FreeSurface( loadedSurface );
        }

        return optimizedSurface;
}
```

Una vez cargada la imagen, se optimiza la superficie cargada.

Mediante **SDL_ConvertSurface**, cuando la imagen esté cargada, se convertirá al mismo formato que la pantalla para que no sea necesaria la conversión cada vez que se combine.

Esta función devuelve una copia del original en un nuevo formato. La imagen original cargada continúa en la memoria después de llamar a la función, es decir, hay que liberar la superficie original cargada o se tendrán dos copias de la misma imagen en la memoria.

Después de que la imagen se cargue y se convierta, se devuelve la imagen final optimizada.

```
//Apply the image stretched

SDL_Rect stretchRect;
stretchRect.x = 0;
stretchRect.y = 0;
stretchRect.w = SCREEN_WIDTH;
stretchRect.h = SCREEN_HEIGHT;
SDL_BlitScaled( gStretchedSurface, NULL, gScreenSurface, &stretchRect
);

//Update the surface

SDL_UpdateWindowSurface( gWindow );

}
```

SDL_BlitScaled es una función dedicada a combinar imágenes de distinto tamaño.

Si se tiene una imagen más pequeña que la pantalla y se quiere hacer de este tamaño, se puede modificar el ancho/alto de la imagen para que tenga el de la pantalla.

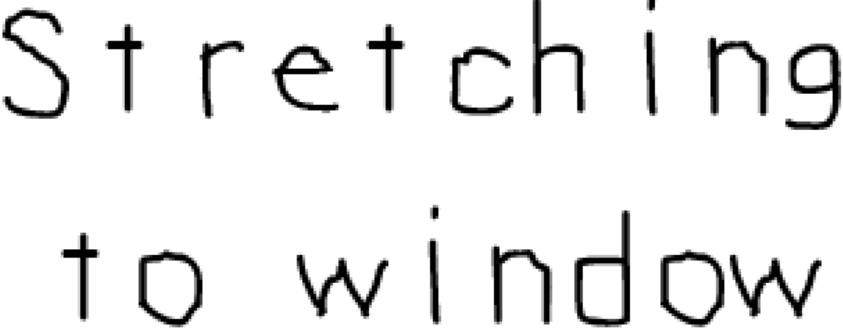


Figura 24: *Resultado ejecutar aplicación Stretching to window*

3.7 Extensión de librerías y carga de otros formatos de imágenes

Las librerías de extensión SDL permiten cargar ficheros de imágenes además de BMP, reproducir música…

SDL_image permite cargar archivos PNG, que hacen que se ahorre gran cantidad de espacio en disco. Una vez configurada **SDL_image**, se mostrará cómo cargar un archivo PNG con SDL.

```
//Using SDL, SDL_image, standard IO, and strings
#include <SDL.h>
#include <SDL_image.h>
#include <stdio.h>
#include <string>
```

Para utilizar cualquier función **SDL_image** o tipos de datos hay que incluir la cabecera **SDL_image**.

```
bool init()
{
      //Initialization flag
      bool success = true;

      //Initialize SDL
      if( SDL_Init( SDL_INIT_VIDEO ) < 0 )

      {
printf( "SDL could not initialize! SDL Error: %s\n",SDL_GetError() );

success = false;
      }

      else
      {

    //Create window

gWindow = SDL_CreateWindow( "SDL Tutorial",SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED,SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN
);
            if( gWindow == NULL )
            {

printf( "Window could not be created! SDL Error: %s\n", SDL_GetError()
);
        success = false;

            }

            else
            {

        //Initialize PNG loading
        int imgFlags = IMG_INIT_PNG;

        if( !( IMG_Init( imgFlags ) & imgFlags ) )
```

```
{
printf("SDL_image could not initialize! SDL_image Error: %s\n",
IMG_GetError() );

            success = false;
}
                  Else

                  {
//Get window surface

gScreenSurface = SDL_GetWindowSurface( gWindow );
                  }
}

}

      return success;
}
```

Puesto que se quiere inicializar **SDL_image** cargando PNG, se convierten los flags de carga PNG en **IMG_Init**, que devuelve los flags cargados correctamente.

- Si estos flags devueltos no contienen ciertos flags requeridos significa que ha ocurrido un error.
- Si hay un error con **SDL_image**, se obtiene una cadena de error mediante **IMG_GetError** en lugar de **SDL_GetError**.

```
SDL_Surface* loadSurface( std::string path )
{
      //The final optimized image
      SDL_Surface* optimizedSurface = NULL;

      //Load image at specified path
      SDL_Surface* loadedSurface=IMG_Load(path.c_str());

      if( loadedSurface == NULL )
      {

printf( "Unable to load image %s! SDL_image Error: %s\n",
path.c_str(), IMG_GetError() );
      }

      else
      {
            //Convert surface to screen format
optimizedSurface = SDL_ConvertSurface( loadedSurface,
gScreenSurface->format, NULL );

            if( optimizedSurface == NULL )
            {

printf( "Unable to optimize image %s! SDL Error: %s\n",
path.c_str(), SDL_GetError() );

            }
```

```
//Get rid of old loaded surface
            SDL_FreeSurface( loadedSurface );
      }

      return optimizedSurface;
}
```

En esta función de carga en lugar de **SDL_LoadBMP** se utiliza **IMG_Load**, que puede cargar diferentes tipos de formatos. Al igual que **IMG_Init**, cuando ocurre un error con **IMG_Load**, se llama a **IMG_GetError** para obtener la cadena de errores.

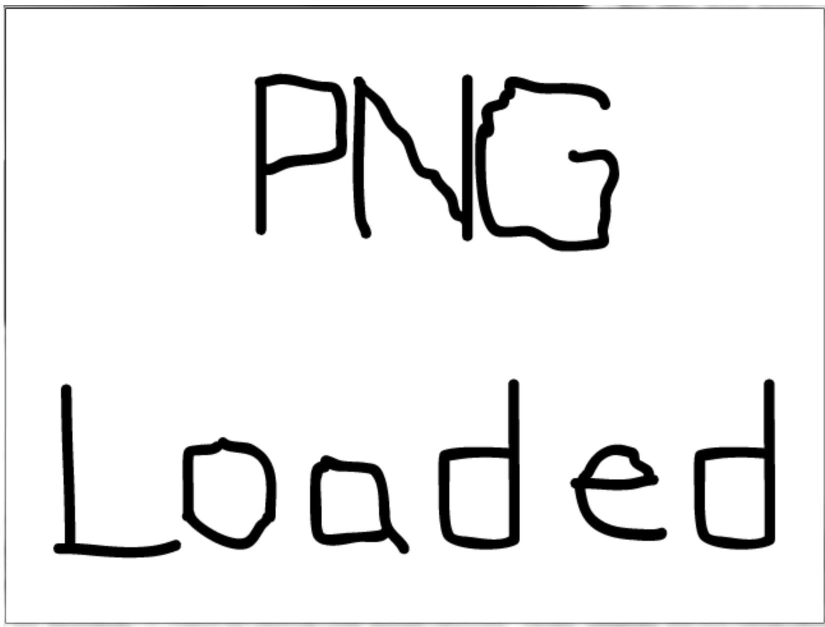


***Figura 25**: Resultado ejecutar aplicación Extensión de librerías y carga de otros formatos de imágenes*

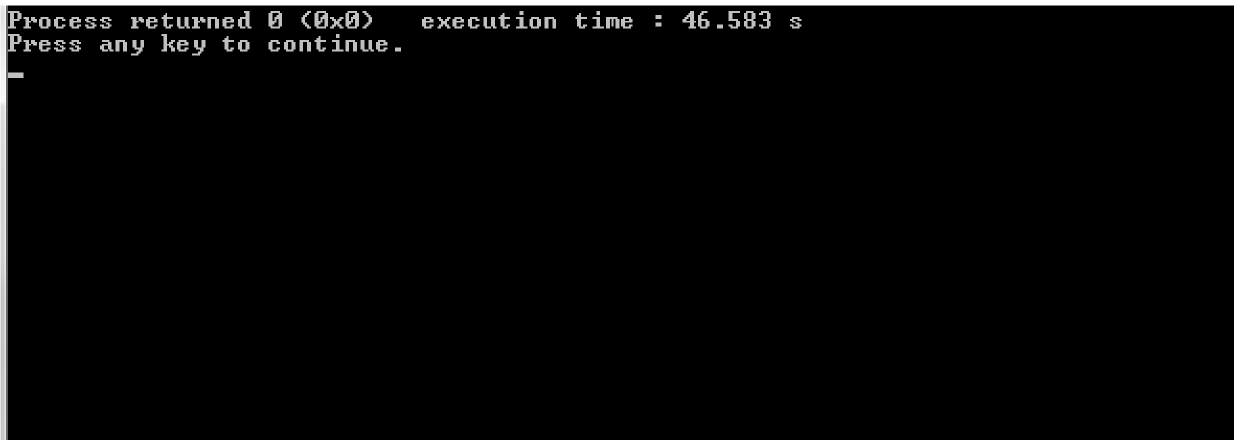


***Figura 26**: Resultado ejecutar aplicación Extensión de librerías y carga de otros formatos de imágenes (II)*

3.8 Carga y Renderizado de Texturas

```
//Loads individual image as texture
SDL_Texture* loadTexture( std::string path );

//The window we'll be rendering to
SDL_Window* gWindow = NULL;

//The window renderer
SDL_Renderer* gRenderer = NULL;

//Current displayed texture
SDL_Texture* gTexture = NULL;
```

En SDL las texturas tienen su propio tipo de datos denominado **SDL_Texture**. Cuando se trata de texturas SDL, se necesita un **SDL_Renderer** para renderizarlas, por lo que se declara un renderizador global llamado “gRenderer”

La carga de imágenes es con **loadTexture** y una textura declarada globalmente que se cargará a continuación.

```
//Create window

gWindow = SDL_CreateWindow( "SDL Tutorial",
SDL_WINDOWPOS_UNDEFINED,SDL_WINDOWPOS_UNDEFINED,SCREEN_WIDTH,
SCREEN_HEIGHT, SDL_WINDOW_SHOWN );

            if( gWindow == NULL )
            {

printf( "Window could not be created! SDL Error: %s\n",
SDL_GetError() );

  success = false;
            }

            else
            {

//Create renderer for window

gRenderer = SDL_CreateRenderer( gWindow, -1,
SDL_RENDERER_ACCELERATED );

                  if( gRenderer == NULL )
                  {

printf("Renderer could not be created! SDL Error: %s\n",
SDL_GetError() );

        success = false;
}

else
```

```
         {
//Initialize renderer color
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );

//Initialize PNG loading
int imgFlags = IMG_INIT_PNG;

if( !( IMG_Init( imgFlags ) & imgFlags ) )
                       {
printf("SDL_image could not initialize! SDL_image Error: %s\n",
IMG_GetError() );

        success = false;
                       }
                 }
           }
     }

     }

     return success;
}
```

Una vez creada la venta, se crea un renderizador para la ventana para que se puedan renderizar las texturas en él. Después de crear el renderizador se inicializa el color de renderizado mediante **SDL_SetRenderDrawColor**, que controla qué color se utiliza para las diversas operaciones de renderizado.

```
SDL_Texture* loadTexture( std::string path )
{
     //The final texture
     SDL_Texture* newTexture = NULL;

     //Load image at specified path
SDL_Surface* loadedSurface=IMG_Load(path.c_str());

if( loadedSurface == NULL )
     {

printf( "Unable to load image %s! SDL_image Error: %s\n",
path.c_str(), IMG_GetError() );

     }

     else
     {
//Create texture from surface pixels
newTexture= SDL_CreateTextureFromSurface(gRenderer,loadedSurface);

           if( newTexture == NULL )
           {

printf( "Unable to create texture from %s! SDL Error: %s\n",
path.c_str(), SDL_GetError() );

           }
```

```
            //Get rid of old loaded surface
            SDL_FreeSurface( loadedSurface );
      }

      return newTexture;
}
```

La función de textura de carga crea una textura a partir de la superficie cargada usando **SDL_CreateTextureFromSurface** en lugar de convertir la superficie cargada a un formato de visualización. Esta función crea una nueva textura a partir de una superficie existente, por lo que es necesario liberar la superficie cargada y, una vez hecho esto, volver a la textura cargada.

```
bool loadMedia()
{
      //Loading success flag
      bool success = true;

      //Load PNG texture
gTexture = loadTexture(07_texture_loading_and_rendering/texture.png");

if( gTexture == NULL )

      {
            printf( "Failed to load texture image!\n" );
            success = false;
      }

      return success;
}

void close()
{
      //Free loaded image

      SDL_DestroyTexture( gTexture );
      gTexture = NULL;

      //Destroy window

      SDL_DestroyRenderer( gRenderer );
      SDL_DestroyWindow( gWindow );
      gWindow = NULL;
      gRenderer = NULL;

      //Quit SDL subsystems
      IMG_Quit();
      SDL_Quit();

}
```

En la función de <limpieza> se designan las texturas empleando **SDL_DestroyTexture**.

```
 //While application is running
while( !quit )
{
        //Handle events on queue
        while( SDL_PollEvent( &e ) != 0 )

       {
                //User requests quit
                if( e.type == SDL_QUIT )
                {
                        quit = true;
                }
        }

//Clear screen
SDL_RenderClear( gRenderer );

//Render texture to screen
SDL_RenderCopy( gRenderer, gTexture, NULL, NULL );

//Update screen
SDL_RenderPresent( gRenderer );
```

En el bucle principal, detrás del bucle de eventos, se llama a **SDL_RenderClear**. Esta función llena la pantalla con el último color establecido por **SDL_SetRenderDrawColor**.

Con la pantalla limpia, se renderiza la textura con **SDL_RenderCopy**. Una vez renderizada la textura, se actualiza la pantalla, pero se utiliza **SDL_RenderPresent** en lugar de **SDL_UpdateWindowSurface** (ya que no se emplea **SDL_Surfaces** para renderizar).

***Figura 27**: Resultado ejecutar aplicación Carga y Renderizado de Texturas*

3.9 Renderizado de Geometría

Mediante SDL pueden renderizarse formas básicas sin que se requiera el uso de gráficos adicionales.

```
bool loadMedia()
{
	//Loading success flag
	bool success = true;

	//Nothing to load
	return success;
}
```

El renderizado primitivo de SDL permite renderizar formas sin cargar gráficos especiales.

```
//While application is running
			while( !quit )
			{
				//Handle events on queue
				while( SDL_PollEvent( &e ) != 0 )
				{
					//User requests quit
					if( e.type == SDL_QUIT )
					{
						quit = true;
					}
				}

	//Clear screen
	SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
	SDL_RenderClear( gRenderer );
```

En la parte superior del bucle principal el evento de salida borra la pantalla. Otro punto a tener en cuenta es que se establece el color transparente a blanco con **SDL_SetRenderDrawColor** en cada fotograma en lugar de establecerlo una vez en la función de inicialización.

```
//Render red filled quad
SDL_Rect fillRect = { SCREEN_WIDTH / 4, SCREEN_HEIGHT / 4,
SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2 };

SDL_SetRenderDrawColor( gRenderer, 0xFF, 0x00, 0x00, 0xFF );
SDL_RenderFillRect( gRenderer, &fillRect );
```

La primera primitiva que se dibujará es un rectángulo relleno, es decir, un rectángulo sólido.

En primer lugar, se define un rectángulo para determinar el área que se quiere rellenar con color. Las variables que componen un rectángulo SDL son x,y,w,h para la posición X, posición Y, anchura y altura respectivamente.

En este código se estable la anchura del rectángulo en un cuarto de la pantalla en la dirección X, la altura en un cuarto de la pantalla en la dirección Y y el ancho y el alto con la mitad de la pantalla.

Una vez se ha definido el área del rectángulo, se fija el color de renderizado con **SDL_SetRenderDrawColor**. Esta función toma el renderizado para la ventana que se está usando y los valores RGBA para el color con el que se quiere renderizar (siendo R la componente roja; G, la verde; B, la azul y A, alfa). Alfa controla el nivel de opacidad. Estos valores van de 0 a 255 (o FF en hexadecimal) y se mezclan todos para crear todos los colores que se ven en la pantalla. La llamada a **SDL_SetRenderDrawColor** fija el color de dibujo a rojo opaco.

Tras establecer el rectángulo y el color, se llama a **SDL_RenderFillRect** para dibujar el rectángulo.

```
//Render green outlined quad
SDL_Rect outlineRect = { SCREEN_WIDTH / 6, SCREEN_HEIGHT / 6,
SCREEN_WIDTH * 2 / 3, SCREEN_HEIGHT * 2 / 3 };

SDL_SetRenderDrawColor( gRenderer, 0x00, 0xFF, 0x00, 0xFF );
SDL_RenderDrawRect( gRenderer, &outlineRect );
```

También se puede dibujar el contorno del rectángulo usando **SDL_RenderDrawRect**. Funciona de forma similar al rectángulo relleno, ya que la mayoría del código es igual que el anterior. La principal diferencia es que este rectángulo es dos terceras veces el tamaño de la pantalla y el color que se emplea es el verde.

Con respecto a la posición del rectángulo, la coordenada más grande se sitúa abajo y la coordenada más pequeña arriba. Esto sucede porque SDL utiliza un sistema de coordenadas diferente.

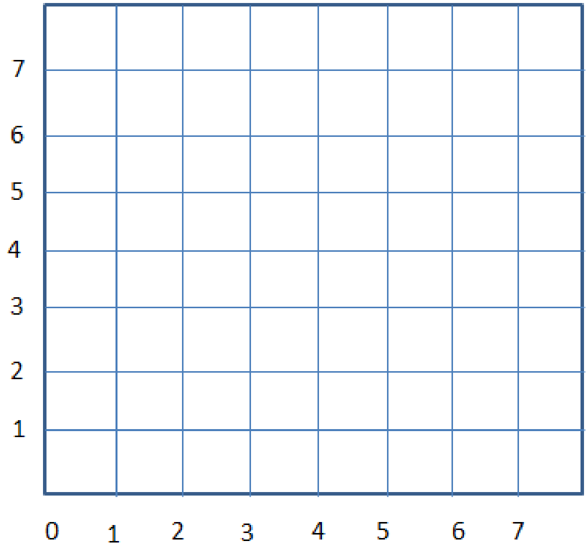


Figura 28: *Sistema de Coordenadas Natural*

Cuando se emplea el sistema de coordenadas natural en el eje X los puntos avanzan hacia la derecha y en el eje Y hacia arriba, estando el origen en la esquina inferior izquierda.

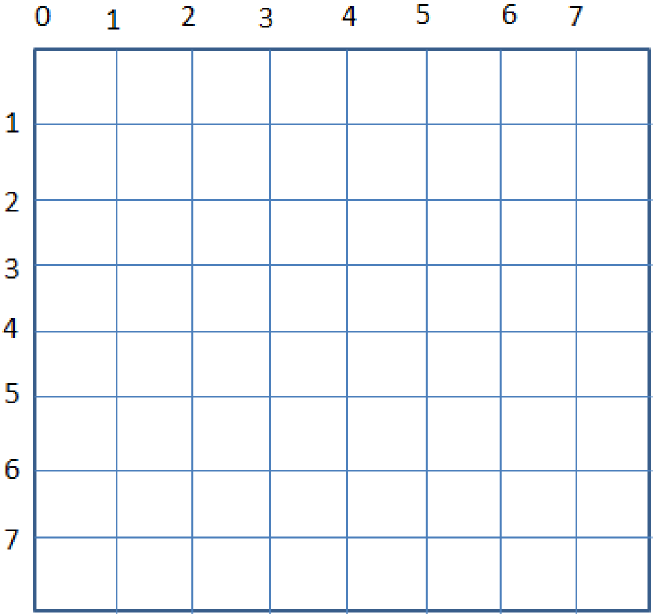


Figura 29: *Sistema de Coordenadas de SDL*

Sin embargo, cuando se emplea el sistema de coordenadas SDLel eje X avanza hacia la derecha, pero el eje Y hacia abajo. En este sistema de coordenadas el origen está en la esquina superior izquierda.

Por lo tanto, cuando se renderice el rectángulo sólido el sistema de coordenada funcionará así:

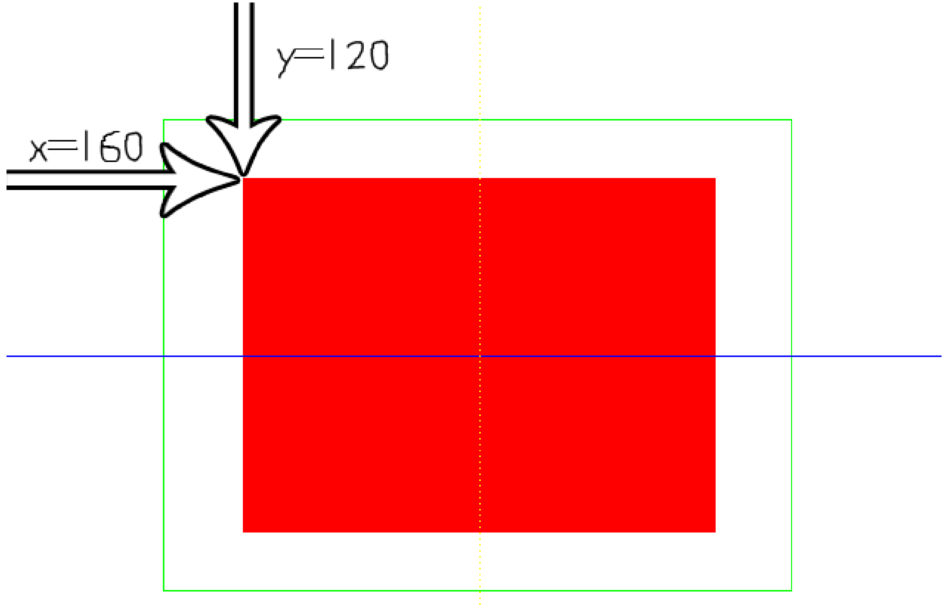


Figura 30: *Origen Sistema de Coordenadas SDL*

```
//Draw blue horizontal line
SDL_SetRenderDrawColor( gRenderer, 0x00, 0x00, 0xFF, 0xFF );

SDL_RenderDrawLine( gRenderer, 0, SCREEN_HEIGHT / 2, SCREEN_WIDTH,
SCREEN_HEIGHT / 2 );
```

Éste es el código para dibujar una línea delgada de píxeles usando **SDL_RenderDrawLine**.

En primer lugar, se establece el color azul y a continuación se hace la llamada al renderizado dando la posición de inicio (X, Y) y fin (X, Y). Estas posiciones hacen que la línea sea horizontalmente recta a través de la pantalla.

```
//Draw vertical line of yellow dots
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0x00, 0xFF );

for( int i = 0; i < SCREEN_HEIGHT; i += 4 )

{
SDL_RenderDrawPoint( gRenderer, SCREEN_WIDTH / 2, i );
}

//Update screen
SDL_RenderPresent( gRenderer );
```

La última parte de la geometría que se renderiza es una secuencia de puntos empleando **SDL_RenderDrawPoint**. Tras haber dibujado toda la geometría se actualiza la pantalla.

En la llamada a **SDL_SetRenderDrawColor** se usa rojo 255 y verde 255 que se combinan juntos para dar lugar al amarillo. Si no estuviera la llamada, la pantalla sería del último color establecido por **SDL_SetRenderDrawColor**, resultando el fondo, en este caso, amarillo.

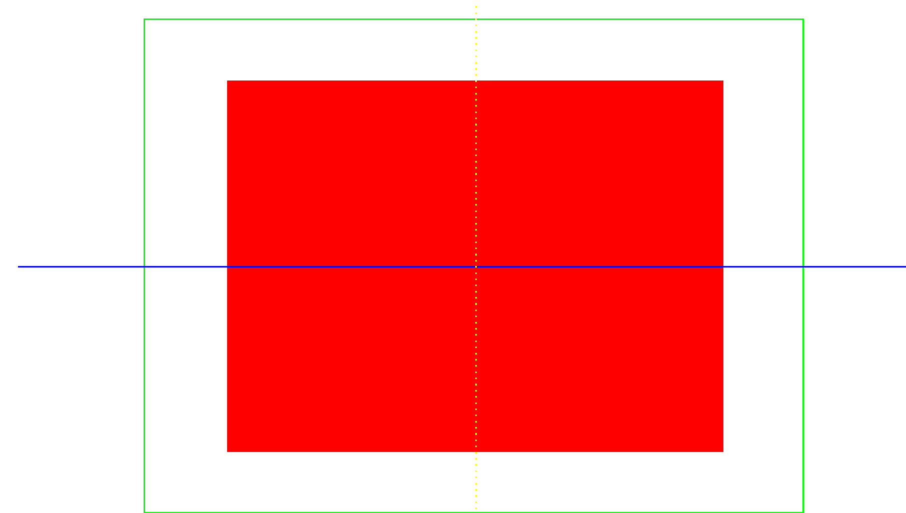


***Figura 31**: Resultado ejecutar aplicación Renderizado de geometrías*

3.10 Ventana de Visualización / Ventana Gráfica

En determinadas ocasiones sólo se quiere renderizar parte de la pantalla para fines como minimaps. Haciendo uso de la ventana gráfica puede controlarse dónde se renderiza en la pantalla.

```
//Top left corner viewport

SDL_Rect topLeftViewport;

topLeftViewport.x = 0;
topLeftViewport.y = 0;
topLeftViewport.w = SCREEN_WIDTH / 2;
topLeftViewport.h = SCREEN_HEIGHT / 2;

SDL_RenderSetViewport(gRenderer,&topLeftViewport);

//Render texture to screen
SDL_RenderCopy( gRenderer, gTexture, NULL, NULL );
```

Tras limpiar la pantalla se empieza a dibujar. Habrá 3 secciones en las que se va a dibujar una imagen a pantalla completa.

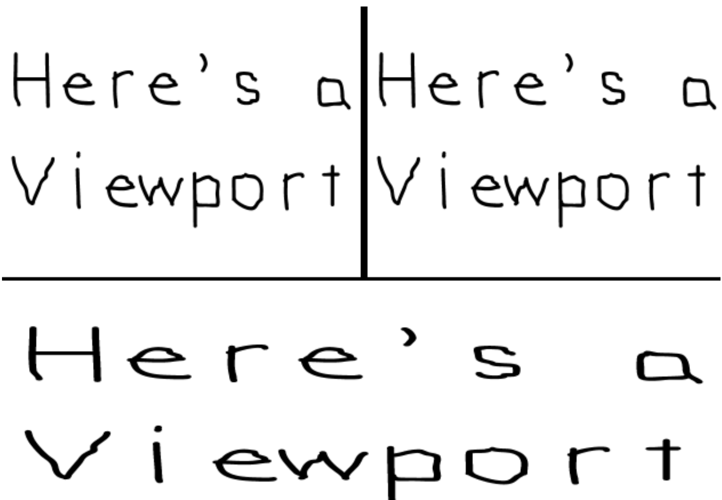


***Figura 32**: Secciones Ventana Gráfica*

En primer lugar, se renderizará la esquina superior izquierda. Es tan fácil como crear un rectángulo con la mitad de la anchura y altura que la pantalla, pasando esta región con **SDL_RenderSetViewport**. Cualquier renderizado hecho después de llamar a la función renderizará dentro de esta región definida por la ventana gráfica determinada.

También se utilizará el sistema de coordenadas de la ventana creada, por lo que la parte inferior de la ventana creada seguirá siendo Y=480 aunque sólo esté a 240 píxeles hacia abajo desde la parte superior.

```
//Top right viewport

SDL_Rect topRightViewport;

topRightViewport.x = SCREEN_WIDTH / 2;
topRightViewport.y = 0;
topRightViewport.w = SCREEN_WIDTH / 2;
topRightViewport.h = SCREEN_HEIGHT / 2;

SDL_RenderSetViewport(gRenderer,&topRightViewport);

//Render texture to screen
SDL_RenderCopy( gRenderer, gTexture, NULL, NULL );
```

A continuación, se define la zona superior derecha y se dibuja. Es muy similar a la anterior, con la salvedad de que ahora la coordenada X está en la mitad de la pantalla.

```
//Bottom viewport

SDL_Rect bottomViewport;
bottomViewport.x = 0;
bottomViewport.y = SCREEN_HEIGHT / 2;
bottomViewport.w = SCREEN_WIDTH;
bottomViewport.h = SCREEN_HEIGHT / 2;

SDL_RenderSetViewport(gRenderer,&bottomViewport);

//Render texture to screen
SDL_RenderCopy( gRenderer, gTexture, NULL, NULL );

//Update screen
SDL_RenderPresent( gRenderer );
```

Finalmente, se renderiza la mitad inferior de la pantalla. Una vez más, la ventana gráfica que se emplee utilizará el mismo sistema de coordenadas, por lo que la imagen aparecerá "aplastada", ya que la ventana gráfica tiene la mitad de altura.

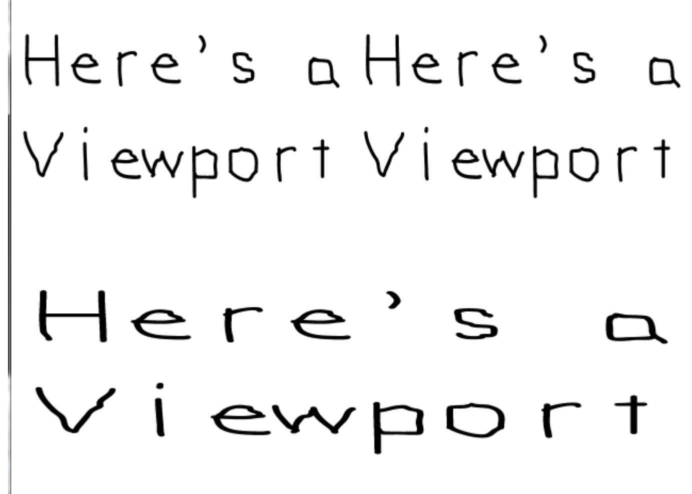


***Figura 33**: Resultado ejecutar aplicación Ventana Gráfica*

3.11 Color

Cuando se renderizan varias imágenes en la pantalla es necesario tener imágenes con fondos transparentes. SDL proporciona una manera fácil para resolver esta cuestión usando la incrustación de color “color keying”.

```
//Texture wrapper class
class LTexture
{
	public:
		//Initializes variables
		LTexture();

		//Deallocates memory
		~LTexture();

		//Loads image at specified path
		bool loadFromFile( std::string path );

		//Deallocates texture
		void free();

		//Renders texture at given point
		void render( int x, int y );

		//Gets image dimensions
		int getWidth();
		int getHeight();

	private:
		//The actual hardware texture
		SDL_Texture* mTexture;

		//Image dimensions
		int mWidth;
		int mHeight;
};
```

Se envolverá SDL_Texture en una clase para que sea más sencillo. Si se deseara obtener cierta información sobre la textura como su anchura o altura, se tendrían que usar algunas funciones SDL para consultar la información de la textura. Sin embargo, se usará una clase para envolver y almacenar la información de la textura.

La clase está formada por un par constructor/destructor, cargador de archivos y renderizador (que adquiere la posición) y funciones para conseguir las dimensiones de la textura.

```
//The window we'll be rendering to
SDL_Window* gWindow = NULL;

//The window renderer
SDL_Renderer* gRenderer = NULL;

//Scene textures
LTexture gFooTexture;
LTexture gBackgroundTexture;
```

En esta escena hay dos texturas que se cargarán declaradas como "gFooTexture" y "gBackgroundTexture".

La imagen de la parte anterior con el fondo coloreado de azul claro:

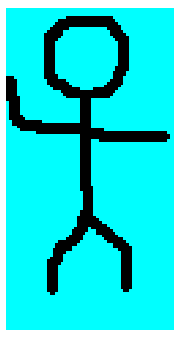


***Figura 34**: Imagen parte anterior*

se renderiza en la parte superior del fondo:

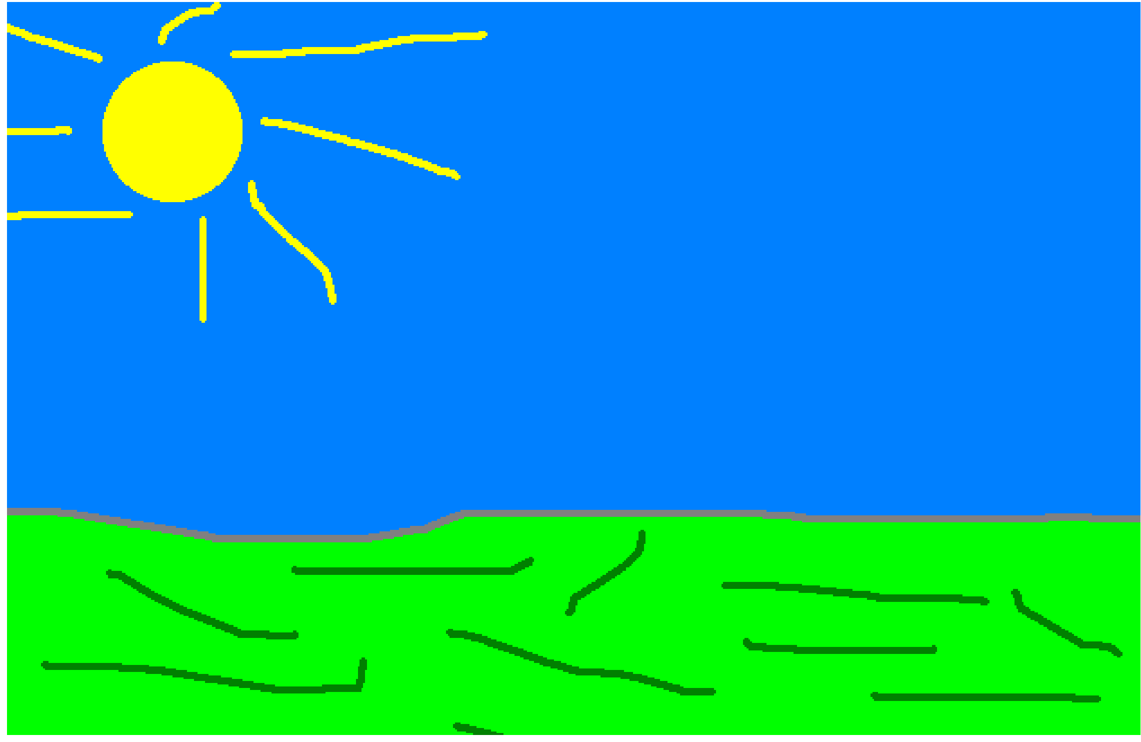


***Figura 35:** fondo*

```
LTexture::LTexture()
{
	//Initialize
	mTexture = NULL;
	mWidth = 0;
	mHeight = 0;
}

LTexture::~LTexture()
{
	//Deallocate
	free();
}
```

El constructor inicializa las variables y el destructor llama al liberador de recursos.

```
bool LTexture::loadFromFile( std::string path )
{
	//Get rid of preexisting texture
	free();
```

En la función de carga: en primer lugar se libera la textura en el caso de que ya haya una que esté cargada.

```
	//The final texture
	SDL_Texture* newTexture = NULL;

	//Load image at specified path
	SDL_Surface* loadedSurface = IMG_Load( path.c_str() );
	if( loadedSurface == NULL )

	{
printf( "Unable to load image %s! SDL_image Error: %s\n",
path.c_str(), IMG_GetError() );
	}

	else
	{

//Color key image

SDL_SetColorKey( loadedSurface, SDL_TRUE, SDL_MapRGB( loadedSurface-
>format, 0, 0xFF, 0xFF ) );
```

A continuación, se colorea la imagen con **SDL_SetColorKey** antes de crear una textura a partir de ella.

- El primer argumento es la superficie a la que se quiere dar color.
- El segundo argumento cubre si se quiere activar el color.
- El último argumento es el pixel al que se quiere dar color.

La forma más transversal para crear un pixel de color RGB es mediante **SDL_MapRGB**.

- El primer argumento es el formato en el que se quiere el pixel.
- Las últimas tres variables son las componentes rojas, verdes y azules para el color que se desea asignar. En este caso se está asignando cyan (0 rojo, 255 verde, 255 azul)

```
//Create texture from surface
newTexture = SDL_CreateTextureFromSurface( gRenderer, loadedSurface );

if( newTexture == NULL )
            {
printf( "Unable to create texture from %s! SDL Error: %s\n",
path.c_str(), SDL_GetError() );
            }

            else
            {
                  //Get image dimensions
                  mWidth = loadedSurface->w;
                  mHeight = loadedSurface->h;
            }

            //Get rid of old loaded surface
            SDL_FreeSurface( loadedSurface );
      }

      //Return success
      mTexture = newTexture;
      return mTexture != NULL;
}
```

Después de dar color a la superficie cargada, se crea una textura de la superficie cargada y coloreada. Si la textura se ha creado con éxito, se almacena la altura y anchura de la textura y se devuelve si la tarea se completa con éxito.

```
void LTexture::free()
{
      //Free texture if it exists
      if( mTexture != NULL )
      {
            SDL_DestroyTexture( mTexture );
            mTexture = NULL;
            mWidth = 0;
            mHeight = 0;
      }
}
```

El liberador de recursos comprueba si existe la textura, la destruye y reinicia las variables miembro.

```
void LTexture::render( int x, int y )
{
	//Set rendering space and render to screen
	SDL_Rect renderQuad = { x, y, mWidth, mHeight };
	SDL_RenderCopy( gRenderer, mTexture, NULL, &renderQuad );
}
```

Cuando se renderizan imágenes a pantalla completa no se necesita especificar la posición, por lo que tan sólo se llama a **SDL_RenderCopy** con los dos últimos argumentos como NULL.

Cuando se renderiza una textura en un lugar determinado hay que especificar un rectángulo de destino que establece la posición (X,Y) así como la altura y anchura. No se puede especificar la altura y anchura sin conocer las dimensiones de la imagen original, por lo que cuando se renderiza la textura, se crea un rectángulo con los argumentos de posición, anchura y altura y se manda este rectángulo a **SDL_RenderCopy**. Por esta razón se necesita una clase contenedora.

```
int LTexture::getWidth()
{
	return mWidth;
}

int LTexture::getHeight()
{
	return mHeight;
}
```

Estas últimas funciones miembro permiten obtener la altura y anchura cuando se necesitan.

```
bool loadMedia()
{
	//Loading success flag
	bool success = true;

	//Load Foo' texture
	if( !gFooTexture.loadFromFile( "10_color_keying/foo.png" ) )
	{
		printf( "Failed to load Foo' texture image!\n" );
		success = false;
	}

	//Load background texture
	if( !gBackgroundTexture.loadFromFile(
"10_color_keying/background.png" ) )
	{
		printf( "Failed to load background texture image!\n" );
		success = false;
	}
	return success;
}
```

Éstas son las funciones de carga de imágenes.

```
void close()
{
	//Free loaded images
	gFooTexture.free();
	gBackgroundTexture.free();

	//Destroy window
	SDL_DestroyRenderer( gRenderer );
	SDL_DestroyWindow( gWindow );
	gWindow = NULL;
	gRenderer = NULL;

	//Quit SDL subsystems
	IMG_Quit();
	SDL_Quit();
}
```

Éstos son los liberadores de recursos.

```
//While application is running
			while( !quit )
			{
				//Handle events on queue
				while( SDL_PollEvent( &e ) != 0 )
				{
					//User requests quit
					if( e.type == SDL_QUIT )
					{
						quit = true;
					}
				}

	//Clear screen
	SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
	SDL_RenderClear( gRenderer );

	//Render background texture to screen
	gBackgroundTexture.render( 0, 0 );

	//Render Foo' to the screen
	gFooTexture.render( 240, 190 );

	//Update screen
	SDL_RenderPresent( gRenderer );
			}
```

Éste es el bucle principal con las texturas renderizadas. Es un bucle básico que se encarga de los sucesos, borra la pantalla, renderiza el fondo, la figura en la parte superior y actualiza la pantalla.

Hay que tener en cuenta que el orden importa cuando se están renderizando múltiples elementos. Si se renderizara en primer lugar la figura del hombre, el fondo se renderizaría sobre él y no se podría ver.

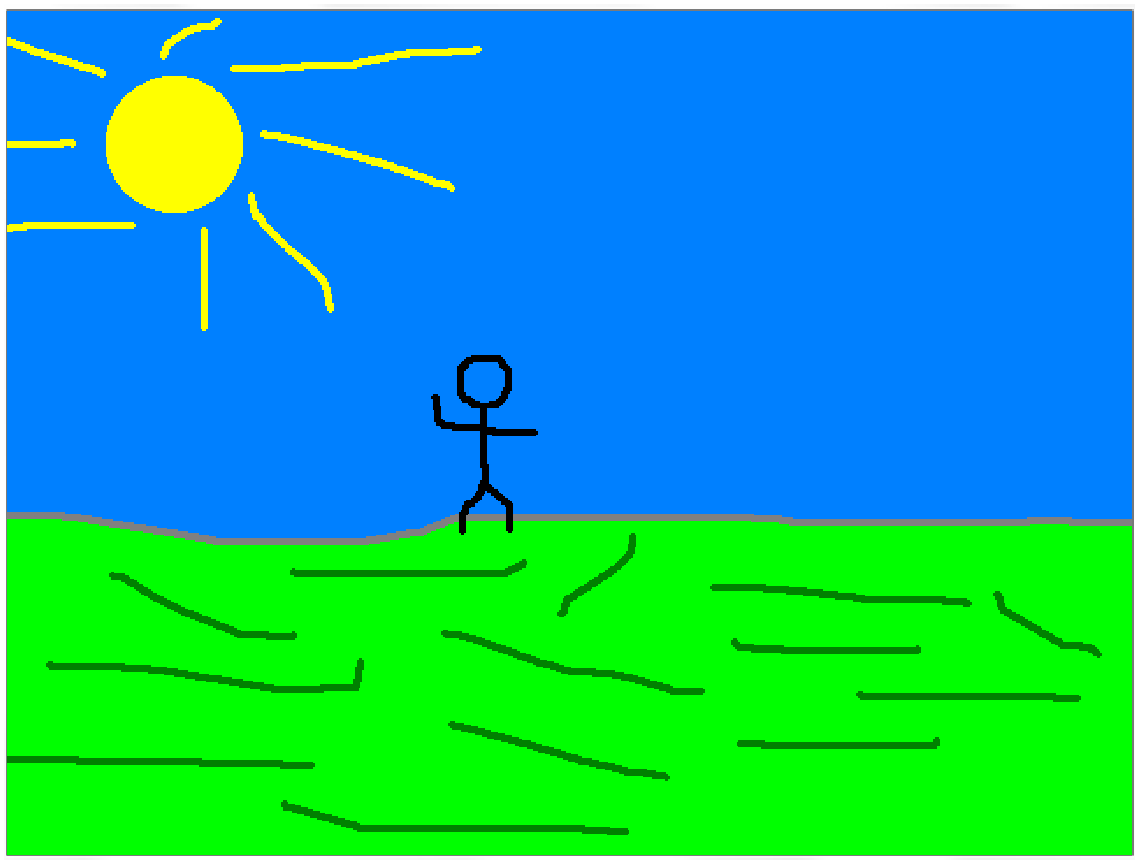


Figura 36: *Resultado ejecutar aplicación Color*

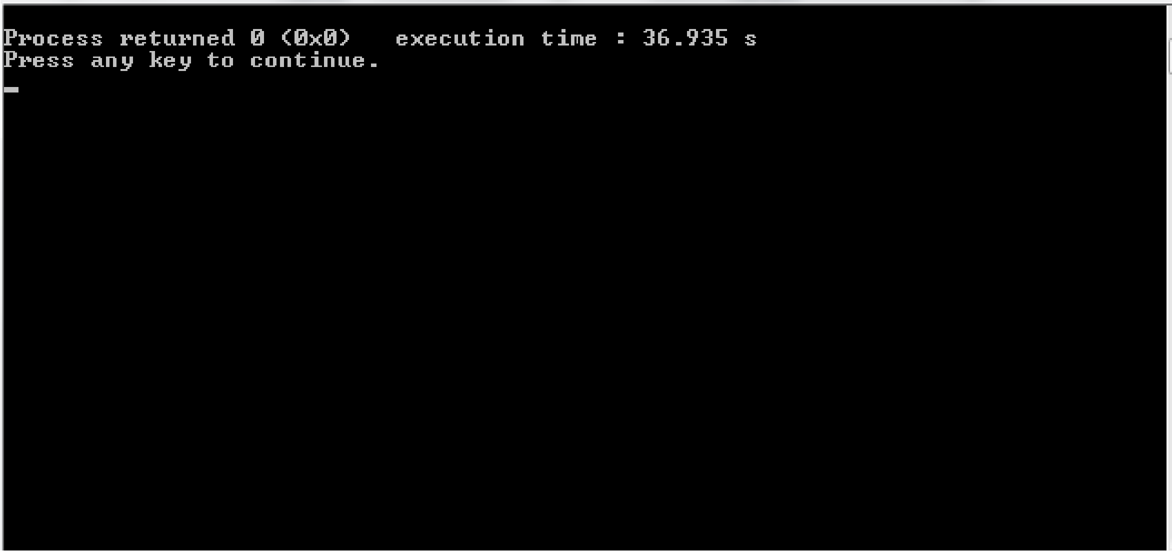


Figura 37: *Resultado ejecutar aplicación Color (II)*

3.12 Clip Rendering - Sprite Sheets

En determinadas ocasiones sólo se desea renderizar parte de una textura. Haciendo uso de clip rendering puede definirse la parte de la textura que se quiere renderizar en lugar de renderizarla por completo.

```
//Texture wrapper class
class LTexture
{
	public:
		//Initializes variables
		LTexture();

		//Deallocates memory
		~LTexture();

		//Loads image at specified path
		bool loadFromFile( std::string path );

		//Deallocates texture
		void free();

		//Renders texture at given point
		void render(int x,int y,SDL_Rect*clip=NULL);

		//Gets image dimensions
		int getWidth();
		int getHeight();

	private:
		//The actual hardware texture
		SDL_Texture* mTexture;

		//Image dimensions
		int mWidth;
		int mHeight;
};
```

La función de renderizado acepta ahora una definición de rectángulo que contiene la parte de la textura que se quiere renderizar. Se le asigna un argumento por defecto NULL en caso de que se quiera renderizar la textura completa.

```
//Scene sprites
SDL_Rect gSpriteClips[ 4 ];
LTexture gSpriteSheetTexture;
```

Se usará la siguiente sprite sheet:

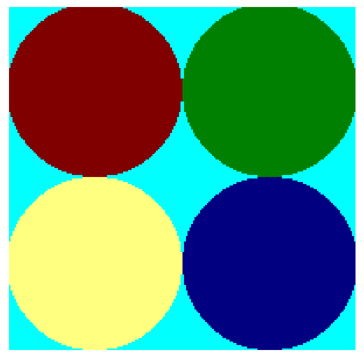


Figura 38: *Sprite sheet*

Y se renderizará cada sprite en una esquina diferente:

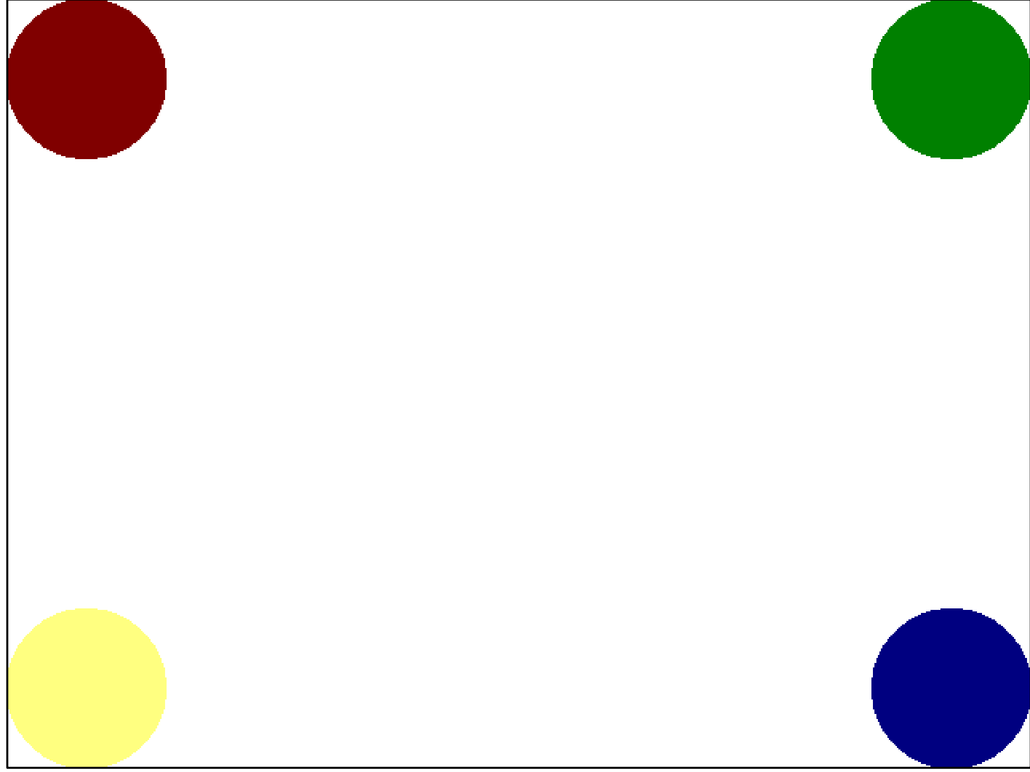


Figura 39: *Sprite sheet (II)*

Para ello se necesitará una imagen de textura y cuatro rectángulos para definir los sprites, que son las variables declaradas.

```
void LTexture::render( int x, int y, SDL_Rect* clip )
{
	//Set rendering space and render to screen
	SDL_Rect renderQuad = { x, y, mWidth, mHeight };

	//Set clip rendering dimensions
	if( clip != NULL )
```

```
{
            renderQuad.w = clip->w;
            renderQuad.h = clip->h;
      }
      //Render to screen
      SDL_RenderCopy(gRenderer,mTexture,clip,&renderQuad);
}
```

Nueva función para la clase textura que soporta el renderizado de superposición. Es prácticamente la misma función de renderizado que la anterior, pero con dos cambios:

- En primer lugar, dado que se está superponiendo y se están usando las dimensiones del rectángulo superpuesto en lugar de la textura, se establece la anchura/altura del rectángulo de destino al tamaño del rectángulo de superposición.

- En segundo lugar, se hace el rectángulo de superposición con **SDL_RenderCopy** como rectángulo de origen, que define qué parte de la textura se quiere renderizar. Cuando el rectángulo de origen es NULL se renderiza toda la textura.

```
bool loadMedia()
{
      //Loading success flag
      bool success = true;

      //Load sprite sheet texture
      if(!gSpriteSheetTexture.loadFromFile("11.png"))

      {
    printf("Failed to load sprite sheet texture!\n");
    success = false;
      }

      else
      {
            //Set top left sprite
            gSpriteClips[ 0 ].x =   0;
            gSpriteClips[ 0 ].y =   0;
            gSpriteClips[ 0 ].w = 100;
            gSpriteClips[ 0 ].h = 100;

            //Set top right sprite
            gSpriteClips[ 1 ].x = 100;
            gSpriteClips[ 1 ].y =   0;
            gSpriteClips[ 1 ].w = 100;
            gSpriteClips[ 1 ].h = 100;

            //Set bottom left sprite
            gSpriteClips[ 2 ].x =   0;
            gSpriteClips[ 2 ].y = 100;
            gSpriteClips[ 2 ].w = 100;
            gSpriteClips[ 2 ].h = 100;
```

```
//Set bottom right sprite
            gSpriteClips[ 3 ].x = 100;
            gSpriteClips[ 3 ].y = 100;
            gSpriteClips[ 3 ].w = 100;
            gSpriteClips[ 3 ].h = 100;
      }

      return success;
}
```

Se carga la textura y se definen los rectángulos de superposición para los círculos sprite si la textura se ha cargado correctamente.

```
//While application is running

while( !quit )
            {
//Handle events on queue

while( SDL_PollEvent( &e ) != 0 )
      {
            //User requests quit

if( e.type == SDL_QUIT )
      {
            quit = true;
      }

}

//Clear screen
SDL_SetRenderDrawColor(gRenderer,0xFF,0xFF,0xFF,0xFF);
SDL_RenderClear( gRenderer );

//Render top left sprite
gSpriteSheetTexture.render(0,0,&gSpriteClips[ 0 ] );

//Render top right sprite
gSpriteSheetTexture.render(SCREEN_WIDTH-gSpriteClips[1].w,
0, &gSpriteClips[ 1 ] );

//Render bottom left sprite
gSpriteSheetTexture.render(0,SCREEN_HEIGHTgSpriteClips[2].h,
&gSpriteClips[ 2 ] );

//Render bottom right sprite
gSpriteSheetTexture.render(SCREEN_WIDTH-gSpriteClips[3].w,
SCREEN_HEIGHT - gSpriteClips[ 3 ].h, &gSpriteClips[ 3 ] );

//Update screen
SDL_RenderPresent( gRenderer );
```

Por último, en el bucle principal se renderiza la misma textura 4 veces, pero se está renderizando una parte distinta de la sprite sheet en un lugar distinto con cada llamada.

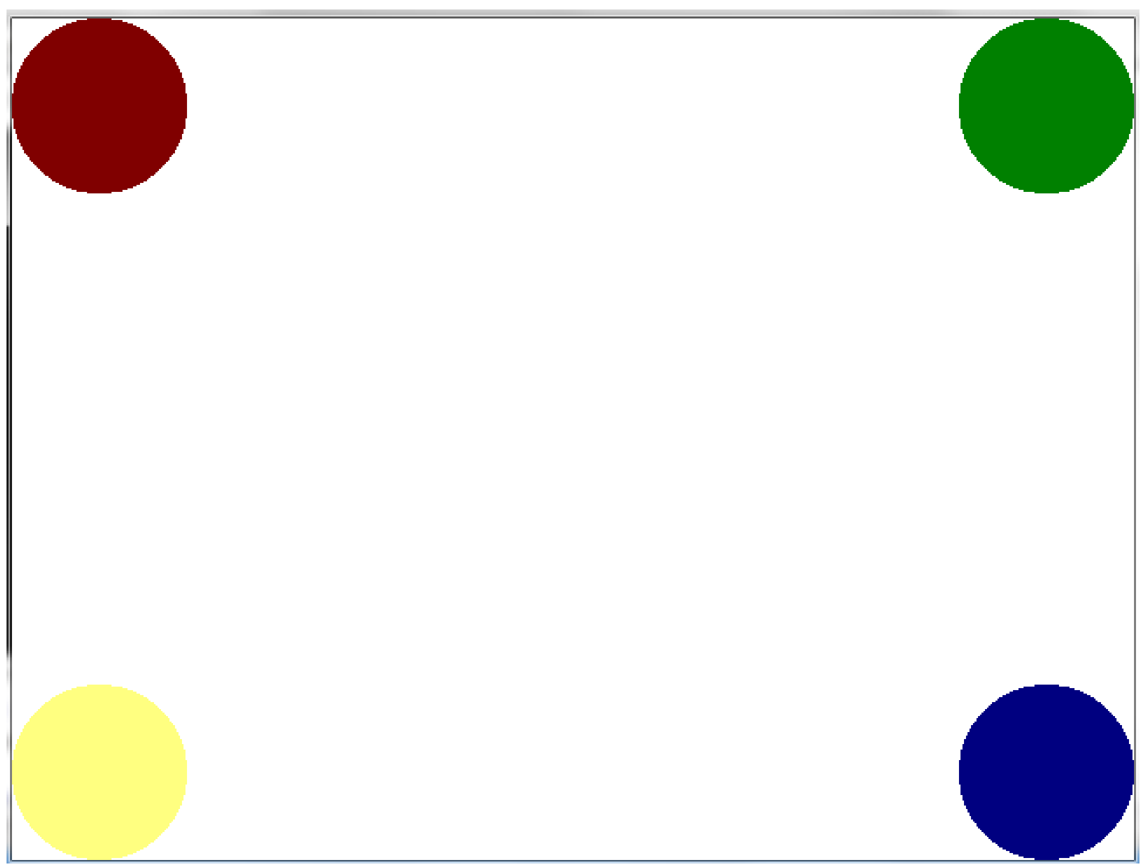


Figura 40: *Resultado ejecutar aplicación Clip Rendering y Sprite Sheets*

```
Process returned 0 (0x0)   execution time : 9.820 s
Press any key to continue.
```

Figura 41: *Resultado ejecutar aplicación Clip Rendering y Sprite Sheets (II)*

3.13 Modulación de Color

La modulación de color permite alterar el color de las texturas renderizadas. El objetivo es modular una textura usando varios colores.

```
//Texture wrapper class

class LTexture
{
      public:

            //Initializes variables
            LTexture();

            //Deallocates memory
            ~LTexture();

            //Loads image at specified path
            bool loadFromFile( std::string path );

            //Deallocates texture
            void free();

            //Set color modulation
            void setColor(Uint8 red,Uint8 green,Uint8 blue);

            //Renders texture at given point
            void render(int x, int y, SDL_Rect*clip = NULL);

            //Gets image dimensions
            int getWidth();
            int getHeight();

      private:

            //The actual hardware texture
            SDL_Texture* mTexture;

            //Image dimensions
            int mWidth;
            int mHeight;
};
```

Se añade una función a la clase contenedora que permitirá que se establezca la modulación de la textura. Todo lo que hay que hacer es tomar componentes de color rojo, verde y azul.

```
void LTexture::setColor(Uint8 red,Uint8 green,Uint8 blue)
{
      //Modulate texture
      SDL_SetTextureColorMod( mTexture,red,green,blue );
}
```

El establecimiento de modulación de la textura se realiza llamando a **SDL_SetTextureColorMod**.

La modulación de color funciona de la siguiente manera:

Partiendo de la siguiente textura:

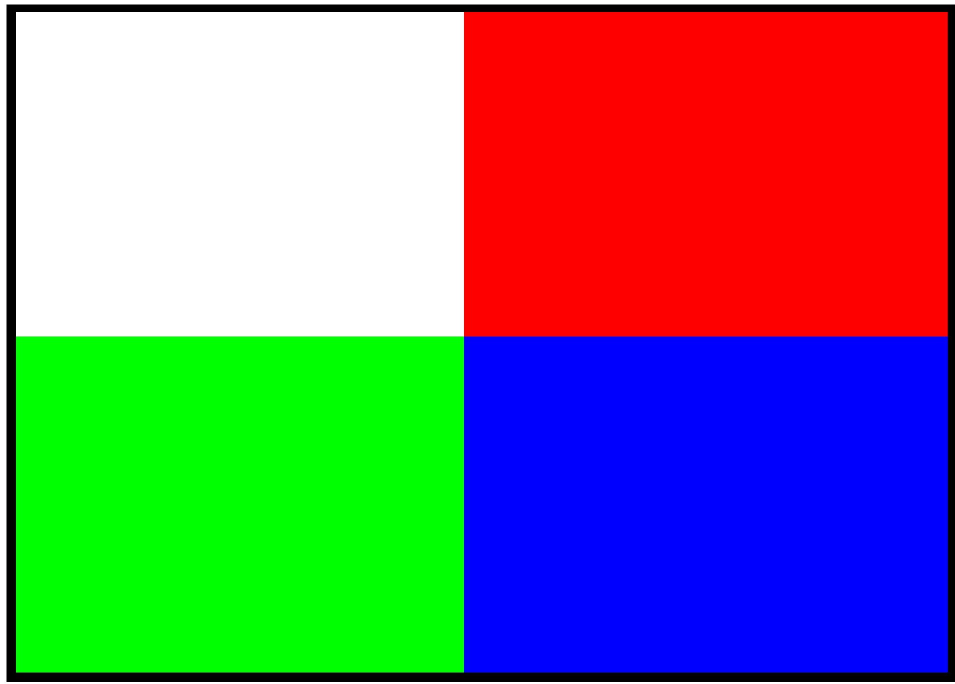


Figura 42: *Modulación de Color (I)*

Se modula con rojo 255, verde 128 y azul 255, obteniéndose:

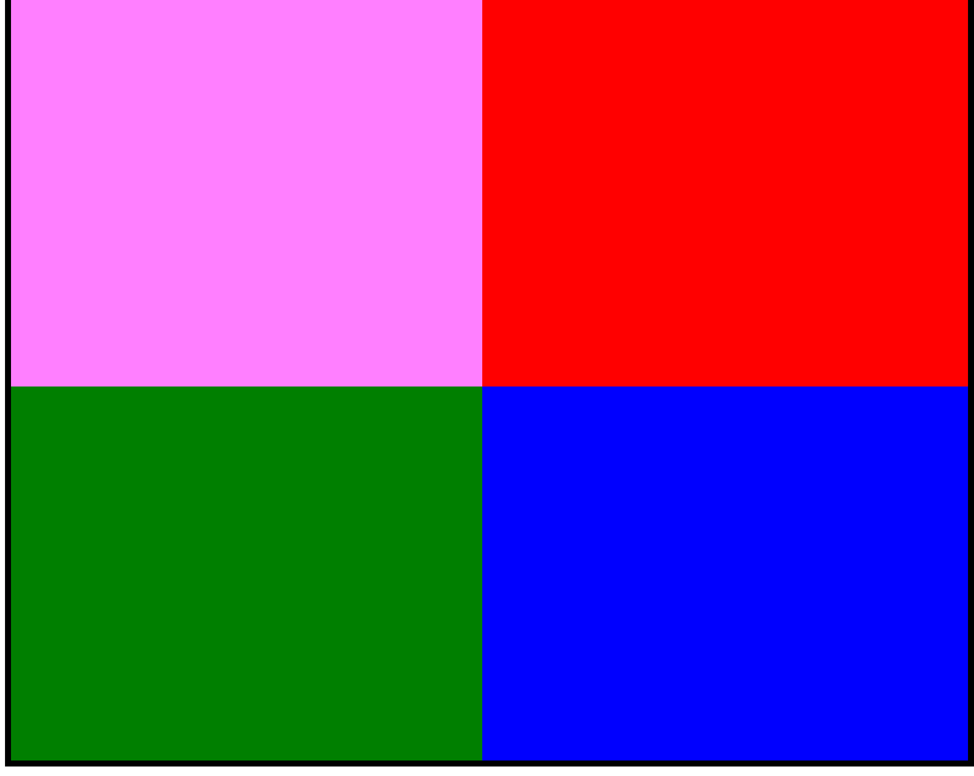


Figura 43: *Modulación de Color (II)*

SDL_SetTextureColorMod acepta Uint8 como argumentos para los componentes de color. Uint8 es un entero sin signo y de 8 bits, lo que significa que irá de 0 hasta 255.

128 está a mitad de camino entre 0 y 255, por lo que al modular el verde a 128 se reduce a la mitad el componente verde de cualquier pixel en la textura.

Los cuadrados rojo y azul no se ven afectados porque no tienen verde en ellos. Sin embargo, el verde se convierte medio brillante y el blanco se vuelve magenta claro (magenta= rojo 255, verde 0, azul 255). La modulación de color es sólo una forma de multiplicar un color a través de toda la textura.

```
//Main loop flag
            bool quit = false;

            //Event handler
            SDL_Event e;

            //Modulation components
            Uint8 r = 255;
            Uint8 g = 255;
            Uint8 b = 255;
```

En este programa se modularán los componentes de color individuales presionando teclas. Para ello, se necesita hacer un seguimiento de los valores de los componentes de color.

```
            //While application is running
            while( !quit )
            {
                //Handle events on queue
                while( SDL_PollEvent( &e ) != 0 )
                {
                    //User requests quit

                    if( e.type == SDL_QUIT )
                    {
                        quit = true;
                    }

                    //On keypress change rgb values
                    else if( e.type == SDL_KEYDOWN )
                    {
                        switch( e.key.keysym.sym )
                        {
                            //Increase red

                            case SDLK_q:
                            r += 32;
                            break;

                            //Increase green

                            case SDLK_w:
                            g += 32;
                            break;
```

```
//Increase blue

case SDLK_e:
b += 32;
break;

//Decrease red

case SDLK_a:
r -= 32;
break;

//Decrease green

case SDLK_s:
g -= 32;
break;

//Decrease blue

case SDLK_d:
b -= 32;
break;
            }
        }
    }
```

En el bucle de eventos las teclas q, w y e aumentan los componentes rojos, verdes y azules y para disminuirlos se tienen las teclas a, s y d. Estas teclas aumentan o disminuyen los componentes en 32, por lo que es notable con cada pulsación.

```
//Clear screen

SDL_SetRenderDrawColor(gRenderer,0xFF,0xFF,0xFF,0xFF);
SDL_RenderClear( gRenderer );

//Modulate and render texture

gModulatedTexture.setColor( r, g, b );
gModulatedTexture.render( 0, 0 );
                                    /
/Update screen
SDL_RenderPresent( gRenderer );
```

Se finaliza estableciendo la modulación de la textura y renderizándola.

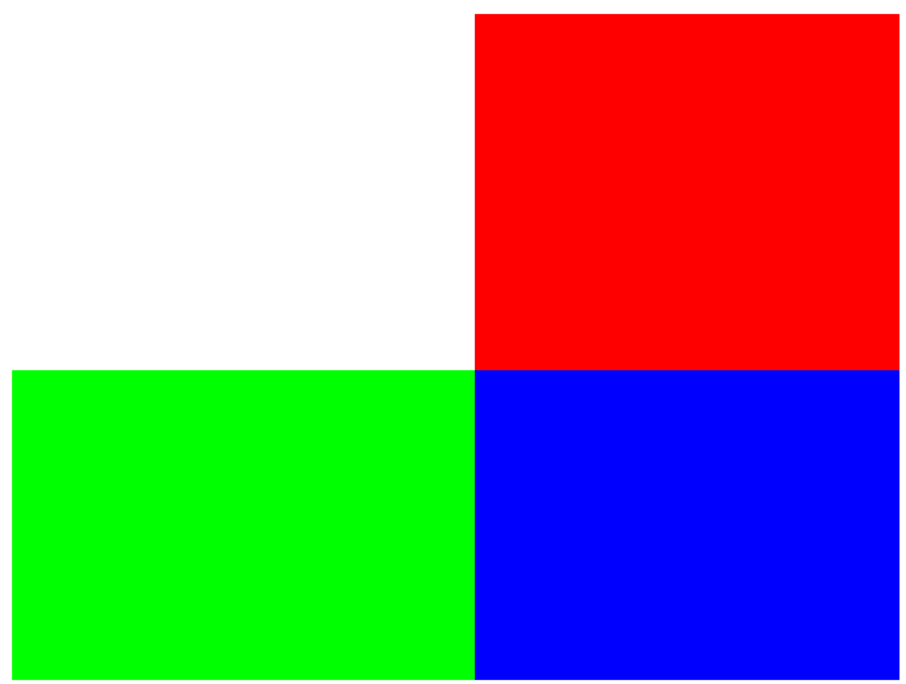


***Figura 44**: Resultado ejecutar aplicación Modulación de Color*

- Cuando se pulsa la tecla Q:

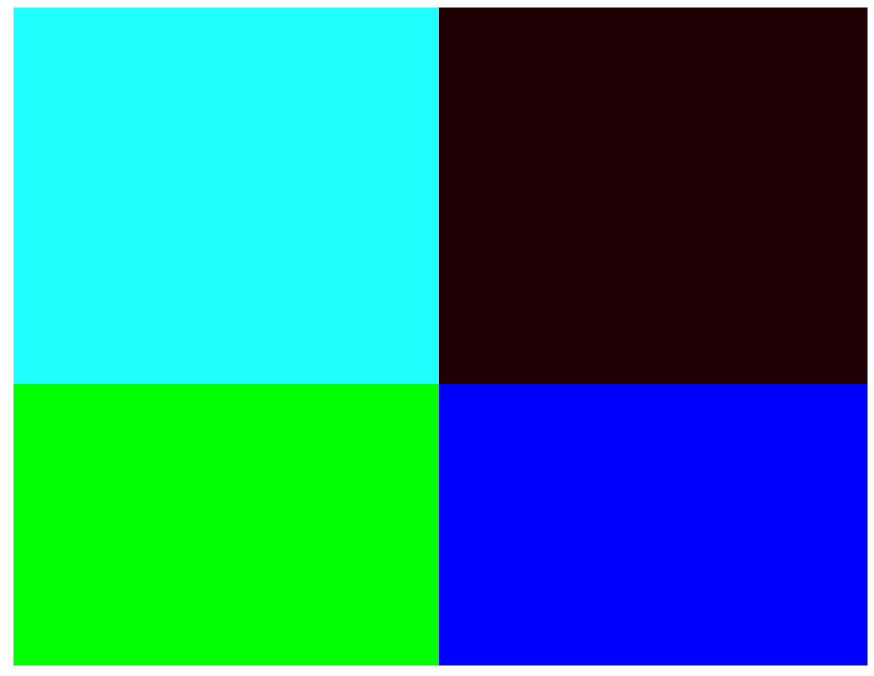


***Figura 45**: Resultado ejecutar aplicación Modulación de Color (II)*

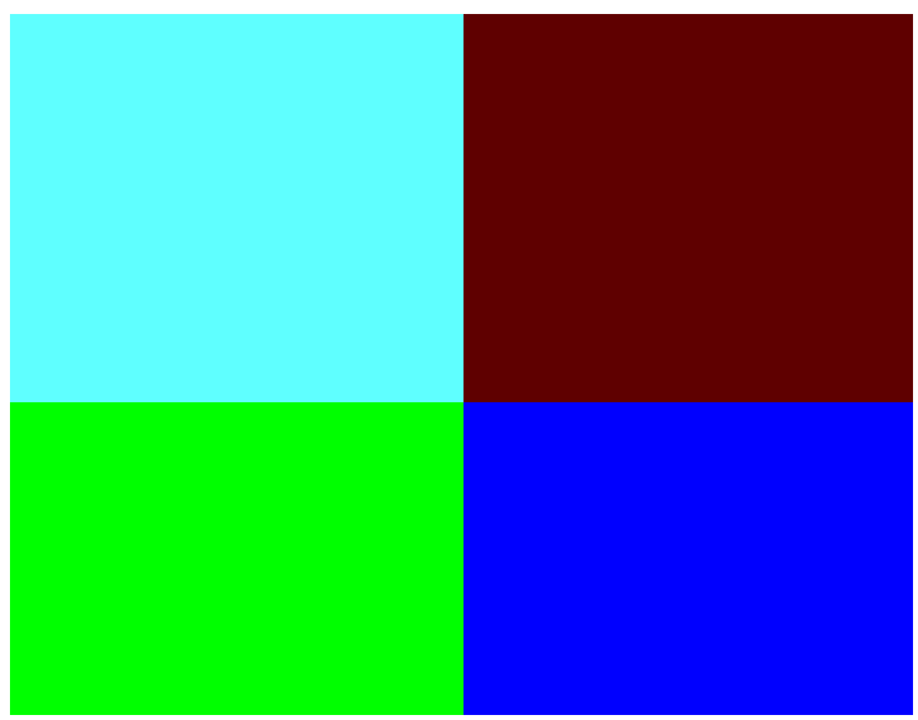


***Figura 46**: Resultado ejecutar aplicación Modulación de Color (III)*

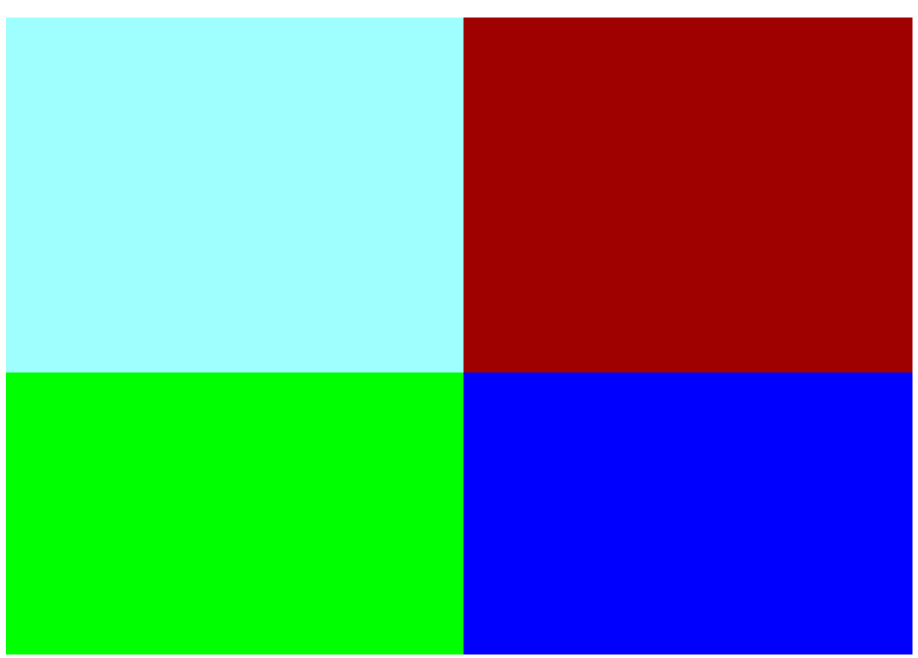


***Figura 47**: Resultado ejecutar aplicación Modulación de Color (IV)*

- Cuando se pulsa la tecla W:

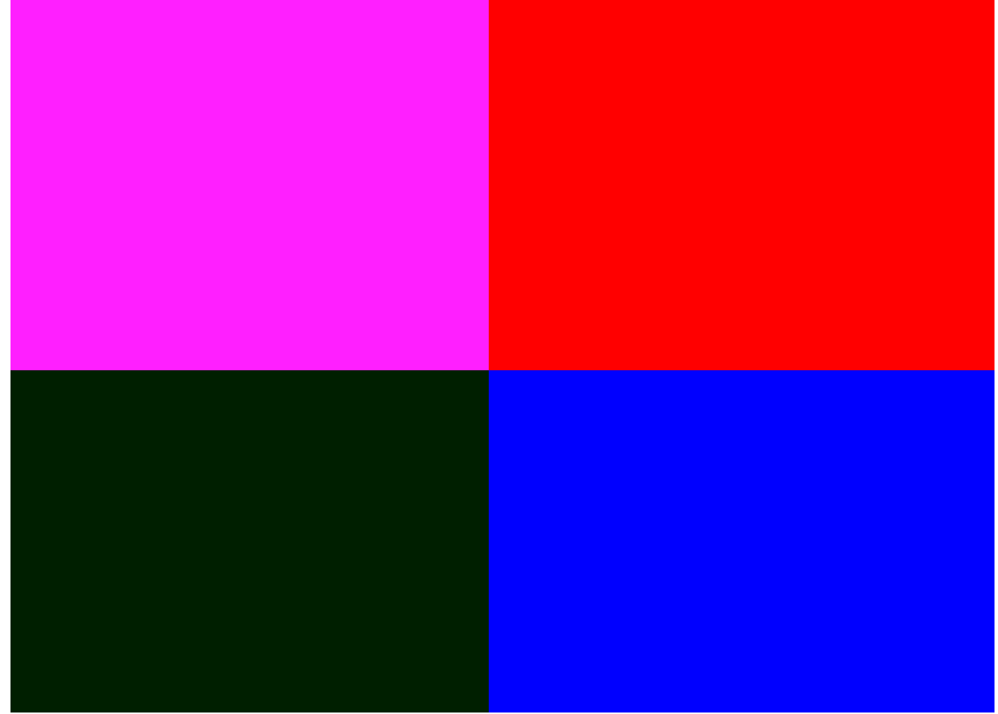


***Figura 48**: Resultado ejecutar aplicación Modulación de Color (V)*

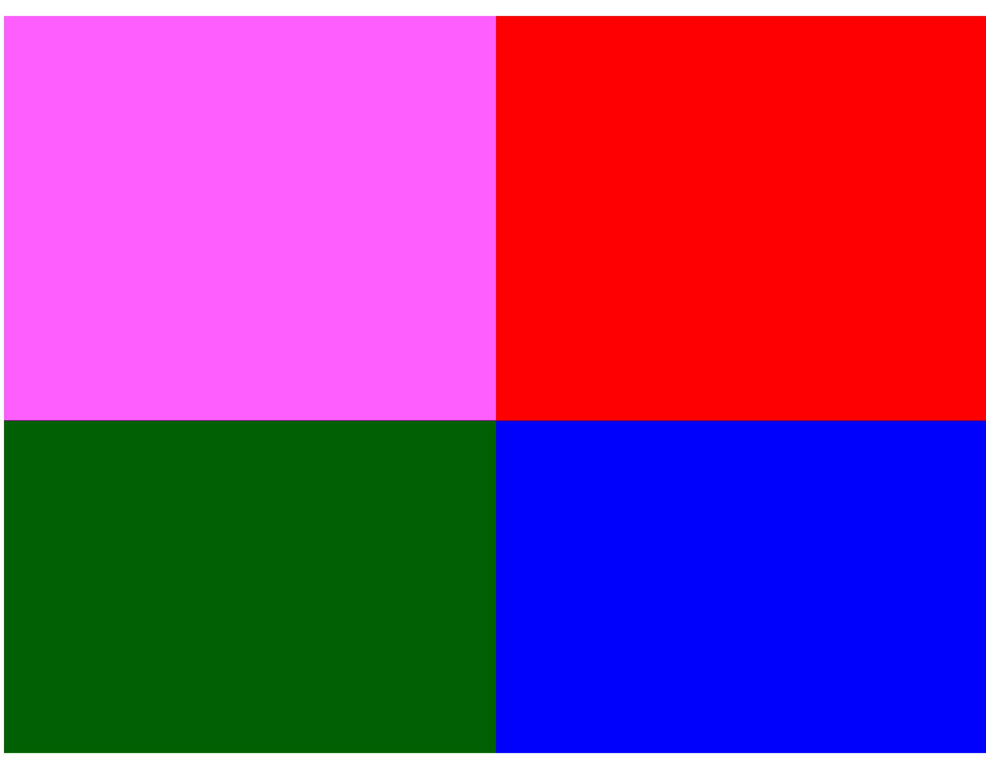


***Figura 49**: Resultado ejecutar aplicación Modulación de Color (VI)*

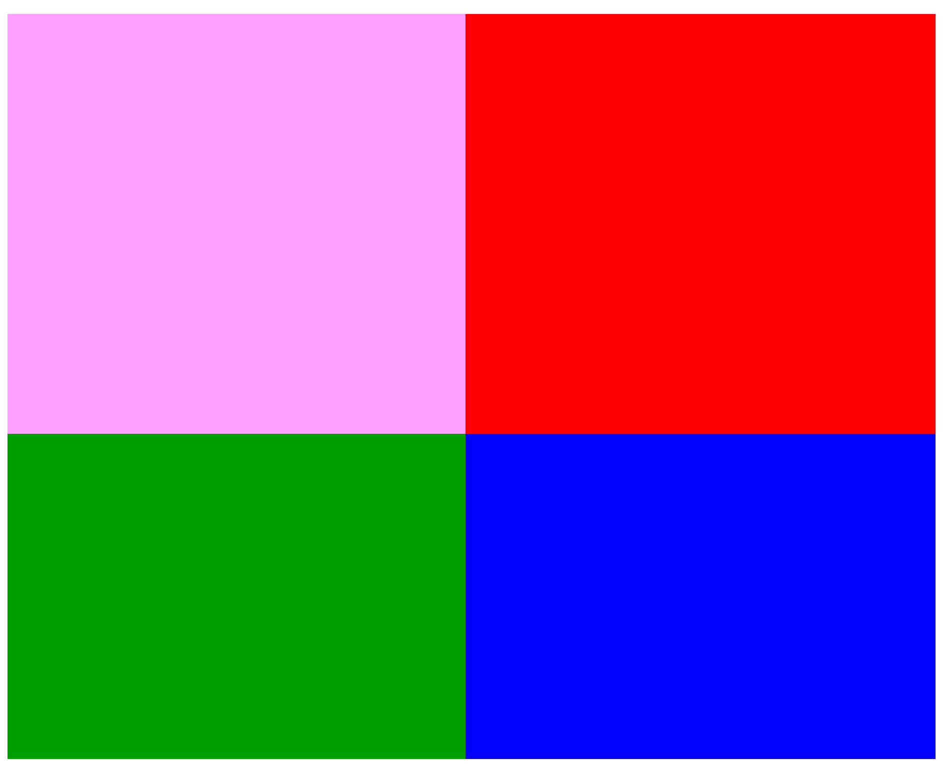


Figura 50*: Resultado ejecutar aplicación Modulación de Color (VII)*

- Cuando se pulsa la tecla E:

Figura 51*: Resultado ejecutar aplicación Modulación de Color (VIII)*

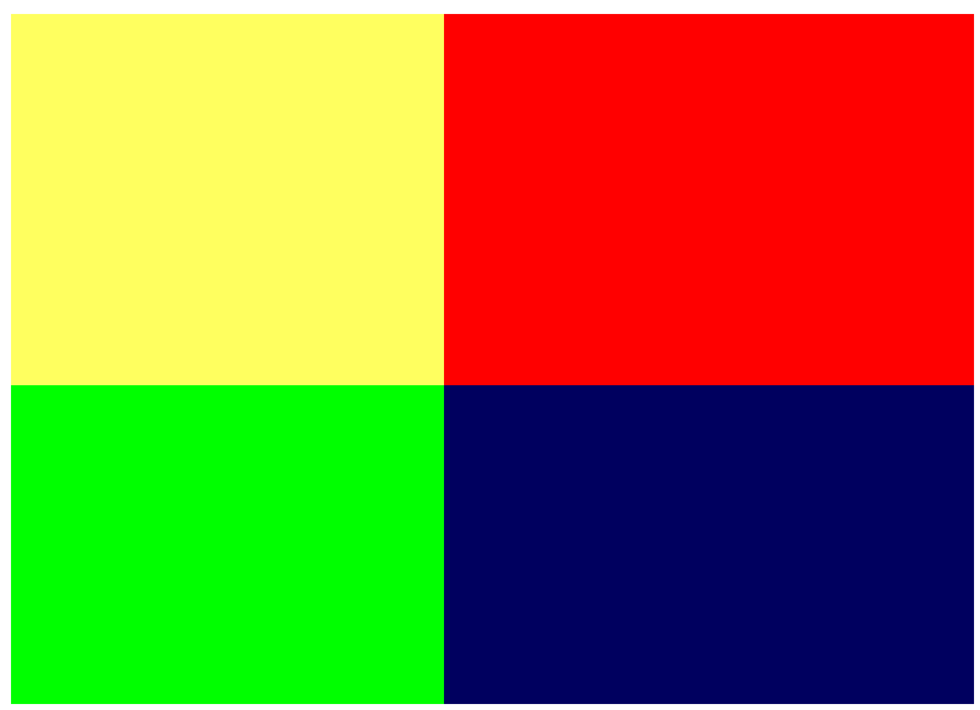


***Figura 52**: Resultado ejecutar aplicación Modulación de Color (IX)*

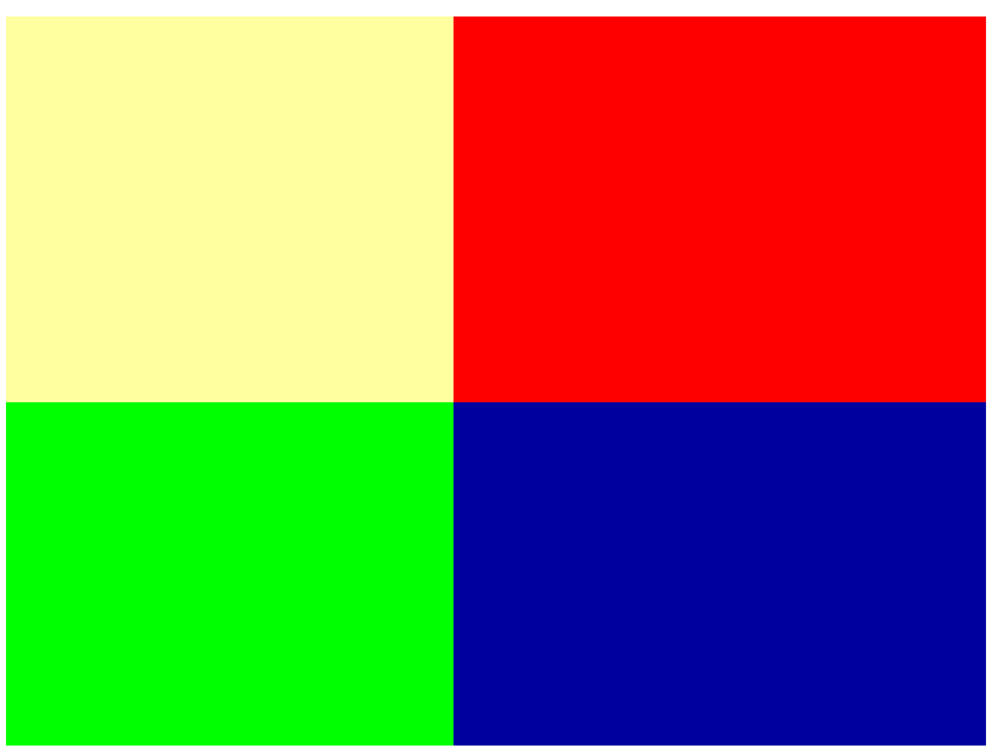


***Figura 53:** Resultado ejecutar aplicación Modulación de Color (X)*

3.14 Composición Alfa

El objetivo es usar la modulación alfa, que trabaja como la modulación de color, para controlar la transparencia de una textura.

```
//Texture wrapper class
class LTexture
{
      public:

            //Initializes variables
            LTexture();

            //Deallocates memory
            ~LTexture();

            //Loads image at specified path
            bool loadFromFile( std::string path );

            //Deallocates texture
            void free();

            //Set color modulation
            void setColor( Uint8 red, Uint8 green, Uint8 blue );

            //Set blending
            void setBlendMode( SDL_BlendMode blending );

            //Set alpha modulation
            void setAlpha( Uint8 alpha );

            //Renders texture at given point
            void render( int x, int y, SDL_Rect* clip = NULL );

            //Gets image dimensions
            int getWidth();
            int getHeight();

      private:

            //The actual hardware texture
            SDL_Texture* mTexture;

            //Image dimensions
            int mWidth;
            int mHeight;
};
```

Se añaden dos funciones para soportar la transparencia alfa en una textura:

- **SetAlpha**, que trabaja de forma muy parecida a setColor.
- **SetBlendMode**, que controlará la forma en la que se mezcla la textura.

Con el fin de conseguir que la composición se haga adecuadamente, se debe establecer el modo de composición de la textura.

```
bool loadMedia()
{
	//Loading success flag
	bool success = true;

	//Load front alpha texture
if( !gModulatedTexture.loadFromFile( "13_alpha_blending/fadeout.png" )
)
	{
		printf( "Failed to load front texture!\n" );
		success = false;
	}

	else

	{
		//Set standard alpha blending
		gModulatedTexture.setBlendMode( SDL_BLENDMODE_BLEND );
	}

//Load background texture
if( !gBackgroundTexture.loadFromFile( "13_alpha_blending/fadein.png" )
)
	{
		printf( "Failed to load background texture!\n" );
		success = false;
	}

	return success;
}
```

En la función de carga de textura se cargará la textura frontal y se le hará la composición alfa con una textura de fondo. Como la textura de delante se volverá más transparente, en el resultado se podrá ver más la textura que hace de fondo.

Tras cargar correctamente la textura frontal, se establece **SDL_BlendMode** para realizar la composición. Así, la composición está activada. Ya que el fondo no será transparente, no es necesario establecer la composición en él.

Alfa representa la opacidad y cuanta menos opacidad, más se podrá ver. Al igual que los componentes de color rojo, verde y azul, cuando modula también va de 0 a 255.

- **Imagen frontal a 255 (100% alfa)**

Press s to
fade out

__Figura 54:__ Resultado ejecutar aplicación Composición Alfa

- **Imagen frontal a 191 (75% alfa)**

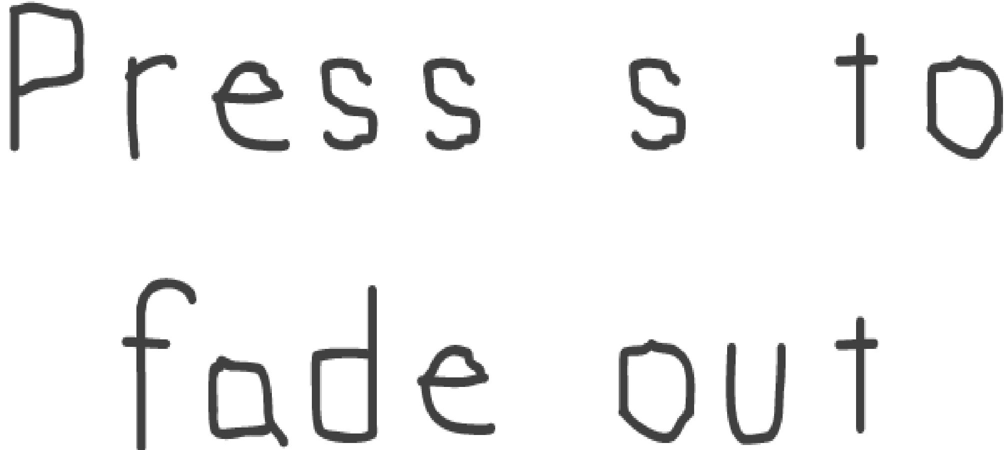


__Figura 55:__ Resultado ejecutar aplicación Composición Alfa (II)

- **Imagen frontal a 127 (50% alfa)**

Figura 56: *Resultado ejecutar aplicación Composición Alfa (III)*

- **Imagen frontal a 63 (25% alfa)**

Figura 57: *Resultado ejecutar aplicación Composición Alfa (IV)*

- **Imagen frontal a 0 (0% alfa)**

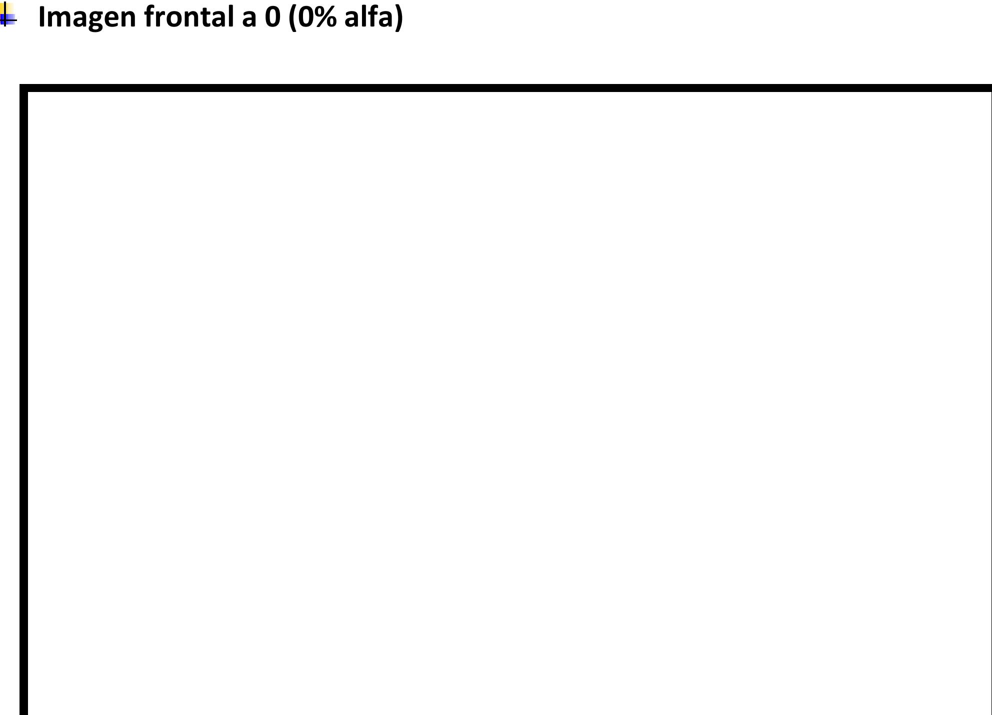


Figura 58: *Resultado ejecutar aplicación Composición Alfa (V)*

Cuanto más bajo sea el nivel de alfa, más transparente será la imagen.

```
void LTexture::setBlendMode( SDL_BlendMode blending )
{
	//Set blending function
	SDL_SetTextureBlendMode( mTexture, blending );
}

void LTexture::setAlpha( Uint8 alpha )
{
	//Modulate texture alpha
	SDL_SetTextureAlphaMod( mTexture, alpha );
}
```

SDL_SetTextureBlendMode en **setBlendMode** permite activar la composición y **SDL_SetTextureAlphaMod** permite establecer la cantidad de alfa para toda la textura.

```
			//Main loop flag
			bool quit = false;

			//Event handler
			SDL_Event e;

			//Modulation component
			Uint8 a = 255;
```

```
                    //While application is running
                    while( !quit )
                    {
```

Justo antes de entrar en el bucle principal, se declara una variable para controlar qué cantidad de alfa tiene la textura. Se inicializa a 255, de manera que la textura frontal se inicia completamente opaca.

```
//Handle events on queue

while( SDL_PollEvent( &e ) != 0 )
      {

//User requests quit

if( e.type == SDL_QUIT )
{
      quit = true;
}

//Handle key presses

else if( e.type == SDL_KEYDOWN )
{

//Increase alpha on w
if( e.key.keysym.sym == SDLK_w )

{

//Cap if over 255
if( a + 32 > 255 )

{
a = 255;
}

//Increment otherwise
else

{
a += 32;
}

}

//Decrease alpha on s
else if( e.key.keysym.sym == SDLK_s )

{

//Cap if below 0
if( a - 32 < 0 )
{
a = 0;
}
```

```
//Decrement otherwise

else

{
a -= 32;
}
```

El bucle de eventos se encarga de parar los eventos y hacer que el valor de alfa suba o baje con las teclas w y s.

```
//Clear screen
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render background
gBackgroundTexture.render( 0, 0 );

//Render front blended
gModulatedTexture.setAlpha( a );
gModulatedTexture.render( 0, 0 );

//Update screen
SDL_RenderPresent( gRenderer );
```

El renderizado se hace al final del bucle principal. Tras limpiar la pantalla, se renderiza el fondo en primer lugar y, en segundo lugar, se renderiza la textura frontal modulada sobre él. Justo antes de renderizar la textura frontal se establece el valor alfa.

Al ejecutar el programa se obtiene:

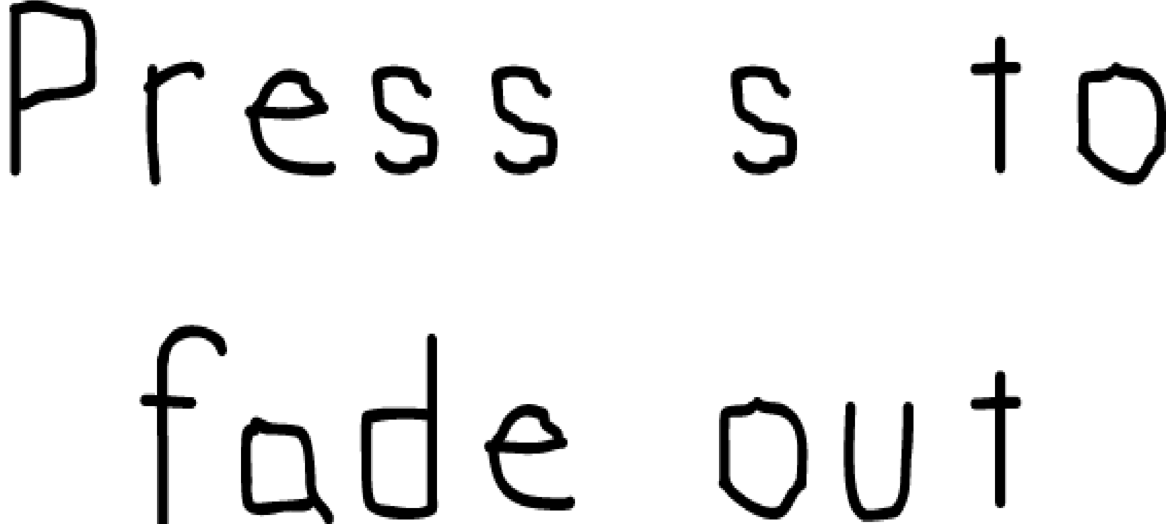


Figura 59: *Resultado ejecutar aplicación Composición Alfa (VI)*

- **Tras pulsar S:**

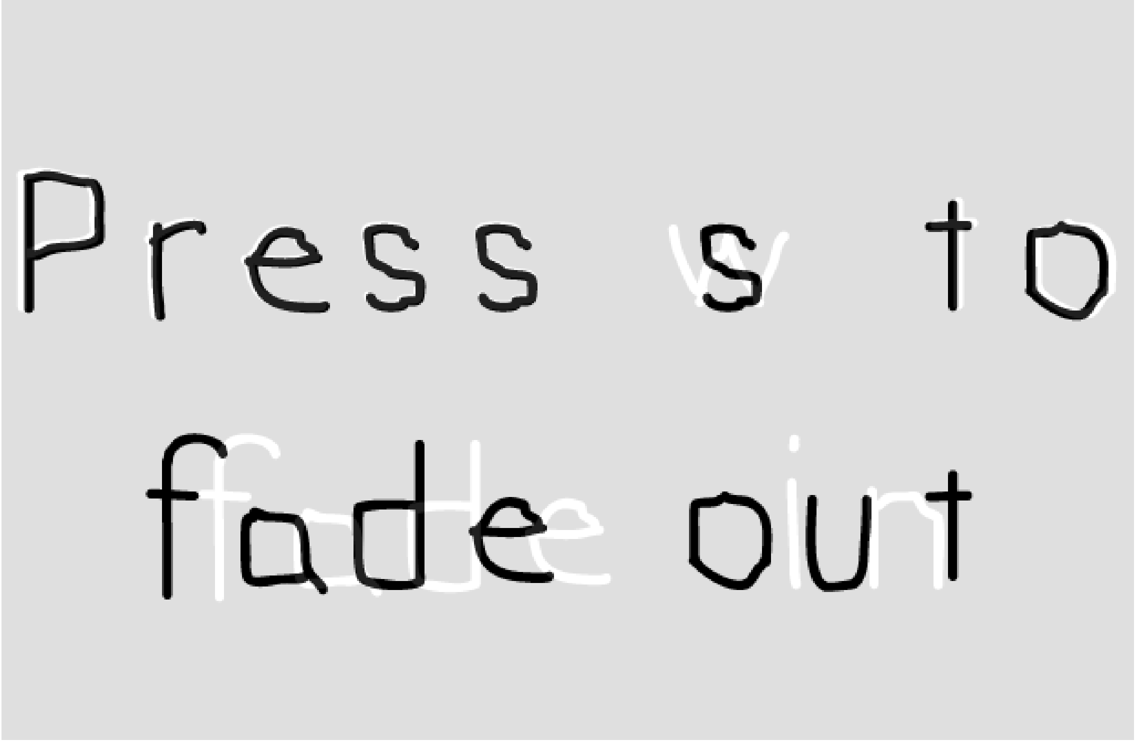


Figura 60: *Resultado ejecutar aplicación Composición Alfa (VII)*

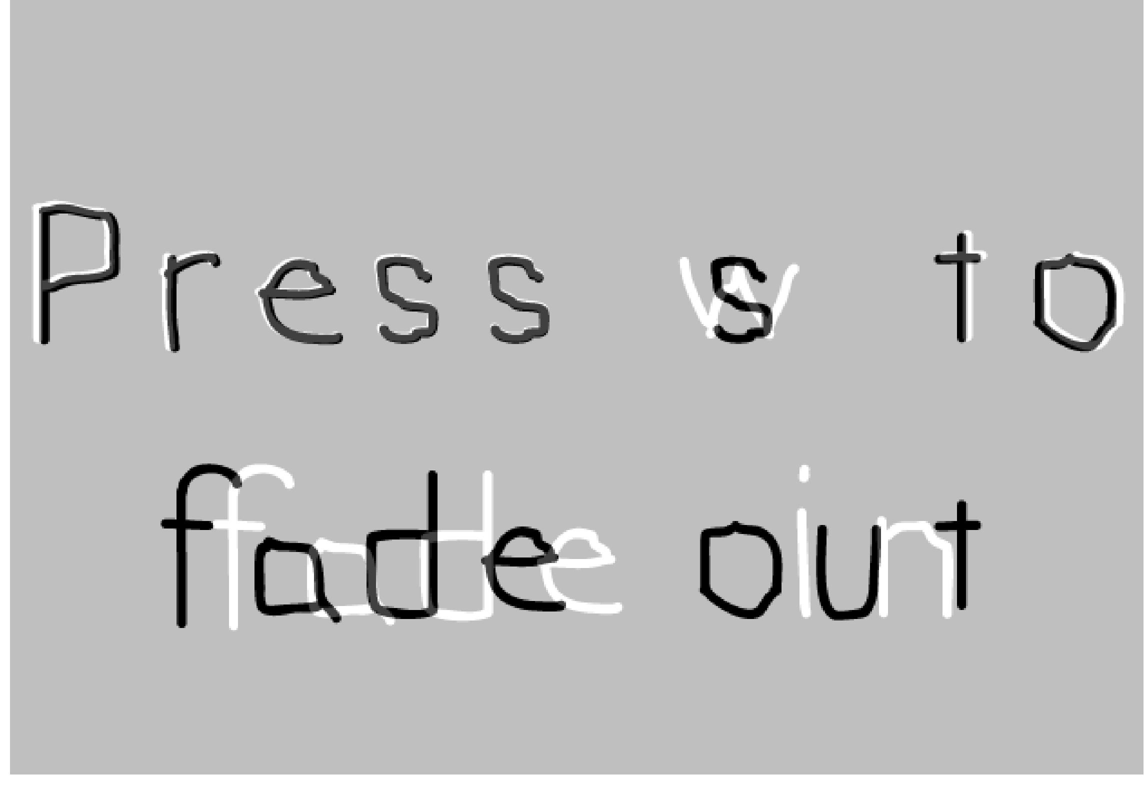


Figura 61: *Resultado ejecutar aplicación Composición Alfa (VIII)*

Figura 62: *Resultado ejecutar aplicación Composición Alfa (IX)*

Hasta que finalmente se consigue:

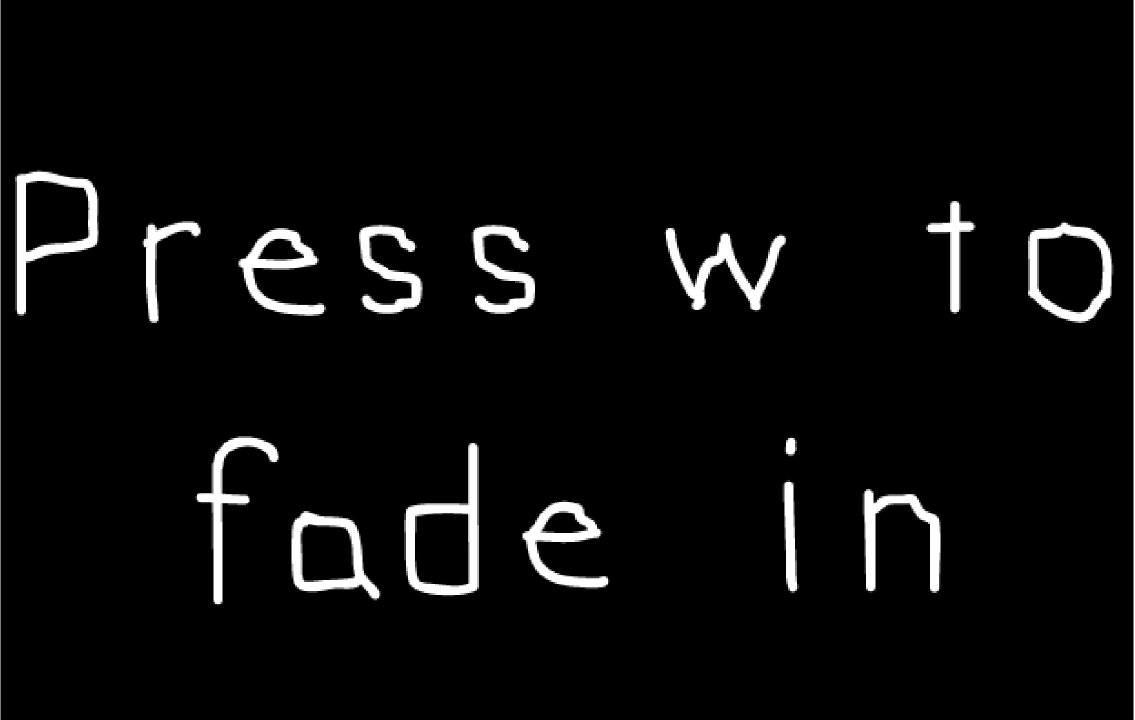


Figura 63: *Resultado ejecutar aplicación Composición Alfa (X)*

- **Si se pulsa W la pantalla se irá aclarando:**

Figura 64: *Resultado ejecutar aplicación Composición Alfa (XI)*

Figura 65: *Resultado ejecutar aplicación Composición Alfa (XII)*

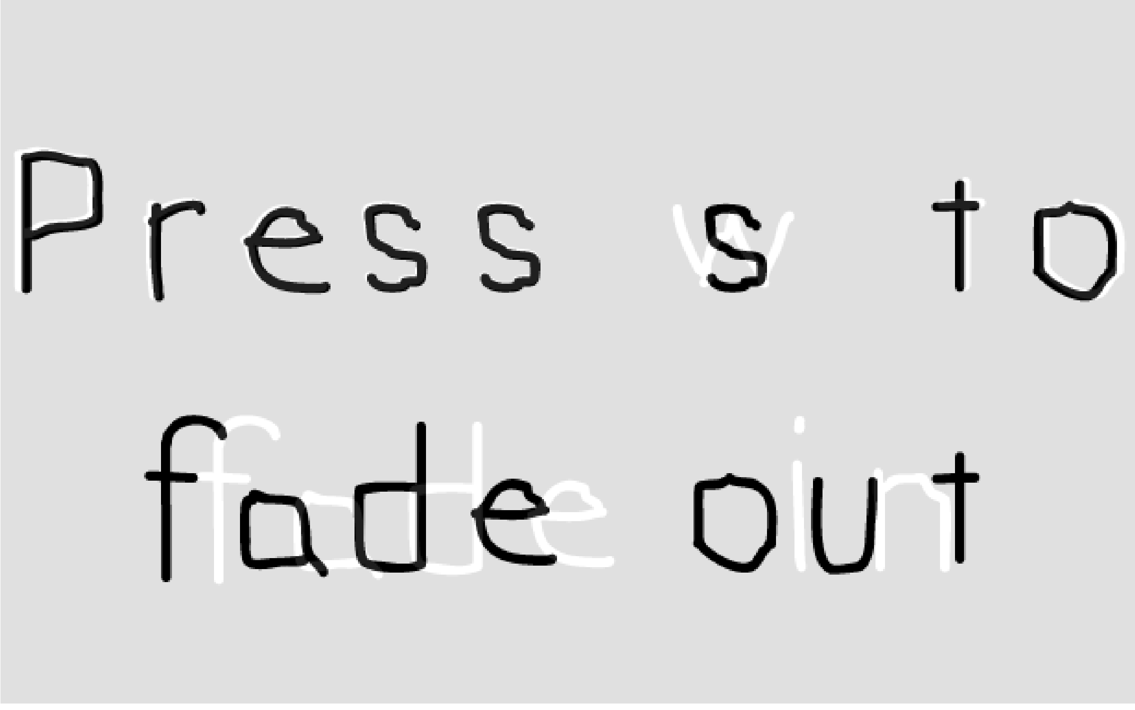


Figura 66: *Resultado ejecutar aplicación Composición Alfa (XIII)*

Hasta obtener de nuevo:

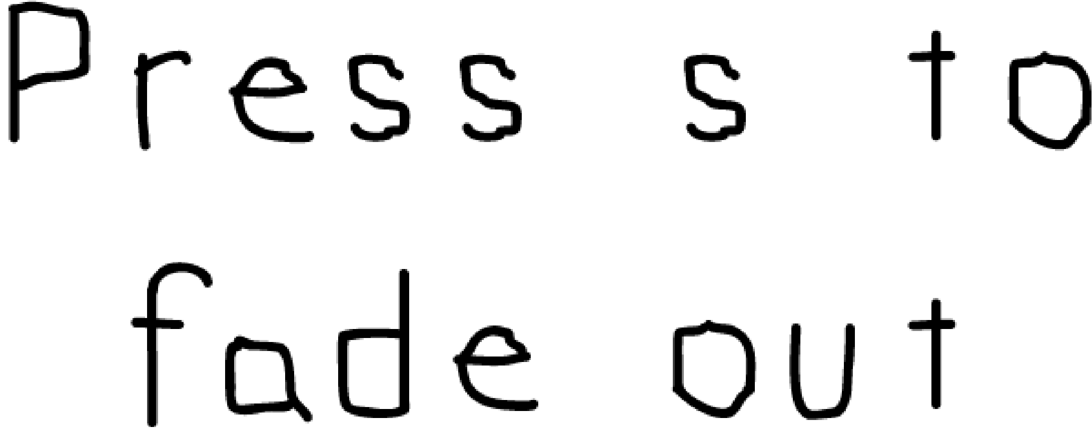


Figura 67: *Resultado ejecutar aplicación Composición Alfa (XIV)*

3.15 Animación de Sprites y Sincronización Vertical

En pocas palabras, la animación consiste en mostrar una imagen tras otra creando la ilusión de movimiento. El objetivo será mostrar diferentes sprites para animar una figura (en este caso la de un hombre de palo).

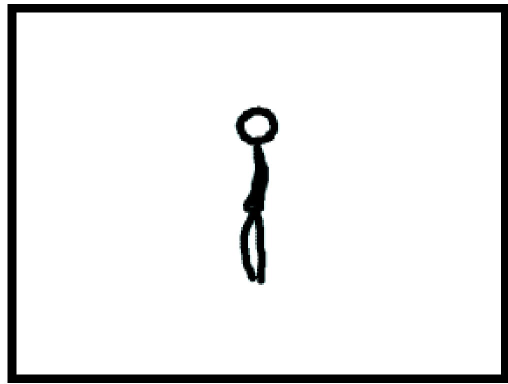


Figura 68: *Sprite para crear animación*

A partir de los siguientes fotogramas de animación:

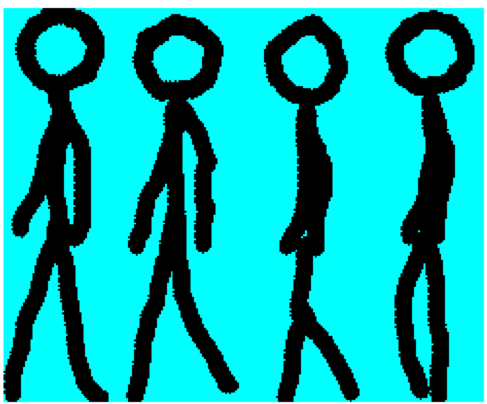


Figura 69*: Fotograma de animación*

Mostrando uno tras otro cada décima de segundo se obtendrá la animación de una persona andando.

Puesto que las imágenes en SDL 2 son generalmente SDL_Textures, la animación es cuestión de mostrar las diferentes partes de las texturas (o distintas texturas completas) una tras otra.

```
//Walking animation

const int WALKING_ANIMATION_FRAMES = 4;
SDL_Rect gSpriteClips[ WALKING_ANIMATION_FRAMES ];
LTexture gSpriteSheetTexture;
```

Sprite sheet con los sprites que se utilizarán para la animación.

```
//Create vsynced renderer for window

gRenderer = SDL_CreateRenderer( gWindow, -1, SDL_RENDERER_ACCELERATED
| SDL_RENDERER_PRESENTVSYNC );

if( gRenderer == NULL )
                  {
printf( "Renderer could not be created! SDL Error: %s\n",
SDL_GetError() );

success = false;
```

La mayoría de monitores funcionan a 60 fotogramas por segundo, por ese motivo se emplea esa suposición aquí. Si se tiene una tasa de actualización del monitor diferente, la animación funcionará demasiado deprisa o despacio.

```
bool loadMedia()
{
      //Loading success flag
      bool success = true;

      //Load sprite sheet texture
if( !gSpriteSheetTexture.loadFromFile(
"14_animated_sprites_and_vsync/foo.png" ) )
      {
            printf( "Failed to load walking animation texture!\n" );
            success = false;
      }

      else
      {
            //Set sprite clips

            gSpriteClips[ 0 ].x =   0;
            gSpriteClips[ 0 ].y =   0;
            gSpriteClips[ 0 ].w =  64;
            gSpriteClips[ 0 ].h = 205;

            gSpriteClips[ 1 ].x =  64;
            gSpriteClips[ 1 ].y =   0;
            gSpriteClips[ 1 ].w =  64;
            gSpriteClips[ 1 ].h = 205;

            gSpriteClips[ 2 ].x = 128;
            gSpriteClips[ 2 ].y =   0;
            gSpriteClips[ 2 ].w =  64;
            gSpriteClips[ 2 ].h = 205;

            gSpriteClips[ 3 ].x = 196;
            gSpriteClips[ 3 ].y =   0;
            gSpriteClips[ 3 ].w =  64;
            gSpriteClips[ 3 ].h = 205;
      }

      return success;
}
```

Tras cargar la sprite sheet, se definen los sprites para los fotogramas de animación individuales.

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//Current animation frame
int frame = 0;
```

Antes del bucle principal se declara una variable que hará un seguimiento al fotograma de animación actual.

```
//Render current frame
SDL_Rect* currentClip = &gSpriteClips[ frame / 4 ];
gSpriteSheetTexture.render( ( SCREEN_WIDTH - currentClip->w ) / 2,
            ( SCREEN_HEIGHT - currentClip->h ) / 2, currentClip );

//Update screen
SDL_RenderPresent( gRenderer );
```

Tras limpiar la pantalla en el bucle principal, se renderiza el fotograma de animación actual.

La animación va de 0 a 3. Dado que se cuenta con 4 fotogramas, se pretende frenar un poco la animación; por ello cuando se llegue al sprite actual, se dividirá el fotograma por 4. De esta manera cada fotograma de animación se actualiza cada 4 fotogramas, ya que con tipos de datos int 0/4=0, 1 / 4 = 0, 2 / 4 = 0, 3 / 4 = 0, 4 / 4 = 1, 5 / 4 = 1...

Después de recibir el sprite actual, se renderiza con la pantalla y ésta se actualiza.

```
//Go to next frame
++frame;

//Cycle animation
if( frame / 4 >= WALKING_ANIMATION_FRAMES )
{
      frame = 0;
}
```

Para que el fotograma se actualice, se necesita incrementar el valor del fotograma cada fotograma. Si no se hiciera, la animación permanecería siempre en el primer fotograma. También se quiere que la animación siga un ciclo, por lo que cuando el fotograma alcanza el valor final (16/4=4) se resetea el fotograma de nuevo a 0 para que la animación comience de nuevo.

Una vez actualizado el fotograma bien incrementándolo o bien siguiendo un ciclo, se llega al final del bucle principal. Este bucle continúa mostrando un fotograma y actualizado el valor de la animación para animar el sprite.

3.16 Rotación y Volteo

El objetivo es utilizar el volteo y rotación para hacer una textura de flecha de ida y vuelta.

```
//Texture wrapper class
class LTexture

{
      public:

            //Initializes variables
            LTexture();

            //Deallocates memory
            ~LTexture();

            //Loads image at specified path
            bool loadFromFile( std::string path );

            //Deallocates texture
            void free();

            //Set color modulation
            void setColor( Uint8 red, Uint8 green, Uint8 blue );

            //Set blending
            void setBlendMode( SDL_BlendMode blending );

            //Set alpha modulation
            void setAlpha( Uint8 alpha );

            //Renders texture at given point
void render( int x, int y, SDL_Rect* clip = NULL, double angle = 0.0,
SDL_Point* center = NULL, DL_RendererFlip flip = SDL_FLIP_NONE );

            //Gets image dimensions
            int getWidth();
            int getHeight();

      private:

            //The actual hardware texture
            SDL_Texture* mTexture;

            //Image dimensions
            int mWidth;
            int mHeight;
};
```

La función renderizar ahora toma un ángulo de rotación, un punto en el que girar la textura y una enumeración de volteo SDL. Al igual que con los rectángulos de superposición, se le asigna los valores por defecto a los argumentos en caso de querer renderizar la textura sin rotación o volteo.

```
void LTexture::render( int x, int y, SDL_Rect* clip,double angle,
```

```
SDL_Point* center, SDL_RendererFlip flip )

{
       //Set rendering space and render to screen
       SDL_Rect renderQuad = { x, y, mWidth, mHeight };

       //Set clip rendering dimensions
       if( clip != NULL )
       {
              renderQuad.w = clip->w;
              renderQuad.h = clip->h;
       }

       //Render to screen
SDL_RenderCopyEx( gRenderer, mTexture, clip, &renderQuad, angle,
center, flip );
}
```

Se pasan los argumentos de la función a **SDL_RenderCopyEx**, que trabaja como la original **SDL_RenderCopy** pero con argumentos adicionales para la rotación y el volteo.

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//Angle of rotation
double degrees = 0;
```

Antes de entrar al bucle principal se declaran variables que hagan un seguimiento del ángulo de rotación y el tipo de volteo.

```
//Handle events on queue

while( SDL_PollEvent( &e ) != 0 )
                        {
//User requests quit

if( e.type == SDL_QUIT )
                                    {
                                           quit = true;
                                    }

else if( e.type == SDL_KEYDOWN )
                                    {
                                           switch( e.key.keysym.sym )

{
case SDLK_a:
degrees -= 60;
break;

case SDLK_d:
degrees += 60;
```

```
break;

case SDLK_q:
flipType = SDL_FLIP_HORIZONTAL;
break;

case SDLK_w:
flipType = SDL_FLIP_NONE;
break;

case SDLK_e:
flipType = SDL_FLIP_VERTICAL;
break;
                    }
                }
            }
```

En el bucle de eventos, se puede incrementar/disminuir la rotación con las teclas a y d y cambiar el tipo de volteo con las teclas q, w y e.

```
//Clear screen
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render arrow
gArrowTexture.render( ( SCREEN_WIDTH - gArrowTexture.getWidth() ) / 2,
( SCREEN_HEIGHT - gArrowTexture.getHeight() ) / 2, NULL, degrees,
NULL, flipType );

//Update screen
SDL_RenderPresent( gRenderer );
```

Éste es el renderizado real.

En primer lugar, se pasan las coordenadas X e Y (todo lo que hace es centrar la imagen). Si la imagen tiene un ancho de 440 pixeles en una pantalla de 640 pixeles, se necesitan rellenar 100 pixeles a cada lado, es decir, la coordenada X tendrá el ancho de la pantalla (640) menos la anchura de la imagen (440) dividido por 2: $(640 - 440)/2 = 100$.

El siguiente argumento es el rectángulo de superposición y ya que se está renderizando la textura completa todo se establece a NULL.

A continuación, el argumento del ángulo de rotación en grados y el del punto en el que se hará la rotación (cuando es NULL girará alrededor del centro de la imagen).

El último argumento es cómo se realiza el volteo de la imagen.

ROTACIÓN:

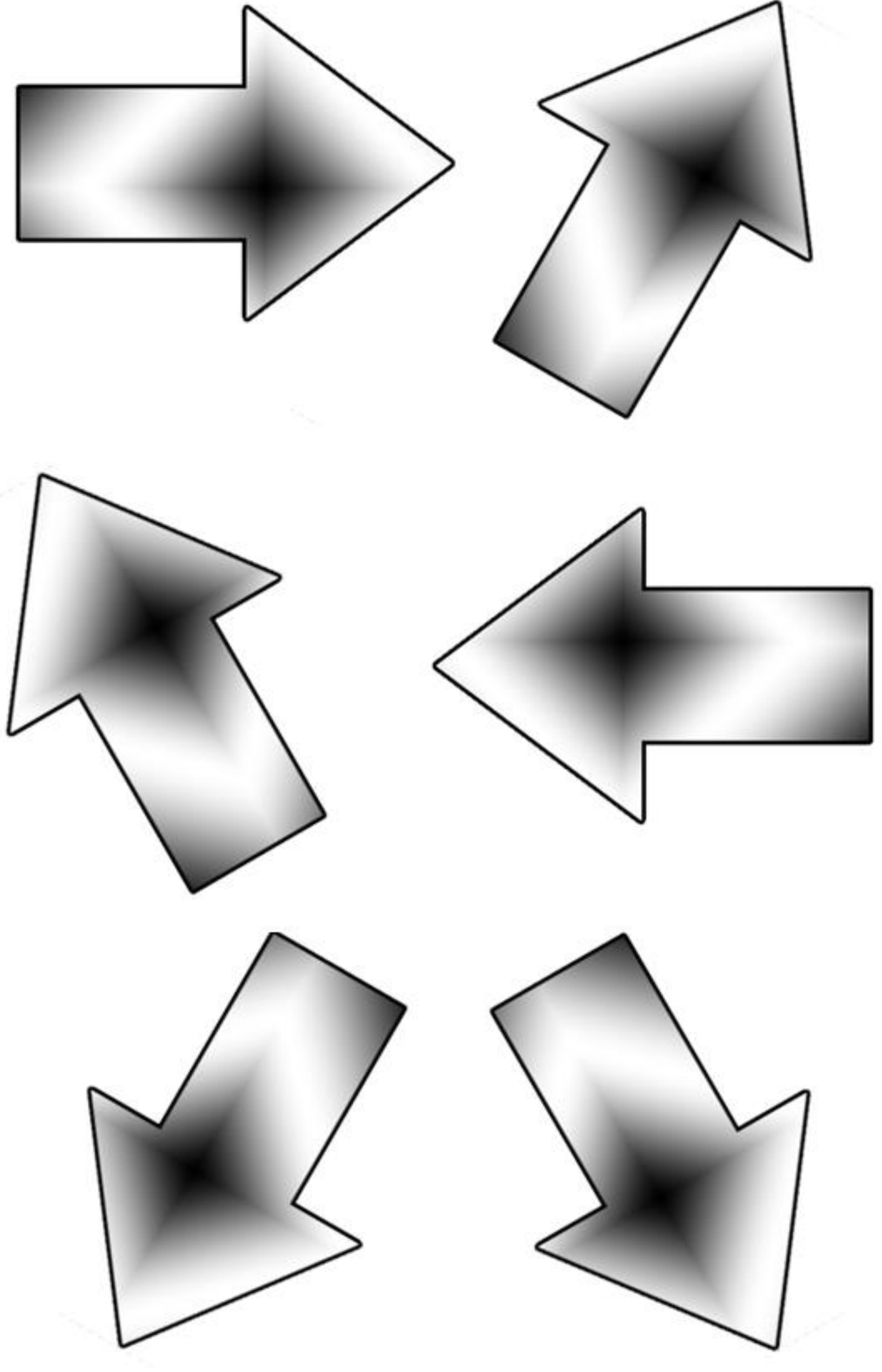


***Figura 70**: Rotación*

- **VOLTEO:**

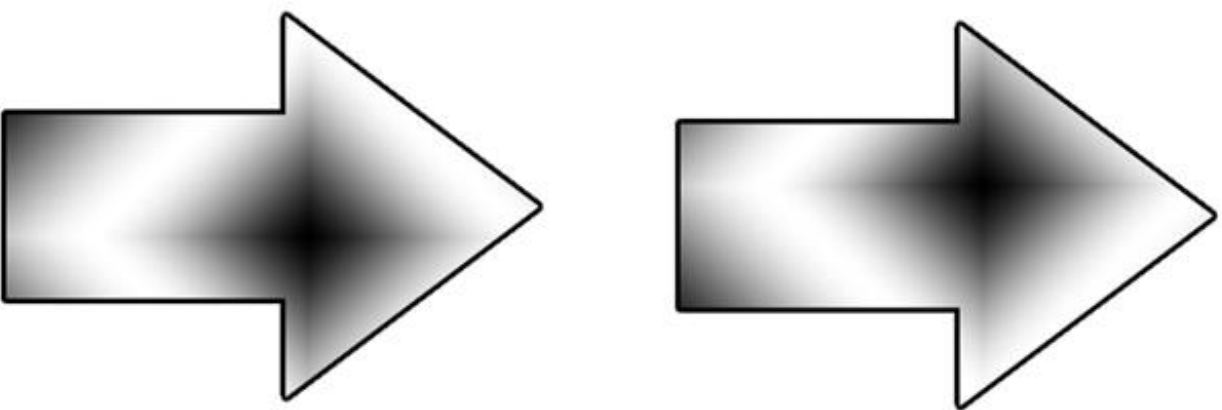


***Figura 71**: Volteo*

3.17 Tipos de Fuentes

Una forma de renderizar texto con SDL es mediante la librería de extensión **SDL_ttf**. Esta librería permite crear imágenes a partir de fuentes TrueType, que se usarán para crear texturas a partir de la fuente de texto.

```
//Using SDL, SDL_image, SDL_ttf, standard IO, math, and strings

#include <SDL.h>
#include <SDL_image.h>
#include <SDL_ttf.h>
#include <stdio.h>
#include <string>
#include <cmath>

//Texture wrapper class

class LTexture
{
	public:

		//Initializes variables
		LTexture();

		//Deallocates memory
		~LTexture();

		//Loads image at specified path
		bool loadFromFile( std::string path );

		//Creates image from font string
bool loadFromRenderedText( std::string textureText, SDL_Color
textColor );

		//Deallocates texture
		void free();

		//Set color modulation
		void setColor( Uint8 red, Uint8 green, Uint8 blue );

		//Set blending
		void setBlendMode( SDL_BlendMode blending );

		//Set alpha modulation
		void setAlpha( Uint8 alpha );

		//Renders texture at given point
void render( int x, int y, SDL_Rect* clip = NULL, double angle = 0.0,
SDL_Point* center = NULL, SDL_RendererFlip flip = SDL_FLIP_NONE );

		//Gets image dimensions
		int getWidth();
		int getHeight();

	private:

		//The actual hardware texture
		SDL_Texture* mTexture;
```

```
            //Image dimensions
            int mWidth;
            int mHeight;
};
```

Se añade una nueva función a la clase de textura denominada **loadFromRenderedText**. La forma de trabajar de **SDL_ttf** consiste en crear una nueva imagen a partir de un tipo de letra y color, es decir, se va a cargar la imagen de texto renderizada a partir de **SDL_ttf** en lugar de un archivo.

```
//The window we'll be rendering to
SDL_Window* gWindow = NULL;

//The window renderer
SDL_Renderer* gRenderer = NULL;

//Globally used font
TTF_Font *gFont = NULL;

//Rendered texture
LTexture gTextTexture;
```

Se usará una fuente global para el texto renderizado.

```
bool LTexture::loadFromRenderedText( std::string textureText, 
SDL_Color textColor )
{
      //Get rid of preexisting texture
      free();

      //Render text surface
SDL_Surface* textSurface = TTF_RenderText_Solid( gFont, 
textureText.c_str(), textColor );

      if( textSurface == NULL )
      {
printf( "Unable to render text surface! SDL_ttf Error: %s\n", 
TTF_GetError() );
      }

      else
      {

//Create texture from surface pixels
mTexture = SDL_CreateTextureFromSurface( gRenderer, textSurface );

if( mTexture == NULL )
            {
printf( "Unable to create texture from rendered text! SDL Error: 
%s\n", SDL_GetError() );
            }

            else
```

```
            {
                    //Get image dimensions
                    mWidth = textSurface->w;
                    mHeight = textSurface->h;
            }

            //Get rid of old surface
            SDL_FreeSurface( textSurface );
        }

//Return success
return mTexture != NULL;

}
```

En esta parte de código es donde realmente se crea la textura que se va a renderizar a partir de la fuente. La función coge la cadena de texto que se va a renderizar y el color que se quiere utilizar para renderizarla.

```
//Initialize renderer color
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );

//Initialize PNG loading
int imgFlags = IMG_INIT_PNG;

if( !( IMG_Init( imgFlags ) & imgFlags ) )
                        {
printf( "SDL_image could not initialize! SDL_image Error: %s\n",
IMG_GetError() );

success = false;
                        }

//Initialize SDL_ttf

if( TTF_Init() == -1 )
                        {
printf( "SDL_ttf could not initialize! SDL_ttf Error: %s\n",
TTF_GetError() );

success = false;
                        }
```

Al igual que **SDL_image**, tiene que ser inicializada o la carga de la fuente y el renderizado no funcionarán correctamente. **TTF_init** pone en marcha **SDL_ttf** y puede comprobarse si hay errores mediante **TTF_GetError.**

```
bool loadMedia()
{
      //Loading success flag
      bool success = true;

      //Open the font
```

```
gFont = TTF_OpenFont( "16_true_type_fonts/lazy.ttf", 28 );

if( gFont == NULL )
       {
printf( "Failed to load lazy font! SDL_ttf Error: %s\n",
TTF_GetError() );

success = false;
       }

       else
       {
              //Render text
              SDL_Color textColor = { 0, 0, 0 };
if( !gTextTexture.loadFromRenderedText( "The quick brown fox jumps
over the lazy dog", textColor ) )

              {
                     printf( "Failed to render text texture!\n" );
                     success = false;
              }
       }

       return success;
```

La fuente se carga usando **TTF_OpenFont.**

```
void close()
{
       //Free loaded images
       gTextTexture.free();

       //Free global font
       TTF_CloseFont( gFont );
       gFont = NULL;

       //Destroy window
       SDL_DestroyRenderer( gRenderer );
       SDL_DestroyWindow( gWindow );
       gWindow = NULL;
       gRenderer = NULL;

       //Quit SDL subsystems
       TTF_Quit();
       IMG_Quit();
       SDL_Quit();
}
```

En la función de <limpieza> se libera la fuente usando **TTF_CloseFont**. Para completar la <limpieza>, se sale de la librería **SDL_ttf**. Para ello se emplea **TTF_Quit**.

```
//While application is running
                  while( !quit )
                  {
                        //Handle events on queue
                        while( SDL_PollEvent( &e ) != 0 )
                        {
                              //User requests quit
                              if( e.type == SDL_QUIT )
                              {
                                    quit = true;
                              }
                        }

//Clear screen
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render current frame
gTextTexture.render( ( SCREEN_WIDTH - gTextTexture.getWidth() ) / 2,
                   ( SCREEN_HEIGHT - gTextTexture.getHeight() ) / 2 );

//Update screen
SDL_RenderPresent( gRenderer );
```

El renderizado de la textura de texto se realiza igual que el de cualquier otra textura.

The quick brown fox jumps over the lazy dog

Figura 72: *Resultado ejecutar aplicación Tipos de Fuentes*

3.18 Acciones con el Ratón. Mouse Events

SDL tiene estructuras para acciones con el ratón tales como el movimiento, pulsación de los botones y el botón de liberación del ratón. El objetivo es realizar una serie de botones con los que se pueda interactuar.

```
//Button constants
const int BUTTON_WIDTH = 300;
const int BUTTON_HEIGHT = 200;
const int TOTAL_BUTTONS = 4;

enum LButtonSprite
{
	BUTTON_SPRITE_MOUSE_OUT = 0,
	BUTTON_SPRITE_MOUSE_OVER_MOTION = 1,
	BUTTON_SPRITE_MOUSE_DOWN = 2,
	BUTTON_SPRITE_MOUSE_UP = 3,
	BUTTON_SPRITE_TOTAL = 4
};
```

En este programa se tendrán cuatro botones en la pantalla. Dependiendo de si el ratón se ha acercado, se ha hecho click con un botón, se ha pulsado el botón de liberación o se ha alejado se mostrará un sprite diferente. Las constantes definidas se utilizarán para este fin.

```
//Texture wrapper class

class LTexture
{
	public:
		//Initializes variables
		LTexture();

		//Deallocates memory
		~LTexture();

		//Loads image at specified path
		bool loadFromFile( std::string path );
		#ifdef _SDL_TTF_H

		//Creates image from font string
bool loadFromRenderedText( std::string textureText, SDL_Color
textColor );
		#endif

		//Deallocates texture
		void free();

		//Set color modulation
		void setColor( Uint8 red, Uint8 green, Uint8 blue );

		//Set blending
		void setBlendMode( SDL_BlendMode blending );
```

```
//Set alpha modulation
void setAlpha( Uint8 alpha );

//Renders texture at given point

void render( int x, int y, SDL_Rect* clip = NULL, double angle = 0.0,
 SDL_Point* center = NULL, SDL_RendererFlip flip = SDL_FLIP_NONE );
```

No se usará **SDL_ttf** para renderizar el texto, lo que significa que no es necesaria la función **loadFromRenderedText**. En lugar de borrar el código (ya que puede ser necesario en el futuro), se envolverá en los estados definidos, así el compilador lo ignorará si no se incluye **SDL_ttf**.

Al igual que #include, #idef es una macro usada para hablar con el compilador. En este caso, al no estar **SDL_ttf** definido se ignorará esta parte del código.

```
//The mouse button
class LButton
{
	public:

		//Initializes internal variables
		LButton();

		//Sets top left position
		void setPosition( int x, int y );

		//Handles mouse event
		void handleEvent( SDL_Event* e );

		//Shows button sprite
		void render();

	private:

		//Top left position
		SDL_Point mPosition;

		//Currently used global sprite
		LButtonSprite mCurrentSprite;

};
```

Ésta es la clase para representar un botón.

Cuenta con un constructor para inicializar, establecedor de la posición, controlador de eventos para el bucle de eventos y función de renderizado. Además, tiene una enumeración y posición de sprites, por lo que se sabrá qué sprite renderiza el botón.

```
#ifdef _SDL_TTF_H

bool LTexture::loadFromRenderedText( std::string textureText,
SDL_Color textColor )
```

```
{
	//Get rid of preexisting texture
	free();

	//Render text surface
SDL_Surface* textSurface = TTF_RenderText_Solid( gFont,
textureText.c_str(), textColor );

	if( textSurface == NULL )
	{
printf( "Unable to render text surface! SDL_ttf Error: %s\n",
TTF_GetError() );
	}

	else
	{
//Create texture from surface pixels
mTexture = SDL_CreateTextureFromSurface( gRenderer, textSurface );

if( mTexture == NULL )
		{
printf( "Unable to create texture from rendered text! SDL Error:
%s\n", SDL_GetError() );
		}

		else
		{
//Get image dimensions
mWidth = textSurface->w;
mHeight = textSurface->h;
		}

//Get rid of old surface
SDL_FreeSurface( textSurface );
	}

//Return success
return mTexture != NULL;

}
#endif
```

Para asegurar que la fuente compila sin **SDL_ttf**, se envuelve de nuevo la carga de la función fuente en otra condición ifdef.

```
LButton::LButton()
{
	mPosition.x = 0;
	mPosition.y = 0;

	mCurrentSprite = BUTTON_SPRITE_MOUSE_OUT;
}

void LButton::setPosition( int x, int y )
{
	mPosition.x = x;
	mPosition.y = y;
}
```

Constructor para el botón y función de ajuste de la posición. Por defecto inicializan el sprite y la posición establecida.

```
void LButton::handleEvent( SDL_Event* e )
{
//If mouse event happened

if( e->type == SDL_MOUSEMOTION || e->type == SDL_MOUSEBUTTONDOWN || e-
>type == SDL_MOUSEBUTTONUP )
      {
            //Get mouse position
            int x, y;
            SDL_GetMouseState( &x, &y );
```

En esta parte del código se controlan las acciones del ratón. La función será llamada en el bucle de eventos y controlará un evento tomado de la cola de eventos para un botón individual.

En primer lugar, se comprueba si el evento que entra es una acción de ratón, concretamente un evento de movimiento del ratón, un evento del botón del ratón hacia abajo (cuando se hace clic con los botones del ratón) o un evento del botón del ratón hacia arriba (al soltar el clic del ratón)

Si se produce uno de los anteriores eventos, se comprueba la posición del ratón mediante **SDL_GetMouseState**. Dependiendo de si el ratón está sobre el botón o no se mostrarán diferentes sprites.

```
            //Check if mouse is in button
            bool inside = true;

            //Mouse is left of the button
            if( x < mPosition.x )
            {
                  inside = false;
            }

            //Mouse is right of the button
            else if( x > mPosition.x + BUTTON_WIDTH )
            {
                  inside = false;
            }

            //Mouse above the button
            else if( y < mPosition.y )
            {
                  inside = false;
            }

            //Mouse below the button
            else if( y > mPosition.y + BUTTON_HEIGHT )
```

```
                {
                        inside = false;
                }
```

Aquí se comprueba si el ratón está sobre el botón o no. Debido a que se usa un sistema de coordenadas distinto con SDL, el origen del botón está en la esquina superior izquierda, es decir, cada coordenada X menor que la posición X estará fuera del botón y lo mismo ocurrirá con cada coordenada Y.

Si la posición del ratón está en cualquier posición fuera del botón, el indicador en su interior marcará false. En caso contrario, mantendrá el valor inicial de true.

```
                //Mouse is outside button
                if( !inside )
                {
                        mCurrentSprite = BUTTON_SPRITE_MOUSE_OUT;
                }

                //Mouse is inside button
                else
                {
                //Set mouse over sprite
                switch( e->type )
                        {

                        case SDL_MOUSEMOTION:
                        mCurrentSprite = BUTTON_SPRITE_MOUSE_OVER_MOTION;
                        break;

                        case SDL_MOUSEBUTTONDOWN:
                        mCurrentSprite = BUTTON_SPRITE_MOUSE_DOWN;
                        break;

                        case SDL_MOUSEBUTTONUP:
                        mCurrentSprite = BUTTON_SPRITE_MOUSE_UP;
                        break;

                        }
```

Se finaliza fijando el botón de sprite dependiendo de si el ratón está dentro del botón y el evento de ratón.

```
void LButton::render()
{
        //Show current button sprite
gButtonSpriteSheetTexture.render( mPosition.x, mPosition.y,
&gSpriteClips[ mCurrentSprite ] );

}
```

Sólo se renderiza el botón de sprite actual con la posición del botón.

```
                //While application is running
                while( !quit )
                {
                        //Handle events on queue
                        while( SDL_PollEvent( &e ) != 0 )
                        {
                                //User requests quit
                                if( e.type == SDL_QUIT )
                                {
                                        quit = true;
                                }

                                //Handle button events
                                for( int i = 0; i < TOTAL_BUTTONS; ++i )
                                {
                                        gButtons[ i ].handleEvent( &e );
                                }
                        }

        //Clear screen
        SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
        SDL_RenderClear( gRenderer );

        //Render buttons
                        for( int i = 0; i < TOTAL_BUTTONS; ++i )
                        {
                                gButtons[ i ].render();
                        }

        //Update screen
        SDL_RenderPresent( gRenderer );

                }
```

Éste es el bucle principal. En el bucle de eventos se controla el evento de salida y los eventos para todos los botones. En la sección de renderizado todos los botones se renderizan con la pantalla.

***(*) NOTA:** La estrella roja representa el ratón.*

- **Opción 1: Se sitúa el ratón sobre el botón superior izquierdo**

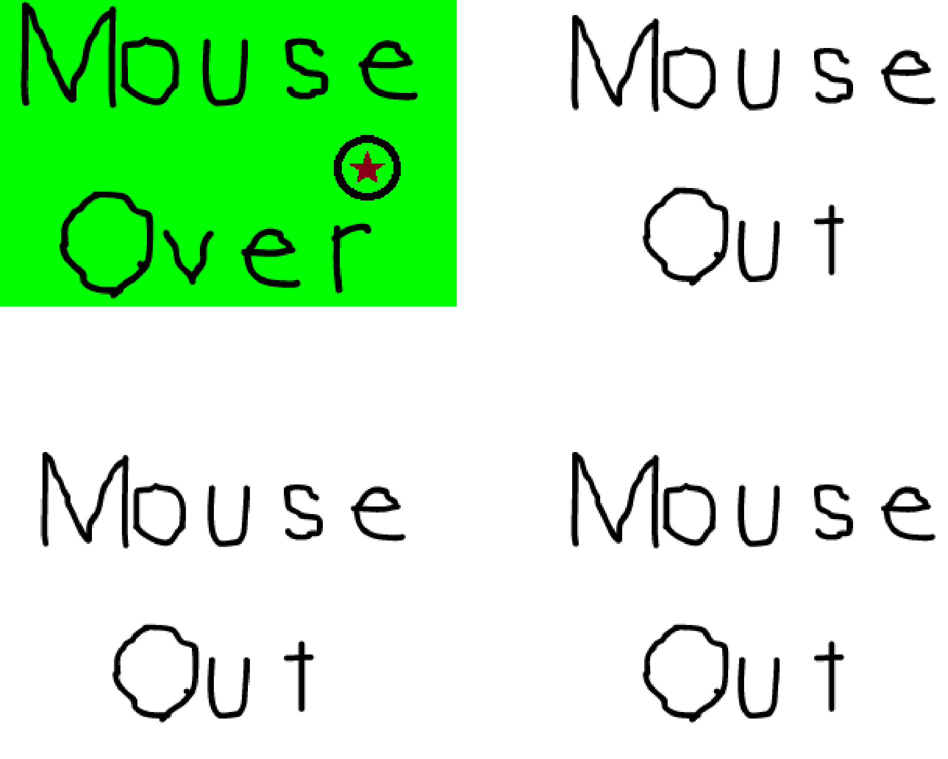


Figura 73: *Resultado ejecutar aplicación Acciones con el Ratón*

Caso A) Haciendo clic

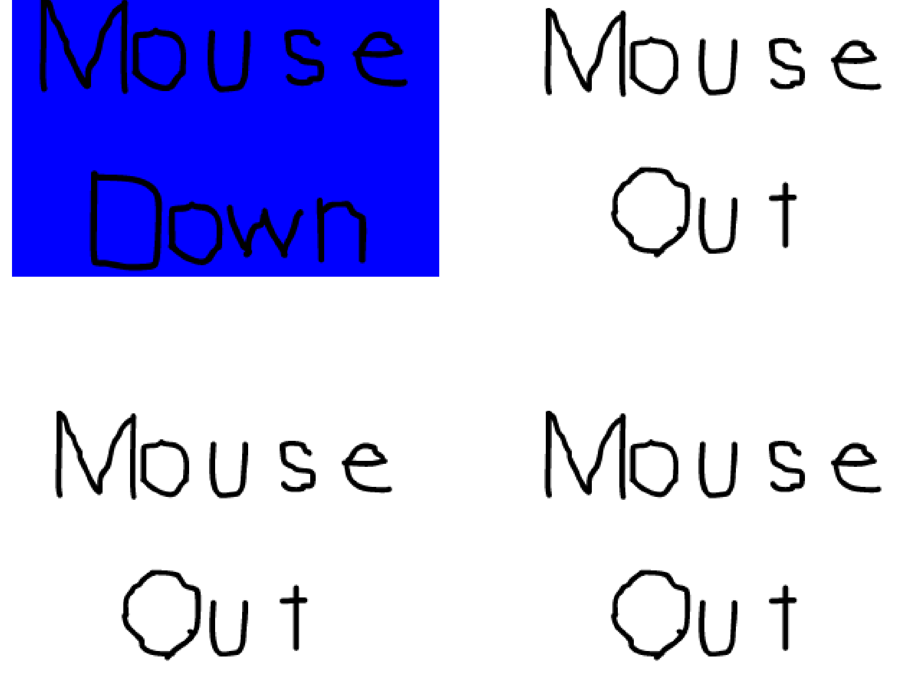


Figura 74: *Resultado ejecutar aplicación Acciones con el Ratón (II)*

Caso B) Soltando el clic

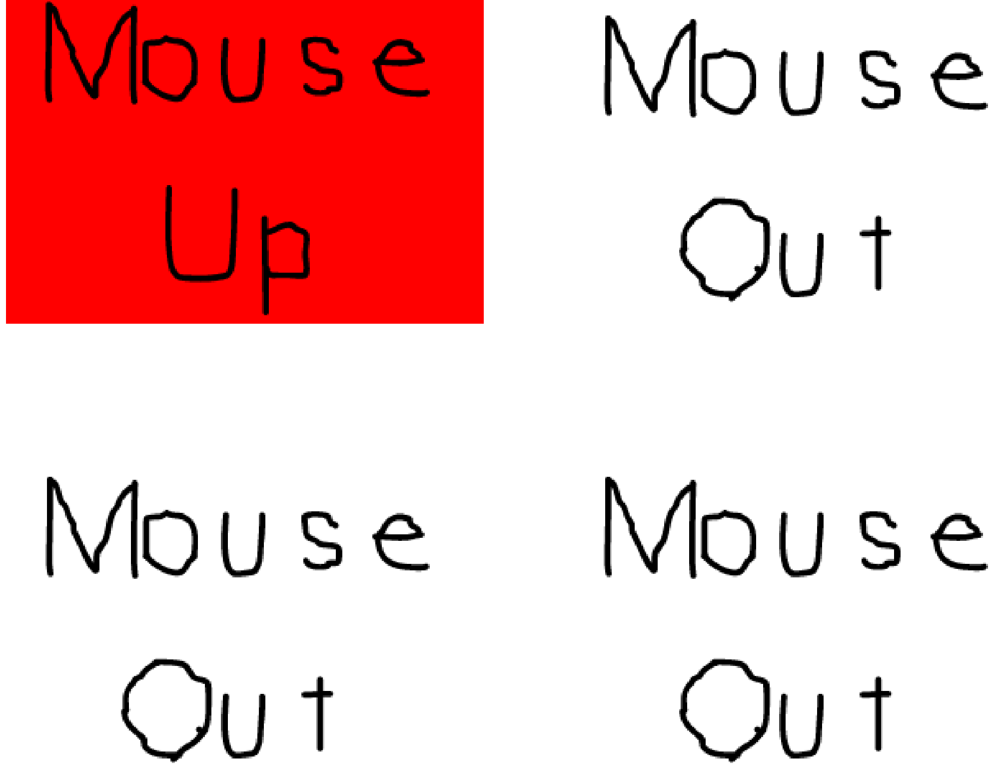


Figura 75: *Resultado ejecutar aplicación Acciones con el Ratón (III)*

- **Opción 2: El ratón se sitúa sobre el botón superior derecho**

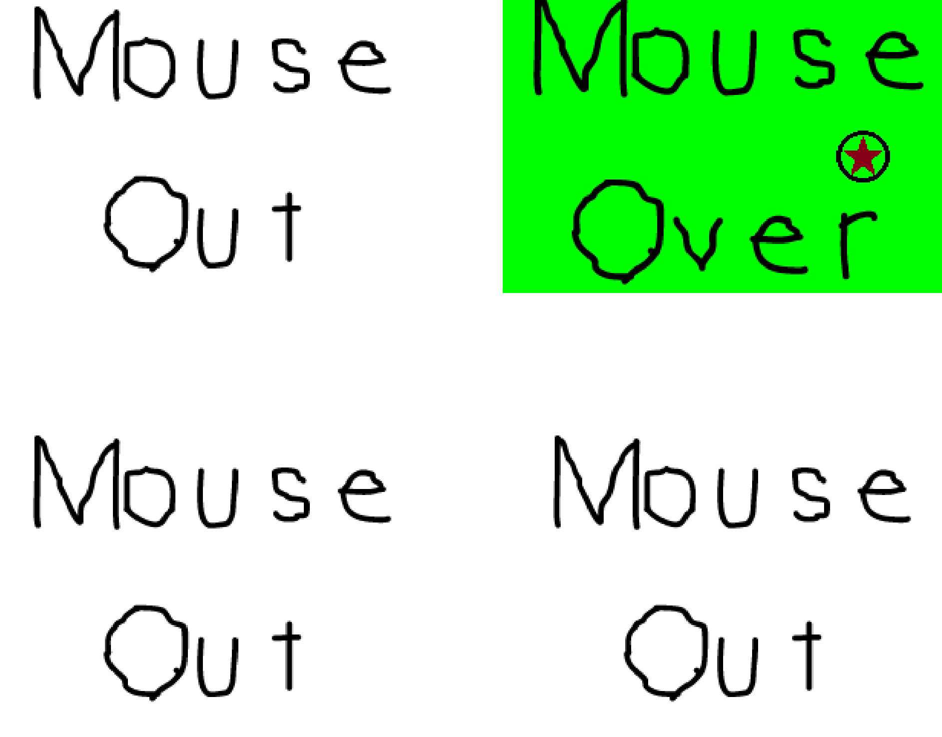


Figura 76: *Resultado ejecutar aplicación Acciones con el Ratón (IV)*

Caso A) Haciendo clic

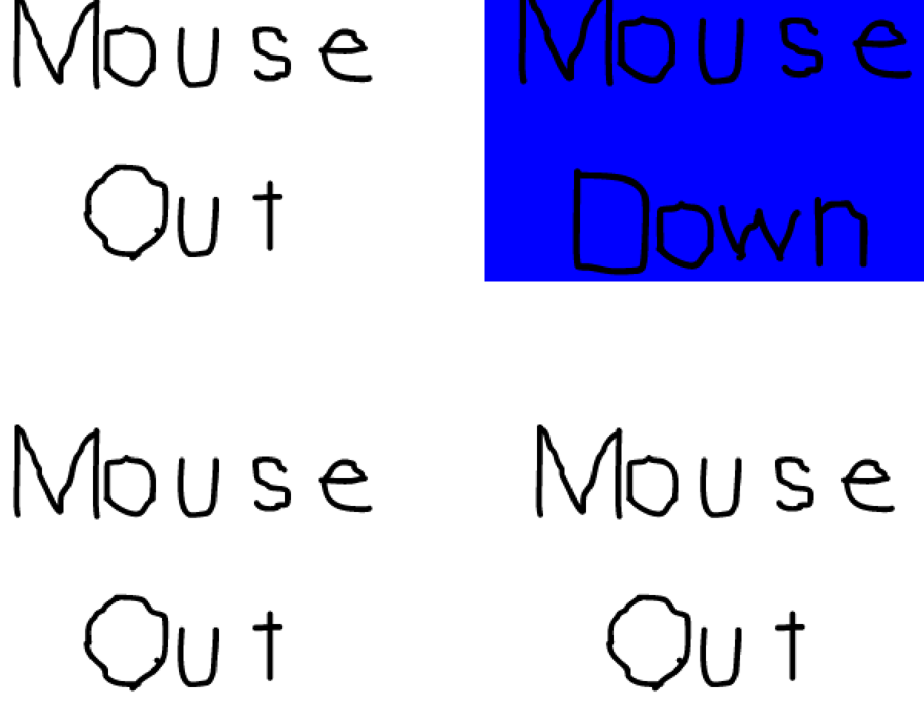


Figura 77: *Resultado ejecutar aplicación Acciones con el Ratón (V)*

Caso B) Soltando el clic

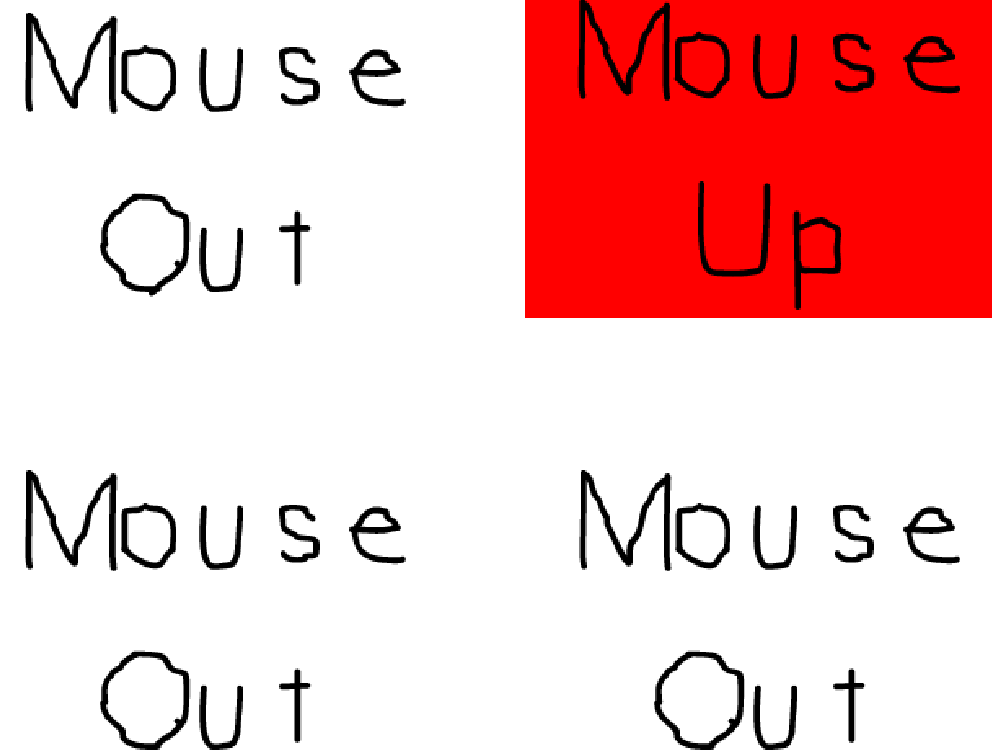


Figura 78: *Resultado ejecutar aplicación Acciones con el Ratón (VI)*

- **Opción 3: El ratón se sitúa sobre el botón inferior izquierdo**

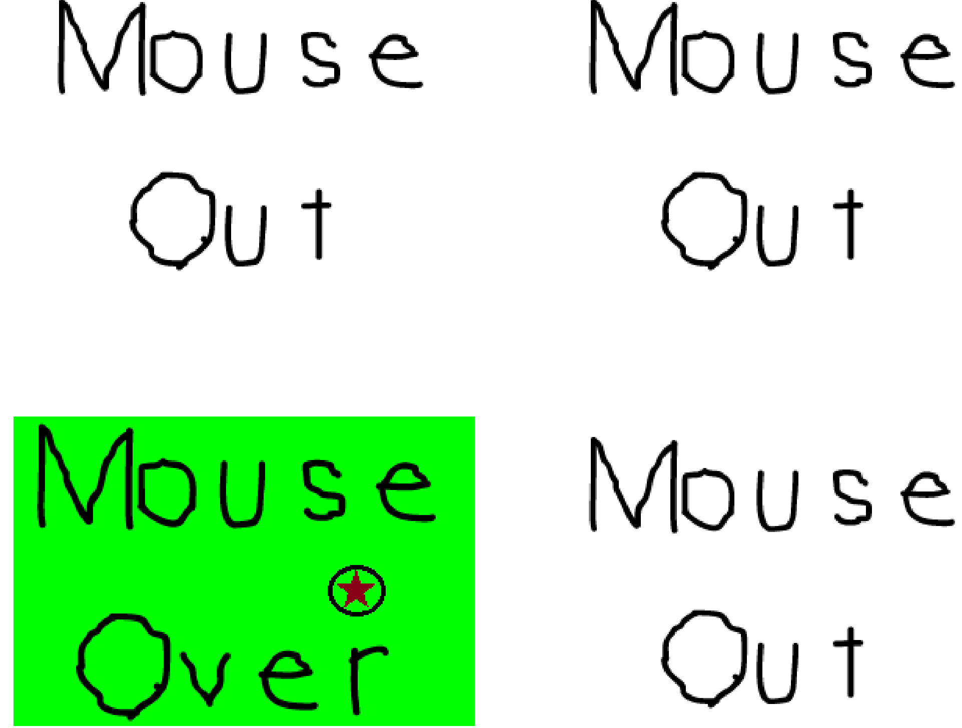


Figura 79: *Resultado ejecutar aplicación Acciones con el Ratón (VII)*

Caso A) Haciendo clic

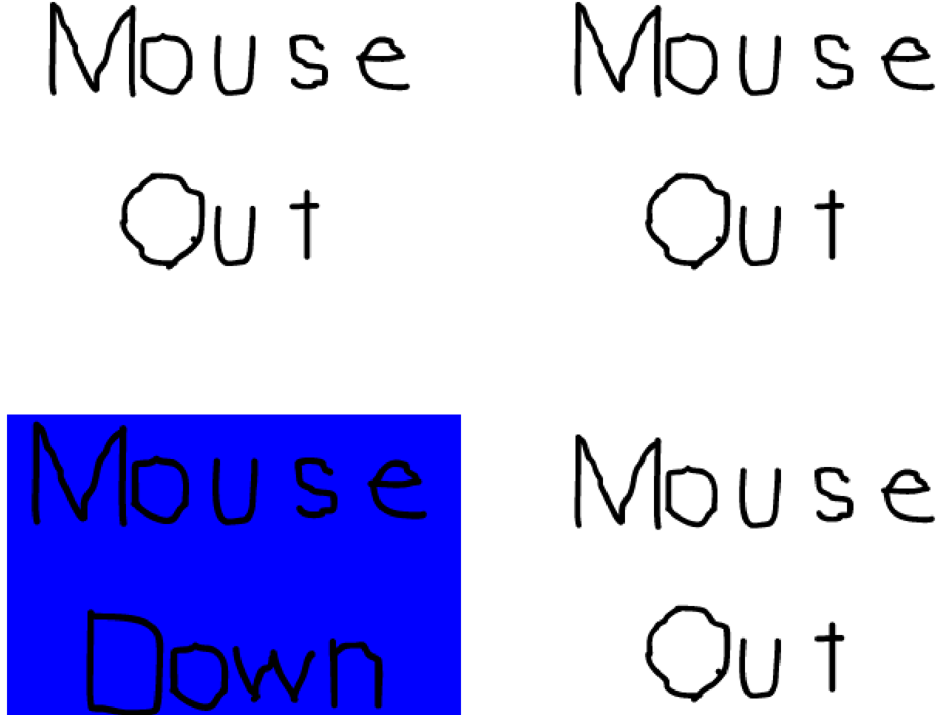


Figura 80: *Resultado ejecutar aplicación Acciones con el Ratón (VIII)*

Caso B) Soltando el clic

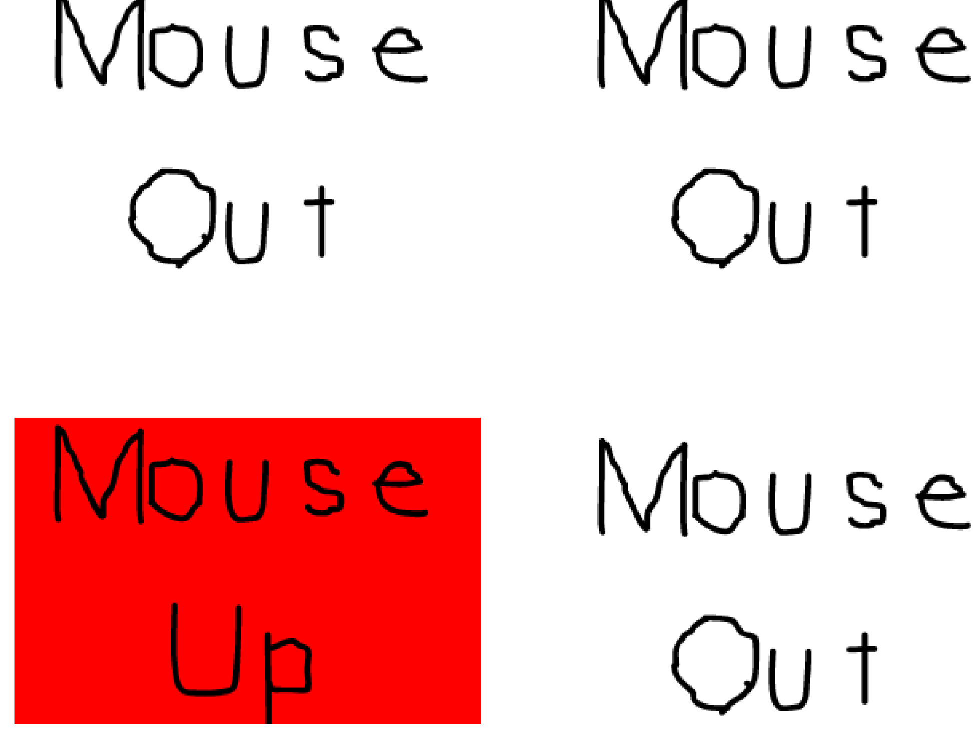


Figura 81: *Resultado ejecutar aplicación Acciones con el Ratón (IX)*

- **Opción 4: El ratón se sitúa sobre el botón inferior derecho**

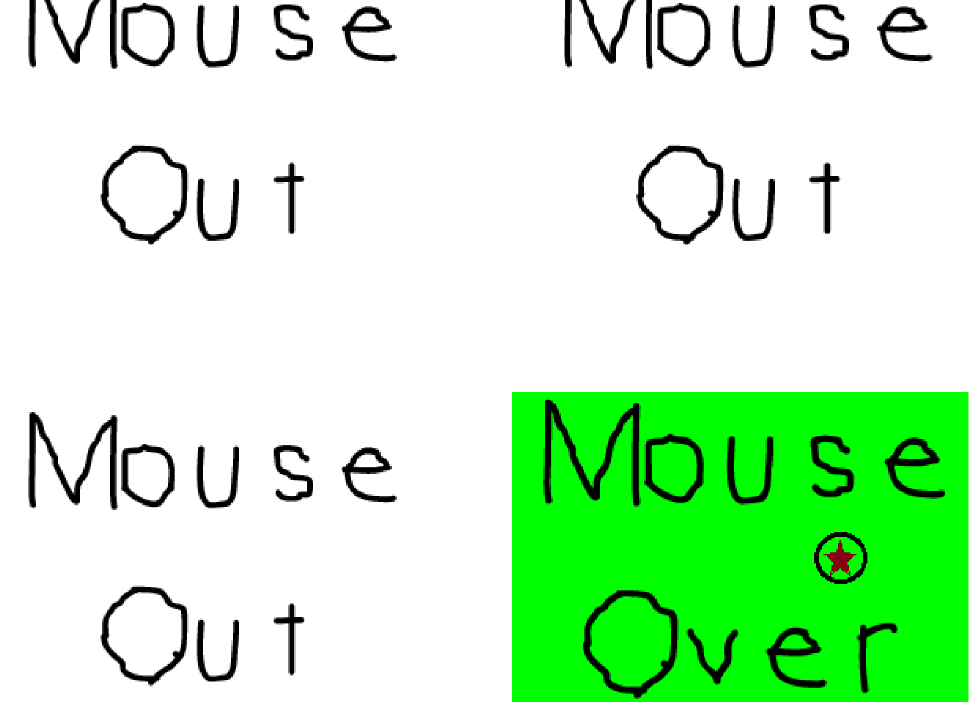


Figura 82: *Resultado ejecutar aplicación Acciones con el Ratón (X)*

Caso A) Haciendo clic

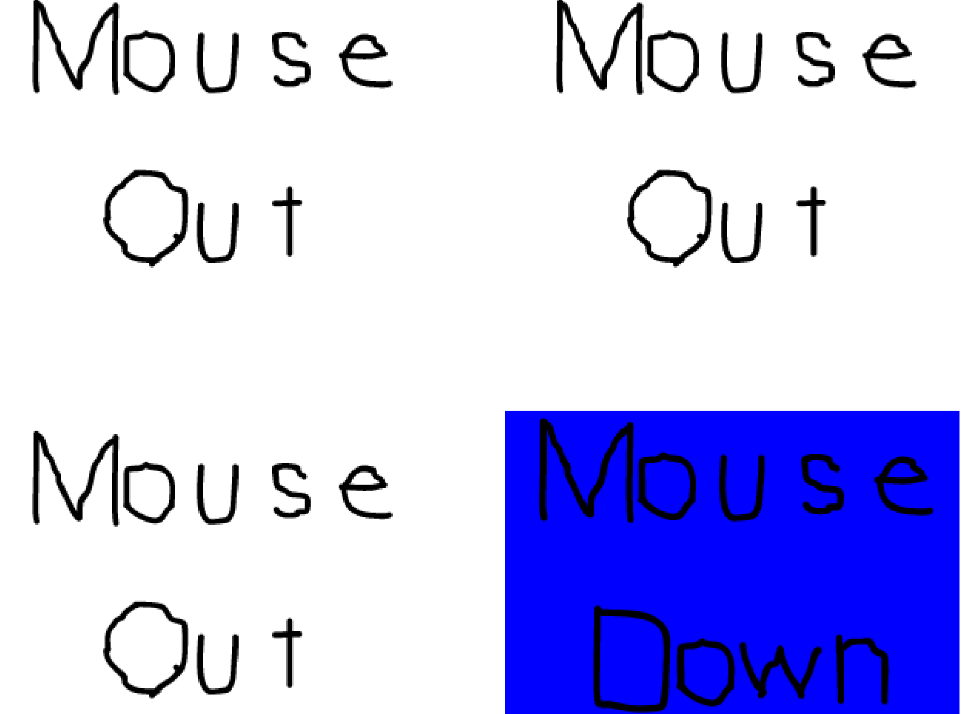


Figura 83: *Resultado ejecutar aplicación Acciones con el Ratón (XI)*

Caso B) Soltando el clic

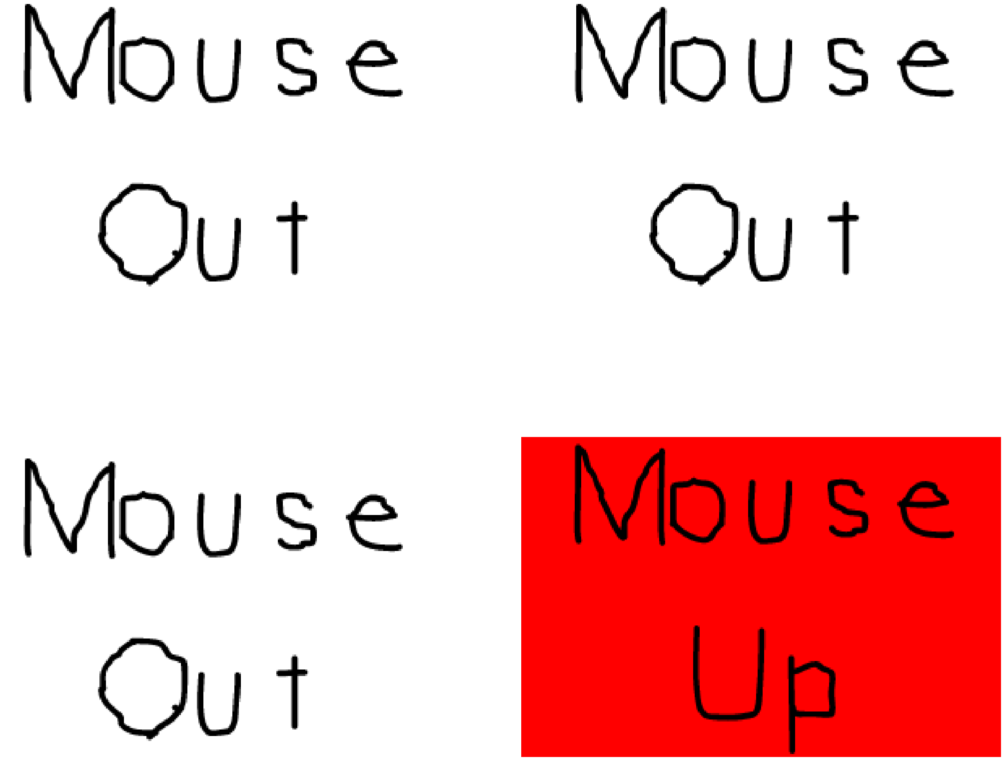


Figura 84: *Resultado ejecutar aplicación Acciones con el Ratón (XII)*

3.19 Key States

Existen diferentes maneras de obtener el estado de los dispositivos de entrada (ratón, teclado,…) distintas al uso de eventos. El objetivo será rehacer las entradas de teclado utilizando estados de teclado en lugar de eventos.

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//Current rendered texture
LTexture* currentTexture = NULL;
```

Justo antes de entrar al bucle principal se declara un puntero textura para controlar la textura que se está renderizando a la pantalla.

```
//While application is running
while( !quit )
{
        //Handle events on queue
        while( SDL_PollEvent( &e ) != 0 )
        {
                //User requests quit
                if( e.type == SDL_QUIT )
                {
                        quit = true;
                }
        }
```

Los eventos de teclado no se comprueban en el bucle principal, sino que se controlarán mediante estados de teclado.

Los estados de teclado internos de SDL se actualizan cada vez que se llama a **SDL_PollEvent**, por lo que es necesario asegurarse de que todos los eventos estén en cola antes de comprobar los estados de teclado.

```
//Set texture based on current keystate
const Uint8* currentKeyStates = SDL_GetKeyboardState( NULL );

if( currentKeyStates[ SDL_SCANCODE_UP ] )
                        {
                                currentTexture = &gUpTexture;
                        }
                        else if( currentKeyStates[ SDL_SCANCODE_DOWN ]
)
                        {
                                currentTexture = &gDownTexture;
                        }
```

```
                else if( currentKeyStates[ SDL_SCANCODE_LEFT ]
)
                {
                    currentTexture = &gLeftTexture;
                }
                else if( currentKeyStates[ SDL_SCANCODE_RIGHT]
)
                {
                    currentTexture = &gRightTexture;
                }
                else
                {
                    currentTexture = &gPressTexture;
                }
```

Se establece la textura que se va a renderizar.

En primer lugar, se asigna un puntero al array de estados de teclado empleando **SDL_GetKeyboardState**. El estado de todas las teclas está clasificado por **SDL_Scancode**.

Para comprobar si una tecla está pulsada tan sólo hay que constatar su estado en el array de estados de teclas. Si la tecla está pulsada se establece la textura actual a la correspondiente textura. En cambio, si ninguna de las teclas está pulsada, se establece la textura por defecto.

```
    //Clear screen
    SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
    SDL_RenderClear( gRenderer );

    //Render current texture
    currentTexture->render( 0, 0 );

    //Update screen
    SDL_RenderPresent( gRenderer );
```

Finalmente, se renderiza la textura actual con la pantalla.

Press: up, down, left, or right!

Figura 85: *Resultado ejecutar aplicación Key States*

Caso 1) Pulsando la tecla de flecha hacia arriba

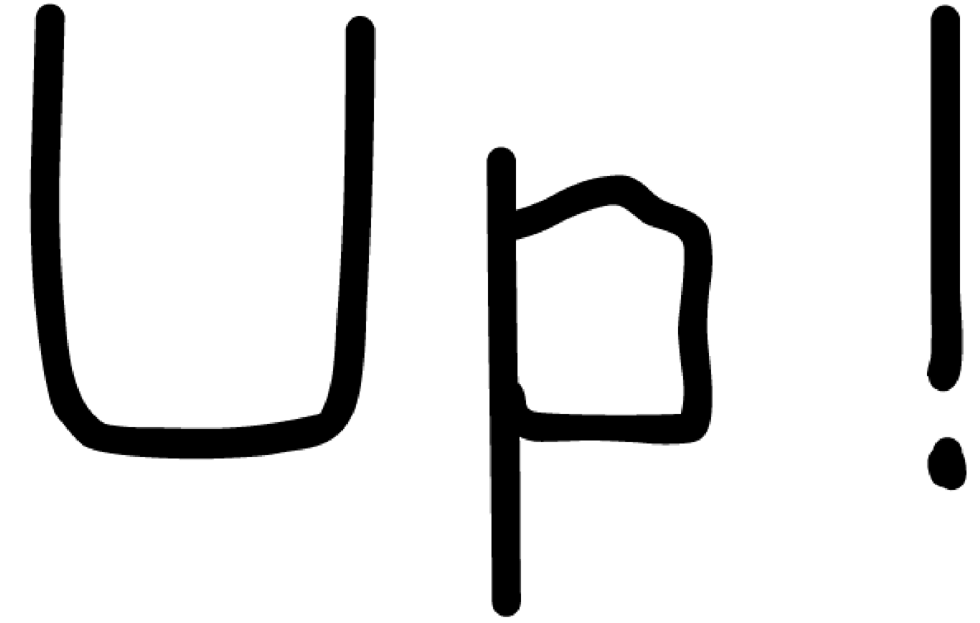


Figura 86: *Resultado ejecutar aplicación Key States (II)*

Tras dejar de pulsar la tecla aparece de nuevo la ventana:

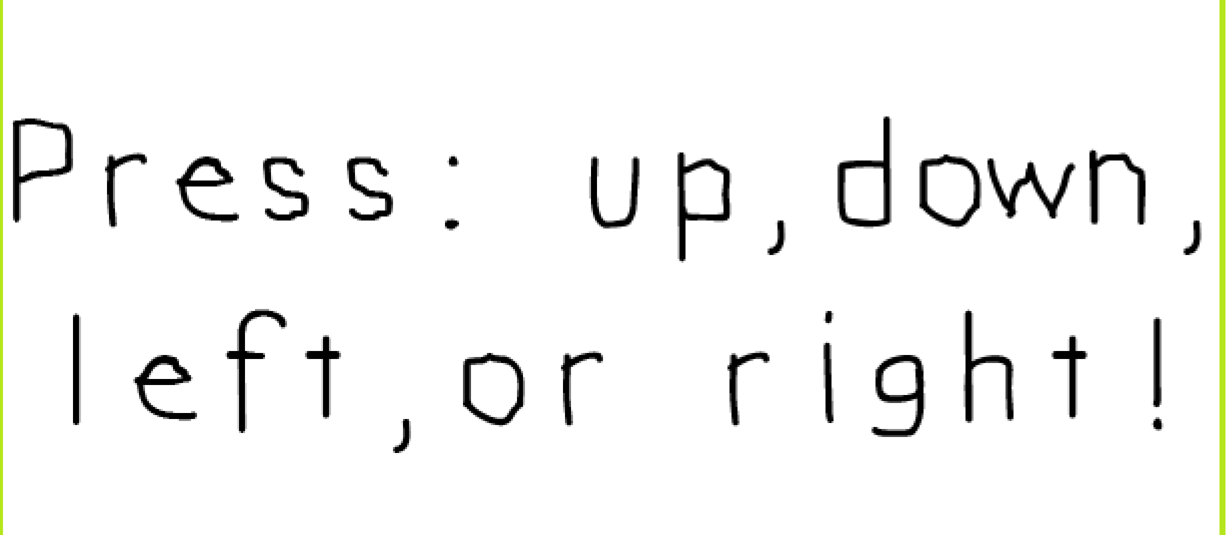


Figura 87: *Resultado ejecutar aplicación Key States (III)*

Caso 2) Pulsando la tecla de flecha hacia abajo

Figura 88: *Resultado ejecutar aplicación Key States (IV)*

Caso 3) Pulsando la tecla de flecha hacia la derecha

Figura 89: *Resultado ejecutar aplicación Key States (V)*

Caso 4) Pulsando la tecla de flecha hacia la izquierda

Figura 90: *Resultado ejecutar aplicación Key States (VI)*

3.20 Gamepads y Joysticks

Al igual que con las entradas del ratón y del teclado, SDL tiene la capacidad de leer la entrada desde un controlador joystick/ gamepad/ juego. El objetivo es rotar una flecha basada en la entrada de un joystick.

```
//Analog joystick dead zone
const int JOYSTICK_DEAD_ZONE = 8000;
```

La manera en la que SDL controla sticks analógicos es convirtiendo su posición en un número entre -32768 y 32767. Con el fin de evitar pulsaciones que ocasionen errores, se crea una zona muerta donde las entradas del joystick se ignoran. Para ello se ha creado la constante.

```
//Game Controller 1 handler
SDL_Joystick* gGameController = NULL;
```

El tipo de datos para un controlador de juego es **SDL_Joystick**. A continuación, se declara el controlador de joystick global que se usará para interactuar con el joystick.

```
bool init()
{
      //Initialization flag
      bool success = true;

      //Initialize SDL
      if(SDL_Init(SDL_INIT_VIDEO|SDL_INIT_JOYSTICK)<0)
      {
 printf( "SDL could not initialize! SDL Error: %s\n",SDL_GetError() );
 success = false;
      }
```

(*) IMPORTANTE: Hasta ahora, se ha estado inicializando sólo video para que se pueda renderizar a la pantalla. Ahora, es necesario inicializar el subsistema del joystick o la lectura del joystick no funcionará.

```
//Set texture filtering to linear
if( !SDL_SetHint( SDL_HINT_RENDER_SCALE_QUALITY, "1" ) )
            {
    printf("Warning: Linear texture filtering not enabled!");
            }

//Check for joysticks
            if( SDL_NumJoysticks() < 1 )
            {
                  printf( "Warning: No joysticks connected!\n" );
            }
```

```
            else
            {

        //Load joystick
        gGameController = SDL_JoystickOpen( 0 );
        if( gGameController == NULL )
        {
 printf("Warning: Unable to open game controller! SDL Error: %s\n",
 SDL_GetError() );
                  }
            }
```

Una vez se ha inicializado el subsistema del joystick, el siguiente paso es abrir el joystick.

En primer lugar, se llama a **SDL_NumJoysticks** para comprobar si hay al menos un joystick conectado. De ser así, se llama a **SDL_JoystickOpen** para abrir el joystick en el índice 0. Tras abrirlo, podrá mandar eventos a la cola de eventos de SDL.

```
void close()
{
      //Free loaded images
      gArrowTexture.free();

      //Close game controller
      SDL_JoystickClose( gGameController );
      gGameController = NULL;

      //Destroy window
      SDL_DestroyRenderer( gRenderer );
      SDL_DestroyWindow( gWindow );
      gWindow = NULL;
      gRenderer = NULL;

      //Quit SDL subsystems
      IMG_Quit();
      SDL_Quit();
}
```

Después de terminar con el joystick, se cierra mediante **SDL_JoystickClose.**

```
                  //Main loop flag
                  bool quit = false;

                  //Event handler
                  SDL_Event e;

                  //Normalized direction
                  int xDir = 0;
                  int yDir = 0;
```

En esta ocasión se quiere hacer un seguimiento de la dirección X e Y.

- Si la X se igualara a -1, la posición X del joystick señalaría a la izquierda, mientras que si se igualase a +1 la posición X señalaría a la derecha.

- En cuanto a la posición Y, para los joysticks será positiva hacia arriba y negativa hacia abajo, es decir, si la Y se igualara a +1 señalaría hacia arriba y si se igualase a -1 señalaría hacia abajo.

- Si el valor de X ó Y es 0, quiere decir que el joystick se encuentra en la zona muerta y estaría situado en el centro.

```
//Handle events on queue
                        while( SDL_PollEvent( &e ) != 0 )
                        {
                              //User requests quit
                              if( e.type == SDL_QUIT )
                              {
                                    quit = true;
                              }

                              else if( e.type == SDL_JOYAXISMOTION )
                              {
                                    //Motion on controller 0
                                    if( e.jaxis.which == 0 )
                                    {
                                          //X axis motion

                                          if( e.jaxis.axis == 0 )
                                          {

                        //Left of dead zone
                        if( e.jaxis.value < -JOYSTICK_DEAD_ZONE )
                                                {
                                                      xDir = -1;
                                                }
                        //Right of dead zone
                        else if( e.jaxis.value > JOYSTICK_DEAD_ZONE )
      {
                                                      xDir =  1;
                                                }
                                                else
                                                {
                                                      xDir = 0;
                                                }
                                          }
```

En el bucle de eventos se comprueba si el joystick se ha movido mediante la verificación de **SDL_JoyAxisEvent**. La variable informa de qué controlador del movimiento de los ejes viene y se constata que el evento vino del joystick 0.

Lo siguiente a comprobar es si fue un movimiento en la dirección X o Y, que indicará la variable “axis” (eje). Normalmente, el eje 0 es el eje X.

La variable “value” aporta qué posición tiene el joystick analógico en el eje. Si la posición X es menor que la de zona muerta, la dirección sería negativa, mientras que si la posición es mayor

que la zona muerta, la posición será positiva. En el caso de estar en la zona muerta, la dirección se fija a 0.

```
//Y axis motion

else if( e.jaxis.axis == 1 )
{

//Below of dead zone
if( e.jaxis.value < -JOYSTICK_DEAD_ZONE )

{
yDir = -1;
}

//Above of dead zone
else if( e.jaxis.value > JOYSTICK_DEAD_ZONE )

{
yDir =  1;
}

else

{
yDir = 0;
}
```

Se repite con el eje Y, que se identifica con el identificador de eje 1.

```
//Clear screen

SDL_SetRenderDrawColor(gRenderer,0xFF,0xFF,0xFF,0xFF);
SDL_RenderClear( gRenderer );

//Calculate angle
double joystickAngle = atan2( (double)yDir,(double)xDir ) * ( 180.0 /
M_PI );

//Correct angle
if( xDir == 0 && yDir == 0 )

{
joystickAngle = 0;
}
```

Antes de renderizar la flecha que señalará la dirección en la que el stick analógico es pulsado, es necesario calcular el ángulo. Para ello se emplea la función cmath atan2, que tiene en cuenta el cuadrante del que vienen los valores. Dará los valores de la posición X e Y, así como el ángulo en radianes.

La rotación de los ángulos en SDL es en grados, por lo que habrá que convertir los radianes a grados. Cuando la posición X e Y son ambas 0, se corrige el valor del ángulo obtenido (casi despreciable) y se iguala a 0.

```
//Render joystick 8 way angle
gArrowTexture.render(( SCREEN_WIDTH -gArrowTexture.getWidth() ) / 2,(
SCREEN_HEIGHT - gArrowTexture.getHeight() ) / 2,NULL, joystickAngle );

//Update screen
SDL_RenderPresent( gRenderer );

}
```

Para finalizar, se renderiza la flecha rotada en la pantalla.

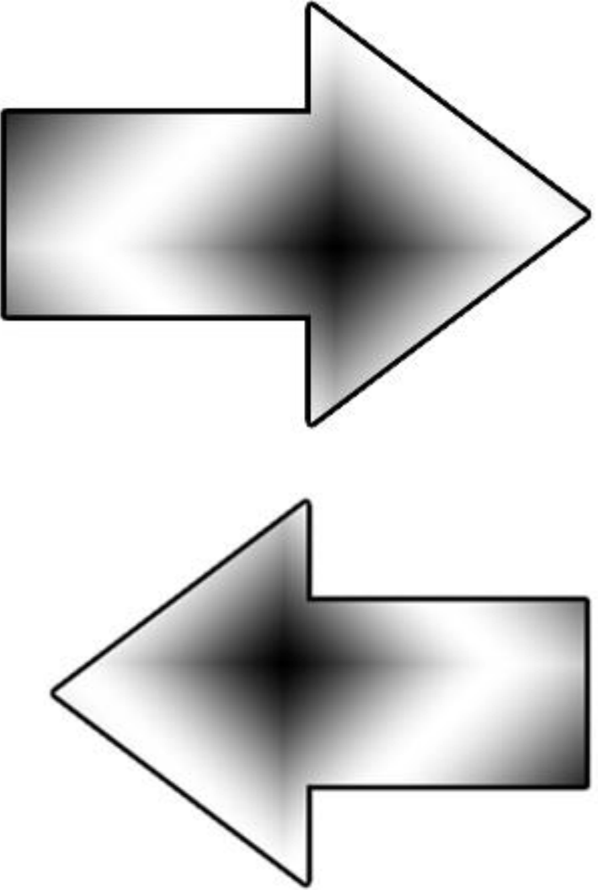


Figura 91: *Resultado ejecutar aplicación Gamepads y Joysticks*

3.21 Música y Efectos de Sonido

El objetivo es usar **SDL_mixer** para reproducir audios.

```
//Using SDL, SDL_image, SDL_mixer, standard IO, math, and strings

#include <SDL.h>
#include <SDL_image.h>
#include <SDL_mixer.h>
#include <stdio.h>
#include <string>
```

Se emplea la librería **SDL_mixer** para reproducir audios de una forma sencilla. Al igual que con **SDL_image**, es necesario configurarla.

```
//The music that will be played
Mix_Music *gMusic = NULL;

//The sound effects that will be used
Mix_Chunk *gScratch = NULL;
Mix_Chunk *gHigh = NULL;
Mix_Chunk *gMedium = NULL;
Mix_Chunk *gLow = NULL;
```

El tipo de datos **SDL_mixer** para música es **Mix_Music** y para sonidos cortos **Mix_Chunk**. Se declaran punteros para la música y los efectos de sonido que se emplearán.

```
//Initialize SDL
      if( SDL_Init( SDL_INIT_VIDEO | SDL_INIT_AUDIO ) < 0 )

      {
printf( "SDL could not initialize! SDL Error: %s\n", SDL_GetError() );
success = false;
      }
```

Ya que se utiliza música y efectos de sonido, hay que inicializar el audio junto con el vídeo.

```
//Initialize PNG loading

int imgFlags = IMG_INIT_PNG;
if( !( IMG_Init( imgFlags ) & imgFlags ) )
{

printf( "SDL_image could not initialize! SDL_image Error: %s\n",
IMG_GetError() );

success = false;

}
```

```
 //Initialize SDL_mixer
if( Mix_OpenAudio( 44100, MIX_DEFAULT_FORMAT, 2, 2048 ) < 0 )
{
printf( "SDL_mixer could not initialize! SDL_mixer Error: %s\n",
Mix_GetError() );
success = false;
}
```

Para inicializar **SDL_mixer** hay que llamar a **Mix_OpenAudio**.

- El primer argumento establece la frecuencia de sonido (44100 es una frecuencia estándar que funciona en la mayoría de los sistemas).
- El segundo argumento determina el formato de la muestra (se emplea el de defecto).
- El tercer argumento es el número de canales del hardware (en este caso se emplean 2 canales para estéreo)
- El último argumento es el tamaño de la muestra, que determina el tamaño de las partes que se usan mientras se reproduce el sonido.

En el caso de haber errores, se informará mediante **Mix_GetError**.

```
bool loadMedia()
{
	//Loading success flag
	bool success = true;

	//Load prompt texture
if( !gPromptTexture.loadFromFile(
"21_sound_effects_and_music/prompt.png" ) )
	{
		printf( "Failed to load prompt texture!\n" );
		success = false;
	}

	//Load music
	gMusic = Mix_LoadMUS( "21_sound_effects_and_music/beat.wav" );
	if( gMusic == NULL )
	{
printf( "Failed to load beat music! SDL_mixer Error: %s\n",
Mix_GetError() );
success = false;
	}

	//Load sound effects
gScratch = Mix_LoadWAV( "21_sound_effects_and_music/scratch.wav" );
if( gScratch == NULL )
	{
```

```
printf( "Failed to load scratch sound effect! SDL_mixer Error: %s\n",
Mix_GetError() );

success = false;
        }

gHigh = Mix_LoadWAV( "21_sound_effects_and_music/high.wav" );

if( gHigh == NULL )
        {

printf( "Failed to load high sound effect! SDL_mixer Error: %s\n",
Mix_GetError() );

success = false;
        }

gMedium = Mix_LoadWAV( "21_sound_effects_and_music/medium.wav" );
        if( gMedium == NULL )
        {

printf( "Failed to load medium sound effect! SDL_mixer Error: %s\n",
Mix_GetError() );

success = false;
        }

gLow = Mix_LoadWAV( "21_sound_effects_and_music/low.wav" );
        if( gLow == NULL )
        {

printf( "Failed to load low sound effect! SDL_mixer Error: %s\n",
Mix_GetError() );

success = false;
        }

        return success;
}
```

Textura y sonido cargado. Para cargar la música se emplea **Mix_LoadMUS** y para los efectos de sonido **Mix_LoadWAV.**

```
void close()
{
        //Free loaded images
        gPromptTexture.free();

        //Free the sound effects
        Mix_FreeChunk( gScratch );
        Mix_FreeChunk( gHigh );
        Mix_FreeChunk( gMedium );
        Mix_FreeChunk( gLow );
        gScratch = NULL;
        gHigh = NULL;
        gMedium = NULL;
        gLow = NULL;
```

```
        //Free the music
        Mix_FreeMusic( gMusic );
        gMusic = NULL;

        //Destroy window
        SDL_DestroyRenderer( gRenderer );
        SDL_DestroyWindow( gWindow );
        gWindow = NULL;
        gRenderer = NULL;

        //Quit SDL subsystems
        Mix_Quit();
        IMG_Quit();
        SDL_Quit();
}
```

Una vez se ha terminado con el audio y se quiere liberar, se llama a **Mix_FreeMusic** para liberar la música y a **Mix_FreeChunk** para liberar los efectos de sonido. Para cerrar **SDL_mixer** se llama a **Mix_Quit**.

```
//Handle key press

else if( e.type == SDL_KEYDOWN )
{
switch( e.key.keysym.sym )
{

//Play high sound effect

case SDLK_1:
Mix_PlayChannel( -1, gHigh, 0 );
break;

//Play medium sound effect

case SDLK_2:
Mix_PlayChannel( -1, gMedium, 0 );
break;

//Play low sound effect

case SDLK_3:
Mix_PlayChannel( -1, gLow, 0 );
break;

//Play scratch sound effect

case SDLK_4:
Mix_PlayChannel( -1, gScratch, 0 );
break;
```

En el bucle de eventos se reproducen efectos de sonido si se presionan las teclas 1, 2, 3 ó 4.

La manera de reproducir un **Mix_Chunk** es llamar a **Mix_PlayChannel**:

- El primer argumento es el canal por el que se quiere reproducir (se establece el canal a -1, por lo que utilizará el canal disponible más cercano).

- El segundo argumento es el efecto de sonido.

- El último argumento es el número de veces que se quiere repetir el efecto (como sólo se quiere reproducir una vez por cada tecla presionada, se establecerán 0 repeticiones).

Un canal, en este caso, no es lo mismo que un canal de hardware, que puede representar el canal izquierdo y derecho de un sistema estéreo. Cada efecto de sonido reproducido tiene un canal asociado. Cuando se pausa o se para un efecto, puede detenerse su canal.

```
case SDLK_9:

//If there is no music playing

if( Mix_PlayingMusic() == 0 )
{
//Play the music
Mix_PlayMusic( gMusic, -1 );
}

//If music is being played

else
{

//If the music is paused

if( Mix_PausedMusic() == 1 )
{

//Resume the music
Mix_ResumeMusic();

}

//If the music is playing

else

{
//Pause the music
Mix_PauseMusic();
}

}

break;

case SDLK_0:
```

```
//Stop the music

Mix_HaltMusic();
break;

}
```

En este ejemplo, se reproducirá/pausará la música mediante la pulsación de la tecla 9 y se detendrá al pulsar el 0.

Cuando la tecla 9 está pulsada, primero se comprueba si la música se está reproduciendo o no mediante **Mix_PlayingMusic**. Si no se está reproduciendo, la música empieza gracias a **Mix_PlayMusic**. El primer argumento es la música que se quiere reproducir y el último es el número de veces que se va a repetir. (El valor -1 es especial, ya que se mantendrá el bucle hasta que se detenga.)

Mix_PausedMusic comprueba si la música está pausada en el caso de que se esté reproduciendo. Si estaba pausada, se reanudará utilizando **Mix_ResumeMusic**, mientras que si nolo estaba se pausará mediante **Mix_PauseMusic**.

Al pulsar el 0, se detendrá la música, si se está reproduciendo, mediante **Mix_HaltMusic**.

Al ejecutar el programa aparece la imagen:

Press 1, 2, 3, or 4 to
play a sound effect

Press 9 to play or pause
the music

Press 0 to stop the music

Figura 92: *Resultado ejecutar aplicación Música y Efectos de Sonido*

De modo que al pulsar 1, 2, 3, 4 se reproducirá un efecto de sonido; al pulsar el 9 se reproducirá música o se pausará y al pulsar 0 la música se detendrá.

3.22 Control del Tiempo (Timing)

Un aspecto importante de cualquier API de juego es la capacidad para controlar el tiempo. El objetivo consiste en hacer un temporizador que se pueda reiniciar.

```
//Using SDL, SDL_image, SDL_ttf, standard IO, strings, and string
streams

#include <SDL.h>
#include <SDL_image.h>
#include <SDL_ttf.h>
#include <stdio.h>
#include <string>
#include <sstream>
```

Dado que se utilizarán streams, será necesario incluir el cabecero sstream.

```
bool loadMedia()
{
	//Loading success flag
	bool success = true;

	//Open the font
	gFont = TTF_OpenFont( "22_timing/lazy.ttf", 28 );
	if( gFont == NULL )
	{
printf( "Failed to load lazy font! SDL_ttf Error: %s\n",
TTF_GetError() );

success = false;
	}

	else
	{
		//Set text color as black
		SDL_Color textColor = { 0, 0, 0, 255 };

		//Load prompt texture
if( !gPromptTextTexture.loadFromRenderedText( "Press Enter to Reset
Start Time.", textColor ) )

		{
			printf( "Unable to render prompt texture!\n" );
			success = false;
		}
	}

	return success;
}
```

Se busca reducir el número de veces que se renderiza el texto. Para ello se tiene una textura para las entradas y una textura para mostrar el tiempo actual en milisegundos. La textura de tiempo cambia cada fotograma, por lo que hay que renderizar cada fotograma. Sin embargo, la textura de la entrada no cambia, por lo que sólo habrá que renderizarla una vez en la función de carga de archivos.

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//Set text color as black
SDL_Color textColor = { 0, 0, 0, 255 };

//Current time start time
Uint32 startTime = 0;

//In memory text stream
std::stringstream timeText;
```

Antes de entrar al bucle principal se declaran algunas variables:

- Variable startTime, que es un entero sin signo de 32 bits.
- Variable timeText, que es un stream de cadena.

```
                    //While application is running
                    while( !quit )
                    {
                          //Handle events on queue
                          while( SDL_PollEvent( &e ) != 0 )
                          {
                                //User requests quit
                                if( e.type == SDL_QUIT )
                                {
                                      quit = true;
                                }
//Reset start time on return keypress

else if( e.type == SDL_KEYDOWN && e.key.keysym.sym == SDLK_RETURN )
                                {
                                      startTime = SDL_GetTicks();
                                }
                          }
```

SDL_GetTicks devuelve el tiempo transcurrido desde el inicio del programa en milisegundos. En este caso, se tendrá un temporizador que se reinicia cada vez que se pulsa la tecla enter.

El tiempo de inicio se inicializó a 0 al comienzo del programa, lo que quiere decir que el tiempo del temporizador es el tiempo actual desde que el programa empezó devuelto por **SDL_GetTicks**.

```
//Set text to be rendered

imeText.str( "" );
timeText << "Milliseconds since start time " << SDL_GetTicks() -
startTime;
```

Aquí es donde se usa el stream de cadena.

En primer lugar, se llama str con una cadena vacía para inicializarla como vacía. Seguidamente se le trata como cout y se imprime: "Milisegundos desde el momento de inicio" y el tiempo actual menos el tiempo de inicio relativo, es decir, se imprimirá el tiempo desde la última vez que se inició el temporizador.

```
//Render text
if( !gTimeTextTexture.loadFromRenderedText( timeText.str().c_str(),
textColor ) )
                        {
            printf( "Unable to render time texture!\n" );
                        }
```

Una vez se tiene el tiempo en un stream de cadena, puede obtenerse una cadena de él y usarla para renderizar el tiempo actual a una textura.

```
//Clear screen

SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render textures

gPromptTextTexture.render( ( SCREEN_WIDTH -
gPromptTextTexture.getWidth() ) / 2, 0 );

gTimeTextTexture.render( ( SCREEN_WIDTH -
gPromptTextTexture.getWidth() ) / 2,
( SCREEN_HEIGHT - gPromptTextTexture.getHeight() ) / 2 );

//Update screen
SDL_RenderPresent( gRenderer );

                  }
```

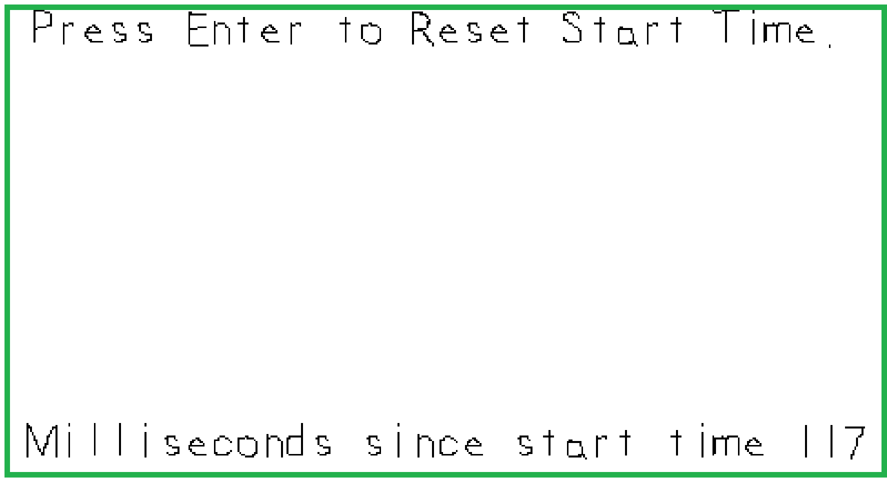


Figura 93: *Resultado ejecutar aplicación Control del Tiempo*

Press Enter to Reset Start Time.

Milliseconds since start time 333

__Figura 94:__ Resultado ejecutar aplicación Control del Tiempo (II)

__Figura 95:__ Resultado ejecutar aplicación Control del Tiempo (III)

3.23 Temporizadores Avanzados

Al tener el tiempo en un stream de cadena, puede obtenerse una cadena de él y usarla para renderizar el tiempo actual en una textura.

```
//The application time based timer
class LTimer
{
    public:

            //Initializes variables
            LTimer();

            //The various clock actions
            void start();
            void stop();
            void pause();
            void unpause();

            //Gets the timer's time
            Uint32 getTicks();

            //Checks the status of the timer
            bool isStarted();
            bool isPaused();

    private:

            //The clock time when the timer started
            Uint32 mStartTicks;

            //The ticks stored when the timer was paused
            Uint32 mPausedTicks;

            //The timer status
            bool mPaused;
            bool mStarted;
};
```

Se hace una clase temporizador para las nuevas características que tendrá todas las funciones básicas para iniciar/ detener/ pausar/ retomar el temporizador y comprobar su estado. El momento de inicio se obtiene como anteriormente: una variable almacena el tiempo cuando se detiene y los flags para hacer un seguimiento con el que se sabrá si el temporizador está en funcionamiento o pausado.

```
LTimer::LTimer()
{
    //Initialize the variables
    mStartTicks = 0;
    mPausedTicks = 0;

    mPaused = false;
    mStarted = false;
}
```

Los datos internos se inicializan mediante el constructor.

```
void LTimer::start()
{
    //Start the timer
    mStarted = true;

    //Unpause the timer
    mPaused = false;

    //Get the current clock time
    mStartTicks = SDL_GetTicks();
    mPausedTicks = 0;
}
```

La función de inicio fija los flags iniciados y pausados, así como obtiene el momento de inicio del temporizador e inicializa el tiempo de pausa a 0. Para reiniciar este temporizador sólo hay que llamarlo de nuevo. (Dado que se puede iniciar el temporizador si está pausado o en funcionamiento, es necesario asegurarse de borrar los datos pausados).

```
void LTimer::stop()
{
    //Stop the timer
    mStarted = false;

    //Unpause the timer
    mPaused = false;

      //Clear tick variables
      mStartTicks = 0;
      mPausedTicks = 0;
}
```

La función de parada simplemente reinicializa todas las variables.

```
void LTimer::pause()
{
    //If the timer is running and isn't already paused
    if( mStarted && !mPaused )
    {
        //Pause the timer
        mPaused = true;

        //Calculate the paused ticks
        mPausedTicks = SDL_GetTicks() - mStartTicks;
        mStartTicks = 0;
    }
}
```

Al pausar, se comprueba si el temporizador está en marcha (ya que no tiene sentido pausar el temporizador si no se ha iniciado).

Si el temporizador está en marcha, se inicia el flag de pausa, se almacena el momento en el que el temporizador se detuvo en mPausedTicks y se resetea el tiempo de inicio.

```
void LTimer::unpause()
{
    //If the timer is running and paused
    if( mStarted && mPaused )
    {
        //Unpause the timer
        mPaused = false;

        //Reset the starting ticks
        mStartTicks = SDL_GetTicks() - mPausedTicks;

        //Reset the paused ticks
        mPausedTicks = 0;
    }
}
```

Cuando se reanuda el temporizador, el flag de pausa toma el valor false y se fija el nuevo tiempo de inicio.

```
Uint32 LTimer::getTicks()
{
      //The actual timer time
      Uint32 time = 0;

    //If the timer is running
    if( mStarted )
    {
        //If the timer is paused
        if( mPaused )
        {
            //Return the number of ticks when the timer was paused
            time = mPausedTicks;
        }
        else
        {
            //Return the current time minus the start time
            time = SDL_GetTicks() - mStartTicks;
        }
    }

    return time;
}
```

Obtener el tiempo es complicado, ya que el temporizador puede estar en marcha, pausado o detenido.

Se tienen los tres siguientes casos:

- Si el temporizador está detenido, se devuelve el valor inicial 0.
- Si el temporizador está pausado, se devuelve el tiempo almacenado cuando se pausó.

- Si el temporizador está en marcha y no está pausado, se devuelve el tiempo relativo a cuando se inició.

```
bool LTimer::isStarted()
{
      //Timer is running and paused or unpaused
    return mStarted;
}

bool LTimer::isPaused()
{
      //Timer is running and paused
    return mPaused && mStarted;
}
```

Éstas son las funciones para comprobar el estado del temporizador.

```
                  //Main loop flag
                  bool quit = false;

                  //Event handler
                  SDL_Event e;

                  //Set text color as black
                  SDL_Color textColor = { 0, 0, 0, 255 };

                  //The application timer
                  LTimer timer;

                  //In memory text stream
                  std::stringstream timeText;
```

Antes del bucle principal se declara un objeto temporizador y un stream de cadena para convertir el valor de tiempo a texto.

```
//Start/stop

if( e.key.keysym.sym == SDLK_s )

{
if( timer.isStarted() )
{
timer.stop();
}

else

{
timer.start();
}
```

```
}

//Pause/unpause

else if( e.key.keysym.sym == SDLK_p )
{
if( timer.isPaused() )
{
timer.unpause();
}

else
{
timer.pause();
}

}

}
```

Al pulsar la tecla S, se comprueba si el temporizador se ha iniciado. Si es así, se detiene. En caso contrario, se iniciará.

Al pulsar la tecla P, se comprueba si el temporizador está pausado. Si es así, se reanuda. En caso contrario, se pausará.

```
//Set text to be rendered

timeText.str( "" );
timeText << "Seconds since start time " << ( timer.getTicks() / 1000.f
) ;

//Render text

if( !gTimeTextTexture.loadFromRenderedText( timeText.str().c_str(),
textColor ) )

{
printf( "Unable to render time texture!\n" );
}

//Clear screen
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render textures
gStartPromptTexture.render( ( SCREEN_WIDTH -
gStartPromptTexture.getWidth() ) / 2, 0 );

gPausePromptTexture.render(( SCREEN_WIDTH -
gPausePromptTexture.getWidth())/2, gStartPromptTexture.getHeight() );

gTimeTextTexture.render( ( SCREEN_WIDTH - gTimeTextTexture.getWidth()
) / 2,( SCREEN_HEIGHT - gTimeTextTexture.getHeight() ) / 2 );

//Update screen
SDL_RenderPresent( gRenderer );
```

Antes de renderizar, se escribe el tiempo actual en un stream de cadena. La razón por la que se divide por 1000 es para obtener el tiempo en segundos en lugar de milisegundos.

Una vez se ha renderizado el texto a la textura, finalmente se dibujan las texturas en la pantalla.

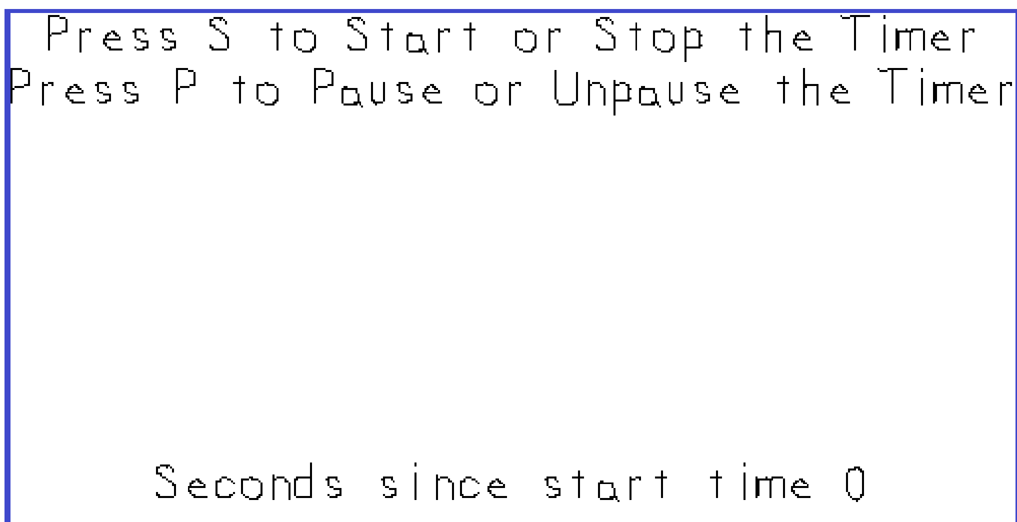


Figura 96: *Temporizadores Avanzados*

Press S to Start or Stop the Timer
Press P to Pause or Unpause the Timer
Seconds since start time 3.731

Figura 97: *Temporizadores Avanzados (II)*

3.24 Cálculo de Fotogramas Por Segundo

El objetivo es medir fps (frecuencias por segundo, velocidad de fotogramas).

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//Set text color as black
SDL_Color textColor = { 0, 0, 0, 255 };

//The frames per second timer
LTimer fpsTimer;

//In memory text stream
std::stringstream timeText;

//Start counting frames per second
int countedFrames = 0;
fpsTimer.start();
```

Para el cálculo de fotogramas por segundo hay que hacer un seguimiento del número de fotogramas procesados y el número de segundos transcurridos. Para ello, antes de entrar al bucle principal, se inicia el temporizador empleado para calcular fps y se declara una variable que controle el número de fotogramas procesados.

```
//Calculate and correct fps

float avgFPS = countedFrames / ( fpsTimer.getTicks() / 1000.f );
if( avgFPS > 2000000 )
                {
                        avgFPS = 0;
                }
```

Para calcular el número de fotogramas por segundo se toma el número de fotogramas renderizados y se divide por el número de segundos que han pasado. Es posible que pase una pequeña cantidad de tiempo en el primer fotograma, de modo que al calcular los fps se obtiene un valor muy alto. Por esta razón se corregirá este valor si es muy alto.

```
//Set text to be rendered

timeText.str( "" );
timeText << "Average Frames Per Second " << avgFPS;

//Render text
if( !gFPSTextTexture.loadFromRenderedText( timeText.str().c_str(),
textColor ) )
```

```
{

printf( "Unable to render FPS texture!\n" );

}

//Clear screen
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render textures
gFPSTextTexture.render( ( SCREEN_WIDTH - gFPSTextTexture.getWidth() )
/ 2,( SCREEN_HEIGHT - gFPSTextTexture.getHeight() ) / 2 );

//Update screen
SDL_RenderPresent( gRenderer );
++countedFrames;

}
```

Cuando se calculan los fps, se renderiza el valor y se muestra en pantalla. Una vez se ha renderizado todo, se incrementa el contador de fotogramas. (Dado que este programa está sincronizado verticalmente probablemente se obtengan 60 fps.)

Average Frames Per Second 61.6175

Figura 98: *Resultado ejecutar aplicación Cálculo de Fotogramas Por Segundo*

Average Frames Per Second 59.9461

Figura 99: *Resultado ejecutar aplicación Cálculo de Fotogramas Por Segundo (II)*

Average Frames Per Second 59.9034

Figura 100: *Resultado ejecutar aplicación Cálculo de Fotogramas Por Segundo (III)*

3.25 Limitación de la Velocidad de Fotogramas Por Segundo

Con los temporizadores de SDL puede limitarse la velocidad de los fotogramas. El objetivo es deshabilitar la sincronización vertical y mantener la velocidad de fotogramas máxima.

```
//Screen dimension constants
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
const int SCREEN_FPS = 60;
const int SCREEN_TICK_PER_FRAME = 1000 / SCREEN_FPS;
```

En este programa se esperará hasta que el tiempo del fotograma se complete. (Por ejemplo, cuando se quiere renderizar a 60 fotogramas por segundo, hay que esperar 16'66666 milisegundos por fotograma, $1000ms/60$ fotogramas) Por este motivo, se calcula el número de ticks por fotograma en milisegundos.

```
//Create renderer for window
gRenderer = SDL_CreateRenderer( gWindow, -1, SDL_RENDERER_ACCELERATED
);
```

Se deshabilita la sincronización vertical, ya que se quiere limitar manualmente la velocidad de los fotogramas.

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//Set text color as black
SDL_Color textColor = { 0, 0, 0, 255 };

//The frames per second timer
LTimer fpsTimer;

//The frames per second cap timer
LTimer capTimer;

//In memory text stream
std::stringstream timeText;

//Start counting frames per second
int countedFrames = 0;
fpsTimer.start();
```

No sólo será necesario un temporizador para la velocidad de los fotogramas, sino también uno que limite los fotogramas por segundo. Antes de entrar al bucle principal se declaran ciertas variables y se inicia el temporizador calculador de fotogramas por segundo.

```
                    //While application is running
                    while( !quit )
                    {
                          //Start cap timer
                          capTimer.start();
```

Para limitar los fotogramas por segundo es necesario conocer la longitud del fotograma que se ha escogido para renderizar. Ésa es la razón por la que se inicia un temporizador al comienzo de cada fotograma.

```
//Handle events on queue
while( SDL_PollEvent( &e ) != 0 )
                        {
                              //User requests quit
                              if( e.type == SDL_QUIT )
                              {
                                    quit = true;
                              }
                        }

//Calculate and correct fps
float avgFPS = countedFrames / ( fpsTimer.getTicks() / 1000.f );

if( avgFPS > 2000000 )
                        {
                              avgFPS = 0;
                        }

//Set text to be rendered

timeText.str( "" );
timeText << "Average Frames Per Second (With Cap) " << avgFPS;

//Render text
if( !gFPSTextTexture.loadFromRenderedText( timeText.str().c_str(),
textColor ) )
                        {
                        printf( "Unable to render FPS texture!\n" );
                        }

//Clear screen
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render textures
gFPSTextTexture.render(( SCREEN_WIDTH - gFPSTextTexture.getWidth())/2,
( SCREEN_HEIGHT - gFPSTextTexture.getHeight() ) / 2 );

//Update screen
SDL_RenderPresent( gRenderer );
++countedFrames;
```

Renderizado de los fotogramas y código del cálculo de fotogramas por segundo.

```
//Update screen
SDL_RenderPresent( gRenderer );
++countedFrames;

//If frame finished early
int frameTicks = capTimer.getTicks();
if( frameTicks < SCREEN_TICK_PER_FRAME )
{

//Wait remaining time
SDL_Delay( SCREEN_TICK_PER_FRAME - frameTicks );

 }

}
```

Código para limitar la velocidad de los fotogramas.

En primer lugar, se obtiene el número de ticks tomados por el fotograma. Si este número es menor que el número de ticks necesitado por fotograma, se retrasa el tiempo restante para evitar que la aplicación se ejecute demasiado rápido.

La razón por la que se emplea la sincronización vertical en lugar de limitar manualmente la velocidad de los fotogramas es que en la segunda, cuando la aplicación se está ejecutando, se ejecuta ligeramente más rápida. Puesto que se usan enteros, los ticks por fotograma serán 16 milisegundos en lugar de 16,6666 milisegundos.

Average Frames Per Second (With Cap) 63.3094

Figura 101: *Resultado ejecutar aplicación Limitación de la Velocidad de Fotogramas Por Segundo*

Average Frames Per Second (With Cap) 63.0311

Figura 102: *Resultado ejecutar aplicación Limitación de la Velocidad de Fotogramas Por Segundo (II)*

Average Frames Per Second (With Cap) 63.4872

Figura 103: *Resultado ejecutar aplicación Limitación de la Velocidad de Fotogramas Por Segundo (III)*

3.26 Movimiento

Lo que se necesita para el movimiento es saber cómo renderizar, controlar la entrada y trabajar con el tiempo. El objetivo será hacer que un punto sencillo se mueva.

```
//The dot that will move around on the screen
class Dot
{
    public:
        //The dimensions of the dot
        static const int DOT_WIDTH = 20;
        static const int DOT_HEIGHT = 20;

        //Maximum axis velocity of the dot
        static const int DOT_VEL = 10;

        //Initializes the variables
        Dot();

        //Takes key presses and adjusts the dot's velocity
        void handleEvent( SDL_Event& e );

        //Moves the dot
        void move();

        //Shows the dot on the screen
        void render();

    private:
        //The X and Y offsets of the dot
        int mPosX, mPosY;

        //The velocity of the dot
        int mVelX, mVelY;
};
```

Ésta es la clase para el punto que estará en movimiento. Tiene constantes determinadas para las dimensiones y velocidad, así como el constructor, controlador de evento, función para mover cada fotograma y función para renderizarlo.

En cuanto a los datos, tiene variables para la posición (X,Y) además de la velocidad (X,Y).

```
Dot::Dot()
{
    //Initialize the offsets
    mPosX = 0;
    mPosY = 0;

    //Initialize the velocity
    mVelX = 0;
    mVelY = 0;
}
```

El constructor simplemente inicializa las variables.

```
void Dot::handleEvent( SDL_Event& e )
{
    //If a key was pressed
      if( e.type == SDL_KEYDOWN && e.key.repeat == 0 )
    {
        //Adjust the velocity
        switch( e.key.keysym.sym )
        {
            case SDLK_UP: mVelY -= DOT_VEL; break;
            case SDLK_DOWN: mVelY += DOT_VEL; break;
            case SDLK_LEFT: mVelX -= DOT_VEL; break;
            case SDLK_RIGHT: mVelX += DOT_VEL; break;
        }
    }
```

En el controlador de eventos se establece la velocidad basándose en la pulsación de las teclas.

Si se tuviera que añadir la posición X cada vez que se pulsa la tecla derecha, se tendría que pulsar repetidamente la tecla derecha para mantener el movimiento. Si se fija la velocidad, sólo habrá que pulsar la tecla una vez.

Se comprueba si la repetición de la tecla es 0, ya que la repetición de la tecla está activa por defecto, y si se mantiene pulsada informará de múltiples pulsaciones de teclas.

La velocidad será la rapidez/dirección de un objeto, es decir, si un objeto se está moviendo a la derecha 10 pixeles por fotograma, tiene una velocidad de 10, mientras que si se mueve a la izquierda 10 pixeles tiene una velocidad de -10. Si la velocidad del punto es 10 significa que 10 fotogramas después, se habrá movido 100 píxeles.

```
    //If a key was released
    else if( e.type == SDL_KEYUP && e.key.repeat == 0 )
    {
        //Adjust the velocity
        switch( e.key.keysym.sym )
        {
            case SDLK_UP: mVelY += DOT_VEL; break;
            case SDLK_DOWN: mVelY -= DOT_VEL; break;
            case SDLK_LEFT: mVelX += DOT_VEL; break;
            case SDLK_RIGHT: mVelX -= DOT_VEL; break;
        }
    }
}
```

Cuando se libera una tecla hay que deshacer el cambio de velocidad cuando se pulsó.

Al pulsar la tecla derecha se añade a la velocidad X y cuando se libera la tecla, se resta a la velocidad X para volver a 0.

```
void Dot::move()
{
    //Move the dot left or right
    mPosX += mVelX;
    //If the dot went too far to the left or right
    if( ( mPosX < 0 ) || ( mPosX + DOT_WIDTH > SCREEN_WIDTH ) )
    {
        //Move back
        mPosX -= mVelX;
    }
```

Función llamada cada fotograma para mover el punto.

- En primer lugar, se mueve el punto a lo largo del eje X en función de su velocidad.
- Tras esto, se comprueba si el punto se ha trasladado fuera de la pantalla. De ser así, se deshace el movimiento a lo largo del eje X.

```
    //Move the dot up or down
    mPosY += mVelY;

    //If the dot went too far up or down
    if( ( mPosY < 0 ) || ( mPosY + DOT_HEIGHT > SCREEN_HEIGHT ) )
    {
        //Move back
        mPosY -= mVelY;
    }
}
```

A continuación se realiza lo mismo con el eje Y.

```
void Dot::render()
{
    //Show the dot
      gDotTexture.render( mPosX, mPosY );
}
```

Se renderiza la textura del punto en la posición del mismo.

```
                //Main loop flag
                bool quit = false;

                //Event handler
                SDL_Event e;

                //The dot that will be moving around on the screen
                Dot dot;
```

Antes de entrar al bucle principal se declara un objeto punto.

```
				//While application is running
				while( !quit )
				{
					//Handle events on queue
					while( SDL_PollEvent( &e ) != 0 )
					{
						//User requests quit
						if( e.type == SDL_QUIT )
						{
							quit = true;
						}

						//Handle input for the dot
						dot.handleEvent( e );
					}

		//Move the dot
		dot.move();

		//Clear screen
		SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
		SDL_RenderClear( gRenderer );

		//Render objects
		dot.render();

		//Update screen
		SDL_RenderPresent( gRenderer );

				}
```

En el bucle de eventos se controlan los eventos para el punto. Tras esto, se actualiza la posición del punto y se renderiza a la pantalla.

Al ejecutar el programa se obtiene:

- Pulsando la tecla de flecha derecha:

Figura 104: *Resultado ejecutar aplicación Movimiento*

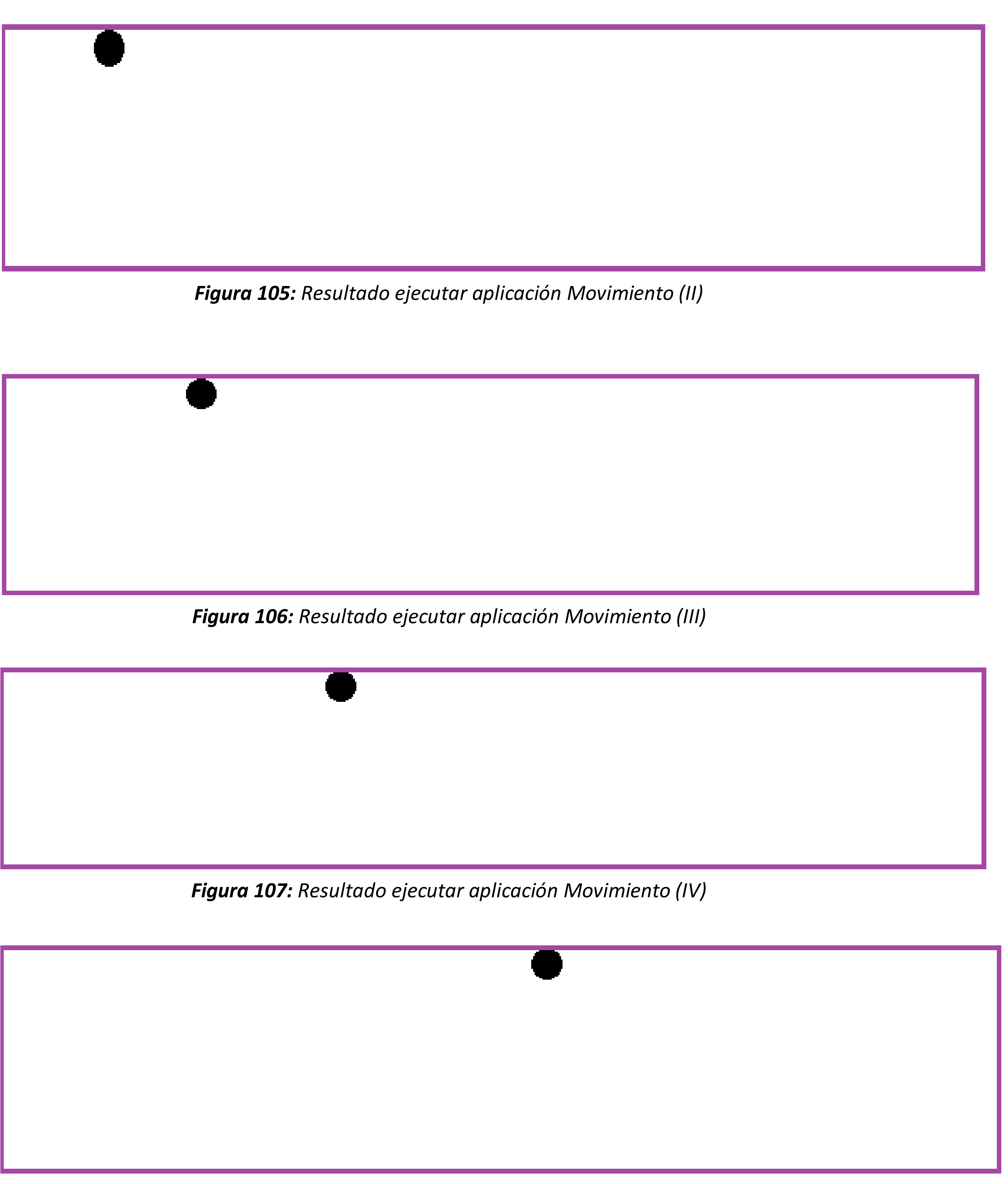


Figura 105: *Resultado ejecutar aplicación Movimiento (II)*

Figura 106: *Resultado ejecutar aplicación Movimiento (III)*

Figura 107: *Resultado ejecutar aplicación Movimiento (IV)*

Figura 108: *Resultado ejecutar aplicación Movimiento (V)*

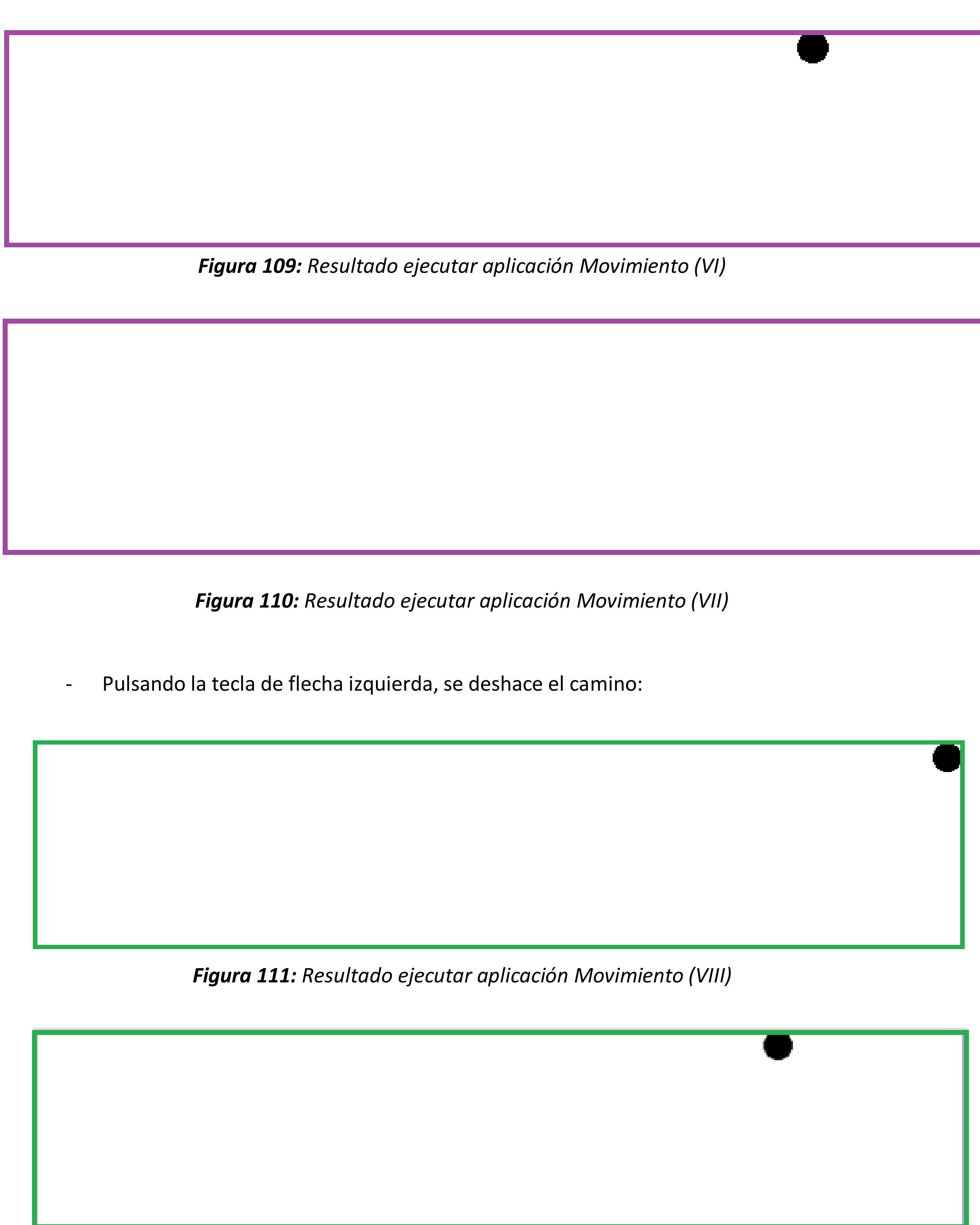


Figura 109: *Resultado ejecutar aplicación Movimiento (VI)*

Figura 110: *Resultado ejecutar aplicación Movimiento (VII)*

- Pulsando la tecla de flecha izquierda, se deshace el camino:

Figura 111: *Resultado ejecutar aplicación Movimiento (VIII)*

Figura 112: *Resultado ejecutar aplicación Movimiento (IX)*

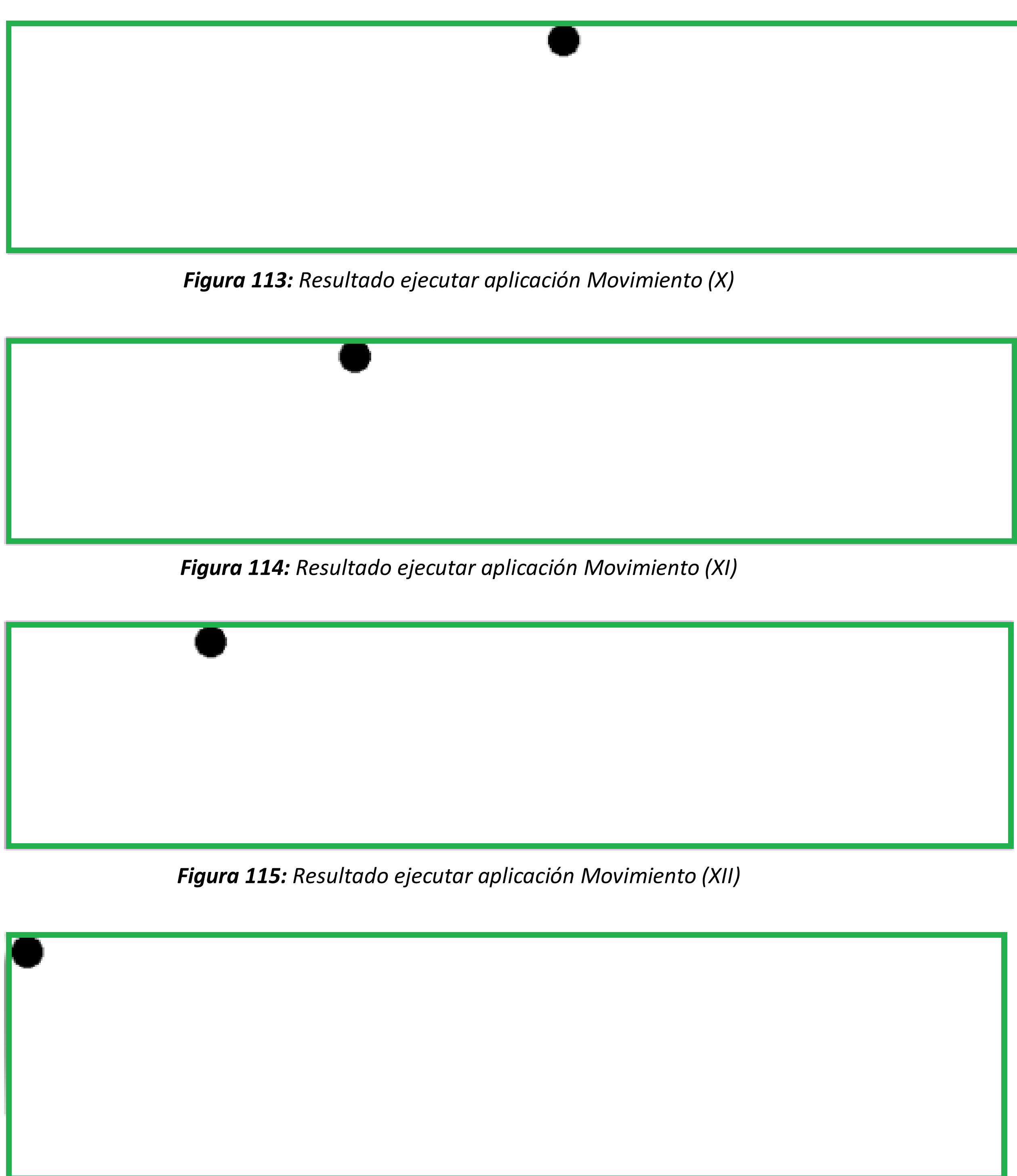


Figura 113: *Resultado ejecutar aplicación Movimiento (X)*

Figura 114: *Resultado ejecutar aplicación Movimiento (XI)*

Figura 115: *Resultado ejecutar aplicación Movimiento (XII)*

Figura 116: *Resultado ejecutar aplicación Movimiento (XII)*

3.27 Detección de la colisión

La colisión de cajas es una forma estándar de comprobar la colisión entre dos objetos. Dos polígonos han colisionado cuando no hay separación entre ellos.

Ejemplo de dos objetos que no han colisionado. Como puede verse, sus proyecciones X están en la base y sus proyecciones Y a la izquierda.

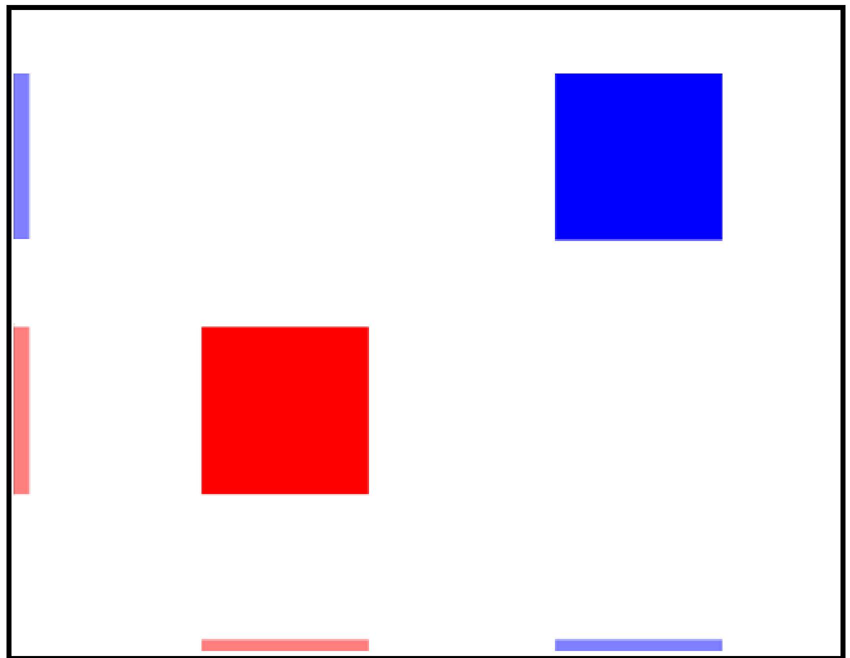


Figura 117: *Ejemplo de no colisión entre dos polígonos*

Ejemplo de dos objetos que han chocado en el eje Y aunque estén separados en el eje X:

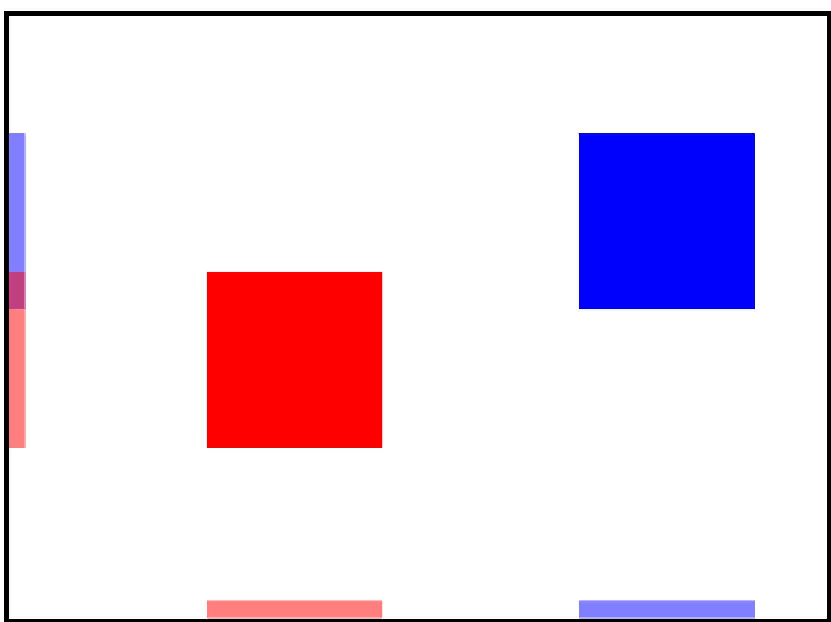


Figura 118: *Ejemplo de colisión entre dos polígonos sólo en el eje Y*

Ejemplo de dos objetos que han chocado en el eje X aunque estén separados en el eje Y:

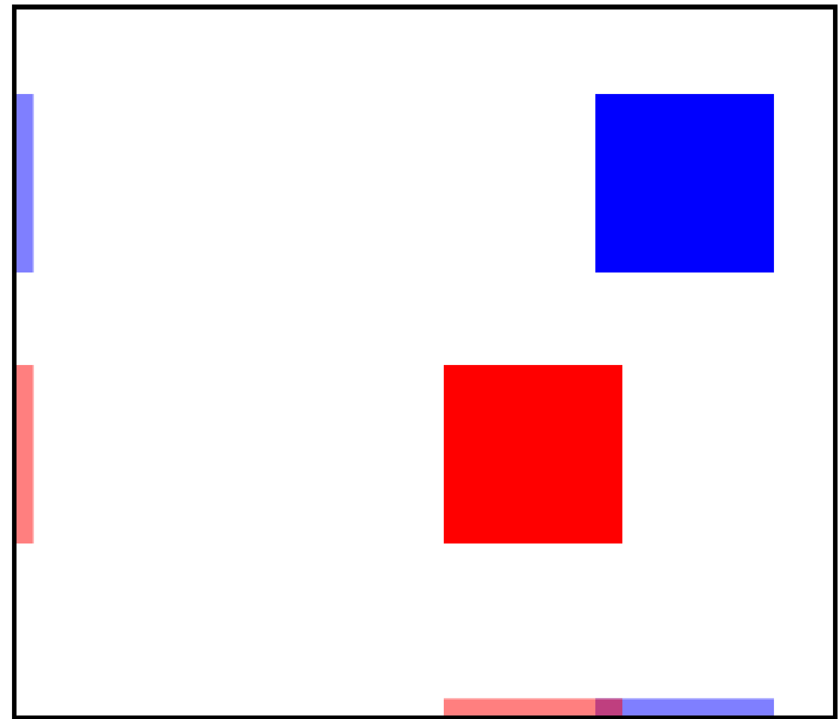


Figura 119: *Ejemplo de colisión entre dos polígonos sólo en el eje X*

Cuando no hay separación en ninguno de los ejes se denomina colisión:

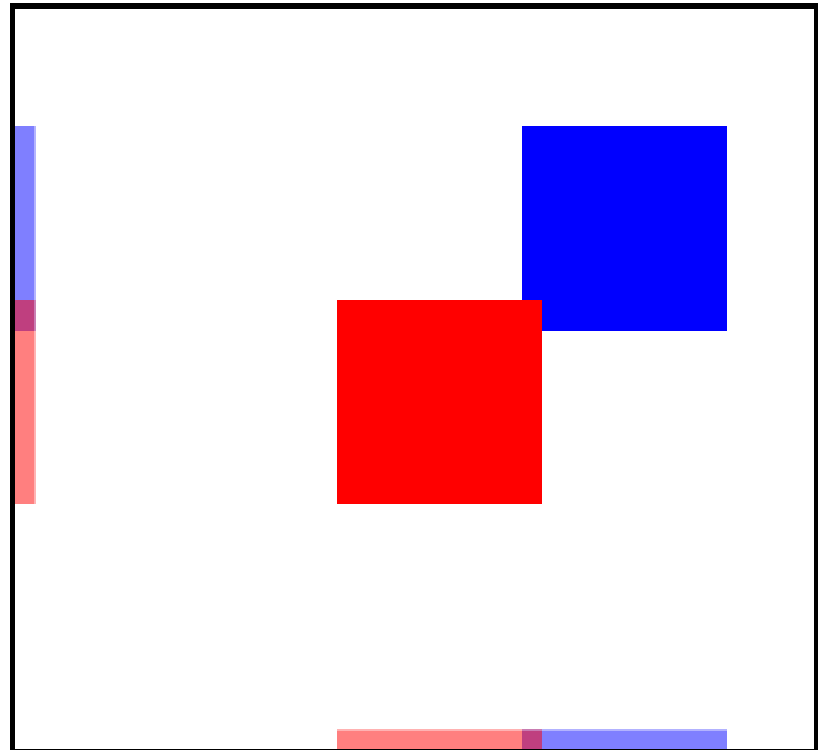


Figura 120: *Ejemplo de colisión entre dos polígonos en el eje X e Y*

Esta forma de detección de colisión se denomina “prueba de separación de ejes” y con ella se intenta encontrar el eje en el que se separan los objetos. Si no hay separación de ejes, los objetos están colisionando.

```
//The dot that will move around on the screen
class Dot
{
    public:
            //The dimensions of the dot
            static const int DOT_WIDTH = 20;
            static const int DOT_HEIGHT = 20;

            //Maximum axis velocity of the dot
            static const int DOT_VEL = 10;

            //Initializes the variables
            Dot();

            //Takes key presses and adjusts the dot's velocity
            void handleEvent( SDL_Event& e );

            //Moves the dot and checks collision
            void move( SDL_Rect& wall );

            //Shows the dot on the screen
            void render();

    private:
            //The X and Y offsets of the dot
            int mPosX, mPosY;

            //The velocity of the dot
            int mVelX, mVelY;

            //Dot's collision box
            SDL_Rect mCollider;
};
```

La función de desplazamiento toma un rectángulo, que es el cuadro de colisión para la pared, y el punto tiene un miembro de dato mCollider para representar el cuadro de colisión.

```
//Starts up SDL and creates window
bool init();

//Loads media
bool loadMedia();

//Frees media and shuts down SDL
void close();

//Box collision detector
bool checkCollision( SDL_Rect a, SDL_Rect b );
```

Se declara una función para comprobar la colisión entre dos objetos.

```
Dot::Dot()
{
    //Initialize the offsets
    mPosX = 0;
    mPosY = 0;

      //Set collision box dimension
      mCollider.w = DOT_WIDTH;
      mCollider.h = DOT_HEIGHT;

    //Initialize the velocity
    mVelX = 0;
    mVelY = 0;
}
```

Mediante el constructor se establecen las dimensiones del cuadro de colisión.

```
void Dot::move( SDL_Rect& wall )
{
    //Move the dot left or right
mPosX += mVelX;
mCollider.x = mPosX;

    //If the dot collided or went too far to the left or right

if((mPosX<0)||(mPosX+DOT_WIDTH>SCREEN_WIDTH)||checkCollision(mCollider
,wall))
    {
        //Move back
      mPosX -= mVelX;
      mCollider.x = mPosX;
    }

    //Move the dot up or down
mPosY += mVelY;
mCollider.y = mPosY;

    //If the dot collided or went too far up or down

if((mPosY<0)||(mPosY+DOT_HEIGHT>SCREEN_HEIGHT)||checkCollision(mCollid
er,wall))
    {
        //Move back
      mPosY -= mVelY;
      mCollider.y = mPosY;
    }
}
```

Ésta es la función de movimiento, que ahora comprueba si se ha golpeado la pared. Funciona igual que antes con la salvedad de que ahora hace que el punto regrese si se sale de la pantalla o golpea la pared.

- En primer lugar, se mueve el punto a lo largo del eje X, pero también se cambia la posición del cuadro de colisión. Siempre que se cambie la posición del punto, la posición del cuadro de colisión tiene que seguirla.

- Seguidamente se comprueba si el punto ha salido de la pantalla o ha golpeado la pares. Si lo ha hecho, el punto regresa su posición.

- Finalmente, se repite para el movimiento a lo largo del eje Y.

```
bool checkCollision( SDL_Rect a, SDL_Rect b )
{
    //The sides of the rectangles
    int leftA, leftB;
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    //Calculate the sides of rect A
    leftA = a.x;
    rightA = a.x + a.w;
    topA = a.y;
    bottomA = a.y + a.h;

    //Calculate the sides of rect B
    leftB = b.x;
    rightB = b.x + b.w;
    topB = b.y;
    bottomB = b.y + b.h;
```

Detección de la colisión. Este código calcula la parte superior/inferior e izquierda/derecha de cada uno de los objetos de la colisión.

```
    //If any of the sides from A are outside of B

if( bottomA <= topB )
    {
        return false;
    }

if( topA >= bottomB )
    {
        return false;
    }

if( rightA <= leftB )
    {
        return false;
    }

if( leftA >= rightB )
    {
        return false;
    }

    //If none of the sides from A are outside B

return true;
}
```

Ésta es la prueba de separación de los ejes.

- En primer lugar, se comprueba la parte superior e inferior de los objetos para determinar si hay separación en el eje Y.

- A continuación, se comprueba la parte izquierda y derecha para determinar si hay separación en el eje X. Si hubiera alguna separación, habría una colisión y se devolvería TRUE.

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//The dot that will be moving around on the screen
Dot dot;

//Set the wall
SDL_Rect wall;
wall.x = 300;
wall.y = 40;
wall.w = 40;
wall.h = 400;
```

Antes de entrar al bucle principal se declara el punto y se define la posición y dimensiones de la pared.

```
//While application is running
while( !quit )
{
	//Handle events on queue
	while( SDL_PollEvent( &e ) != 0 )
	{
		//User requests quit
		if( e.type == SDL_QUIT )
		{
			quit = true;
		}

		//Handle input for the dot
		dot.handleEvent( e );
	}

//Move the dot and check collision
dot.move( wall );

//Clear screen
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render wall
SDL_SetRenderDrawColor( gRenderer, 0x00, 0x00, 0x00, 0xFF );
```

```
SDL_RenderDrawRect( gRenderer, &wall );

//Render dot
dot.render();

//Update screen
SDL_RenderPresent( gRenderer );

}
```

Bucle principal con los eventos para controlar el punto, que se mueve mientras se comprueba si hay colisión con la pared. Finalmente se renderiza la pared y el punto en la pantalla.

Otro aspecto importante es que los objetos son AABB u objetos de ejes alineados, es decir, sus lados están alineados con el eje X e Y.

(*) En el caso de querer usar objetos rotados, puede seguir usándose la prueba de separación de ejes. En lugar de proyectar las esquinas en los ejes X e Y, se proyectan todas las esquinas de los objetos en el eje X e Y de cada uno de ellos. A continuación, se comprueba si los objetos están separados a lo largo de cada eje).

Al ejecutar el programa se obtiene:

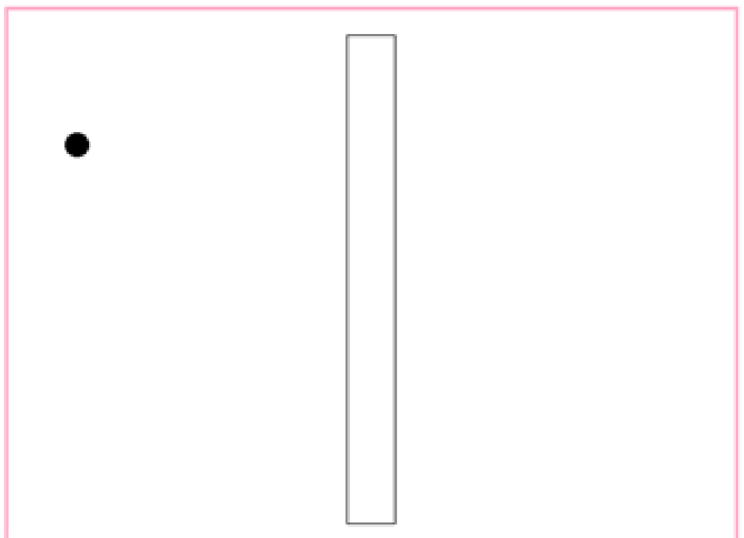


Figura 121: *Resultado ejecutar aplicación Detección de la Colisión*

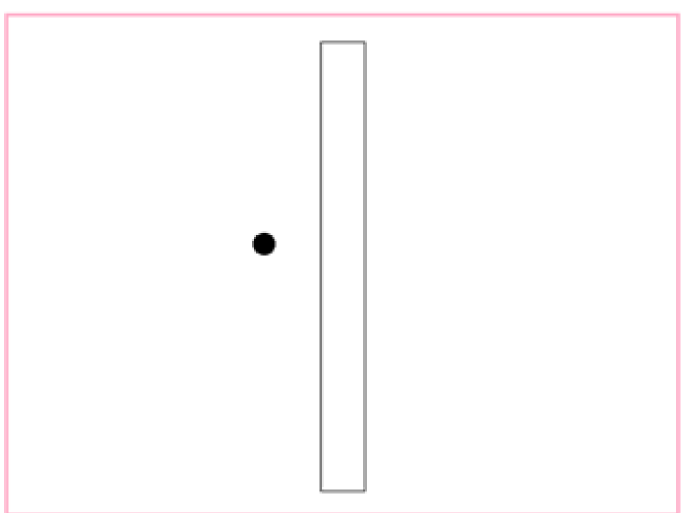


Figura 122: *Resultado ejecutar aplicación Detección de la Colisión (II)*

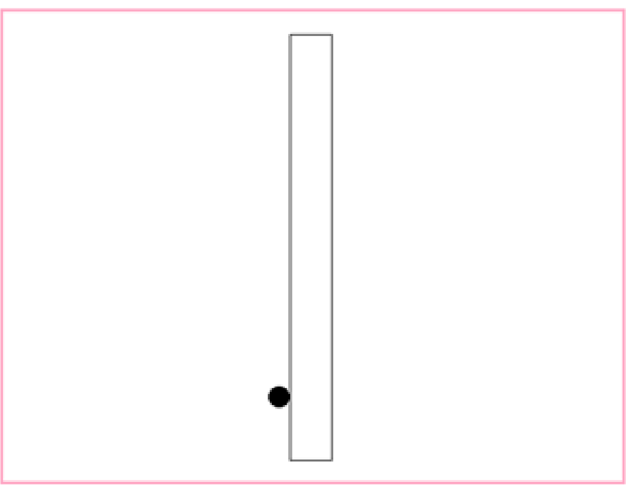


Figura 123: *Resultado ejecutar aplicación Detección de la Colisión (III)*

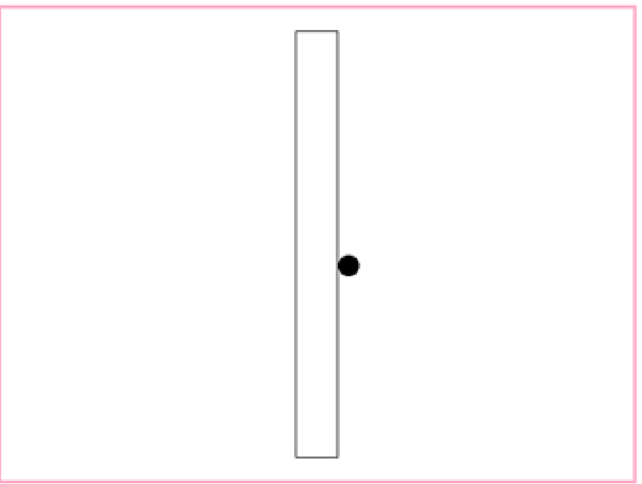


Figura 124: *Resultado ejecutar aplicación Detección de la Colisión (IV)*

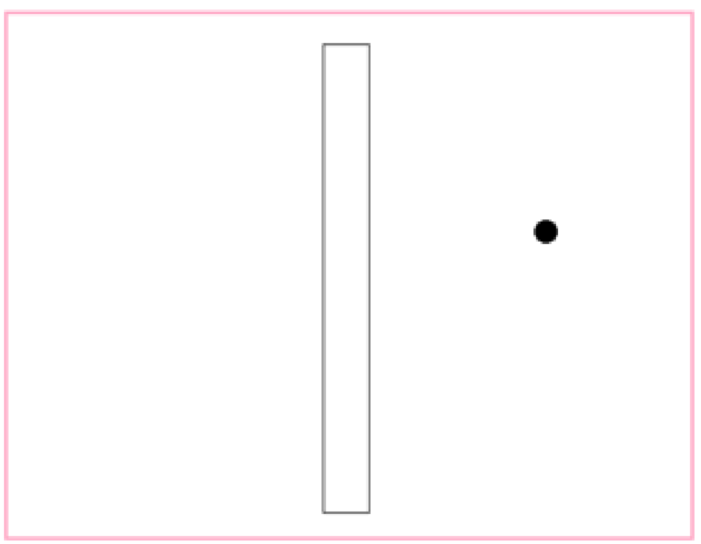


Figura 125: *Resultado ejecutar aplicación Detección de la Colisión (V)*

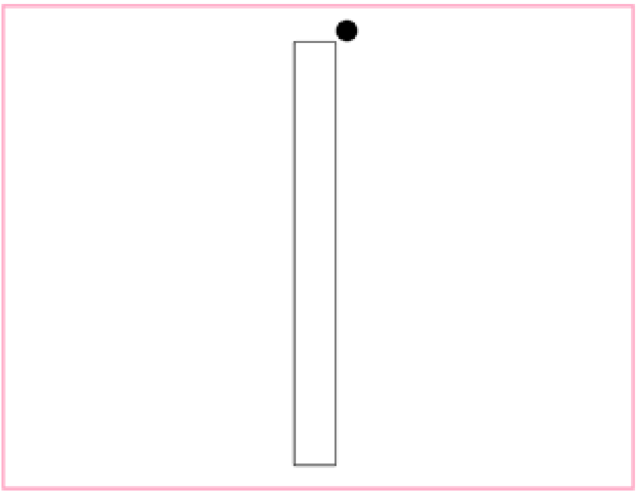


Figura 126: *Resultado ejecutar aplicación Detección de la Colisión (VI)*

3.28 Detección de la colisión por pixel

Una vez determinado cómo comprobar la colisión entre dos objetos rectángulos, puede comprobarse la colisión entre dos imágenes cualesquiera, ya que todas las imágenes están compuestas por rectángulos.

Todo está hecho por rectángulos, incluso puntos como éste:

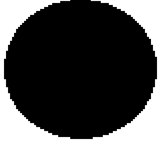


Figura 127: *Punto*

Para comprobarlo, se amplía el zoom:

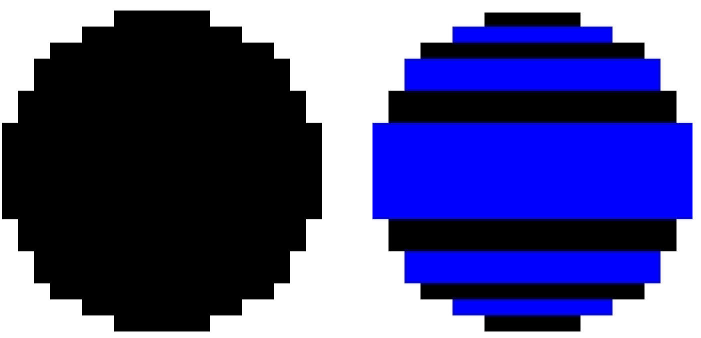


Figura 128: *Ejemplo puntos formados por rectángulos*

Las imágenes están hechas de píxeles, que a su vez son rectángulos. Para la detección de colisión por pixel lo único que hay que hacer es que cada objeto tenga un conjunto de cuadros de colisión y comprobar la colisión de los cuadros de colisión de la siguiente manera:

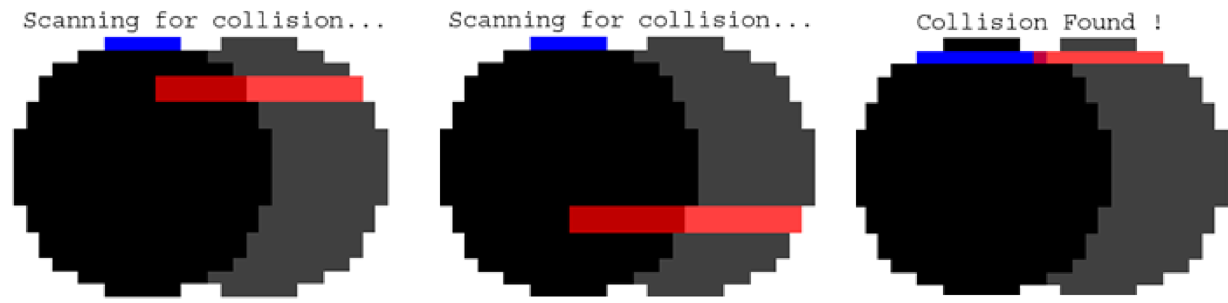


Figura 129: *Comprobación colisión por pixel*

```
//The dot that will move around on the screen
class Dot
{
    public:
        //The dimensions of the dot
        static const int DOT_WIDTH = 20;
        static const int DOT_HEIGHT = 20;

        //Maximum axis velocity of the dot
        static const int DOT_VEL = 1;

        //Initializes the variables
        Dot( int x, int y );

        //Takes key presses and adjusts the dot's velocity
        void handleEvent( SDL_Event& e );

        //Moves the dot and checks collision
        void move( std::vector<SDL_Rect>& otherColliders );

        //Shows the dot on the screen
        void render();

        //Gets the collision boxes
        std::vector<SDL_Rect>& getColliders();

    private:
        //The X and Y offsets of the dot
        int mPosX, mPosY;

        //The velocity of the dot
        int mVelX, mVelY;

        //Dot's collision boxes
        std::vector<SDL_Rect> mColliders;

        //Moves the collision boxes relative to the dot's offset
        void shiftColliders();
};
```

Clase Punto con la detección de colisión por pixel. Su velocidad se reduce a 1 pixel por fotograma para que la colisión sea más fácil de ver. La función desplazamiento toma un vector de objetos de colisión, de manera que pueden comprobarse dos conjuntos (uno contra otro). Dado que se tienen que tener dos puntos chocando es necesario obtener los cuadros de colisión, por lo que se cuenta con una función para ello.

En lugar de tener un único objeto de colisión se tiene un vector de cuadros de colisión. Además, se cuenta con una función interna para desplazar los cuadros de colisión hasta hacerlos coincidir con la posición del punto.

```
//Starts up SDL and creates window
bool init();

//Loads media
bool loadMedia();
```

```
//Frees media and shuts down SDL
void close();

//Box set collision detector
bool checkCollision(std::vector<SDL_Rect>&a,std::vector<SDL_Rect>&b);
```

Detector de las colisiones que comprueba que los cuadros de colisión están unos contra otros.

```
Dot::Dot( int x, int y )
{
    //Initialize the offsets
    mPosX = x;
    mPosY = y;

    //Create the necessary SDL_Rects
    mColliders.resize( 11 );

    //Initialize the velocity
    mVelX = 0;
    mVelY = 0;

    //Initialize the collision boxes' width and height
    mColliders[ 0 ].w = 6;
    mColliders[ 0 ].h = 1;
    mColliders[ 1 ].w = 10;
    mColliders[ 1 ].h = 1;
    mColliders[ 2 ].w = 14;
    mColliders[ 2 ].h = 1;
    mColliders[ 3 ].w = 16;
    mColliders[ 3 ].h = 2;
    mColliders[ 4 ].w = 18;
    mColliders[ 4 ].h = 2;
    mColliders[ 5 ].w = 20;
    mColliders[ 5 ].h = 6;
    mColliders[ 6 ].w = 18;
    mColliders[ 6 ].h = 2;
    mColliders[ 7 ].w = 16;
    mColliders[ 7 ].h = 2;
    mColliders[ 8 ].w = 14;
    mColliders[ 8 ].h = 1;
    mColliders[ 9 ].w = 10;
    mColliders[ 9 ].h = 1;
    mColliders[ 10 ].w = 6;
    mColliders[ 10 ].h = 1;

      //Initialize colliders relative to position
      shiftColliders();
}
```

A continuación se establecen las dimensiones de los cuadros de colisión en el constructor. La única diferencia es que se tienenque configurar varios cuadros de colisión.

```
void Dot::move( std::vector<SDL_Rect>& otherColliders )
{
    //Move the dot left or right
    mPosX += mVelX;
    shiftColliders();

    //If the dot collided or went too far to the left or right
    if((mPosX<0) || (mPosX+DOT_WIDTH>SCREEN_WIDTH) ||
checkCollision(mColliders,otherColliders))
    {
        //Move back
        mPosX -= mVelX;
            shiftColliders();
    }

    //Move the dot up or down
    mPosY += mVelY;
      shiftColliders();

    //If the dot collided or went too far up or down
    if((mPosY<0) || (mPosY+DOT_HEIGHT>SCREEN_HEIGHT) ||
checkCollision(mColliders,otherColliders))
    {
        //Move back
        mPosY -= mVelY;
            shiftColliders();
    }
}
```

Siempre que el punto se mueva, los cuadros de colisión se moverán con él. Tras mover el punto se comprueba si se ha movido fuera de la pantalla o ha golpeado algo. De ser así, se mueve el punto a su posición, así como sus cuadros de colisión.

```
void Dot::shiftColliders()
{
//The row offset
    int r = 0;

//Go through the dot's collision boxes
    for( int set = 0; set < mColliders.size(); ++set )
    {

//Center the collision box
mColliders[ set ].x = mPosX + ( DOT_WIDTH - mColliders[ set ].w ) / 2;

//Set the collision box at its row offset
mColliders[ set ].y = mPosY + r;

//Move the row offset down the height of the collision box
r += mColliders[ set ].h;
    }
}
```

Tras **shiftColliders** se tiene una función de acceso para obtener los cuadros de colisión.

```
bool checkCollision( std::vector<SDL_Rect>& a, std::vector<SDL_Rect>&
b )
{
    //The sides of the rectangles
    int leftA, leftB;
    int rightA, rightB;
    int topA, topB;
    int bottomA, bottomB;

    //Go through the A boxes
    for( int Abox = 0; Abox < a.size(); Abox++ )
    {
        //Calculate the sides of rect A
        leftA = a[ Abox ].x;
        rightA = a[ Abox ].x + a[ Abox ].w;
        topA = a[ Abox ].y;
        bottomA = a[ Abox ].y + a[ Abox ].h;
```

En la función de detección de la colisión se tiene un bucle for que calcula la parte de arriba/abajo/izquierda/derecha de cada cuadro de colisión en el objeto A.

```
        //Go through the B boxes
        for( int Bbox = 0; Bbox < b.size(); Bbox++ )
        {
            //Calculate the sides of rect B
            leftB = b[ Bbox ].x;
            rightB = b[ Bbox ].x + b[ Bbox ].w;
            topB = b[ Bbox ].y;
            bottomB = b[ Bbox ].y + b[ Bbox ].h;

            //If no sides from A are outside of B
if( ( ( bottomA <= topB ) || ( topA >= bottomB ) || ( rightA <= leftB
) || ( leftA >= rightB ) ) == false )
            {
                //A collision is detected
                return true;
            }
        }
    }
    //If neither set of collision boxes touched
    return false;
}
```

A continuación, se comprueba el cálculo de la parte de arriba/abajo/izquierda/derecha de cada cuadro de colisión en el objeto B. Seguidamente se comprueba si no hay separación de ejes. De ser así, se devolverá TRUE. Si no se producen colisiones, se devolverá FALSE.

```
            //Main loop flag
            bool quit = false;

            //Event handler
            SDL_Event e;
```

```
                        //The dot that will be moving around on the screen
                        Dot dot( 0, 0 );

                        //The dot that will be collided against
                        Dot otherDot( SCREEN_WIDTH / 4, SCREEN_HEIGHT / 4 );
```

Antes de entrar al bucle principal se declara un punto y el otro contra el que colisionará.

```
                        //While application is running
                        while( !quit )
                        {
                                //Handle events on queue
                                while( SDL_PollEvent( &e ) != 0 )

                                {
                                        //User requests quit
                                        if( e.type == SDL_QUIT )
                                        {
                                                quit = true;
                                        }

                                        //Handle input for the dot
                                        dot.handleEvent( e );
                                }

                //Move the dot and check collision
                dot.move( otherDot.getColliders() );

                //Clear screen
                SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
                SDL_RenderClear( gRenderer );

                //Render dots
                dot.render();
                otherDot.render();

                //Update screen
                SDL_RenderPresent( gRenderer );
                        }
```

Una vez más, en el bucle principal se controlan los eventos para el punto, éste se mueve y se comprueba la colisión. Finalmente se renderizan los objetos.

(*) El punto B está fijo y el usuario puede mover el punto A.

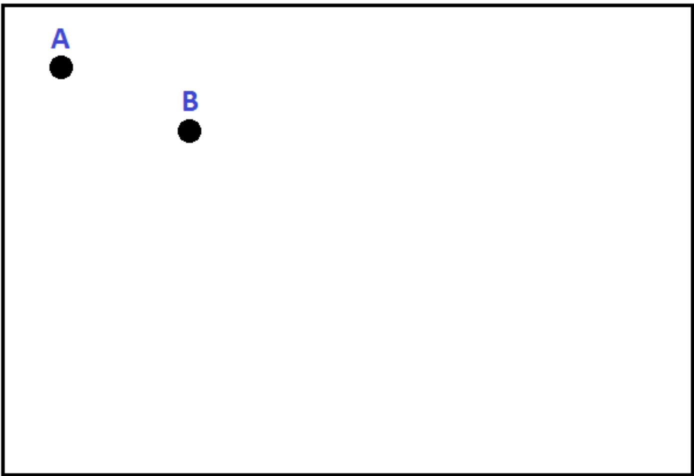


Figura 130: *Resultado ejecutar aplicación Detección de la colisión por pixel*

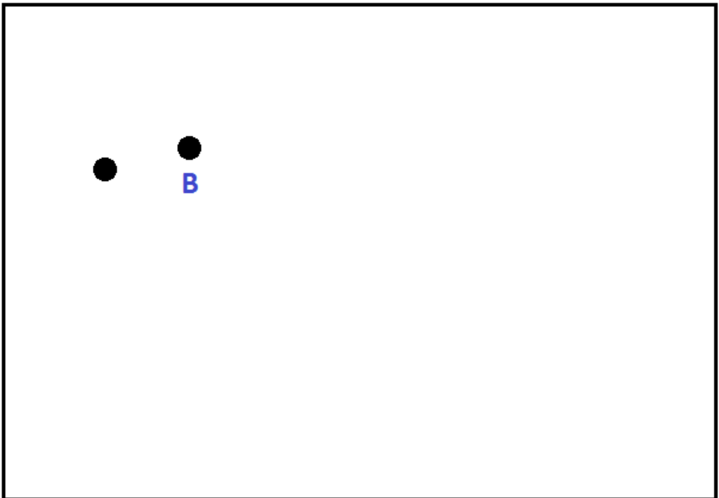


Figura 131: *Resultado ejecutar aplicación Detección de la colisión por pixel (II)*

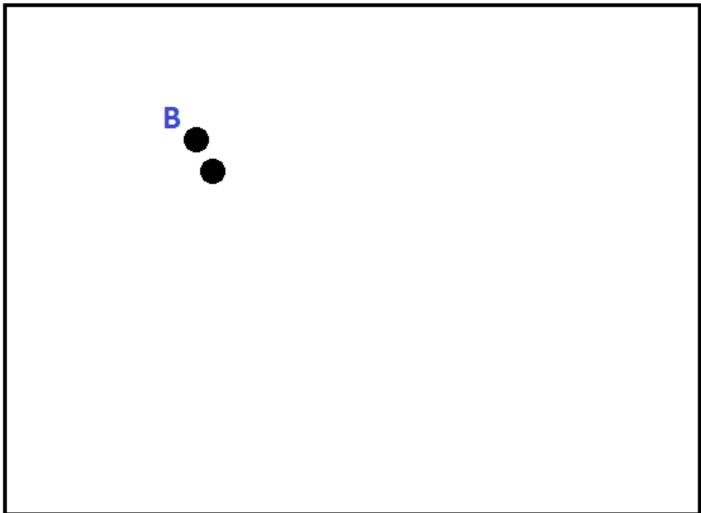


Figura 132: *Resultado ejecutar aplicación Detección de la colisión por pixel (III)*

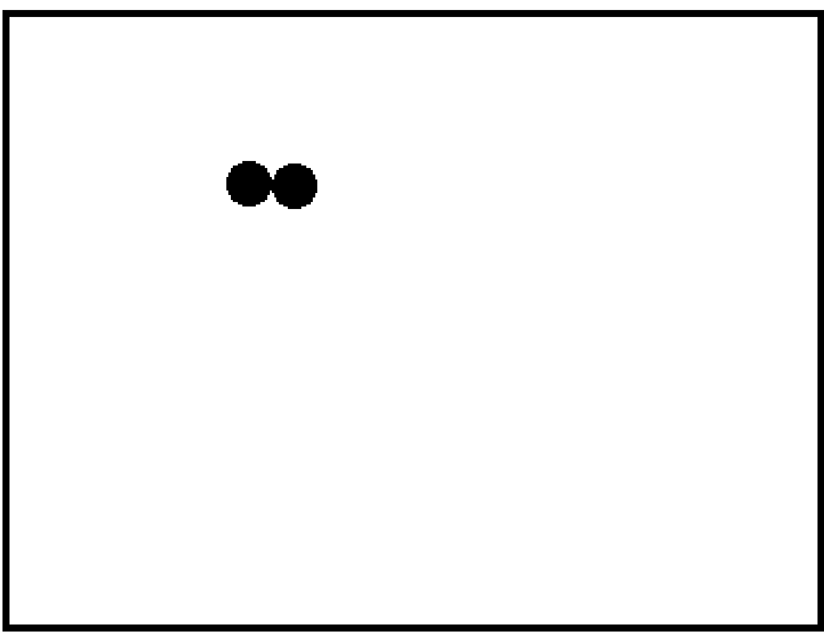


Figura 133: *Resultado ejecutar aplicación Detección de la colisión por pixel (IV)*

3.29 Detección de colisión circular

Junto con los rectángulos, los círculos son la forma más común de cuadro de colisión. El objetivo será comprobar la colisión entre dos círculos y un círculo y un rectángulo.

Comprobar la colisión entre dos círculos es sencillo. Tan sólo hay que verificar si la distancia entre el centro de cada círculo es menor que la suma de sus radios.

Para la colisión circular/rectangular se tiene que encontrar el puntero en el cuadro de colisión más cercano al centro del círculo. Si ese punto es menor que el radio del círculo, hay una colisión.

```
//A circle stucture
struct Circle
{
	int x, y;
	int r;
};
```

SDL ha construido en la estructura del rectángulo, pero hay que hacer una estructura de círculo propia con la posición y el radio.

```
//The dot that will move around on the screen
class Dot
{
    public:

		//The dimensions of the dot
		static const int DOT_WIDTH = 20;
		static const int DOT_HEIGHT = 20;

		//Maximum axis velocity of the dot
		static const int DOT_VEL = 1;

		//Initializes the variables
		Dot( int x, int y );

		//Takes key presses and adjusts the dot's velocity
		void handleEvent( SDL_Event& e );

		//Moves the dot and checks collision
		void move( SDL_Rect& square, Circle& circle );

		//Shows the dot on the screen
		void render();

		//Gets collision circle
		Circle& getCollider();

    private:

		//The X and Y offsets of the dot
		int mPosX, mPosY;
```

```
            //The velocity of the dot
            int mVelX, mVelY;

            //Dot's collision circle
            Circle mCollider;

            //Moves the collision circle relative to the dot's offset
            void shiftColliders();
};
```

La función desplazamiento toma un círculo y un rectángulo para comprobar la colisión de uno contra otro cuando se mueven. Además, se cuenta con un círculo de colisión en lugar de rectángulo de colisión.

```
//Circle/Circle collision detector
bool checkCollision( Circle& a, Circle& b );

//Circle/Box collision detector
bool checkCollision( Circle& a, SDL_Rect& b );

//Calculates distance squared between two points
double distanceSquared( int x1, int y1, int x2, int y2 );
```

En este programa se tienen funciones para detectar la colisión de círculo/círculo y círculo/rectángulo, además de una función que calcula la distancia entre dos puntos al cuadrado.

```
Dot::Dot( int x, int y )
{
    //Initialize the offsets
    mPosX = x;
    mPosY = y;

      //Set collision circle size
      mCollider.r = DOT_WIDTH / 2;

    //Initialize the velocity
    mVelX = 0;
    mVelY = 0;

      //Move collider relative to the circle
      shiftColliders();
}
```

El constructor toma la posición e inicializa los cuadros de colisión y la velocidad.

```
void Dot::move( SDL_Rect& square, Circle& circle )
```

```
{
    //Move the dot left or right
    mPosX += mVelX;
      shiftColliders();

    //If the dot collided or went too far to the left or right
if((mPosX-mCollider.r <0)||(mPosX+mCollider.r >SCREEN_WIDTH)
||checkCollision(mCollider,square)||checkCollision(mCollider,circle))

    {
        //Move back
        mPosX -= mVelX;
            shiftColliders();
    }

    //Move the dot up or down
    mPosY += mVelY;
      shiftColliders();

    //If the dot collided or went too far up or down
if((mPosY-mCollider.r <0)|| (mPosY+mCollider.r>SCREEN_HEIGHT)
||checkCollision(mCollider,square)||checkCollision(mCollider,circle))

    {
        //Move back
        mPosY -= mVelY;
            shiftColliders();
    }
}
```

El movimiento se realiza a lo largo del eje X, se comprueba la colisión con los bordes de la pantalla así como contra los demás objetos. Si el punto choca contra algo, vuelve atrás. Siempre que el punto se mueva, sus cuadros de colisión se mueven con él.

A continuación, se realiza lo mismo para el eje Y.

```
void Dot::render()
{
    //Show the dot
      gDotTexture.render( mPosX - mCollider.r, mPosY - mCollider.r );
}
```

SDL_Rects tiene las posiciones en la parte superior izquierda, donde la estructura de círculo tiene la posición en el centro. Esto significa que hay que contrarrestar la posición de renderizado a la parte superior izquierda del círculo restando el radio desde la posición X a la posición Y.

```
bool checkCollision( Circle& a, Circle& b )
{
      //Calculate total radius squared

      int totalRadiusSquared = a.r + b.r;
      totalRadiusSquared=totalRadiusSquared*totalRadiusSquared;
```

```
//If the distance between the centers of the circles is less than the
sum of their radio

    if(distanceSquared(a.x,a.y,b.x,b.y)<(totalRadiusSquared))
    {
        //The circles have collided
        return true;
    }

    //If not
    return false;
}
```

Detector de la colisión circular. Simplemente comprueba si la distancia al cuadrado entre los centros es menor que la suma de los radios al cuadrado. Si esto se cumple, hay una colisión.

```
bool checkCollision( Circle& a, SDL_Rect& b )
{
    //Closest point on collision box
    int cX, cY;

    //Find closest x offset
    if( a.x < b.x )
    {
        cX = b.x;
    }
    else if( a.x > b.x + b.w )
    {
        cX = b.x + b.w;
    }
    else
    {
        cX = a.x;
    }
```

Para comprobar si un rectángulo y un círculo han chocado entre sí hay que encontrar el punto más cercano al rectángulo.

- Si el centro del círculo está a la izquierda del rectángulo, la posición X del punto más cercano está en el lado izquierdo del rectángulo.

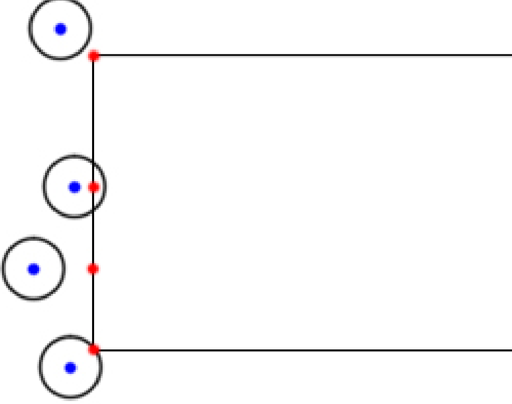


Figura 134: *Ejemplo de colisión entre círculo y rectángulo*

- Si el centro del círculo está a la derecha del rectángulo, la posición X del punto más cercano está en el lado derecho del rectángulo.

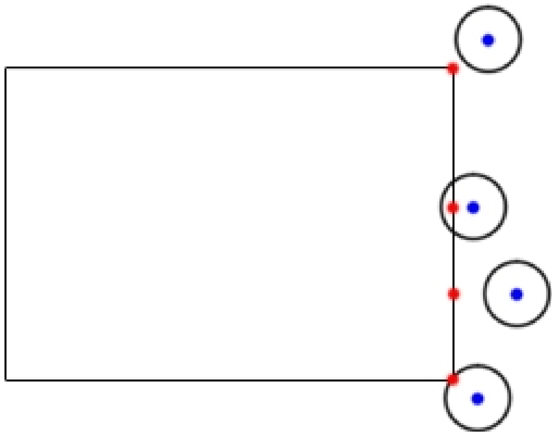


Figura 135: *Ejemplo de colisión entre círculo y rectángulo (II)*

- Si el centro del círculo está dentro del rectángulo, la posición X del punto más cercano es la misma que la posición X del círculo.

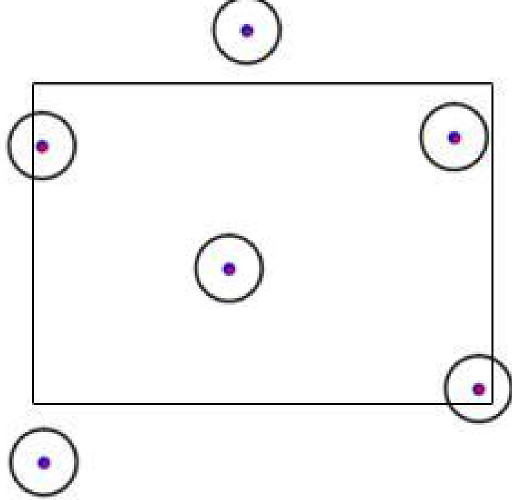


Figura 136: *Ejemplo de colisión entre círculo y rectángulo (III)*

```
//Find closest y offset
if( a.y < b.y )
{
    cY = b.y;
}
else if( a.y > b.y + b.h )
{
    cY = b.y + b.h;
}
```

```
    else
    {
        cY = a.y;
    }

    //If the closest point is inside the circle

    if( distanceSquared( a.x, a.y, cX, cY ) < a.r * a.r )
    {
        //This box and the circle have collided
        return true;
    }

    //If the shapes have not collided
    return false;
}
```

Al igual que con la posición X, se encuentra la posición más cercana. Si la distancia al cuadrado entre el punto más cercano en el rectángulo y el centro del círculo es menor que el radio del círculo al cuadrado, hay colisión.

```
double distanceSquared( int x1, int y1, int x2, int y2 )
{
      int deltaX = x2 - x1;
      int deltaY = y2 - y1;
      return deltaX*deltaX + deltaY*deltaY;
}
```

Función de la distancia al cuadrado.

```
//The dot that will be moving around on the screen
Dot dot( Dot::DOT_WIDTH / 2, Dot::DOT_HEIGHT / 2 );
Dot otherDot( SCREEN_WIDTH / 4, SCREEN_HEIGHT / 4 );

//Set the wall
SDL_Rect wall;
wall.x = 300;
wall.y = 40;
wall.w = 40;
wall.h = 400;
```

Antes de entrar al bucle principal se definen los objetos de la escena.

```
                  //While application is running
                  while( !quit )
                  {
                        //Handle events on queue
                        while( SDL_PollEvent( &e ) != 0 )
                        {
```

```
                    //User requests quit
                    if( e.type == SDL_QUIT )
                    {
                        quit = true;
                    }

                    //Handle input for the dot
                    dot.handleEvent( e );
                }

                //Move the dot and check collision
                dot.move( wall, otherDot.getCollider() );

    //Clear screen
    SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
    SDL_RenderClear( gRenderer );

    //Render wall
    SDL_SetRenderDrawColor( gRenderer, 0x00, 0x00, 0x00, 0xFF );
    SDL_RenderDrawRect( gRenderer, &wall );

                //Render dots
                dot.render();
                otherDot.render();

                //Update screen
                SDL_RenderPresent( gRenderer );
            }
        }
    }
```

Finalmente, en el bucle principal se controla la entrada, se mueve el punto, se comprueba si hay colisión y se renderizan los objetos de la escena a la pantalla.

- **Sin colisión:**

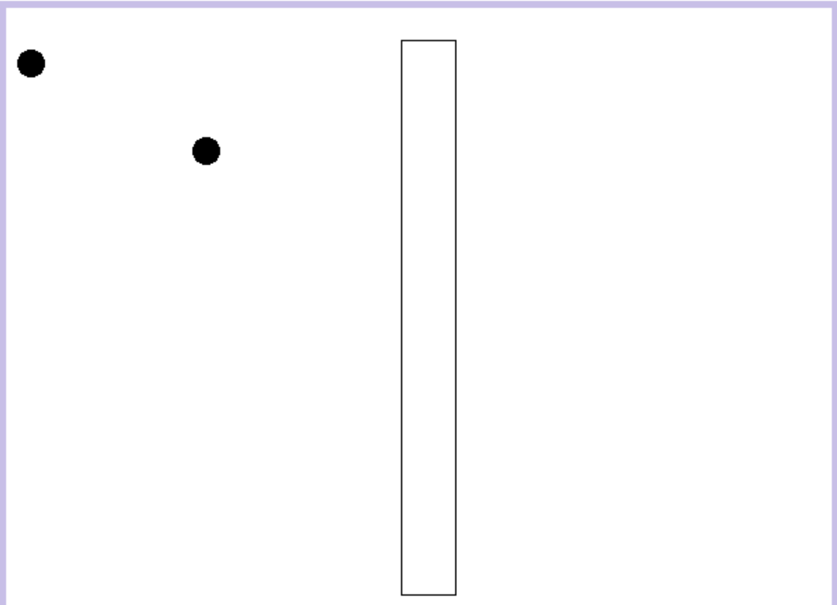


Figura 137: *Resultado ejecutar aplicación Detección de colisión circular*

- **Colisión entre dos círculos:**

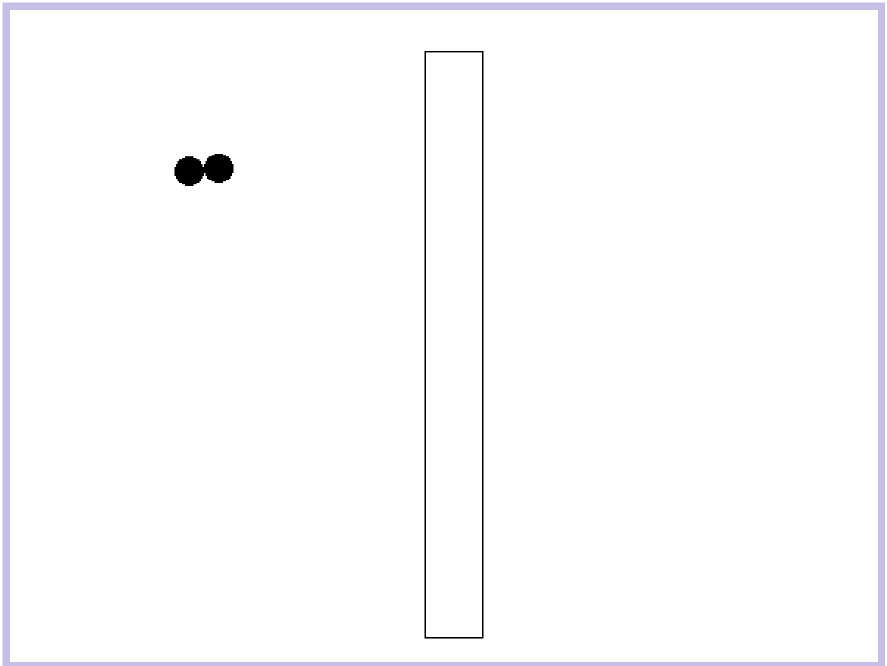


Figura 138: *Resultado ejecutar aplicación Detección de colisión circular (II)*

- **Colisión entre círculo y rectángulo:**

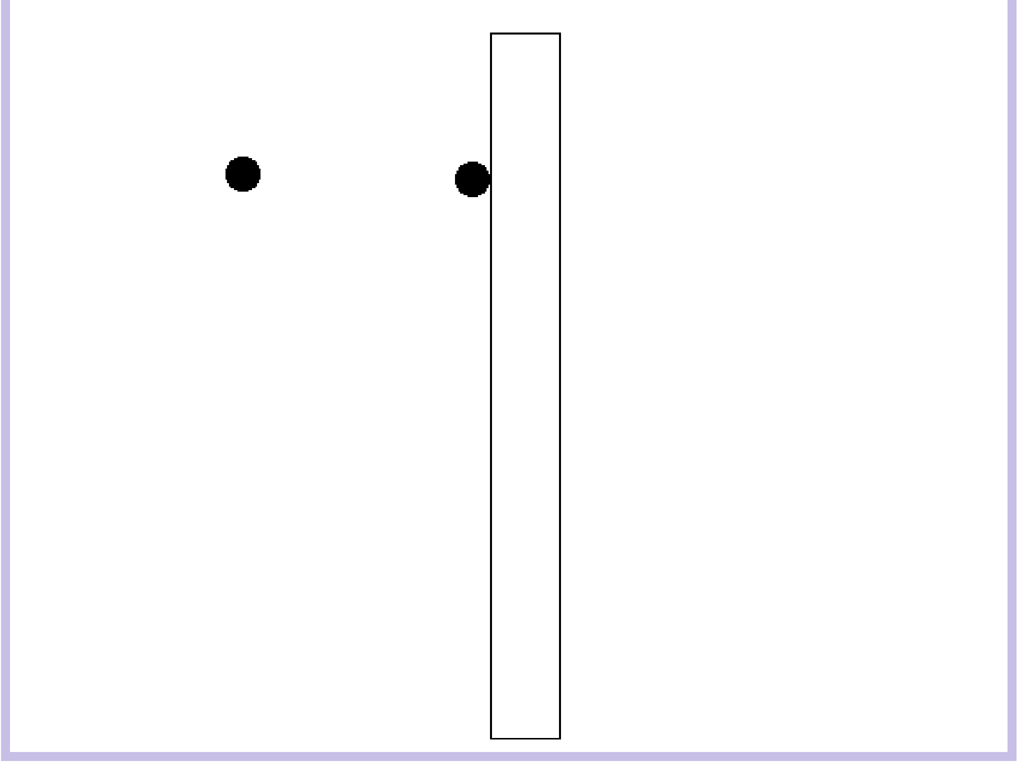


Figura 139: *Resultado ejecutar aplicación Detección de colisión circular (III)*

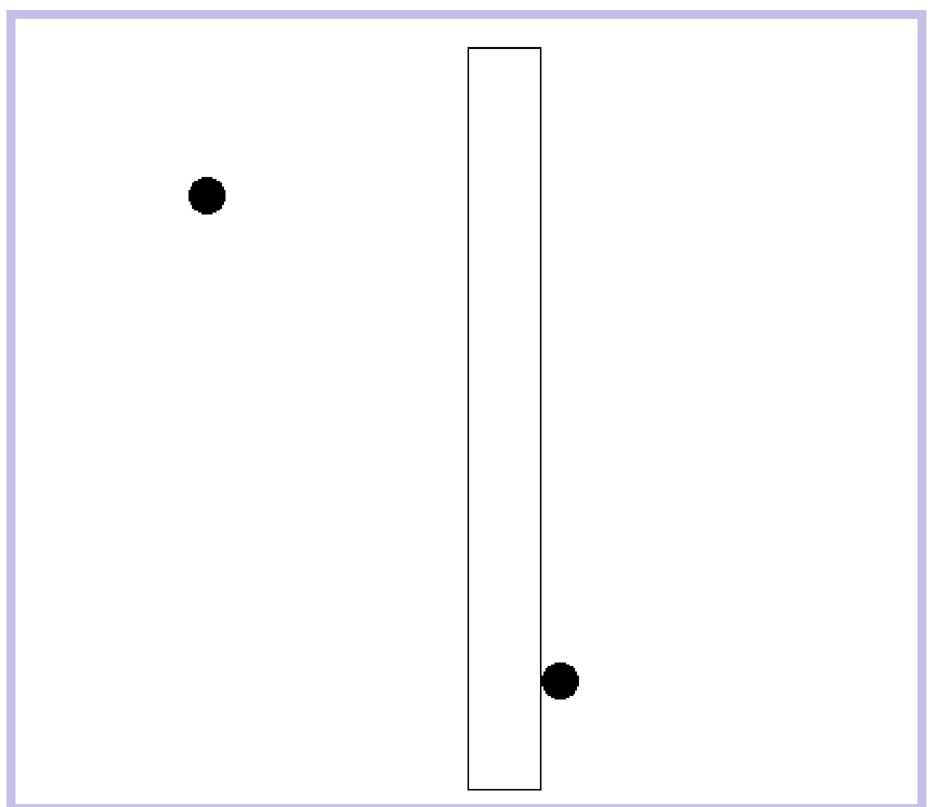


Figura 140: *Resultado ejecutar aplicación Detección de colisión circular (IV)*

3.30 Scrolling

Mediante scrolling puede navegarse a través de niveles de cualquier tamaño al renderizar todo lo relativo a una cámara.

El principio básico del scrolling es que se tiene un rectángulo que funciona como una cámara:

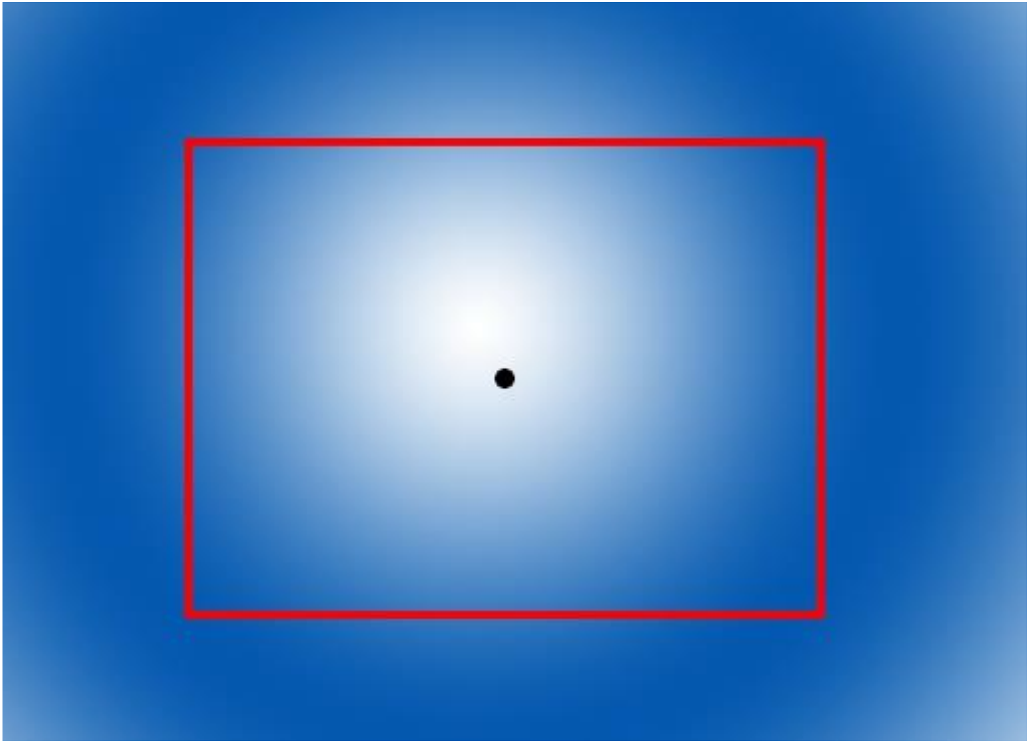


Figura 141: *Scrolling*

Y sólo se renderiza lo que hay dentro de la cámara, que por lo general implica el renderizado de lo relativo a la cámara o sólo muestra partes de los objetos dentro de la cámara.

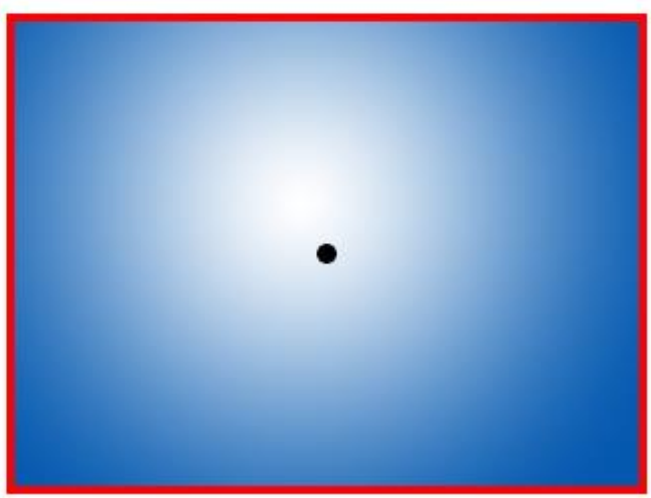


Figura 142: *Scrolling (II)*

```
//The dimensions of the level
const int LEVEL_WIDTH = 1280;
const int LEVEL_HEIGHT = 960;

//Screen dimension constants
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
```

Dado que el nivel no es del mismo tamaño que la pantalla, hay que tener un conjunto separado de constantes para definir el tamaño del nivel.

```
//The dot that will move around on the screen
class Dot
{
    public:
        //The dimensions of the dot
        static const int DOT_WIDTH = 20;
        static const int DOT_HEIGHT = 20;

        //Maximum axis velocity of the dot
        static const int DOT_VEL = 10;

        //Initializes the variables
        Dot();

        //Takes key presses and adjusts the dot's velocity
        void handleEvent( SDL_Event& e );

        //Moves the dot
        void move();

        //Shows the dot on the screen relative to the camera
        void render( int camX, int camY );

        //Position accessors
        int getPosX();
        int getPosY();

    private:
        //The X and Y offsets of the dot
        int mPosX, mPosY;

        //The velocity of the dot
        int mVelX, mVelY;
};
```

El punto se tiene que renderizar con respecto a la cámara, por lo que su función de renderizado toma la posición de la cámara.

```
void Dot::move()
{
    //Move the dot left or right
    mPosX += mVelX;
```

```
    //If the dot went too far to the left or right
    if( ( mPosX < 0 ) || ( mPosX + DOT_WIDTH > LEVEL_WIDTH ) )
    {
        //Move back
        mPosX -= mVelX;
    }

    //Move the dot up or down
    mPosY += mVelY;

    //If the dot went too far up or down
    if( ( mPosY < 0 ) || ( mPosY + DOT_HEIGHT > LEVEL_HEIGHT ) )
    {
        //Move back
        mPosY -= mVelY;
    }
}
```

Cuando el punto se mueve, se comprueba si se ha alejado del nivel en lugar de comprobar si se ha alejado de la pantalla, ya que la pantalla se va a mover alrededor del nivel.

```
void Dot::render( int camX, int camY )
{
    //Show the dot relative to the camera
      gDotTexture.render( mPosX - camX, mPosY - camY );
}
```

Cuando se rendericen objetos a la pantalla, se hará en relación con la cámara restando el desplazamiento de ésta.

```
                //Main loop flag
                bool quit = false;
                //Event handler
                SDL_Event e;
                //The dot that will be moving around on the screen
                Dot dot;
                //The camera area
            SDL_Rect camera = { 0, 0, SCREEN_WIDTH, SCREEN_HEIGHT };
```

Antes de entrar al bucle principal se declara el punto así como la cámara que va a seguirlo.

```
//Move the dot
dot.move();

//Center the camera over the dot
camera.x = ( dot.getPosX() + Dot::DOT_WIDTH / 2 ) - SCREEN_WIDTH / 2;
camera.y = ( dot.getPosY() + Dot::DOT_HEIGHT / 2 ) - SCREEN_HEIGHT /
2;

//Keep the camera in bounds
if( camera.x < 0 )
```

```
{
        camera.x = 0;
}

if( camera.y < 0 )
{
        camera.y = 0;
}

if( camera.x > LEVEL_WIDTH - camera.w )
{
        camera.x = LEVEL_WIDTH - camera.w;
}
if( camera.y > LEVEL_HEIGHT - camera.h )
{
        camera.y = LEVEL_HEIGHT - camera.h;
}
```

Tras mover el punto se cambia la posición de la cámara para centrarla sobre él. (Es importante que la cámara no se mueva fuera del nivel, por lo que hay que mantener los límites después de moverlo)

```
        //Clear screen
        SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
        SDL_RenderClear( gRenderer );

        //Render background
        gBGTexture.render( 0, 0, &camera );

        //Render objects
        dot.render( camera.x, camera.y );

        //Update screen
        SDL_RenderPresent( gRenderer );
```

Una vez la cámara está en su lugar, se renderiza la porción del fondo dentro de la cámara y se renderiza el punto relativo a la posición de la cámara.

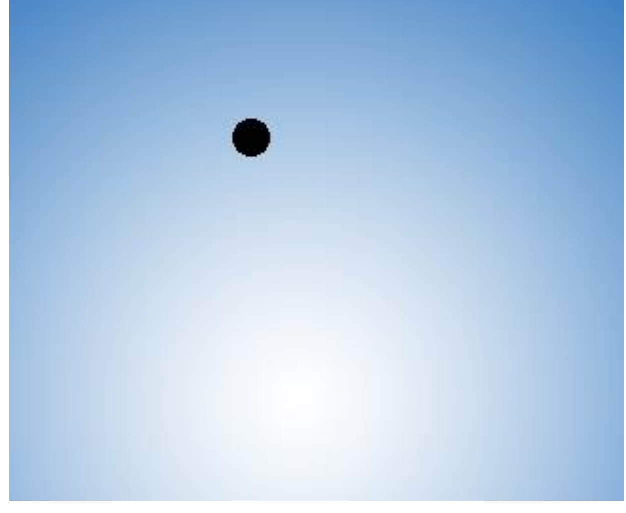


Figura 143: *Scrolling (III)*

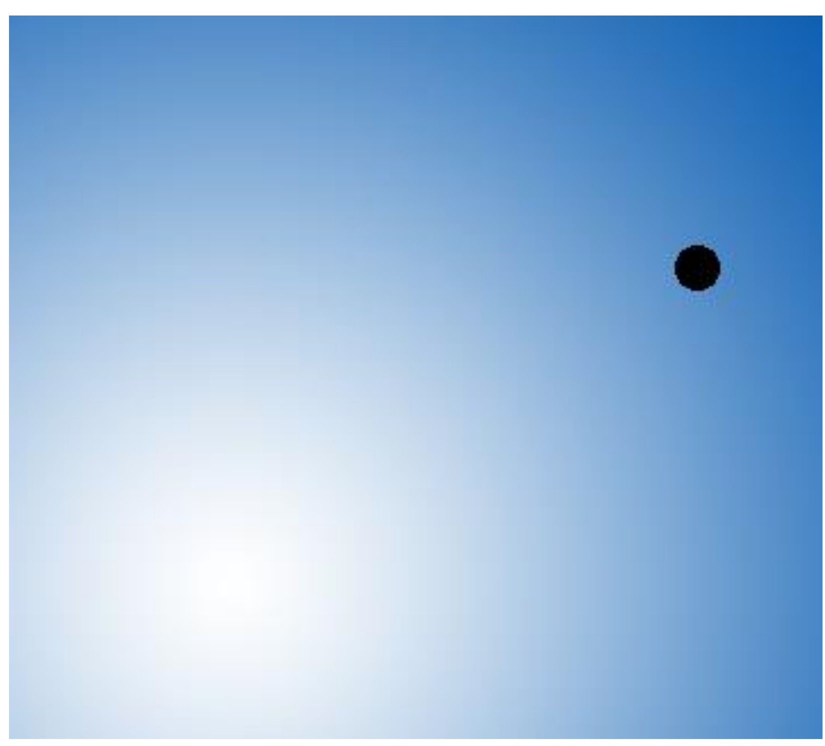


Figura 144: *Scrolling (IV)*

Figura 145: *Scrolling (V)*

3.31 Scrolling Backgrounds (Fondos de desplazamiento)

Mediante fondos de desplazamiento (fondo infinito o en bucle) puede alternarse un fondo para que se vea siempre.

Si quiere moverse un punto en torno a un fondo infinito:

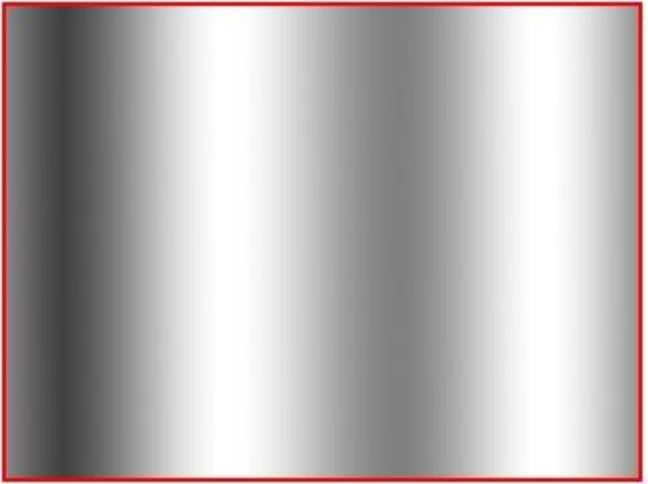


Figura 146: *Scrolling Backgrounds*

Todo lo que hay que hacer es renderizar dos iteraciones del fondo una junto a la otra y moverlas un poco cada fotograma. Cuando el fondo se ha movido completamente se reinicia el movimiento.

```
//The dot that will move around on the screen
class Dot

{
    public:
            //The dimensions of the dot
            static const int DOT_WIDTH = 20;
            static const int DOT_HEIGHT = 20;

            //Maximum axis velocity of the dot
            static const int DOT_VEL = 10;

            //Initializes the variables
            Dot();

            //Takes key presses and adjusts the dot's velocity
            void handleEvent( SDL_Event& e );

            //Moves the dot
            void move();

            //Shows the dot on the screen
            void render();
```

```
    private:
        //The X and Y offsets of the dot
        int mPosX, mPosY;

        //The velocity of the dot
        int mVelX, mVelY;
};
```

Se usará una versión simple del punto que sólo permanecerá en la pantalla.

```
            //The dot that will be moving around on the screen
            Dot dot;

            //The background scrolling offset
            int scrollingOffset = 0;
```

Antes de entrar al bucle principal se declara un objeto Punto y la compensación del desplazamiento.

```
                //Move the dot
                dot.move();

                //Scroll background
                --scrollingOffset;
                if( scrollingOffset < -gBGTexture.getWidth() )
                {
                    scrollingOffset = 0;
                }
```

Se actualiza el punto así como el desplazamiento del fondo.

Para actualizar la posición del desplazamiento del fondo tan sólo hay que disminuir la posición X y si ésta es menor que la anchura del fondo significará que el fondo se ha movido completamente a lo largo de la pantalla y habrá que reiniciar su posición.

```
        //Clear screen
        SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
                        SDL_RenderClear( gRenderer );

        //Render background
        gBGTexture.render( scrollingOffset, 0 );
        gBGTexture.render( scrollingOffset + gBGTexture.getWidth(), 0 );

        //Render objects
        dot.render();

        //Update screen
        SDL_RenderPresent( gRenderer );
```

Se renderiza el punto y el fondo. En primer lugar, se renderiza el desplazamiento del fondo mediante el renderizado de dos iteraciones de texturas una junto a la otra y seguidamente se renderiza el punto sobre ella. Así se consigue el efecto de un fondo de desplazamiento infinito.

Figura 147: *Scrolling Backgrounds (II)*

Figura 148: *Scrolling Backgrounds (III)*

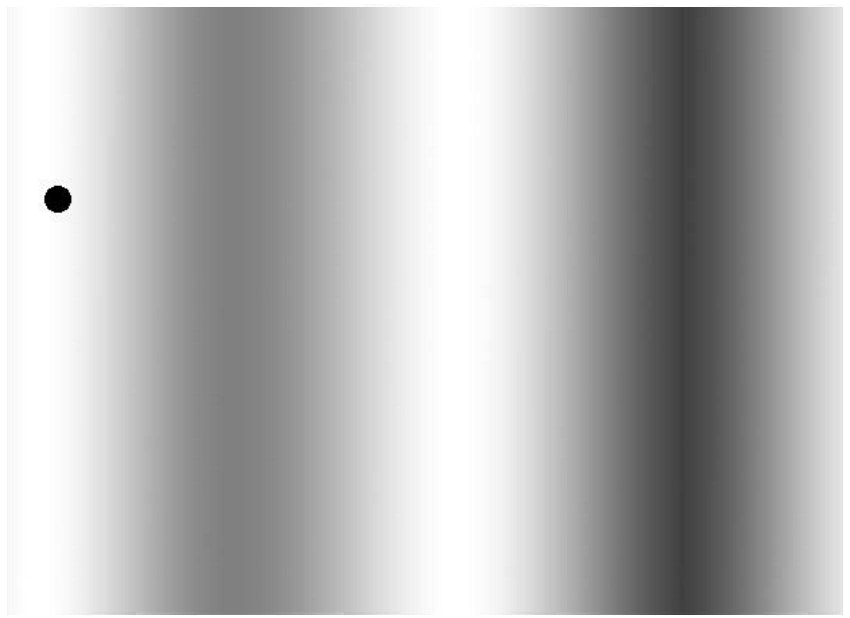


Figura 149: *Scrolling Backgrounds (IV)*

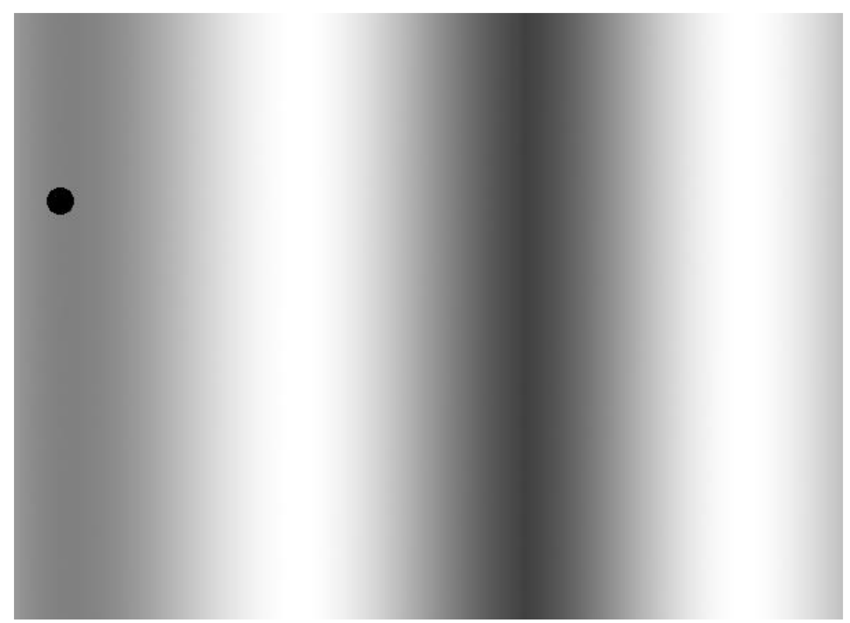


Figura 150: *Scrolling Backgrounds (V)*

Figura 151: *Scrolling Backgrounds (VI)*

Figura 152: *Scrolling Backgrounds (VII)*

3.32 Archivos de lectura y escritura

El controlador de archivo **SDL_RWops** permite hacer archivos multiplataforma IO para guardar los datos.

```
//Data points
Sint32 gData[ TOTAL_DATA ];
```

Se declara un array de enteros con signo de 32 bits de longitud. Éste representa los datos que se cargarán y guardarán. En este caso el array será de longitud 10.

```
//Open file for reading in binary
SDL_RWops* file = SDL_RWFromFile(
"33_file_reading_and_writing/nums.bin", "r+b" );
```

En la función de carga se abre el archivo a guardar usando **SDL_RWFromFile**.

- El primer argumento es la ruta del archivo.
- El segundo argumento define cómo se abrirá.
- “r+b” significa que se está abriendo para lectura en binario.

```
//File does not exist

if( file == NULL )
      {

printf( "Warning: Unable to open file! SDL Error: %s\n",
SDL_GetError() );

//Create file for writing
file = SDL_RWFromFile( "33_file_reading_and_writing/nums.bin", "w+b");
```

Ha de tenerse en cuenta que sólo porque el archivo no exista no significa exactamente que haya un error, sino que puede ser la primera vez que el programa se ejecuta y el archivo no se haya creado aún. Si el archivo no existe, aparece una advertencia y se crea un archivo al abrir un archivo con “w+b”, que abre un nuevo archivo para escribir en binario.

```
if( file != NULL )
            {
                  printf( "New file created!\n" );

                  //Initialize data
                  for( int i = 0; i < TOTAL_DATA; ++i )
                  {
                  gData[ i ] = 0;
                  SDL_RWwrite( file, &gData[ i ], sizeof(Sint32), 1 );
                  }
```

```
                        //Close file handler
                        SDL_RWclose( file );
                }

                else
                {
printf( "Error: Unable to create file! SDL Error: %s\n",
SDL_GetError() );
                        success = false;
                }
        }
```

Si se crea correctamente un nuevo archivo, se comienzan a escribir los datos inicializados empleando **SDL_RWwrite**.

- El primer argumento es el archivo en el que se está escribiendo.
- El segundo argumento es la dirección de los objetos en memoria que se están escribiendo.
- El tercer argumento corresponde con el número de bytes por objeto que se está escribiendo.
- El último se refiere al número de objetos que se escribe.

Una vez se han terminado de escribir todos los objetos, se cierra el archivo de escritura mediante **SDL_RWclose.**

Si el archivo nunca llegó a crearse:

- En primer lugar, se informa del error
- A continuación, el flag de éxito toma el valor FALSE.

```
//File exists
        else
        {
                //Load data
                printf( "Reading file...!\n" );
                for( int i = 0; i < TOTAL_DATA; ++i )
                {
                        SDL_RWread( file, &gData[ i ], sizeof(Sint32), 1 );
                }

                //Close file handler
                SDL_RWclose( file );
        }
```

Si el archivo se ha cargado con éxito en el primer intento, lo único que hay que hacer es leer los datos usando **SDL_RWread** (funciona como **SDL_RWwrite**, pero a la inversa).

```
//Initialize data textures
gDataTextures[ 0 ].loadFromRenderedText( std::to_string(
(_Longlong)gData[ 0 ] ), highlightColor );
```

```
for( int i = 1; i < TOTAL_DATA; ++i )
	{
gDataTextures[ i ].loadFromRenderedText( std::to_string(
(_Longlong)gData[ i ] ), textColor );
	}

	return success;
}
```

Una vez cargado el archivo, se renderizan las texturas de texto que se corresponden con cada uno de los datos. **LoadFromRenderedText** únicamente acepta cadenas, por lo que hay que convertir los números enteros a cadenas.

```
void close()
{
	//Open data for writing
SDL_RWops* file = SDL_RWFromFile(
"33_file_reading_and_writing/nums.bin", "w+b" );

	if( file != NULL )
	{
		//Save data
		for( int i = 0; i < TOTAL_DATA; ++i )
		{
			SDL_RWwrite( file, &gData[ i ], sizeof(Sint32), 1 );
		}

		//Close file handler
		SDL_RWclose( file );
	}

	else
	{
	printf( "Error: Unable to save file! %s\n", SDL_GetError() );
	}
```

Cuando se cierra el programa, vuelve a abrirse el archivo para escribir todos los datos.

```
			//Main loop flag
			bool quit = false;

			//Event handler
			SDL_Event e;

			//Text rendering color
			SDL_Color textColor = { 0, 0, 0, 0xFF };
			SDL_Color highlightColor = { 0xFF, 0, 0, 0xFF };

			//Current input point
			int currentData = 0;
```

Antes de entrar al bucle principal se declara **CurrentData** para hacer un seguimiento de qué números enteros se están alterando. Además, se declara un color de texto sin formato y un color que resalte para renderizar el texto.

```
else if( e.type == SDL_KEYDOWN )
{

switch( e.key.keysym.sym )
{

//Previous data entry
case SDLK_UP:

//Rerender previous entry input point

gDataTextures[ currentData ].loadFromRenderedText( std::to_string(
(_Longlong)gData[ currentData ] ), textColor );
--currentData;

if( currentData < 0 )

{
currentData = TOTAL_DATA - 1;
}

//Rerender current entry input point

gDataTextures[ currentData ].loadFromRenderedText( std::to_string(
(_Longlong)gData[ currentData ] ), highlightColor );

break;


//Next data entry
case SDLK_DOWN:

//Rerender previous entry input point

gDataTextures[ currentData ].loadFromRenderedText( std::to_string(
(_Longlong)gData[ currentData ] ), textColor );
++currentData;

if( currentData == TOTAL_DATA )

{
currentData = 0;
}

//Rerender current entry input point

gDataTextures[ currentData ].loadFromRenderedText( std::to_string(
(_Longlong)gData[ currentData ] ), highlightColor );

break;
```

Cuando se pulsa arriba o abajo, se renderiza el anterior dato actual en color plano, se pasa al siguiente punto de datos (con comprobación de los límites) y se vuelve a renderizar el nuevo dato actual con un color que resalte.

```
//Decrement input point

case SDLK_LEFT:

--gData[ currentData ];
gDataTextures[ currentData ].loadFromRenderedText( std::to_string(
(_Longlong)gData[ currentData ] ), highlightColor );

break;


//Increment input point

case SDLK_RIGHT:

++gData[ currentData ];
gDataTextures[ currentData ].loadFromRenderedText( std::to_string(
(_Longlong)gData[ currentData ] ), highlightColor );

break;

}

}

}
```

Cuando se pulsa la tecla izquierda/derecha se disminuye/incrementa el dato actual y se renderiza la textura asociada a él.

```
//Clear screen

SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render text textures
gPromptTextTexture.render( ( SCREEN_WIDTH -
gPromptTextTexture.getWidth() ) / 2, 0 );

for( int i = 0; i < TOTAL_DATA; ++i )
                        {
gDataTextures[ i ].render( ( SCREEN_WIDTH - gDataTextures[ i
].getWidth() ) / 2, gPromptTextTexture.getHeight() + gDataTextures[ 0
].getHeight() * i );
                        }

//Update screen
SDL_RenderPresent( gRenderer );

            }
```

Al final del bucle principal se renderizan todas las texturas.

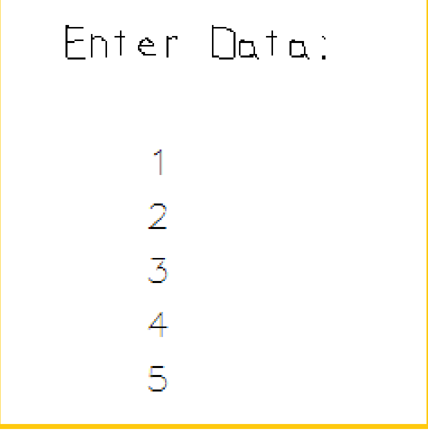


Figura 153: *Resultado ejecutar aplicación Archivos de lectura y escritura*

3.33 Window Events

El objetivo es controlar eventos adicionales cuando se tienen ventanas de tamaño variable.

```
class LWindow
{
	public:

		//Intializes internals
		LWindow();

		//Creates window
		bool init();

		//Creates renderer from internal window
		SDL_Renderer* createRenderer();

		//Handles window events
		void handleEvent( SDL_Event& e );

		//Deallocates internals
		void free();

		//Window dimensions
		int getWidth();
		int getHeight();

		//Window focii
		bool hasMouseFocus();
		bool hasKeyboardFocus();
		bool isMinimized();

	private:

		//Window data
		SDL_Window* mWindow;

		//Window dimensions
		int mWidth;
		int mHeight;

		//Window focus
		bool mMouseFocus;
		bool mKeyboardFocus;
		bool mFullScreen;
		bool mMinimized;
};
```

Clase Ventana que se usará para envolver el **SDL_Window**.

Está formada por el constructor, inicializador que crea la ventana, función para renderizar la ventana, controlador de eventos, liberador, así como funciones de acceso para obtener diversos atributos de la ventana.

En cuanto a los datos, se tiene la ventana que se está envolviendo, sus dimensiones y flags.

```
//Our custom window
LWindow gWindow;

//The window renderer
SDL_Renderer* gRenderer = NULL;

//Scene textures
LTexture gSceneTexture;
```

La ventana se usará como un objeto global.

```
LWindow::LWindow()
{
	//Initialize non-existant window
	mWindow = NULL;
	mMouseFocus = false;
	mKeyboardFocus = false;
	mFullScreen = false;
	mMinimized = false;
	mWidth = 0;
	mHeight = 0;
}
```

En el constructor se inicializan todas las variables.

```
bool LWindow::init()
{
	//Create window

mWindow = SDL_CreateWindow( "SDL Tutorial", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN
| SDL_WINDOW_RESIZABLE );

	if( mWindow != NULL )
	{
		mMouseFocus = true;
		mKeyboardFocus = true;
		mWidth = SCREEN_WIDTH;
		mHeight = SCREEN_HEIGHT;
	}

	return mWindow != NULL;
}
```

La función de inicialización crea la ventana con el flag **SDL_WINDOW_RESIZABLE**, que permite a la ventana ser de tamaño variable. Si la función tiene éxito, se establecen los flags correspondientes y sus dimensiones. A continuación, se devuelve si la ventana es nula o no.

```
SDL_Renderer* LWindow::createRenderer()
{
return SDL_CreateRenderer( mWindow, -1, SDL_RENDERER_ACCELERATED |
SDL_RENDERER_PRESENTVSYNC );
}
```

Control de la creación de un renderizador para la ventana miembro. Se devuelve el renderizador creado, ya que el renderizado se controlará fuera de la clase.

```
void LWindow::handleEvent( SDL_Event& e )
{
	//Window event occured
	if( e.type == SDL_WINDOWEVENT )
	{
		//Caption update flag
		bool updateCaption = false;
```

En el controlador de eventos de la ventana se buscarán eventos del tipo **SDL_WINDOWEVENT**. Dependiendo del tipo de evento, puede ser necesario actualizar el título de la ventana, por lo que se emplea un flag que lleve esta cuenta.

```
switch( e.window.event )
{

//Get new dimensions and repaint on window size change

case SDL_WINDOWEVENT_SIZE_CHANGED:
mWidth = e.window.data1;
mHeight = e.window.data2;
SDL_RenderPresent( gRenderer );

break;

//Repaint on exposure

case SDL_WINDOWEVENT_EXPOSED:
SDL_RenderPresent( gRenderer );

break;
```

Cuando se tiene un evento de ventana se comprueba el **SDL_WindowEventID** para determinar qué tipo de evento es. Un **SDL_WINDOWEVENT_SIZE_CHANGED** es un evento de cambio de tamaño, por lo que se obtienen las nuevas dimensiones y se actualiza la imagen en la pantalla.

```
				//Mouse entered window
				case SDL_WINDOWEVENT_ENTER:
				mMouseFocus = true;
				updateCaption = true;
				break;
```

```
                    //Mouse left window
                    case SDL_WINDOWEVENT_LEAVE:
                    mMouseFocus = false;
                    updateCaption = true;
                    break;

                    //Window has keyboard focus
                    case SDL_WINDOWEVENT_FOCUS_GAINED:
                    mKeyboardFocus = true;
                    updateCaption = true;
                    break;

                    //Window lost keyboard focus
                    case SDL_WINDOWEVENT_FOCUS_LOST:
                    mKeyboardFocus = false;
                    updateCaption = true;
                    break;
```

SDL_WINDOWEVENT_ENTER/SDL_WINDOWEVENT_LEAVE controla el movimiento del ratón dentro y fuera de la ventana.

SDL_WINDOWEVENT_FOCUS_GAINED/SDL_WINDOWEVENT_FOCUS_LOST actúa cuando la ventana obtiene entradas de teclado.

```
              //Window minimized
              case SDL_WINDOWEVENT_MINIMIZED:
              mMinimized = true;
              break;

              //Window maxized
              case SDL_WINDOWEVENT_MAXIMIZED:
              mMinimized = false;
              break;

              //Window restored
              case SDL_WINDOWEVENT_RESTORED:
              mMinimized = false;
              break;

              }
```

Finalmente se controla cuándo se minimiza, maximiza o se restaura el minimizado de la ventana.

```
              //Update window caption with new data
              if( updateCaption )
              {

std::stringstream caption;
caption<<"SDL Tutorial-MouseFocus:"<<((mMouseFocus) ? "On" : "Off" )<<
" KeyboardFocus:" << ( ( mKeyboardFocus ) ? "On" : "Off" );
```

```
SDL_SetWindowTitle( mWindow, caption.str().c_str() );
            }
        }
```

Si hay que actualizar el título, se carga un stream de cadena que actualiza los datos y el título mediante **SDL_SetWindowTitle**.

```
//Enter exit full screen on return key

      else if(e.type==SDL_KEYDOWN&&e.key.keysym.sym==SDLK_RETURN)
      {
            if( mFullScreen )
            {
                  SDL_SetWindowFullscreen(mWindow,SDL_FALSE);
                  mFullScreen = false;
            }

            else
            {
                  SDL_SetWindowFullscreen(mWindow,SDL_TRUE);
                  mFullScreen = true;
                  mMinimized = false;
            }
      }
}
```

- Para pasar a modo pantalla completa se empleará la tecla enter.
- Para configurar dicho modo se emplea **SDL_SetWindowFullscreen.**

```
            //Create window

            if( !gWindow.init() )
            {
printf("Window could not be created! SDL Error: %s\n",SDL_GetError());
                  success = false;
            }

            else
            {
            //Create renderer for window

gRenderer = gWindow.createRenderer();
                  if( gRenderer == NULL )
                  {
printf("Renderer could not be created! SDL Error:
%s\n",SDL_GetError());
                        success = false;
                  }
```

En la función de inicialización se crea la ventana y se renderiza con la envoltura de ventana.

```
void close()
{
	//Free loaded images
	gSceneTexture.free();

	//Destroy window
	SDL_DestroyRenderer( gRenderer );
	gWindow.free();

	//Quit SDL subsystems
	IMG_Quit();
	SDL_Quit();
}
```

En la función de limpieza se libera la ventana y se renderiza.

```
			//While application is running
			while( !quit )
			{
				//Handle events on queue
				while( SDL_PollEvent( &e ) != 0 )
				{
					//User requests quit
					if( e.type == SDL_QUIT )
					{
						quit = true;
					}

					//Handle window events
					gWindow.handleEvent( e );
				}

				//Only draw when not minimized
				if( !gWindow.isMinimized() )
				{
					//Clear screen
	SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
	SDL_RenderClear( gRenderer );

					//Render text textures

gSceneTexture.render((gWindow.getWidth()-gSceneTexture.getWidth())/2,
( gWindow.getHeight() - gSceneTexture.getHeight() ) / 2 );

					//Update screen
					SDL_RenderPresent( gRenderer );
				}
			}
		}
	}
```

En el bucle principal se asegura que los eventos se pasan a la envoltura de ventana para controlar los eventos de cambio de tamaño y en la parte de renderizado sólo se renderiza cuando la ventana no esté minimizada, ya que puede causar sobrecarga al tratar de renderizar una ventana minimizada.

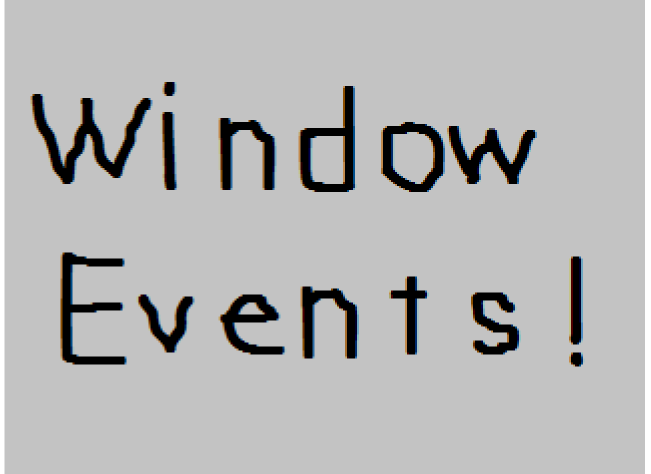


Figura 154: *Resultado ejecutar aplicación Window Events*

Figura 155: *Resultado ejecutar aplicación Window Events (II)*

3.34 Ventanas mú4ltiples

SDL 2.0 es capaz de controlar múltiples ventanas a la vez. El objetivo será ser capaz de moverse alrededor de 3 ventanas de tamaño variable.

```
//Total windows
const int TOTAL_WINDOWS = 3;

class LWindow
{
	public:

		//Intializes internals
		LWindow();

		//Creates window
		bool init();

		//Handles window events
		void handleEvent( SDL_Event& e );

		//Focuses on window
		void focus();

		//Shows windows contents
		void render();

		//Deallocates internals
		void free();

		//Window dimensions
		int getWidth();
		int getHeight();

		//Window focii
		bool hasMouseFocus();
		bool hasKeyboardFocus();
		bool isMinimized();
		bool isShown();

	private:
		//Window data
		SDL_Window* mWindow;
		SDL_Renderer* mRenderer;
		int mWindowID;

		//Window dimensions
		int mWidth;
		int mHeight;

		//Window focus
		bool mMouseFocus;
		bool mKeyboardFocus;
		bool mFullScreen;
		bool mMinimized;
		bool mShown;
};
```

Se pretende ser capaz de tomar el enfoque y saber si la ventana se muestra para añadir funciones que lo hagan. Cada ventana tendrá su propio renderizador; para ello se añade una variable miembro.

Además, se hace un seguimiento de la ventana ID para decir qué eventos pertenecen a cada ventana así como un flag que lleve un registro de si se muestra la ventana.

```
//Our custom windows
LWindow gWindows[ TOTAL_WINDOWS ];
```

En este caso habrá 3 ventanas liberadas.

```
bool LWindow::init()
{

//Create window

mWindow = SDL_CreateWindow( "SDL Tutorial", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN
| SDL_WINDOW_RESIZABLE );

      if( mWindow != NULL )
      {
            mMouseFocus = true;
            mKeyboardFocus = true;
            mWidth = SCREEN_WIDTH;
            mHeight = SCREEN_HEIGHT;

            //Create renderer for window
mRenderer = SDL_CreateRenderer( mWindow, -1, SDL_RENDERER_ACCELERATED
| SDL_RENDERER_PRESENTVSYNC );

            if( mRenderer == NULL )
            {
printf( "Renderer could not be created! SDL Error: %s\n",
SDL_GetError() );
SDL_DestroyWindow( mWindow );
mWindow = NULL;
            }

            else
            {

      //Initialize renderer color
      SDL_SetRenderDrawColor( mRenderer, 0xFF, 0xFF, 0xFF, 0xFF );

      //Grab window identifier
      mWindowID = SDL_GetWindowID( mWindow );

      //Flag as opened
      mShown = true;

            }
      }
```

```
        else
        {
printf( "Window could not be created! SDL Error: %s\n", SDL_GetError()
);
        }

        return mWindow != NULL && mRenderer != NULL;
}
```

Código de creación de ventana y renderizado.

```
void LWindow::handleEvent( SDL_Event& e )
{
//If an event was detected for this window
if( e.type == SDL_WINDOWEVENT && e.window.windowID == mWindowID )
        {
                //Caption update flag
                bool updateCaption = false;
```

Todos los eventos de todas las ventanas van en la misma cola de eventos, por lo que hay que saber qué eventos pertenecen a cada ventana. A continuación, se comprueba que el evento de la ventana ID coincide.

```
switch( e.window.event )

                {
                        //Window appeared
                        case SDL_WINDOWEVENT_SHOWN:
                        mShown = true;
                        break;

                        //Window disappeared
                        case SDL_WINDOWEVENT_HIDDEN:
                        mShown = false;
                        break;

                        //Get new dimensions and repaint
                        case SDL_WINDOWEVENT_SIZE_CHANGED:
                        mWidth = e.window.data1;
                        mHeight = e.window.data2;
                        SDL_RenderPresent( mRenderer );
                        break;

                        //Repaint on expose
                        case SDL_WINDOWEVENT_EXPOSED:
                        SDL_RenderPresent( mRenderer );
                        break;

                        //Mouse enter
                        case SDL_WINDOWEVENT_ENTER:
                        mMouseFocus = true;
                        updateCaption = true;
                        break;
```

```
                    //Mouse exit
                    case SDL_WINDOWEVENT_LEAVE:
                    mMouseFocus = false;
                    updateCaption = true;
                    break;

                    //Keyboard focus gained
                    case SDL_WINDOWEVENT_FOCUS_GAINED:
                    mKeyboardFocus = true;
                    updateCaption = true;
                    break;

                    //Keyboard focus lost
                    case SDL_WINDOWEVENT_FOCUS_LOST:
                    mKeyboardFocus = false;
                    updateCaption = true;
                    break;

                    //Window minimized
                    case SDL_WINDOWEVENT_MINIMIZED:
                    mMinimized = true;
                    break;

                    //Window maxized
                    case SDL_WINDOWEVENT_MAXIMIZED:
                    mMinimized = false;
                    break;

                    //Window restored
                    case SDL_WINDOWEVENT_RESTORED:
                    mMinimized = false;
                    break;
```

Cuando se tienen varias ventanas, pulsar X no significa necesariamente salir del programa. En este caso, en su lugar se ocultará cada ventana al pulsar X. Para ello, es necesario hacer un seguimiento de cuándo la ventana se muestra o se oculta mediante la comprobación de eventos **SDL_WINDOWEVENT_SHOWN/SDL_WINDOWEVENT_HIDDEN.**

```
                    //Hide on close
                    case SDL_WINDOWEVENT_CLOSE:
                    SDL_HideWindow( mWindow );
                    break;
                }

                //Update window caption with new data
                if( updateCaption )
                {
                std::stringstream caption;
caption << "SDL Tutorial - ID: " << mWindowID << " MouseFocus:" << ( (
mMouseFocus ) ? "On" : "Off" ) << " KeyboardFocus:" << ( (
mKeyboardFocus ) ? "On" : "Off" );

SDL_SetWindowTitle( mWindow, caption.str().c_str() );
                }
        }
}
```

Con múltiples ventanas, pulsar X se relaciona con eventos de ventana tipo **SDL_WINDOWEVENT_CLOSE.** Al obtener estos eventos se oculta la ventana usando **SDL_HideWindow**.

```
void LWindow::focus()
{
	//Restore window if needed
	if( !mShown )
	{
		SDL_ShowWindow( mWindow );
	}

	//Move window forward
	SDL_RaiseWindow( mWindow );
}
```

- En primer lugar se comprueba si la ventana está siendo mostrada.
- A continuación se muestra mediante **SDL_ShowWindow** si ésta no está siendo mostrada.
- Finalmente, se llama a **SDL_RaiseWindow** para enfocar la ventana.

```
void LWindow::render()
{
	if( !mMinimized )
	{
		//Clear screen
	SDL_SetRenderDrawColor( mRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
	SDL_RenderClear( mRenderer );

		//Update screen
		SDL_RenderPresent( mRenderer );
	}
}
```

Sólo se renderiza si la ventana no está minimizada.

```
bool init()
{
	//Initialization flag
	bool success = true;

	//Initialize SDL
	if( SDL_Init( SDL_INIT_VIDEO ) < 0 )

	{
printf( "SDL could not initialize! SDL Error: %s\n", SDL_GetError() );
success = false;
	}
```

```
	else
	{
		//Set texture filtering to linear
		if( !SDL_SetHint( SDL_HINT_RENDER_SCALE_QUALITY, "1" ) )
		{
	printf( "Warning: Linear texture filtering not enabled!" );
		}

		//Create window
		if( !gWindows[ 0 ].init() )
		{
			printf( "Window 0 could not be created!\n" );
			success = false;
		}
	}

	return success;
}
```

En la función de inicialización se abre una sola ventana para comprobar si la creación de la ventana funciona correctamente.

```
void close()
{
	//Destroy windows
	for( int i = 0; i < TOTAL_WINDOWS; ++i )
	{
		gWindows[ i ].free();
	}

	//Quit SDL subsystems
	SDL_Quit();
}
```

En la función de limpieza se cierra cualquier ventana que pudiera estar abierta.

```
		//Initialize the rest of the windows
		for( int i = 1; i < TOTAL_WINDOWS; ++i )
		{
			gWindows[ i ].init();
		}

		//Main loop flag
		bool quit = false;

		//Event handler
		SDL_Event e;
```

Antes de entrar al bucle principal se abre el resto de las ventanas.

```
		//While application is running
		while( !quit )
		{
			//Handle events on queue
			while( SDL_PollEvent( &e ) != 0 )
			{
				//User requests quit
				if( e.type == SDL_QUIT )
				{
					quit = true;
				}

				//Handle window events
				for( int i = 0; i < TOTAL_WINDOWS; ++i )
				{
					gWindows[ i ].handleEvent( e );
				}

				//Pull up window
				if( e.type == SDL_KEYDOWN )
				{
					switch( e.key.keysym.sym )
					{
						case SDLK_1:
						gWindows[ 0 ].focus();
						break;

						case SDLK_2:
						gWindows[ 1 ].focus();
						break;

						case SDLK_3:
						gWindows[ 2 ].focus();
						break;
					}
				}
			}
```

En el bucle principal se controlan los eventos para todas las ventanas, así como las pulsaciones de teclas especiales. En este ejemplo, cuando se pulse 1, 2 ó 3 se abrirá la ventana correspondiente.

```
			//Update all windows
			for( int i = 0; i < TOTAL_WINDOWS; ++i )
			{
				gWindows[ i ].render();
			}

			//Check all windows
			bool allWindowsClosed = true;
			for( int i = 0; i < TOTAL_WINDOWS; ++i )
			{
				if( gWindows[ i ].isShown() )
				{
					allWindowsClosed = false;
					break;
```

```
                }
            }

            //Application closed all windows
            if( allWindowsClosed )
            {
                quit = true;
            }
        }
    }
```

A continuación, se renderizan todas las ventanas y se pasa por cada una de ellas para comprobar si alguna de ella está mostrada. Si todas están cerradas, se establece el flag de salida a true para finalizar el programa. (En este caso no se mostrará nada dentro de las ventanas).

Figura 156: *Resultado ejecutar aplicación Ventanas múltiples*

3.35 Pantallas múltiples

SDL 2.0 tiene la capacidad de controlar múltiples pantallas. El objetivo será hacer una ventana propia para saltar de una pantalla a otra.

```
class LWindow
{
	public:
		//Intializes internals
		LWindow();

		//Creates window
		bool init();

		//Handles window events
		void handleEvent( SDL_Event& e );

		//Focuses on window
		void focus();

		//Shows windows contents
		void render();

		//Deallocates internals
		void free();

		//Window dimensions
		int getWidth();
		int getHeight();

		//Window focii
		bool hasMouseFocus();
		bool hasKeyboardFocus();
		bool isMinimized();
		bool isShown();

	private:
		//Window data
		SDL_Window* mWindow;
		SDL_Renderer* mRenderer;
		int mWindowID;
		int mWindowDisplayID;

		//Window dimensions
		int mWidth;
		int mHeight;

		//Window focus
		bool mMouseFocus;
		bool mKeyboardFocus;
		bool mFullScreen;
		bool mMinimized;
		bool mShown;
};
```

Ventana con dispositivo para hacer un seguimiento de en qué pantalla se muestra la ventana.

```
//Our custom window
LWindow gWindow;

//Display data
int gTotalDisplays = 0;
SDL_Rect* gDisplayBounds = NULL;
```

Todas las pantallas tienen un ID entero y un rectángulo asociado a él para saber la posición y las dimensiones de cada pantalla en el escritorio.

```
bool LWindow::init()
{
	//Create window
mWindow=SDL_CreateWindow("SDL Tutorial",
SDL_WINDOWPOS_UNDEFINED,SDL_WINDOWPOS_UNDEFINED,SCREEN_WIDTH,
SCREEN_HEIGHT,SDL_WINDOW_SHOWN | SDL_WINDOW_RESIZABLE );

	if( mWindow != NULL )
	{
		mMouseFocus = true;
		mKeyboardFocus = true;
		mWidth = SCREEN_WIDTH;
		mHeight = SCREEN_HEIGHT;

		//Create renderer for window
mRenderer = SDL_CreateRenderer(mWindow, -1,SDL_RENDERER_ACCELERATED |
SDL_RENDERER_PRESENTVSYNC );

		if( mRenderer == NULL )
		{
printf("Renderer could not be created! SDL Error: %s\n",
SDL_GetError() );

SDL_DestroyWindow( mWindow );
mWindow = NULL;
		}

		else
		{
	//Initialize renderer color
	SDL_SetRenderDrawColor( mRenderer,0xFF,0xFF,0xFF,0xFF );

	//Grab window identifiers
	mWindowID = SDL_GetWindowID( mWindow );
	mWindowDisplayID = SDL_GetWindowDisplayIndex( mWindow );

	//Flag as opened
	mShown = true;
		}
	}

	else

	{
```

```
printf("Window could not be created! SDL Error: %s\n",SDL_GetError());
        }

        return mWindow != NULL && mRenderer != NULL;
}
```

Código de creación de ventana. Se llama a **SDL_GetWindowDisplayIndex** para saber en qué pantalla se ha creado la ventana.

```
void LWindow::handleEvent( SDL_Event& e )
{
        //Caption update flag
        bool updateCaption = false;

        //If an event was detected for this window
        if(e.type== SDL_WINDOWEVENT&&e.window.windowID==mWindowID)
        {
                switch( e.window.event )
                {
                        //Window moved

                        case SDL_WINDOWEVENT_MOVED:
                        mWindowDisplayID=SDL_GetWindowDisplayIndex(mWindow);
                        updateCaption = true;
                        break;

                        //Window appeared

                        case SDL_WINDOWEVENT_SHOWN:
                        mShown = true;
                        break;

                        //Window disappeared

                        case SDL_WINDOWEVENT_HIDDEN:
                        mShown = false;
                        break;

                        //Get new dimensions and repaint

                        case SDL_WINDOWEVENT_SIZE_CHANGED:
                        mWidth = e.window.data1;
                        mHeight = e.window.data2;
                        SDL_RenderPresent( mRenderer );
                        break;

                        //Repaint on expose

                        case SDL_WINDOWEVENT_EXPOSED:
                        SDL_RenderPresent( mRenderer );
                        break;

                        //Mouse enter

                        case SDL_WINDOWEVENT_ENTER:
                        mMouseFocus = true;
                        updateCaption = true;
```

```
                break;

                //Mouse exit

                case SDL_WINDOWEVENT_LEAVE:
                mMouseFocus = false;
                updateCaption = true;
                break;

                //Keyboard focus gained

                case SDL_WINDOWEVENT_FOCUS_GAINED:
                mKeyboardFocus = true;
                updateCaption = true;
                break;

                //Keyboard focus lost

                case SDL_WINDOWEVENT_FOCUS_LOST:
                mKeyboardFocus = false;
                updateCaption = true;
                break;

                //Window minimized

                case SDL_WINDOWEVENT_MINIMIZED:
                mMinimized = true;
                break;

                //Window maxized

                case SDL_WINDOWEVENT_MAXIMIZED:
                mMinimized = false;
                break;

                //Window restored

                case SDL_WINDOWEVENT_RESTORED:
                mMinimized = false;
                break;

                //Hide on close

                case SDL_WINDOWEVENT_CLOSE:
                SDL_HideWindow( mWindow );
                break;
        }
}
```

En el controlador de eventos de la ventana se controlan eventos **SDL_WINDOWEVENT_MOVED**, de modo que pueda actualizarse la pantalla en la que se está usando la ventana empleando **SDL_GetWindowDisplayIndex**.

```
        else if( e.type == SDL_KEYDOWN )
        {
                //Display change flag
                bool switchDisplay = false;
```

```
            //Cycle through displays on up/down
            switch( e.key.keysym.sym )
            {
                  case SDLK_UP:
                  ++mWindowDisplayID;
                  switchDisplay = true;
                  break;

                  case SDLK_DOWN:
                  --mWindowDisplayID;
                  switchDisplay = true;
                  break;
            }
```

Cuando se pulsa o se suelta el clic, se cambia el índice de pantalla para pasar a la siguiente pantalla.

```
            //Display needs to be updated
            if( switchDisplay )
            {
                  //Bound display index
                  if( mWindowDisplayID < 0 )
                  {
                        mWindowDisplayID = gTotalDisplays - 1;
                  }

                  else if( mWindowDisplayID >= gTotalDisplays )
                  {
                        mWindowDisplayID = 0;
                  }

            //Move window to center of next display

SDL_SetWindowPosition(mWindow,gDisplayBounds[mWindowDisplayID].x + (
gDisplayBounds[ mWindowDisplayID ].w - mWidth ) / 2, gDisplayBounds[
mWindowDisplayID ].y + ( gDisplayBounds[ mWindowDisplayID ].h -
mHeight ) / 2 );

updateCaption = true;

            }
      }

      //Update window caption with new data

      if( updateCaption )
      {
            std::stringstream caption;
caption << "SDL Tutorial - ID: " << mWindowID << " Display: " <<
mWindowDisplayID << " MouseFocus:" <<( ( mMouseFocus ) ? "On" : "Off"
) <<" KeyboardFocus:" << ( ( mKeyboardFocus ) ? "On" : "Off" );

SDL_SetWindowTitle( mWindow, caption.str().c_str() );

      }
}
```

Si se tiene que pasar a la siguiente pantalla, lo primero es asegurarse de que la pantalla tiene un índice válido limitándola. A continuación, se actualiza la posición de la ventana con **SDL_SetWindowPosition**. Esta llamada centrará la ventana en la siguiente pantalla.

```
bool init()
{
	//Initialization flag
	bool success = true;

	//Initialize SDL
	if( SDL_Init( SDL_INIT_VIDEO ) < 0 )

	{
printf("SDL could not initialize! SDL Error: %s\n",SDL_GetError() );
success = false;
	}

	else
	{
		//Set texture filtering to linear

		if( !SDL_SetHint( SDL_HINT_RENDER_SCALE_QUALITY, "1" ) )
		{
	printf("Warning: Linear texture filtering not enabled!");
		}

		//Get number of displays

		gTotalDisplays = SDL_GetNumVideoDisplays();
		if( gTotalDisplays < 2 )
		{
			printf( "Warning: Only one display connected!" );
		}
```

En la función de inicialización se muestran cuántas pantallas están conectadas al ordenador mediante **SDL_GetNumVideoDisplays**. Si sólo hay una pantalla, se muestra una advertencia.

```
		//Get bounds of each display
		gDisplayBounds = new SDL_Rect[gTotalDisplays];
		for( int i = 0; i < gTotalDisplays; ++i )
		{
			SDL_GetDisplayBounds(i,&gDisplayBounds[i]);
		}
		//Create window
		if( !gWindow.init() )
		{
			printf( "Window could not be created!\n" );
			success = false;
		}
	}

	return success;
}
```

Se liberan rectángulos para cada una de las pantallas conectadas y se obtienen sus límites mediante **SDL_GetDisplayBounds**. Una vez hecho esto, se inicializa la ventana.

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//While application is running
while( !quit )
{
	//Handle events on queue
	while( SDL_PollEvent( &e ) != 0 )
	{
		//User requests quit
		if( e.type == SDL_QUIT )
		{
			quit = true;
		}

		//Handle window events
		gWindow.handleEvent( e );
	}

	//Update window
	gWindow.render();
}
```

Dado que el código está encapsulado, el bucle principal no ha cambiado.

Al ejecutar la aplicación, dado que sólo hay una pantalla conectada:

Figura 157: *Resultado ejecutar aplicación Pantallas múltiples*

3.36 Partículas en movimiento

Las partículas son mini animaciones. El objetivo de este programa será tomar las partículas y repartirlas alrededor de un punto para crear un rastro de partículas de colores brillantes.

```
//Particle count
const int TOTAL_PARTICLES = 20;

class Particle
{
	public:

		//Initialize position and animation
		Particle( int x, int y );

		//Shows the particle
		void render();

		//Checks if particle is dead
		bool isDead();

	private:

		//Offsets
		int mPosX, mPosY;

		//Current frame of animation
		int mFrame;

		//Type of particle
		LTexture *mTexture;
};
```

Clase Partícula sencilla. Consta de:

- Constructor para establecer la posición.
- Función para renderizar.
- Función para saber si la partícula está <muerta>.

En cuanto a los miembros de datos se tiene:

- Posición.
- Fotograma de animación.
- Textura con la que se renderizará.

```
//The dot that will move around on the screen
class Dot
{
    public:

		//The dimensions of the dot
		static const int DOT_WIDTH = 20;
		static const int DOT_HEIGHT = 20;
```

```
            //Maximum axis velocity of the dot
            static const int DOT_VEL = 10;

            //Initializes the variables and allocates particles
            Dot();

            //Deallocates particles
            ~Dot();

            //Takes key presses and adjusts the dot's velocity
            void handleEvent( SDL_Event& e );

            //Moves the dot
            void move();

            //Shows the dot on the screen
            void render();

    private:
            //The particles
            Particle* particles[ TOTAL_PARTICLES ];

            //Shows the particles
            void renderParticles();

            //The X and Y offsets of the dot
            int mPosX, mPosY;

            //The velocity of the dot
            int mVelX, mVelY;
};
```

Clase Punto con un array de partículas y una función para renderizar las partículas con el punto.

```
Particle::Particle( int x, int y )
{
    //Set offsets
    mPosX = x - 5 + ( rand() % 25 );
    mPosY = y - 5 + ( rand() % 25 );

    //Initialize animation
    mFrame = rand() % 5;

    //Set type
    switch( rand() % 3 )
    {
        case 0: mTexture = &gRedTexture; break;
        case 1: mTexture = &gGreenTexture; break;
        case 2: mTexture = &gBlueTexture; break;
    }
}
```

En el constructor se inicializa la posición de la partícula alrededor de la posición obtenida de forma aleatoria. A continuación, se inicializa el fotograma de animación con cierta aleatoriedad, por lo que las partículas tendrán vida variable hasta que finalmente se elige el tipo de textura que se usará para la partícula, también al azar.

```
void Particle::render()
{
    //Show image
      mTexture->render( mPosX, mPosY );

    //Show shimmer
    if( mFrame % 2 == 0 )
    {
            gShimmerTexture.render( mPosX, mPosY );
    }

    //Animate
    mFrame++;
}
```

- Se renderiza la textura seleccionada en el constructor.

- Cada fotograma se renderiza con una textura de brillo semitransparente para hacer que parezca que la partícula está brillando.

- A continuación, se actualiza el fotograma de animación.

```
bool Particle::isDead()
{
    return mFrame > 10;
}
```

Una vez se ha renderizado la partícula para un máximo de 10 fotogramas, se marca como muerta.

```
Dot::Dot()
{
    //Initialize the offsets

    mPosX = 0;
    mPosY = 0;

    //Initialize the velocity

    mVelX = 0;
    mVelY = 0;

    //Initialize particles
```

```
    for( int i = 0; i < TOTAL_PARTICLES; ++i )
    {
        particles[ i ] = new Particle( mPosX, mPosY );
    }
}

Dot::~Dot()
{
    //Delete particles

    for( int i = 0; i < TOTAL_PARTICLES; ++i )
    {
        delete particles[ i ];
    }
}
```

El constructor/ destructor tiene que asignar/desasignar las partículas renderizadas con el punto.

```
void Dot::render()
{
      //Show the dot
      gDotTexture.render( mPosX, mPosY );

      //Show particles on top of dot
      renderParticles();
}

void Dot::renderParticles()
{
      //Go through particles
    for( int i = 0; i < TOTAL_PARTICLES; ++i )
    {
        //Delete and replace dead particles
        if( particles[ i ]->isDead() )
        {
            delete particles[ i ];
            particles[ i ] = new Particle( mPosX, mPosY );
        }
    }

    //Show particles
    for( int i = 0; i < TOTAL_PARTICLES; ++i )
    {
        particles[ i ]->render();
    }
}
```

La función de renderizado del punto llama a la función de renderizado de la partícula. Ésta última comprueba si hay partículas muertas para reemplazarlas. Una vez han sido reemplazadas, se renderizan las nuevas partículas.

```
bool loadMedia()
{
	//Loading success flag
	bool success = true;

	//Load dot texture
	if(!gDotTexture.loadFromFile("38_particle_engines/dot.bmp"))
	{
		printf( "Failed to load dot texture!\n" );
		success = false;
	}

	//Load red texture
	if(!gRedTexture.loadFromFile("38_particle_engines/red.bmp"))
	{
		printf( "Failed to load red texture!\n" );
		success = false;
	}

	//Load green texture
	if(!gGreenTexture.loadFromFile("38_particle_engines/green.bmp"))
	{
		printf( "Failed to load green texture!\n" );
		success = false;
	}

	//Load blue texture
	if(!gBlueTexture.loadFromFile("38_particle_engines/blue.bmp"))
	{
		printf( "Failed to load blue texture!\n" );
		success = false;
	}

	//Load shimmer texture
if(!gShimmerTexture.loadFromFile("38_particle_engines/shimmer.bmp"))
	{
		printf( "Failed to load shimmer texture!\n" );
		success = false;
	}

	//Set texture transparency

	gRedTexture.setAlpha( 192 );
	gGreenTexture.setAlpha( 192 );
	gBlueTexture.setAlpha( 192 );
	gShimmerTexture.setAlpha( 192 );

	return success;
}
```

Para que las partículas sean semitransparentes se establece su valor de alfa en 192.

```
			//Main loop flag
			bool quit = false;

			//Event handler
			SDL_Event e;
```

```
                //The dot that will be moving around on the screen
                Dot dot;

                //While application is running
                while( !quit )
                {
                        //Handle events on queue
                        while( SDL_PollEvent( &e ) != 0 )
                        {
                                //User requests quit
                                if( e.type == SDL_QUIT )
                                {
                                        quit = true;
                                }

                                //Handle input for the dot
                                dot.handleEvent( e );
                        }

        //Move the dot
        dot.move();

        //Clear screen
        SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
        SDL_RenderClear( gRenderer );

        //Render objects
        dot.render();

        //Update screen
        SDL_RenderPresent( gRenderer );

                }
        }
}
```

Como el código está encapsulado, el código en el bucle principal apenas cambia.

```
//Using SDL, SDL_image, standard IO, and strings
#include <SDL.h>
#include <SDL_image.h>
#include <stdio.h>
#include <string>
#include <cstdlib>
```

Por el momento no existe una función aleatoria estándar en C++, hay que definirla en cstdlib.

Figura 158: *Resultado ejecutar aplicación Partículas en movimiento*

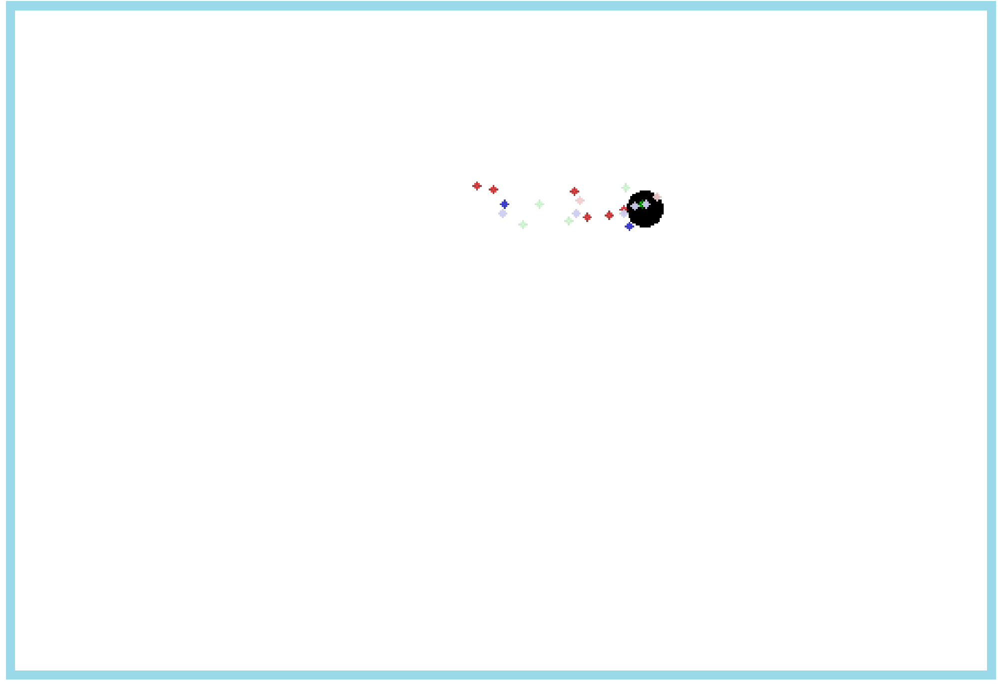


Figura 159: *Resultado ejecutar aplicación Partículas en movimiento (II)*

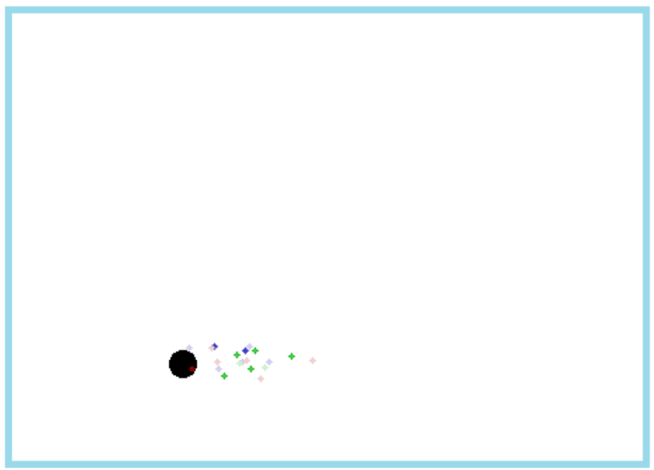


Figura 160: *Resultado ejecutar aplicación Partículas en movimiento (III)*

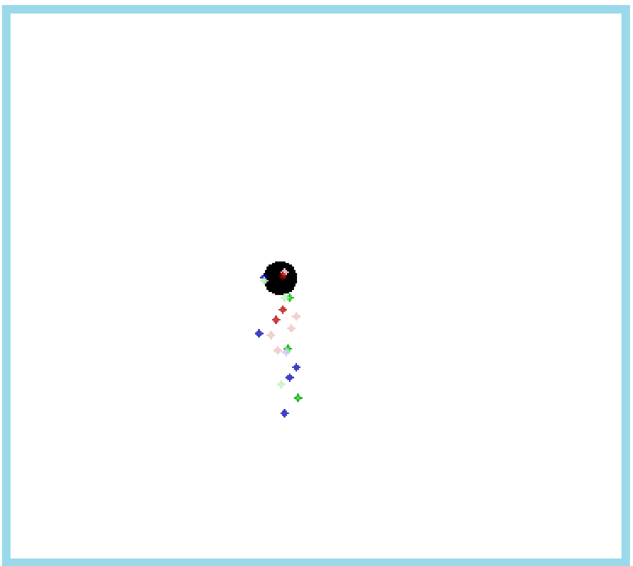


Figura 161: *Resultado ejecutar aplicación Partículas en movimiento (IV)*

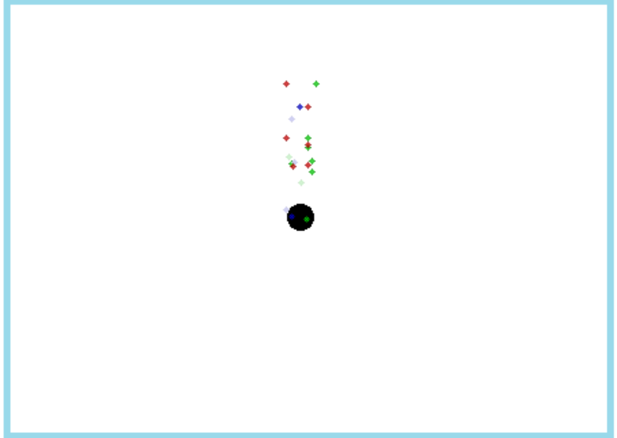


Figura 162: *Resultado ejecutar aplicación Partículas en movimiento (V)*

3.37 Tiling (Embaldosado)

El uso de azulejos es una forma de hacer niveles de piezas reutilizables de tamaño uniforme. El objetivo es realizar un nivel de tamaño 1280x960 con un conjunto de azulejos en la parte de fuera de 160x120.

Si quisiera hacerse un nivel tal que:

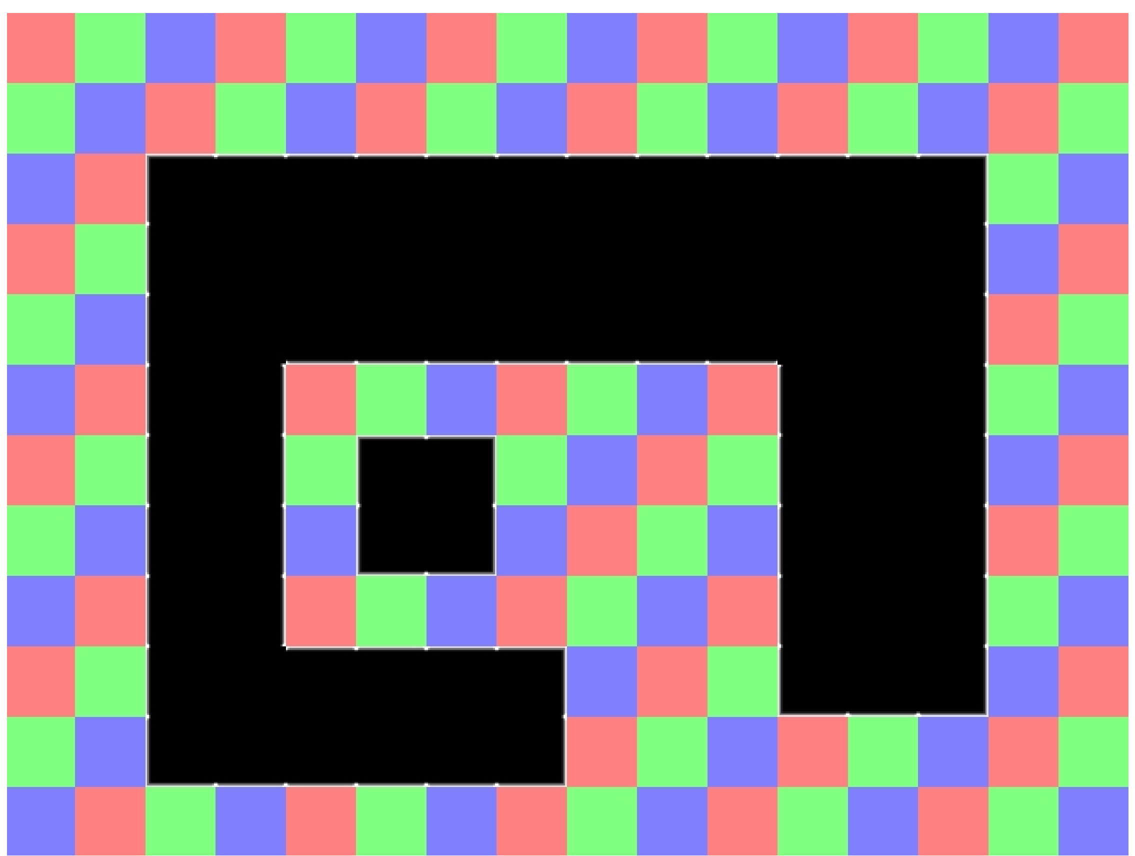


Figura 163: *Tiling*

Podría crearse un gran nivel o crear un conjunto de azulejos de 12 piezas:

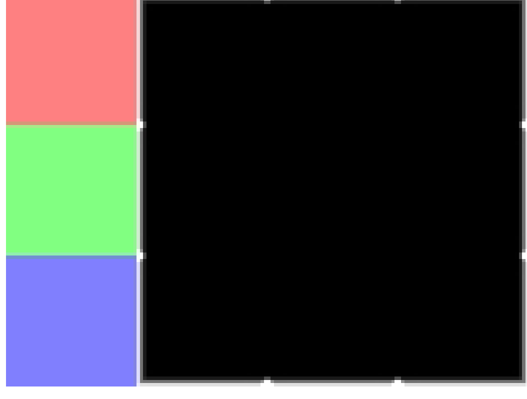


Figura 164: *Tiling (II)*

Y, a continuación, crear un nivel de esas piezas que permita ahorrar memoria y tiempo mediante la reutilización de piezas.

```
//Using SDL, SDL_image, standard IO, strings, and file streams
#include <SDL.h>
#include <SDL_image.h>
#include <stdio.h>
#include <string>
#include <fstream>
```

Se usará la cabecera <fstream>, que es parte de las librerías estándar de C++, al ser relativamente fácil de usar con archivos de texto.

```
//Screen dimension constants
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;

//The dimensions of the level
const int LEVEL_WIDTH = 1280;
const int LEVEL_HEIGHT = 960;

//Tile constants
const int TILE_WIDTH = 80;
const int TILE_HEIGHT = 80;
const int TOTAL_TILES = 192;
const int TOTAL_TILE_SPRITES = 12;

//The different tile sprites
const int TILE_RED = 0;
const int TILE_GREEN = 1;
const int TILE_BLUE = 2;
const int TILE_CENTER = 3;
const int TILE_TOP = 4;
const int TILE_TOPRIGHT = 5;
const int TILE_RIGHT = 6;
const int TILE_BOTTOMRIGHT = 7;
const int TILE_BOTTOM = 8;
const int TILE_BOTTOMLEFT = 9;
const int TILE_LEFT = 10;
const int TILE_TOPLEFT = 11;
```

Ésta es la definición de las constantes.

Se usará el desplazamiento, por lo que serán necesarias contantes tanto para el nivel como para la pantalla. Además, se usan constantes para definir las baldosas así com los tipos de baldosas.

```
//The tile
class Tile
{
    public:
```

```
        //Initializes position and type
        Tile( int x, int y, int tileType );

        //Shows the tile
        void render( SDL_Rect& camera );

        //Get the tile type
        int getType();

        //Get the collision box
        SDL_Rect getBox();

    private:
        //The attributes of the tile
        SDL_Rect mBox;

        //The tile type
        int mType;
};
```

Ésta es la Clase Baldosa. Consta de:

- Constructor, que define la posición y tipo.
- Renderizador.
- Descriptores de acceso para obtener el tipo de baldosa y la colisión con ella.

En cuanto a los miembros de datos, se tiene un rectángulo de colisión e indicador del tipo.

```
//The dot that will move around on the screen
class Dot

{
    public:
        //The dimensions of the dot
        static const int DOT_WIDTH = 20;
        static const int DOT_HEIGHT = 20;

        //Maximum axis velocity of the dot
        static const int DOT_VEL = 10;

        //Initializes the variables
        Dot();

        //Takes key presses and adjusts the dot's velocity
        void handleEvent( SDL_Event& e );

        //Moves the dot and check collision against tiles
        void move( Tile *tiles[] );

        //Centers the camera over the dot
        void setCamera( SDL_Rect& camera );


        //Shows the dot on the screen
        void render( SDL_Rect& camera );

    private:
```

```
            //Collision box of the dot
            SDL_Rect mBox;

            //The velocity of the dot
            int mVelX, mVelY;
};
```

Ésta es la Clase Punto, con la capacidad de comprobar la colisión contra las baldosas cuando se mueve.

```
//Starts up SDL and creates window
bool init();

//Loads media
bool loadMedia( Tile* tiles[] );

//Frees media and shuts down SDL
void close( Tile* tiles[] );

//Box collision detector
bool checkCollision( SDL_Rect a, SDL_Rect b );

//Checks collision box against set of tiles
bool touchesWall( SDL_Rect box, Tile* tiles[] );

//Sets tiles from tile map
bool setTiles( Tile *tiles[] );
```

La función de carga también inicializará baldosas, por lo que necesita tomarlas como argumento.

La función **touchesWall**, que comprueba el cuadro de colisión contra toda pared en un conjunto de baldosas, se utilizará cuando sea necesario comprobar que el punto ha chocado contra todo el conjunto de baldosas.

Finalmente, **setTiles** carga y establece las baldosas.

```
Tile::Tile( int x, int y, int tileType )
{
    //Get the offsets
    mBox.x = x;
    mBox.y = y;

    //Set the collision box
    mBox.w = TILE_WIDTH;
    mBox.h = TILE_HEIGHT;


    //Get the tile type
    mType = tileType;
}
```

El constructor inicializa la posición, dimensiones y tipo.

```
void Tile::render( SDL_Rect& camera )
{
    //If the tile is on screen
    if( checkCollision( camera, mBox ) )
    {
        //Show the tile

gTileTexture.render( mBox.x - camera.x, mBox.y - camera.y,
&gTileClips[ mType ] );

    }
}
```

Cuando se renderiza, sólo se muestran las baldosas que están dentro de la visión de la cámara:

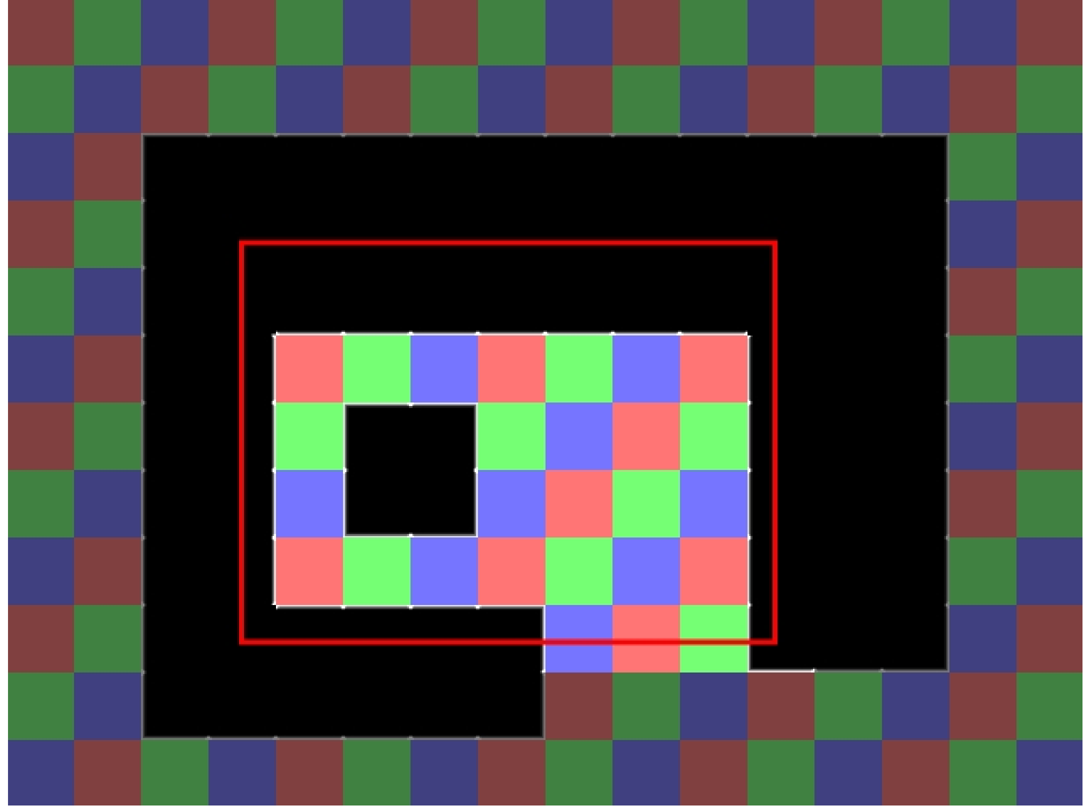


Figura 165: *Tiling (III)*

De modo que se comprueba si el azulejo choca con la cámara antes de renderizarlos. (Se renderiza el azulejo relativo a la cámara).

```
int Tile::getType()
{
    return mType;
}
```

```
SDL_Rect Tile::getBox()
{
    return mBox;
}
```

Descriptores de acceso para obtener el tipo de baldosa y la colisión.

```
void Dot::move( Tile *tiles[] )
{
    //Move the dot left or right
    mBox.x += mVelX;

    //If the dot went too far to the left or right or touched a wall
    if( ( mBox.x < 0 ) || ( mBox.x + DOT_WIDTH > LEVEL_WIDTH ) ||
touchesWall( mBox, tiles ) )
    {
        //move back
        mBox.x -= mVelX;
    }

    //Move the dot up or down
    mBox.y += mVelY;

    //If the dot went too far up or down or touched a wall
    if( ( mBox.y < 0 ) || ( mBox.y + DOT_HEIGHT > LEVEL_HEIGHT ) ||
touchesWall( mBox, tiles ) )
    {
        //move back
        mBox.y -= mVelY;
    }
}
```

Cuando el punto se mueve, se comprueba si se ha salido del nivel o si ha golpeado la pared. De ser así, se corrige.

```
void Dot::setCamera( SDL_Rect& camera )

{
      //Center the camera over the dot
      camera.x = ( mBox.x + DOT_WIDTH / 2 ) - SCREEN_WIDTH / 2;
      camera.y = ( mBox.y + DOT_HEIGHT / 2 ) - SCREEN_HEIGHT / 2;

      //Keep the camera in bounds
      if( camera.x < 0 )
      {
            camera.x = 0;
      }


      if( camera.y < 0 )
      {
            camera.y = 0;
      }
```

```
        if( camera.x > LEVEL_WIDTH - camera.w )
        {
                camera.x = LEVEL_WIDTH - camera.w;
        }

        if( camera.y > LEVEL_HEIGHT - camera.h )
        {
                camera.y = LEVEL_HEIGHT - camera.h;
        }
}

void Dot::render( SDL_Rect& camera )

{
    //Show the dot
        gDotTexture.render( mBox.x - camera.x, mBox.y - camera.y );
}
```

Éste es el código de renderizado.

```
bool loadMedia( Tile* tiles[] )
{
        //Loading success flag
        bool success = true;

        //Load dot texture
        if( !gDotTexture.loadFromFile( "39_tiling/dot.bmp" ) )
        {
                printf( "Failed to load dot texture!\n" );
                success = false;
        }

        //Load tile texture
        if( !gTileTexture.loadFromFile( "39_tiling/tiles.png" ) )
        {
                printf( "Failed to load tile set texture!\n" );
                success = false;
        }

        //Load tile map
        if( !setTiles( tiles ) )
        {
                printf( "Failed to load tile set!\n" );
                success = false;
        }

        return success;
}
```

En la función de carga no sólo se cargan las texturas, sino también el conjunto de baldosas.

```
bool setTiles( Tile* tiles[] )
{
    //Success flag
    bool tilesLoaded = true;
```

```
    //The tile offsets
    int x = 0, y = 0;

    //Open the map
    std::ifstream map( "39_tiling/lazy.map" );

    //If the map couldn't be loaded
    if( map == NULL )
    {
            printf( "Unable to load map file!\n" );
            tilesLoaded = false;
    }
```

En la parte superior de la función setTiles se declaran compensaciones (X, Y) que definen el lugar de las baldosas.

Mediante **fstream** pueden leerse textos de archivos de forma muy similar a como se leen las entradas de teclado con **iostream**.

Antes de continuar, se comprueba si el mapa se ha cargado correctamente comprobando si es null. De ser así, no se continúa. En caso contrario, se sigue cargando el archivo.

```
      else
      {
            //Initialize the tiles
            for( int i = 0; i < TOTAL_TILES; ++i )
            {
                  //Determines what kind of tile will be made
                  int tileType = -1;

                  //Read tile from map file
                  map >> tileType;

                  //If the was a problem in reading the map
                  if( map.fail() )
                  {
                        //Stop loading map

            printf( "Error loading map: Unexpected end of file!\n" );
            tilesLoaded = false;
            break;
                  }

                  //If the number is a valid tile number

      if( ( tileType >= 0 ) && ( tileType < TOTAL_TILE_SPRITES ) )
                  {
                        tiles[ i ] = new Tile( x, y, tileType );
                  }

//If we don't recognize the tile type
                  else
                  {
      //Stop loading map
```

```
printf( "Error loading map: Invalid tile type at %d!\n", i );
tilesLoaded = false;
break;
            }
```

Si el archivo se ha cargado correctamente:

- El bucle for lee todos los números del archivo de texto.
- Se lee el número de la variable tileType y se comprueba si la lectura ha fallado. De ser así, no se continúa. En caso contrario, se verifica que el número del tipo de baldosa es válido.
- Si ocurre así, se crea una nueva azulejo a partir de la obtenida. Si no, se imprime un error y se dejan de cargar baldosas.

```
            //Move to next tile spot
            x += TILE_WIDTH;

            //If we've gone too far
            if( x >= LEVEL_WIDTH )
            {
                  //Move back
                  x = 0;

                  //Move to the next row
                  y += TILE_HEIGHT;
            }
```

Tras cargar una baldosa, se pasa a la posición del texto de la baldosa a la derecha. Si se alcanza el final de la línea de baldosas, se pasará a la siguiente línea de abajo.

```
      //Clip the sprite sheet

      if( tilesLoaded )
      {

            gTileClips[ TILE_RED ].x = 0;
            gTileClips[ TILE_RED ].y = 0;
            gTileClips[ TILE_RED ].w = TILE_WIDTH;
            gTileClips[ TILE_RED ].h = TILE_HEIGHT;

            gTileClips[ TILE_GREEN ].x = 0;
            gTileClips[ TILE_GREEN ].y = 80;
            gTileClips[ TILE_GREEN ].w = TILE_WIDTH;
            gTileClips[ TILE_GREEN ].h = TILE_HEIGHT;



            gTileClips[ TILE_BLUE ].x = 0;
            gTileClips[ TILE_BLUE ].y = 160;
            gTileClips[ TILE_BLUE ].w = TILE_WIDTH;
            gTileClips[ TILE_BLUE ].h = TILE_HEIGHT;
```

```
                gTileClips[ TILE_TOPLEFT ].x = 80;
                gTileClips[ TILE_TOPLEFT ].y = 0;
                gTileClips[ TILE_TOPLEFT ].w = TILE_WIDTH;
                gTileClips[ TILE_TOPLEFT ].h = TILE_HEIGHT;

                gTileClips[ TILE_LEFT ].x = 80;
                gTileClips[ TILE_LEFT ].y = 80;
                gTileClips[ TILE_LEFT ].w = TILE_WIDTH;
                gTileClips[ TILE_LEFT ].h = TILE_HEIGHT;

                gTileClips[ TILE_BOTTOMLEFT ].x = 80;
                gTileClips[ TILE_BOTTOMLEFT ].y = 160;
                gTileClips[ TILE_BOTTOMLEFT ].w = TILE_WIDTH;
                gTileClips[ TILE_BOTTOMLEFT ].h = TILE_HEIGHT;

                gTileClips[ TILE_TOP ].x = 160;
                gTileClips[ TILE_TOP ].y = 0;
                gTileClips[ TILE_TOP ].w = TILE_WIDTH;
                gTileClips[ TILE_TOP ].h = TILE_HEIGHT;

                gTileClips[ TILE_CENTER ].x = 160;
                gTileClips[ TILE_CENTER ].y = 80;
                gTileClips[ TILE_CENTER ].w = TILE_WIDTH;
                gTileClips[ TILE_CENTER ].h = TILE_HEIGHT;

                gTileClips[ TILE_BOTTOM ].x = 160;
                gTileClips[ TILE_BOTTOM ].y = 160;
                gTileClips[ TILE_BOTTOM ].w = TILE_WIDTH;
                gTileClips[ TILE_BOTTOM ].h = TILE_HEIGHT;

                gTileClips[ TILE_TOPRIGHT ].x = 240;
                gTileClips[ TILE_TOPRIGHT ].y = 0;
                gTileClips[ TILE_TOPRIGHT ].w = TILE_WIDTH;
                gTileClips[ TILE_TOPRIGHT ].h = TILE_HEIGHT;

                gTileClips[ TILE_RIGHT ].x = 240;
                gTileClips[ TILE_RIGHT ].y = 80;
                gTileClips[ TILE_RIGHT ].w = TILE_WIDTH;
                gTileClips[ TILE_RIGHT ].h = TILE_HEIGHT;

                gTileClips[ TILE_BOTTOMRIGHT ].x = 240;
                gTileClips[ TILE_BOTTOMRIGHT ].y = 160;
                gTileClips[ TILE_BOTTOMRIGHT ].w = TILE_WIDTH;
                gTileClips[ TILE_BOTTOMRIGHT ].h = TILE_HEIGHT;
            }
        }

    //Close the file
    map.close();

    //If the map was loaded fine
    return tilesLoaded;
}
```

Una vez cargadas todas las baldosas se establecen los rectángulos de recorte para los sprites de baldosas. Finalmente se carga el archivo de mapa y el retorno.

```
bool touchesWall( SDL_Rect box, Tile* tiles[] )
{
    //Go through the tiles
    for( int i = 0; i < TOTAL_TILES; ++i )
    {
        //If the tile is a wall type tile

if( ( tiles[ i ]->getType() >= TILE_CENTER ) && ( tiles[ i ]-
>getType() <= TILE_TOPLEFT ) )
        {
            //If the collision box touches the wall tile
            if( checkCollision( box, tiles[ i ]->getBox() ) )
            {
                return true;
            }
        }
    }

    //If no wall tiles were touched
    return false;
}
```

TouchesWall comprueba un cuadro de colisión dado contra baldosas del tipo TILE_CENTER, TILE_TOP, TILE_TOPRIGHT, TILE_RIGHT, TILE_BOTTOMRIGHT, TILE_BOTTOM, TILE_BOTTOMLEFT, TILE_LEFT, and TILE_TOPLEFT, que son todas las baldosas de pared.

Si el cuadro de colisión choca con cualquier baldosa que haya en la pared, la función devuelve True.

```
            //The level tiles
            Tile* tileSet[ TOTAL_TILES ];

            //Load media
            if( !loadMedia( tileSet ) )
            {
                  printf( "Failed to load media!\n" );
            }
```

En la función principal se declara el array de punteros de baldosas.

```
//While application is running
                  while( !quit )
                  {
                        //Handle events on queue
                        while( SDL_PollEvent( &e ) != 0 )

                        {

                              //User requests quit
                              if( e.type == SDL_QUIT )
                              {
                                    quit = true;
```

```
                        }

                        //Handle input for the dot
                        dot.handleEvent( e );
                }

                //Move the dot
                dot.move( tileSet );
                dot.setCamera( camera );

    //Clear screen
    SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
    SDL_RenderClear( gRenderer );

    //Render level
                for( int i = 0; i < TOTAL_TILES; ++i )
                {
                        tileSet[ i ]->render( camera );
                }

                //Render dot
                dot.render( camera );

                //Update screen
                SDL_RenderPresent( gRenderer );
        }
    }
```

Cuando se mueve el punto, se pasa el conjunto de baldosas y se fija la cámara sobre el punto después de moverlo. A continuación, se renderiza el azulejo y finalmente el punto sobre el nivel.

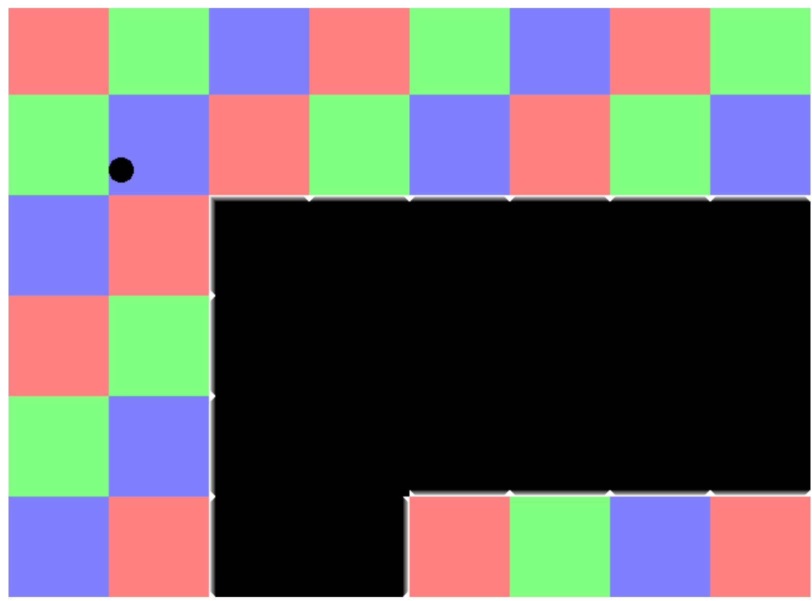


Figura 166: *Resultado ejecutar aplicación Tiling*

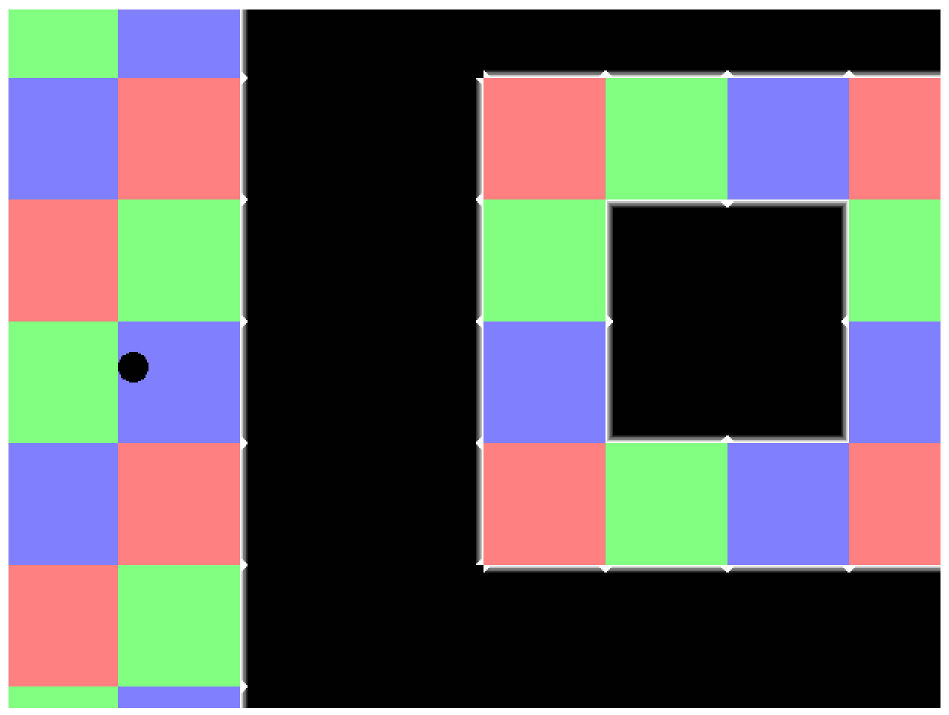


Figura 167: *Resultado ejecutar aplicación Tiling (II)*

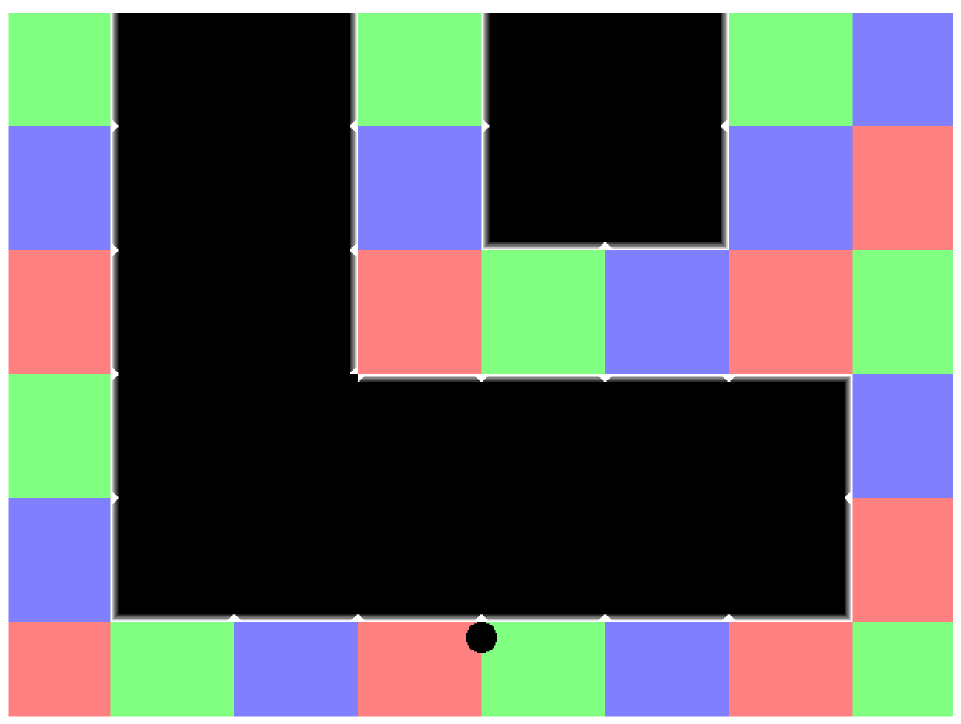


Figura 168: *Resultado ejecutar aplicación Tiling (III)*

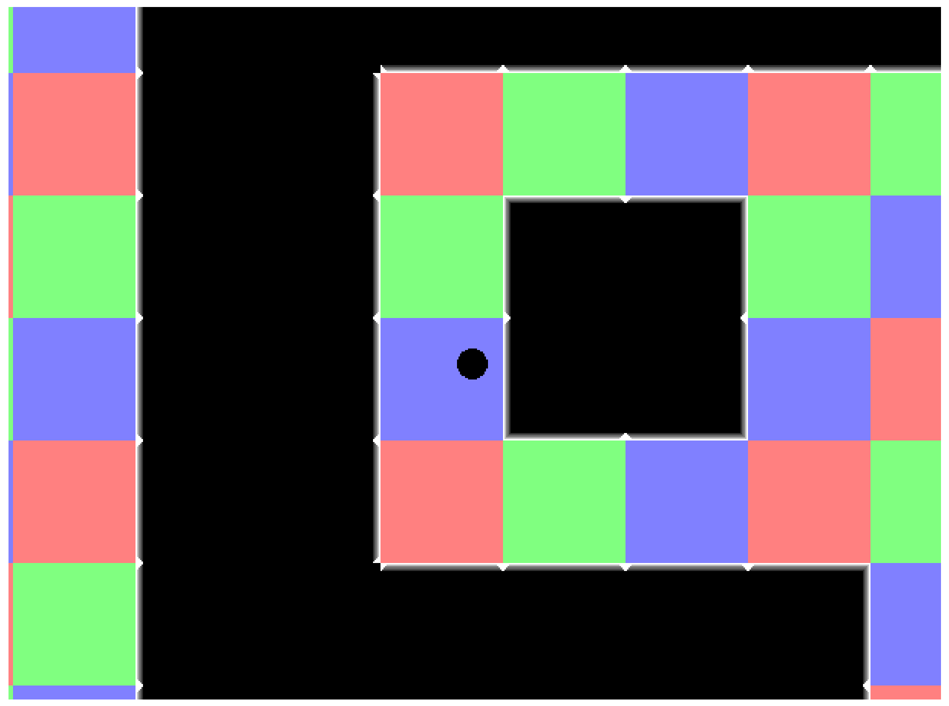


Figura 169: *Resultado ejecutar aplicación Tiling (IV)*

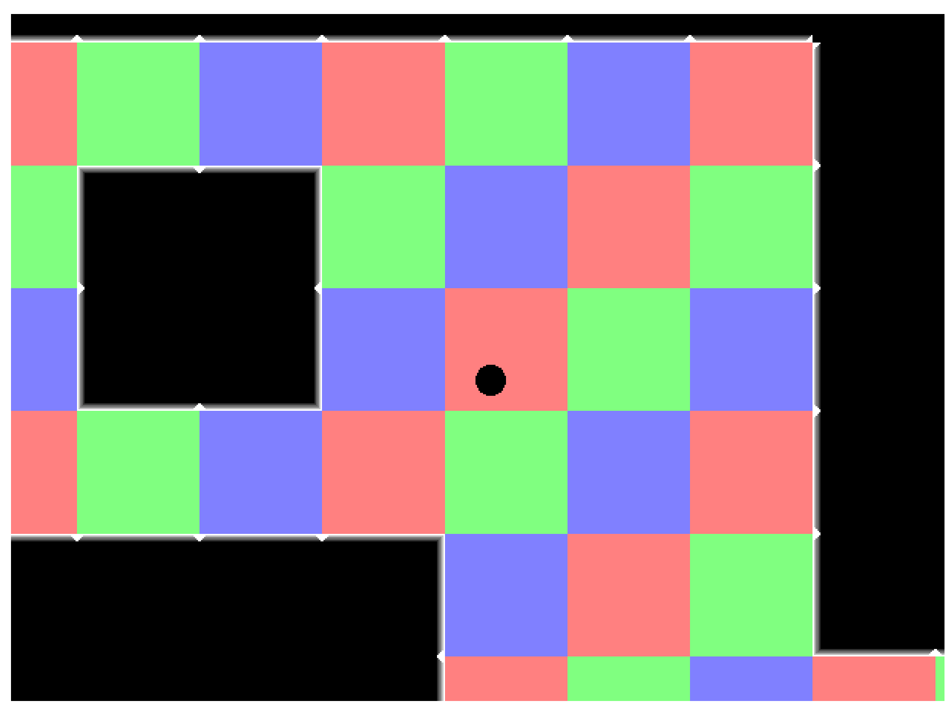


Figura 170: *Resultado ejecutar aplicación Tiling (V)*

3.38 Manipulación de texturas

Para hacer efectos gráficos a menudo se requiere acceso pixel. El objetivo será alterar los píxeles de una imagen para corregir el fondo.

```
//Texture wrapper class
class LTexture
{
	public:
		//Initializes variables
		LTexture();

		//Deallocates memory
		~LTexture();

		//Loads image at specified path
		bool loadFromFile( std::string path );

		#ifdef _SDL_TTF_H
		//Creates image from font string
bool loadFromRenderedText( std::string textureText, SDL_Color
textColor );
		#endif

		//Deallocates texture
		void free();

		//Set color modulation
		void setColor( Uint8 red, Uint8 green, Uint8 blue );

		//Set blending
		void setBlendMode( SDL_BlendMode blending );

		//Set alpha modulation
		void setAlpha( Uint8 alpha );

		//Renders texture at given point
void render( int x, int y, SDL_Rect* clip = NULL, double angle = 0.0,
SDL_Point* center = NULL, SDL_RendererFlip flip = SDL_FLIP_NONE );

		//Gets image dimensions
		int getWidth();
		int getHeight();

		//Pixel manipulators
		bool lockTexture();
		bool unlockTexture();
		void* getPixels();
		int getPitch();

	private:

		//The actual hardware texture
		SDL_Texture* mTexture;
		void* mPixels;
		int mPitch;
```

```
            //Image dimensions
            int mWidth;
            int mHeight;
};
```

Hay funciones para bloquear/ desbloquear la textura, ya que para acceder a los pixeles de una textura hay que bloquearla y, una vez se ha terminado con los pixeles, desbloquearla.

Además, se cuenta con una función para obtener los pixeles primos así como otra para obtener el paso. El paso es la anchura de la textura en memoria. Mediante el paso puede saberse cómo se encuentra la imagen en la memoria.

En cuanto a los datos, se tiene un puntero para los pixeles después de que se bloquee la textura y el paso.

```
bool LTexture::loadFromFile( std::string path )
{
      //Get rid of preexisting texture
      free();

      //The final texture
      SDL_Texture* newTexture = NULL;

      //Load image at specified path
      SDL_Surface* loadedSurface = IMG_Load( path.c_str() );
      if( loadedSurface == NULL )
      {
printf( "Unable to load image %s! SDL_image Error: %s\n",
path.c_str(), IMG_GetError() );
      }

      else
      {
            //Convert surface to display format
            SDL_Surface* formattedSurface = SDL_ConvertSurface(
loadedSurface, SDL_GetWindowSurface( gWindow )->format, NULL );
            if( formattedSurface == NULL )
            {
printf( "Unable to convert loaded surface to display format! SDL
Error: %s\n", SDL_GetError() );
            }

            else
            {
                  //Create blank streamable texture
newTexture = SDL_CreateTexture( gRenderer, SDL_GetWindowPixelFormat(
gWindow ), SDL_TEXTUREACCESS_STREAMING, formattedSurface->w,
formattedSurface->h );

if( newTexture == NULL )
                  {
printf( "Unable to create blank texture! SDL Error: %s\n",
SDL_GetError() );
                  }
```

Si se quiere poder editar una textura, ésta tiene que cargarse de forma diferente. Cuando se crean texturas a partir de superficies, no pueden modificarse una vez estén creadas.

Para editar los pixeles de la textura se crea la textura con **SDL_TEXTUREACCESS_STREAMING**.

- En primer lugar, hay que cargar la imagen como una superficie.

- A continuación se convierte la superficie al mismo formato de píxel que la ventana y se crea una textura en blanco mediante **SDL_CreateTexture**.

```
                  else
                  {
            //Lock texture for manipulation
            SDL_LockTexture( newTexture, NULL, &mPixels, &mPitch );

            //Copy loaded/formatted surface pixels
memcpy( mPixels, formattedSurface->pixels, formattedSurface->pitch *
formattedSurface->h );

            //Unlock texture to update
            SDL_UnlockTexture( newTexture );
            mPixels = NULL;

            //Get image dimensions
            mWidth = formattedSurface->w;
            mHeight = formattedSurface->h;
                  }

//Get rid of old formatted surface
SDL_FreeSurface( formattedSurface );
            }

//Get rid of old loaded surface
SDL_FreeSurface( loadedSurface );
      }

//Return success
mTexture = newTexture;
return mTexture != NULL;

}
```

Cuando la textura se ha creado, hay que copiar manualmente los pixeles de la superficie a la textura. Para obtener los pixeles de la textura se utiliza **SDL_LockTexture**.

- El primer argumento es la textura de la que se van a obtener los pixeles.

- El segundo hace referencia a la región de la que se quieren obtener los pixeles y, dado que se obtienen de toda la textura, se estable este argumento como null.

- El tercer argumento es el puntero que fijará la dirección de los pixeles.

- El último argumento corresponde con el paso de la textura.

Tras obtener los pixeles de la textura, se copian de la superficie a ésta empleando **memcpy**.

- El primer argumento es el destino.

- El segundo argumetno hace referencia a la fuente.

- El tercer argumento corresponde all número de bytes que se está copiando.

Una vez se han copiado los pixeles, se desbloquea la textura para actualizarla con los nuevos pixeles empleado **SDL_UnlockTexture**. Cuando la textura está desbloqueada, el puntero de pixel se invalida, por lo que toma el valor null.

Finalmente se destruyen las antiguan superficies y se devuelve true si la textura se carga correctamente.

```
bool LTexture::lockTexture()
{
	bool success = true;

	//Texture is already locked
	if( mPixels != NULL )
	{
		printf( "Texture is already locked!\n" );
		success = false;
	}
	//Lock texture
	else
	{

	if( SDL_LockTexture( mTexture, NULL, &mPixels, &mPitch ) != 0 )
		{
		printf( "Unable to lock texture! %s\n", SDL_GetError() );
		success = false;
		}
	}

	return success;
}
```

Función para bloquear la textura.

```
bool LTexture::unlockTexture()
{
	bool success = true;

	//Texture is not locked
	if( mPixels == NULL )
	{
		printf( "Texture is not locked!\n" );
		success = false;
	}
```

```
        //Unlock texture
        else


        {
                SDL_UnlockTexture( mTexture );
                mPixels = NULL;
                mPitch = 0;
        }

        return success;
}
```

Función para desbloquear la textura tras cargar la imagen.

```
void* LTexture::getPixels()
{
        return mPixels;
}

int LTexture::getPitch()
{
        return mPitch;
}
```

Descriptores de acceso para obtener los pixeles y el paso mientras la textura está bloqueada.

```
bool loadMedia()
{
        //Loading success flag
        bool success = true;

        //Load foo' texture
if( !gFooTexture.loadFromFile( "40_texture_manipulation/foo.png" ) )
        {
                printf( "Failed to load corner texture!\n" );
                success = false;
        }
        else
        {
                //Lock texture
                if( !gFooTexture.lockTexture() )
                {
                        printf( "Unable to lock Foo' texture!\n" );
                }
```

En la función de carga se bloquea la textura tras cargarla, de modo que puedan modificarse sus pixeles.

```
//Manual color key
            else

            {

//Get pixel data
Uint32* pixels = (Uint32*)gFooTexture.getPixels();
int pixelCount = ( gFooTexture.getPitch() / 4 ) *
gFooTexture.getHeight();
```

Cuando la textura está bloqueada, se busca hacer todos los pixeles de fondo transparente.

- En primer lugar se obtienen los pixeles. El descriptor de acceso devuelve un puntero vacío y, como se quieren pixeles de 32 bits, se colocan en enteros sin signo de 32 bits.

- A continuación se obtiene el número de pixeles. Se necesita la anchura entre pixeles y dado que hay 4 bytes por pixel, tan sólo hay que dividir por 4 el paso entre pixeles. Seguidamente se multiplica la anchura del paso por la altura para determinar el número total de pixeles.

```
//Map colors
Uint32 colorKey = SDL_MapRGB( SDL_GetWindowSurface( gWindow )->format,
0, 0xFF, 0xFF );

Uint32 transparent = SDL_MapRGBA( SDL_GetWindowSurface( gWindow )-
>format, 0xFF, 0xFF, 0xFF, 0x00 );

//Color key pixels
                  for( int i = 0; i < pixelCount; ++i )
                  {
                        if( pixels[ i ] == colorKey )
                        {
                              pixels[ i ] = transparent;
                        }
                  }

                  //Unlock texture
                  gFooTexture.unlockTexture();
            }
      }

      return success;
}
```

Hay que encontrar los pixeles con color y reemplazarlos por transparentes.

En primer lugar, se hace un mapa de color y otro transparente y se comprueban todos los píxeles por si coinciden con el mapa de color. De ser así, se les asignará el valor de pixel transparente.

Una vez se han terminado de comprobar todos los pixeles, se desbloquea la textura para actualizarla con los nuevos pixeles.

Imagen previa:

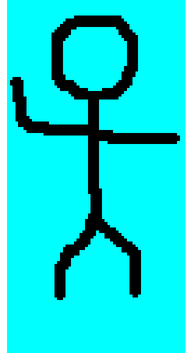


Figura 171: *Imagen previa aplicación Manipulación de texturas*

Tras ejecutar el programa se obtiene:

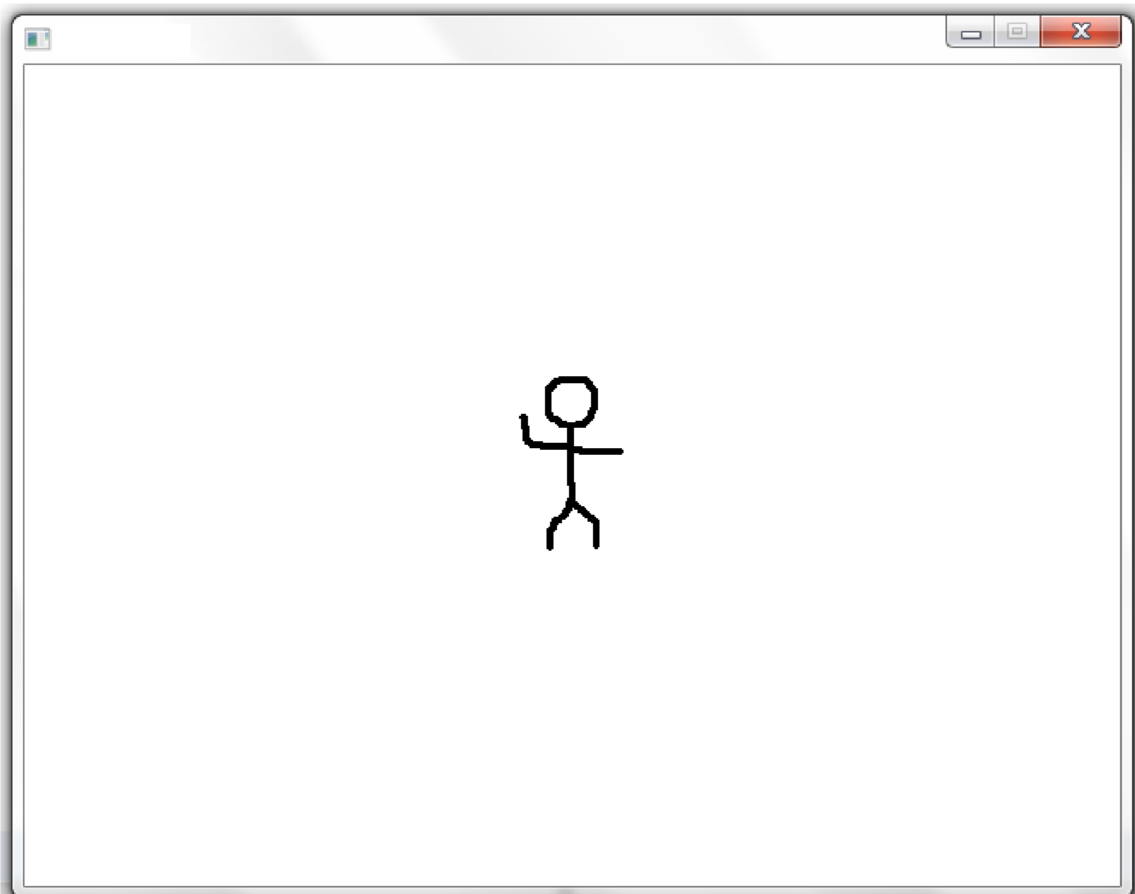


Figura 172: *Resultado ejecutar aplicación Manipulación de texturas*

3.39 Fuentes de mapas de bits

En determinadas ocasiones las fuentes TTF no son lo suficientemente flexibles. Dado que renderizar texto es renderizar imágenes de caracteres, pueden utilizarse fuentes de mapas de bits para renderizar texto.

Si se imagina cada carácter en una cadena como un sprite, puede entenderse la renderización de una fuente como la organización de un conjunto de sprites.

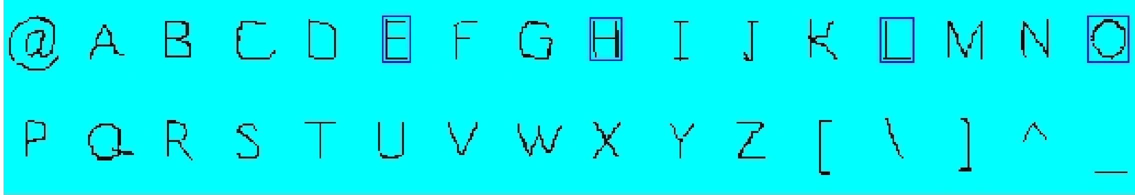


Figura 173: Caracteres previos

A partir de estos caracteres se formará la palabra "Hello".

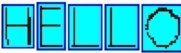


Figura 174: Palabra formada a partir de los caracteres previos

Las fuentes de mapas de bits funcionan tomando una sprite sheet de glyphs (imágenes de caracteres) y renderizándolos para formar cadenas en la pantalla.

```
//Texture wrapper class
class LTexture
{
	public:
		//Initializes variables
		LTexture();
		//Deallocates memory
		~LTexture();
		//Loads image at specified path
		bool loadFromFile( std::string path );
			#ifdef _SDL_TTF_H
		//Creates image from font string
		bool loadFromRenderedText
		( std::string textureText, SDL_Color textColor );
		#endif

		//Deallocates texture
		void free();
		//Set color modulation
		void setColor( Uint8 red, Uint8 green, Uint8 blue );
```

```
            //Set blending
            void setBlendMode( SDL_BlendMode blending );

            //Set alpha modulation
            void setAlpha( Uint8 alpha );

            //Renders texture at given point
void render( int x, int y, SDL_Rect* clip = NULL,double angle = 0.0,
SDL_Point* center = NULL,SDL_RendererFlip flip = SDL_FLIP_NONE );

            //Gets image dimensions
            int getWidth();
            int getHeight();

            //Pixel manipulators
            bool lockTexture();
            bool unlockTexture();
            void* getPixels();
            int getPitch();
            Uint32 getPixel32( unsigned int x, unsigned int y );

      private:

            //The actual hardware texture
            SDL_Texture* mTexture;
            void* mPixels;
            int mPitch;

            //Image dimensions
            int mWidth;
            int mHeight;
};
```

Se necesita obtener los pixeles con unas coordenadas (X, Y) exactas, por lo que se añade la función **getPixel32**. Esta función trabaja específicamente para pixeles de 32 bits.

```
//Our bitmap font
class LBitmapFont
{
    public:
            //The default constructor
            LBitmapFont();

            //Generates the font
            bool buildFont( LTexture *bitmap );

            //Shows the text
            void renderText( int x, int y, std::string text );

    private:

            //The font texture
            LTexture* mBitmap;

            //The individual characters in the surface
            SDL_Rect mChars[ 256 ];
```

```
            //Spacing Variables
            int mNewLine, mSpace;
};
```

Fuente de mapa de bit que funciona como envoltura para la sprite sheet de glyphs. Consta de un constructor para inicializar las variables internas, una función para construir la fuente y una función para renderizar el texto.

Cuando la fuente de mapa de bits está construida, se buscan todos los sprites de 256 caracteres (que están almacenados en el array mChars) y se calcula la distancia para una nueva línea y un espacio.

```
bool LTexture::loadFromFile( std::string path )
{
      //Get rid of preexisting texture
      free();

      //The final texture
      SDL_Texture* newTexture = NULL;

      //Load image at specified path
      SDL_Surface*loadedSurface=IMG_Load(path.c_str());
      if( loadedSurface == NULL )
      {
printf("Unable to load image %s! SDL_image Error: %s\n",
path.c_str(), IMG_GetError() );
      }

      else
      {

    //Convert surface to display format
SDL_Surface* formattedSurface = SDL_ConvertSurfaceFormat (
loadedSurface, SDL_PIXELFORMAT_RGBA8888, NULL );

            if( formattedSurface == NULL )
            {
printf("Unable to convert loaded surface to display format! %s\n",
SDL_GetError() );
            }

            else
{
        //Create blank streamable texture

newTexture=SDL_CreateTexture(gRenderer,SDL_PIXELFORMAT_RGBA8888,
SDL_TEXTUREACCESS_STREAMING,formattedSurface->w,formattedSurface->h);

                  if( newTexture == NULL )
                  {
printf( "Unable to create blank texture! SDL Error: %s\n",
SDL_GetError() );
                  }

                  else
```

```
                {
            //Enable blending on texture
            SDL_SetTextureBlendMode( newTexture, SDL_BLENDMODE_BLEND );

            //Lock texture for manipulation
SDL_LockTexture( newTexture, &formattedSurface->clip_rect,
&mPixels, &mPitch );

            //Copy loaded/formatted surface pixels
memcpy( mPixels, formattedSurface->pixels, formattedSurface->pitch *
formattedSurface->h );

            //Get image dimensions
                        mWidth = formattedSurface->w;
                        mHeight = formattedSurface->h;

//Get pixel data in editable format
                        Uint32* pixels = (Uint32*)mPixels;
                        int pixelCount = ( mPitch / 4 ) * mHeight;

//Map colors
Uint32 colorKey = SDL_MapRGB( formattedSurface->format,
0, 0xFF, 0xFF );

Uint32 transparent=SDL_MapRGBA( formattedSurface->format,
0x00, 0xFF, 0xFF, 0x00 );

//Color key pixels
                        for( int i = 0; i < pixelCount; ++i )
                        {
                              if( pixels[ i ] == colorKey )
                              {
                                    pixels[ i ] = transparent;
                              }
                        }

                        //Unlock texture to update
                        SDL_UnlockTexture( newTexture );
                        mPixels = NULL;
                  }

                  //Get rid of old formatted surface
                  SDL_FreeSurface( formattedSurface );
            }

            //Get rid of old loaded surface
            SDL_FreeSurface( loadedSurface );
      }

      //Return success
      mTexture = newTexture;
      return mTexture != NULL;
}
```

La coloración se realiza en este caso internamente en la función de carga de textura.

En segundo lugar, se especifica el formato del pixel de la textura mediante **SDL_PIXELFORMAT_RGBA8888**, por lo que se obtendrán pixeles RGBA de 32 bit.

```
Uint32 LTexture::getPixel32( unsigned int x, unsigned int y )

{
    //Convert the pixels to 32 bit
    Uint32 *pixels = (Uint32*)mPixels;

    //Get the pixel requested
    return pixels[ ( y * ( mPitch / 4 ) ) + x ];
}
```

Función para obtener el pixel con una compensación específica.

Aunque se tenga una imagen de textura bidimensional tal que:

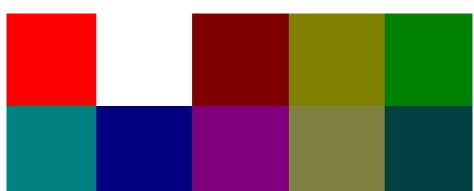


Figura 175: *Imagen de textura bidimensional*

Los pixeles se almacenan en una única dimensión de la siguiente manera:

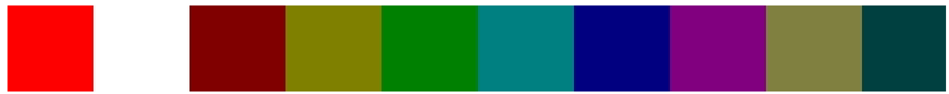


Figura 176: *Imagen de textura unidimensional*

De este modo, si se quiere obtener el pixel azul en la fila 1, columna 1 (la primera fila/columna es la fila/columna 0), la compensación tendría que calcularse así:

$$Compensación\ Y * Paso\ +\ Compensación\ X$$

Es decir:

$$1 * 5 + 1 = 6$$

```
LBitmapFont::LBitmapFont()
{
    //Initialize variables
    mBitmap = NULL;
    mNewLine = 0;
    mSpace = 0;
}
```

En el constructor se inicializan las variables internas.

```
bool LBitmapFont::buildFont( LTexture* bitmap )
{
	bool success = true;

	//Lock pixels for access

	if( !bitmap->lockTexture() )
	{
		printf( "Unable to lock bitmap font texture!\n" );
		success = false;
	}
```

Función que irá a través de la fuente de mapa de bits y define todos los rectángulos de recorte para todos los sprites. Para ello hay que bloquear la textura para acceder a sus pixeles.

```
	else
	{
		//Set the background color
		Uint32 bgColor = bitmap->getPixel32( 0, 0 );

		//Set the cell dimensions
		int cellW = bitmap->getWidth() / 16;
		int cellH = bitmap->getHeight() / 16;

		//New line variables
		int top = cellH;
		int baseA = cellH;

		//The current character we're setting
		int currentChar = 0;
```

Para que la carga de fuentes de mapas de bits funcione, los caracteres glyphs se tienen que organizar en celdas:

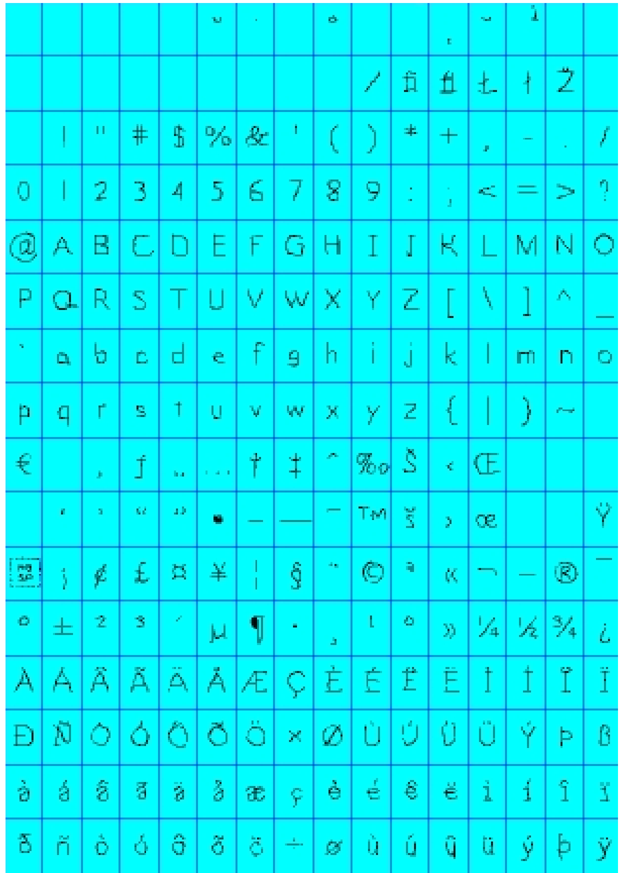


Figura 177: *Caracteres organizados en celdas*

Todas las celdas deben tener la misma anchura y altura, dispuestas en 16 columnas y 16 filas, además de estar en orden ASCII.

En primer lugar, se obtiene el color del fondo que será necesario para encontrar los bordes de los sprites y se calcula la altura y anchura de la celda.

La variable “top” hace un seguimiento de la parte superior del glyph de mayor altura de la hoja de sprites. La variable baseA hace un seguimiento de la compensación de la parte inferior de la A que se usará como line base para la renderización de caracteres.

Por último, currentChar hace un seguimiento del carácter glyph actual que se está buscando.

```
//Go through the cell rows
for( int rows = 0; rows < 16; ++rows )
{
      //Go through the cell columns
      for( int cols = 0; cols < 16; ++cols )

      {
            //Set the character offset
            mChars[ currentChar ].x = cellW * cols;
            mChars[ currentChar ].y = cellH * rows;
```

```
//Set the dimensions of the character
mChars[ currentChar ].w = cellW;
mChars[ currentChar ].h = cellH;
```

Estos dos bucles for anidados son para ir a través de las filas/columnas de celdas.

En la parte superior del bucle se inicializa la posición del sprite glyph en la parte superior de la celda y las dimensiones del sprite serán las dimensiones de la celda, es decir, por defecto el sprite glyph es la celda completa.

```
//Find Left Side
//Go through pixel columns
for( int pCol = 0; pCol < cellW; ++pCol )
{
      //Go through pixel rows
for( int pRow = 0; pRow < cellH; ++pRow )

      {
            //Get the pixel offsets
            int pX = ( cellW * cols ) + pCol;
            int pY = ( cellH * rows ) + pRow;

            //If a non colorkey pixel is found
if( bitmap->getPixel32( pX, pY ) != bgColor )

            {
            //Set the x offset
            mChars[ currentChar ].x = pX;

            //Break the loops
            pCol = cellW;
            pRow = cellH;

            }
      }
}
```

Hay que moverse a través de todos los pixeles de la celda para encontrar el borde del sprite glyph (para cada una de las celdas). En este bucle se comprueba cada columna de arriba abajo y se busca el primer pixel que no es del color del fondo. Una vez se encuentra dicho pixel, se habrá hallado el borde izquierdo del sprite.

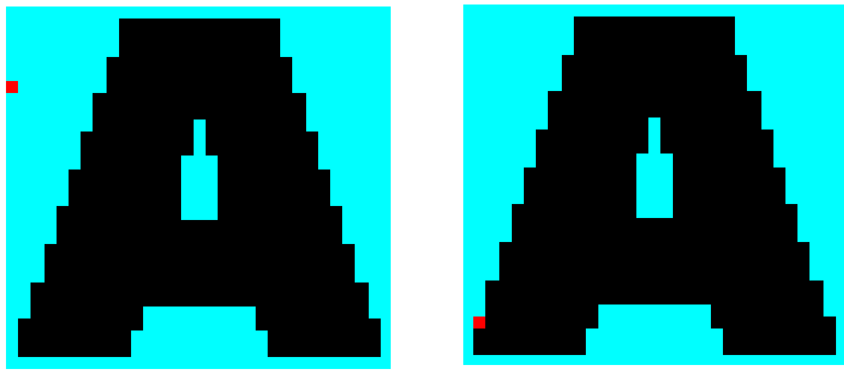


Figura 178: *Detección de pixel de distinto color al fondo*

Tras encontrar el lado izquierdo se establece como posición X del sprite y a continuación se sale de los bucles.

```
                    //Find Right Side
                    //Go through pixel columns
                    for( int pColW = cellW - 1; pColW >= 0; --pColW )
                          {

                          //Go through pixel rows
                          for( int pRowW = 0; pRowW < cellH; ++pRowW )
                                {
                                      //Get the pixel offsets
                                      int pX = ( cellW * cols ) + pColW;
                                      int pY = ( cellH * rows ) + pRowW;

                                      //If a non colorkey pixel is found
                          if( bitmap->getPixel32( pX, pY ) != bgColor )
                                      {

                          //Set the width
                mChars[ currentChar ].w = (pX-mChars[currentChar].x)+ 1;

                          //Break the loops
                          pColW = -1;
                          pRowW = cellH;
                                      }
                                }
                          }
```

Se realiza lo mismo que para el lado izquierdo. En este caso el movimiento será de derecha a izquierda en lugar de izquierda a derecha.

Una vez se ha encontrado el pixel derecho, se emplea para fijar el ancho. Dado que el array de pixel empieza en 0, hay que añadir 1 a la anchura.

```
                          //Find Top
                          //Go through pixel rows

                          for( int pRow = 0; pRow < cellH; ++pRow )
                          {

                          //Go through pixel columns
                          for( int pCol = 0; pCol < cellW; ++pCol )
                                {
                                      //Get the pixel offsets
                                      int pX = ( cellW * cols ) + pCol;
                                      int pY = ( cellH * rows ) + pRow;

                                      //If a non colorkey pixel is found
                          if( bitmap->getPixel32( pX, pY ) != bgColor )
                                      {
                                            //If new top is found
                                            if( pRow < top )
                                            {
```

```
                        top = pRow;
                    }

                    //Break the loops
                    pCol = cellW;
                    pRow = cellH;
                }
            }
        }
```

Código para encontrar la parte superior del sprite. Cuando se encuentra una más alta que la actual, se establece como nueva parte superior. (Al estar el eje Y invertido, la parte superior más alta es la que en realidad tiene la compensación más baja)

```
        //Find Bottom of A
        if( currentChar == 'A' )
        {
        //Go through pixel rows
        for( int pRow = cellH - 1; pRow >= 0; --pRow )
            {
        //Go through pixel columns
        for( int pCol = 0; pCol < cellW; ++pCol )
                {
        //Get the pixel offsets
        int pX = ( cellW * cols ) + pCol;
        int pY = ( cellH * rows ) + pRow;

        //If a non colorkey pixel is found
        if( bitmap->getPixel32( pX, pY ) != bgColor )
                    {
                        //Bottom of a is found
                        baseA = pRow;

                        //Break the loops
                        pCol = cellW;
                        pRow = -1;
                    }
                }
            }
        }
```

Para buscar la parte inferior de los glyphs, sólo importa la A. Para esta fuente de mapa de bits, se usará la parte inferior de A del glyph sprite como línea base para los caracteres como “g, j, y” que están por debajo de la línea base.

```
    //Calculate space
    mSpace = cellW / 2;

    //Calculate new line
    mNewLine = baseA - top;
```

```
            //Lop off excess top pixels
            for( int i = 0; i < 256; ++i )
            {
                  mChars[ i ].y += top;
                  mChars[ i ].h -= top;
            }

            bitmap->unlockTexture();
            mBitmap = bitmap;
      }

      return success;
}
```

De esta manera se termina la definición de todos los sprites.

- En primer lugar, se calcula el tamaño de un espacio (se está definiendo como la mitad de ancho de una celda) y a continuación se calcula la altura de una nueva línea empleando la línea base y la parte superior más alta del sprite.

- Seguidamente se recorta el espacio extra en la parte superior de cada glyph para evitar que haya demasiado espacio entre líneas.

- Finalmente, se desbloquea la textura y se fija el mapa de bits para la fuente.

```
void LBitmapFont::renderText( int x, int y, std::string text )
{
    //If the font has been built
    if( mBitmap != NULL )
    {
            //Temp offsets
            int curX = x, curY = y;
```

En primer lugar se verifica que hay un mapa de bits que renderizar y se declaran las compensaciones (X, Y) que se usarán para renderizar el actual sprite glyph.

```
        //Go through the text
        for( int i = 0; i < text.length(); ++i )
        {
            //If the current character is a space
            if( text[ i ] == ' ' )
            {
                //Move over
                curX += mSpace;
            }
            //If the current character is a newline
            else if( text[ i ] == '\n' )
            {
                //Move down
                curY += mNewLine;
```

```
                //Move back
                curX = x;
            }
```

Bucle for que se mueve a través de la cadena para renderizar cada sprite glyph. Sin embargo, hay dos valores ASCII que no renderizarán:

- Cuando se tiene un espacio, lo único que se hace es moverse por toda la anchura del espacio.

- Cuando se tiene una nueva línea, se realiza movimiento hacia abajo hasta una nueva línea y se vuelve a la compensación base X.

```
            else
            {
        //Get the ASCII value of the character
        int ascii = (unsigned char)text[ i ];

        //Show the character
        mBitmap->render( curX, curY, &mChars[ ascii ] );

        //Move over the width of the character with one pixel of padding
        curX += mChars[ ascii ].w + 1;

            }
```

Cuando se trata de caracteres no especiales, se renderiza el sprite. Es tan sencillo como obtener el valor ASCII, renderizar el sprite asociado a este valor y moverse sobre la anchura del sprite.

El bucle for seguirá moviéndose a través de todos los caracteres y renderizando el sprite para cada uno de ellos, uno detrás de otro.

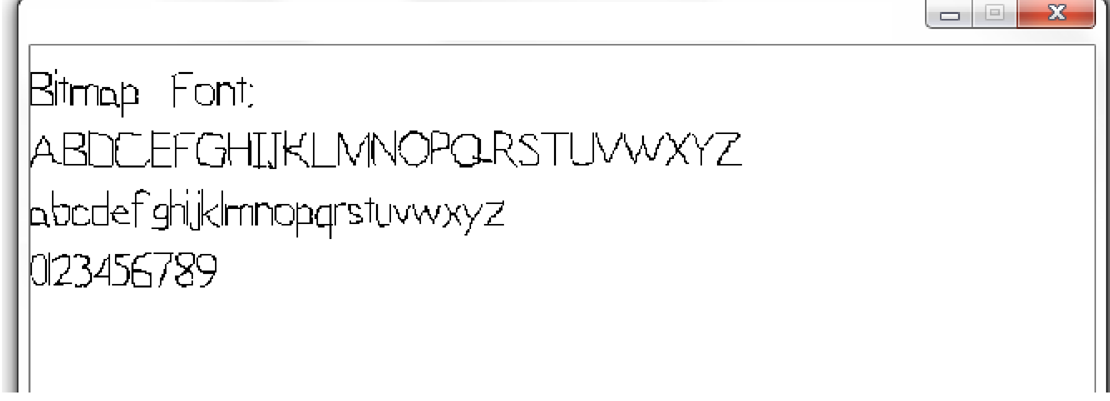


Figura 179: *Resultado ejecutar aplicación Fuentes de mapas de bits*

3.40 Texture Streaming

En determinadas ocasiones se necesita renderizar datos de los pixeles de una fuente que no sea un mapa de bits, como una web cam. Mediante texture stream pueden renderizarse pixeles de cualquier fuente.

```
//Texture wrapper class
class LTexture
{
      public:

            //Initializes variables
            LTexture();

            //Deallocates memory
            ~LTexture();

            //Loads image at specified path
            bool loadFromFile( std::string path );

            #ifdef _SDL_TTF_H
            //Creates image from font string
            bool loadFromRenderedText( std::string textureText,
SDL_Color textColor );
            #endif

            //Creates blank texture
            bool createBlank( int width, int height );

            //Deallocates texture
            void free();

            //Set color modulation
            void setColor( Uint8 red, Uint8 green, Uint8 blue );

            //Set blending
            void setBlendMode( SDL_BlendMode blending );

            //Set alpha modulation
            void setAlpha( Uint8 alpha );

            //Renders texture at given point
            void render( int x, int y, SDL_Rect* clip = NULL, double
angle = 0.0, SDL_Point* center = NULL, SDL_RendererFlip flip =
SDL_FLIP_NONE );

            //Gets image dimensions
            int getWidth();
            int getHeight();

            //Pixel manipulators
            bool lockTexture();
            bool unlockTexture();
            void* getPixels();
            void copyPixels( void* pixels );
            int getPitch();
            Uint32 getPixel32( unsigned int x, unsigned int y );
```

```
	private:
		//The actual hardware texture
		SDL_Texture* mTexture;
		void* mPixels;
		int mPitch;

		//Image dimensions
		int mWidth;
		int mHeight;
};
```

La función **createBlank** libera una textura en blanco para que se puedan copiar los datos cuando se transmiten.

La función **copyPixels** copia los datos de pixeles que se quieren transmitir.

```
//A test animation stream
class DataStream
{
	public:

		//Initializes internals
		DataStream();

		//Loads initial data
		bool loadMedia();

		//Deallocator
		void free();

		//Gets current frame data
		void* getBuffer();

	private:

		//Internal data
		SDL_Surface* mImages[ 4 ];
		int mCurrentImage;
		int mDelayFrames;
};
```

Clase Transmisión de Datos. Mediante **GetBuffer** se obtienen los pixeles actuales de la memoria de datos.

```
bool LTexture::createBlank( int width, int height )
{
	//Create uninitialized texture
mTexture = SDL_CreateTexture( gRenderer,
SDL_PIXELFORMAT_RGBA8888,SDL_TEXTUREACCESS_STREAMING, width, height );

	if( mTexture == NULL )
	{
```

```
printf( "Unable to create blank texture! SDL Error: %s\n",
SDL_GetError() );
        }

        else
        {
                mWidth = width;
                mHeight = height;
        }

        return mTexture != NULL;
}
```

Esta función crea una textura RGBA de 32 bits con acceso de transmisión. Es necesario asegurarse de que el formato de los pixeles de la textura es el mismo que el de los pixeles que se están transmitiendo.

```
void LTexture::copyPixels( void* pixels )
{
        //Texture is locked
        if( mPixels != NULL )
        {
                //Copy to locked pixels
                memcpy( mPixels, pixels, mPitch * mHeight );
        }
}
```

Función para copiar los pixeles de la transmisión. Esta función asume que la textura está bloqueada y que los pixeles son de una imagen del mismo tamaño que la textura.

```
                        //While application is running
                        while( !quit )
                        {
                                //Handle events on queue
                                while( SDL_PollEvent( &e ) != 0 )
                                {
                                        //User requests quit
                                        if( e.type == SDL_QUIT )

                                        {
                                                quit = true;
                                        }
                                }

                                //Clear screen
        SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
        SDL_RenderClear( gRenderer );

                //Copy frame from buffer
                gStreamingTexture.lockTexture();
                gStreamingTexture.copyPixels( gDataStream.getBuffer() );
                gStreamingTexture.unlockTexture();
```

```
//Render frame
gStreamingTexture.render((SCREEN_WIDTH-
gStreamingTexture.getWidth())/2,( SCREEN_HEIGHT -
gStreamingTexture.getHeight() ) / 2 );

//Update screen
SDL_RenderPresent( gRenderer );

                  }
            }
      }
```

En el bucle principal se bloquea la textura de transmisión, se copian los pixeles de la transmisión y, a continuación, se desbloquea la textura. Se renderiza la última imagen de la transmisión almacenada en la textura.

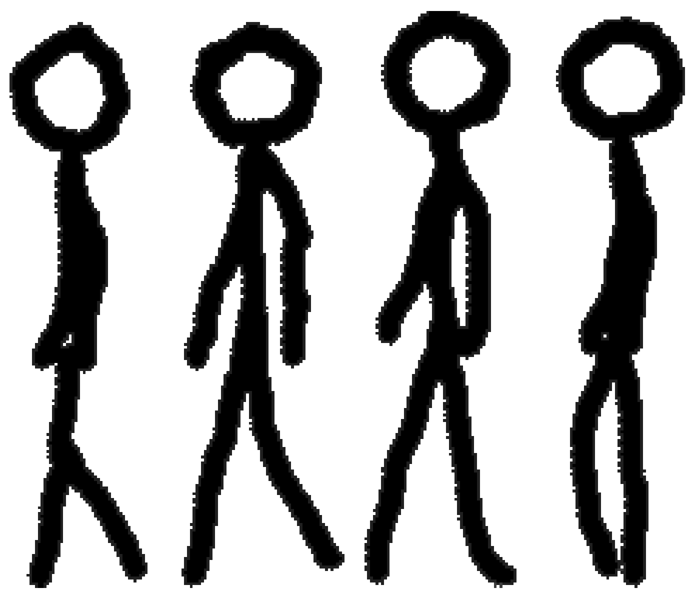


Figura 179: *Resultado ejecutar aplicación Texture Streaming*

3.41 Renderizar a una textura

Para determinados efectos es necesario poder renderizar una textura a una escena. El objetivo será renderizar una escena a una textura para lograr el efecto de escena girada.

```
//Texture wrapper class
class LTexture
{
	public:

		//Initializes variables
		LTexture();

		//Deallocates memory
		~LTexture();

		//Loads image at specified path
		bool loadFromFile( std::string path );

#ifdef _SDL_TTF_H
//Creates image from font string
bool loadFromRenderedText( std::string textureText,
SDL_Color textColor );
#endif

//Creates blank texture
bool createBlank( int width, int height,SDL_TextureAccess =
SDL_TEXTUREACCESS_STREAMING );

		//Deallocates texture
		void free();

		//Set color modulation
		void setColor( Uint8 red, Uint8 green, Uint8 blue );

		//Set blending
		void setBlendMode( SDL_BlendMode blending );

		//Set alpha modulation
		void setAlpha( Uint8 alpha );

//Renders texture at given point
void render( int x, int y, SDL_Rect* clip = NULL, double angle = 0.0,
SDL_Point* center = NULL,SDL_RendererFlip flip = SDL_FLIP_NONE );

		//Set self as render target
		void setAsRenderTarget();

		//Gets image dimensions
		int getWidth();
		int getHeight();

		//Pixel manipulators
		bool lockTexture();
		bool unlockTexture();
		void* getPixels();
		void copyPixels( void* pixels );
```

```
        int getPitch();
        Uint32 getPixel32( unsigned int x, unsigned int y );

    private:

        //The actual hardware texture
        SDL_Texture* mTexture;
        void* mPixels;
        int mPitch;

        //Image dimensions
        int mWidth;
        int mHeight;
};
```

CreateBlanck toma un argumento que define la forma en que se accede.

SetAsRenderTarget hace que se pueda renderizar a la textura.

```
bool LTexture::createBlank( int width, int height, SDL_TextureAccess
access )
{
    //Create uninitialized texture
mTexture = SDL_CreateTexture( gRenderer, SDL_PIXELFORMAT_RGBA8888,
access, width, height );

    if( mTexture == NULL )
    {
printf( "Unable to create blank texture! SDL Error: %s\n",
SDL_GetError() );
    }

    else
    {
        mWidth = width;
        mHeight = height;
    }

    return mTexture != NULL;
}
```

Para renderizar a una textura es necesario fijar su acceso a **SDL_TEXTUREACCESS_TARGET**, por esta razón la función tiene un argumento adicional.

```
void LTexture::setAsRenderTarget()
{
    //Make self render target
    SDL_SetRenderTarget( gRenderer, mTexture );
}
```

Para renderizar a una textura se establece como objetivo llamando a **SDL_SetRenderTarget**.

```
bool loadMedia()
{
	//Loading success flag
	bool success = true;

	//Load texture target
	if( !gTargetTexture.createBlank( SCREEN_WIDTH,
	  SCREEN_HEIGHT, SDL_TEXTUREACCESS_TARGET ) )
	{
		printf( "Failed to create target texture!\n" );
		success = false;
	}

	return success;
}
```

La textura objetivo se crea en la función de carga.

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//Rotation variables
double angle = 0;
SDL_Point screenCenter = { SCREEN_WIDTH / 2,
SCREEN_HEIGHT / 2 };
```

En este caso se renderiza cierta geometría a la textura y el giro de la textura en torno al centro de la pantalla. Por ello, se cuenta con variables para el ángulo de giro y el centro de la pantalla.

```
			//While application is running
			while( quit == false )
			{
				//Handle events on queue
				while( SDL_PollEvent( &e ) != 0 )
				{
					//User requests quit
					if( e.type == SDL_QUIT )
					{
						quit = true;
					}
				}

				//rotate

				angle += 2;

				if( angle > 360 )
				{
					angle -= 360;
				}
```

```
//Set self as render target
gTargetTexture.setAsRenderTarget();

//Clear screen
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render red filled quad

SDL_Rect fillRect = { SCREEN_WIDTH / 4, SCREEN_HEIGHT / 4,
SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2 };

SDL_SetRenderDrawColor( gRenderer, 0xFF, 0x00, 0x00, 0xFF );
SDL_RenderFillRect( gRenderer, &fillRect );

//Render green outlined quad
SDL_Rect outlineRect = { SCREEN_WIDTH / 6, SCREEN_HEIGHT / 6,
SCREEN_WIDTH * 2 / 3, SCREEN_HEIGHT * 2 / 3 };

SDL_SetRenderDrawColor( gRenderer, 0x00, 0xFF, 0x00, 0xFF );

SDL_RenderDrawRect( gRenderer, &outlineRect );

//Draw blue horizontal line

SDL_SetRenderDrawColor( gRenderer, 0x00, 0x00, 0xFF, 0xFF );

SDL_RenderDrawLine( gRenderer, 0, SCREEN_HEIGHT / 2,
SCREEN_WIDTH, SCREEN_HEIGHT / 2 );

//Draw vertical line of yellow dots

SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0x00, 0xFF );

for( int i = 0; i < SCREEN_HEIGHT; i += 4 )
                        {
            SDL_RenderDrawPoint( gRenderer, SCREEN_WIDTH / 2, i );
                        }

//Reset render target
SDL_SetRenderTarget( gRenderer, NULL );

//Show rendered to texture
gTargetTexture.render( 0, 0, NULL, angle, &screenCenter );

//Update screen
SDL_RenderPresent( gRenderer );

                  }
            }
```

En el bucle principal, antes de renderizar se fija la textura objetivo como objetivo. A continuación se renderiza la escena con toda la geometría y, una vez esto está hecho, se renderiza a la textura. Dicha textura se denomina **SDL_SetRenderTarget** con una textura nula, por lo que cualquier renderización hecha tras ella se hará a la pantalla.

Con la escena renderizada a la textura, se renderiza la textura objetivo con el ángulo de giro.

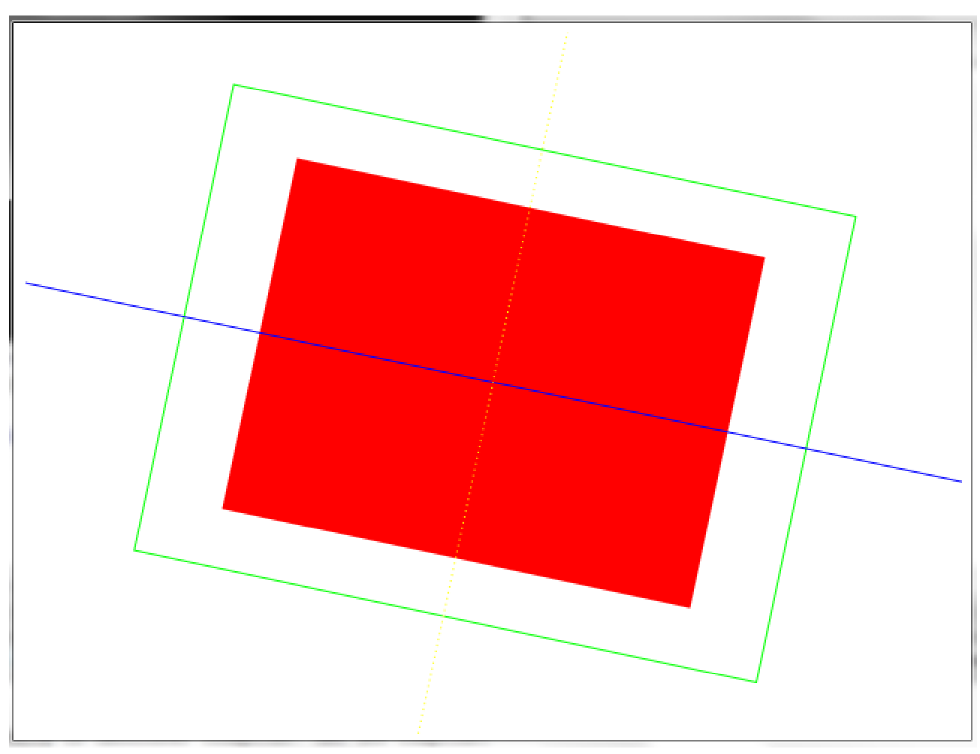


Figura 180: *Resultado ejecutar aplicación Renderizar a textura*

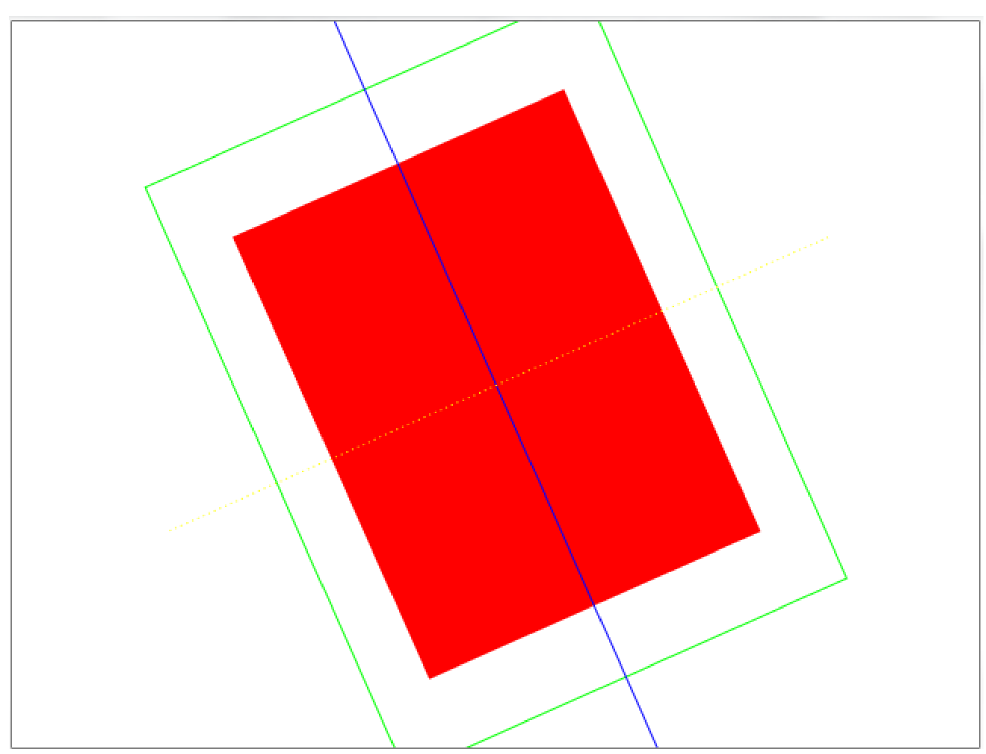


Figura 181: *Resultado ejecutar aplicación Renderizar a textura (II)*

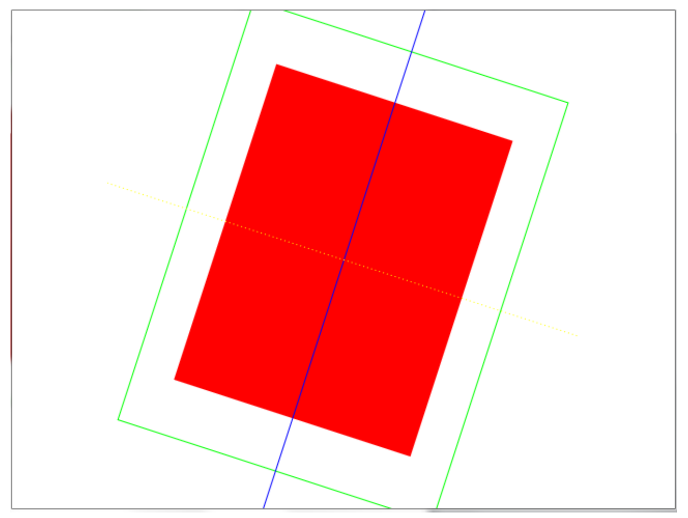


Figura 182: *Resultado ejecutar aplicación Renderizar a textura (III)*

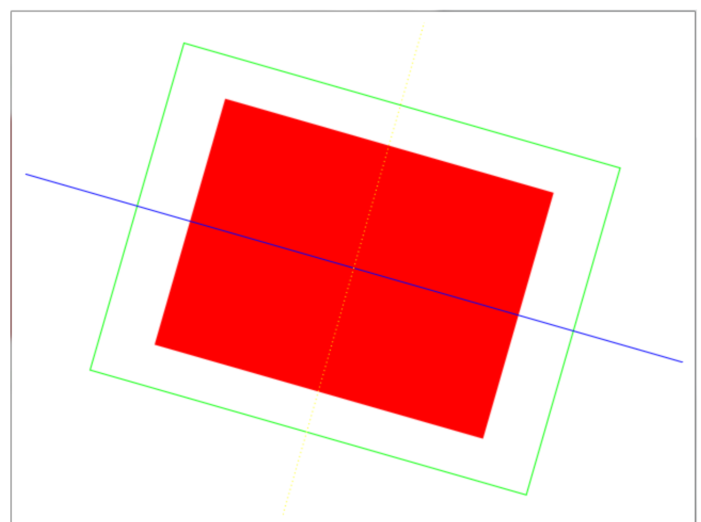


Figura 183: *Resultado ejecutar aplicación Renderizar a textura (IV)*

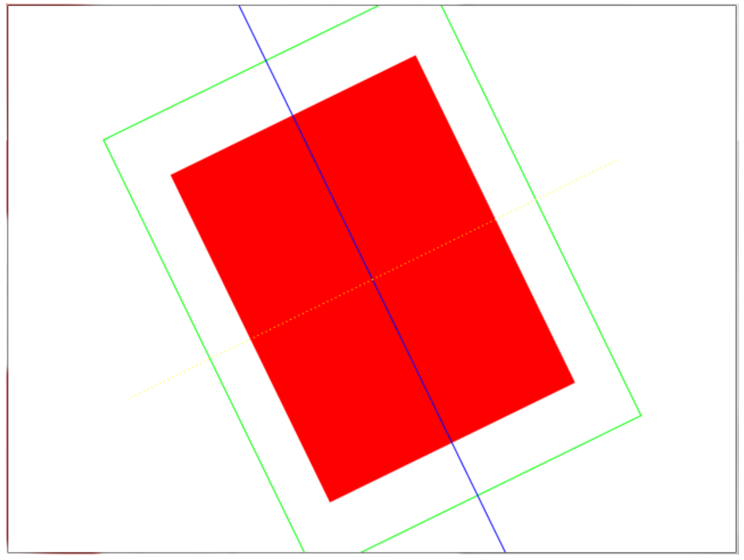


Figura 184: *Resultado ejecutar aplicación Renderizar a textura (V)*

3.42 Movimiento independiente de la velocidad de los fotogramas

Para controlar velocidades variables de fotogramas o soportar múltiples frecuencias de fotograma, puede fijarse el movimiento basándose en el tiempo para que sea independiente de la velocidad de los fotogramas.

```
//The dot that will move around on the screen
class Dot
{
    public:

            //The dimensions of the dot
            static const int DOT_WIDTH = 20;
            static const int DOT_HEIGHT = 20;

            //Maximum axis velocity of the dot
            static const int DOT_VEL = 640;

            //Initializes the variables
            Dot();

            //Takes key presses and adjusts the dot's velocity
            void handleEvent( SDL_Event& e );

            //Moves the dot
            void move( float timeStep );

            //Shows the dot on the screen
            void render();

    private:

            float mPosX, mPosY;
            float mVelX, mVelY;
};
```

En este programa todo se basará en el tiempo. Ahora la velocidad es 640: 640 pixeles por segundo se traducen en 10 pixeles por imagen, es decir, 60 fotogramas por segundo.

Con el fin de avanzar basándose en el tiempo de fotograma, la función movimiento necesita saber cuánto tiempo se está moviendo por fotograma. Por esta razón, la función toma un intervalo de tiempo, que es el tiempo pasado desde la última actualización.

Además, la posición y la velocidad son floats en lugar de enteros. Si se usaran enteros el movimiento se pararía siempre en el entero más cercano y esto podría causar mayores imprecisiones.

```
void Dot::move( float timeStep )
{
    //Move the dot left or right
    mPosX += mVelX * timeStep;
```

```
    //If the dot went too far to the left or right
      if( mPosX < 0 )
      {
            mPosX = 0;
      }

      else if( mPosX > SCREEN_WIDTH - DOT_WIDTH )
      {
            mPosX = SCREEN_WIDTH - DOT_WIDTH;
      }

    //Move the dot up or down
    mPosY += mVelY * timeStep;

    //If the dot went too far up or down
      if( mPosY < 0 )
      {
            mPosY = 0;
      }

      else if( mPosY > SCREEN_HEIGHT - DOT_HEIGHT )
      {
            mPosY = SCREEN_HEIGHT - DOT_HEIGHT;
      }
}
```

Se actualiza la posición moviéndola sobre la velocidad * tiempo. Si se tuviera una velocidad de 600 pixeles por segundo y un intervalo de tiempo de 1/60 segundos, se moverían $600 * (1/60)$ pixeles, es decir, 10 pixeles.

Debido al movimiento no uniforme, no puede retrocederse cuando se mueve el punto fuera de la pantalla, sino que hay que corregir el valor.

```
void Dot::render()
{
    //Show the dot
      gDotTexture.render( (int)mPosX, (int)mPosY );
}
```

Para evitar problemas con el compilador, se convierten las posiciones en enteros cuando se renderiza el punto.

```
                  //Main loop flag
                  bool quit = false;

                  //Event handler
                  SDL_Event e;

                  //The dot that will be moving around on the screen
                  Dot dot;

                  //Keeps track of time between steps
                  LTimer stepTimer;
```

Se deshabilita la sincronización vertical para mostrar que puede ejecutarse independientemente de la velocidad de fotogramas. Para saber cuánto tiempo ha pasado entre renderizados, se emplea un temporizador que hace un seguimiento del tiempo que transcurre.

```
            //While application is running
            while( !quit )
            {
                //Handle events on queue
                while( SDL_PollEvent( &e ) != 0 )
                {
                    //User requests quit
                    if( e.type == SDL_QUIT )
                    {
                        quit = true;
                    }

                    //Handle input for the dot
                    dot.handleEvent( e );
                }

            //Calculate time step
            float timeStep = stepTimer.getTicks() / 1000.f;

            //Move for time step
            dot.move( timeStep );

            //Restart step timer
            stepTimer.start();

            //Clear screen
    SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
    SDL_RenderClear( gRenderer );

                //Render dot
                dot.render();

                //Update screen
                SDL_RenderPresent( gRenderer );
            }
        }
    }
```

Cuando se mueve el punto se obtiene, en primer lugar, el tiempo del intervalo de tiempo para conocer cuánto tiempo ha pasado desde el último movimiento. A continuación, se convierte a milisegundos y se pasa a la función de desplazamiento.

Una vez se ha hecho el movimiento se reinicia el temporizador, de este modo será posible saber el tiempo transcurrido cuando haya un nuevo movimiento. Finalmente se renderiza.

Figura 184: *Resultado ejecutar aplicación Movimiento independiente de la velocidad de los fotogramas*

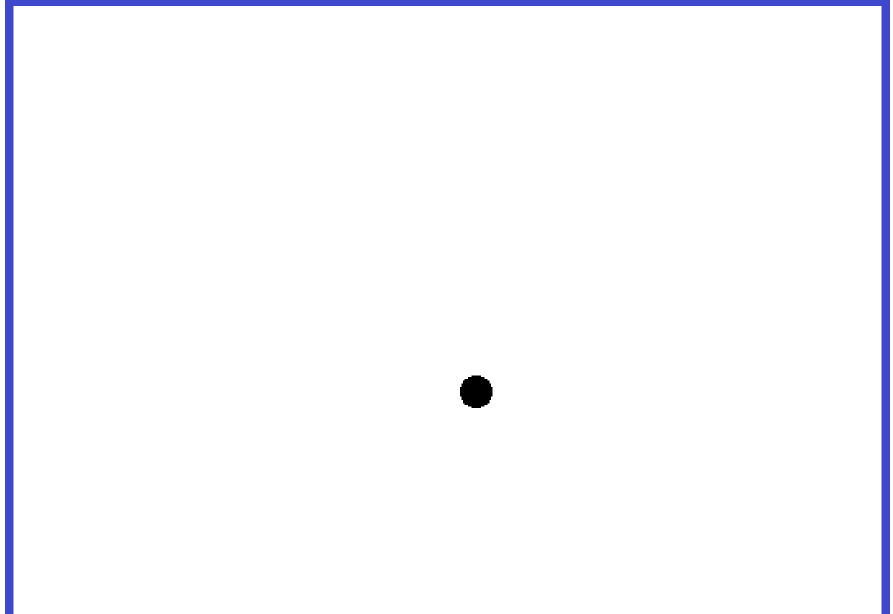


Figura 185: *Resultado ejecutar aplicación Movimiento independiente de la velocidad de los fotogramas (II)*

3.43 Retrollamadas temporizadas

Existen retrollamadas temportizadas que ejecutan una función después de pasar una cierta cantidad de tiempo. El objetivo será realizar un programa sencillo que imprima en la pantalla después de un tiempo establecido.

```
//Our test callback function
Uint32 callback( Uint32 interval, void* param );
```

Cuando se crea una función de retrollamada, ésta tiene que ser declarada de una manera concreta. No puede crearse cualquier tipo de función y usarla como retrollamada.

Estas funciones necesitan tener un entero de 32 bits en su primer argumento, un puntero vacío en el segundo y, además, deben devolver un entero de 32 bits.

```
Uint32 callback( Uint32 interval, void* param )
{
//Print callback message
printf( "Callback called back with message: %s\n", (char*)param );

return 0;
}
```

Función de retrollamada sencilla que imprime un mensaje en la consola después de un periodo de tiempo determinado. No se utilizará el argumento intervalo, aunque suele utilizarse para retrollamadas temporizadas que deben repetirse.

Dado que un puntero vacío puede apuntar a cualquier elemento, esta función tomará una cadena y la imprimirá en la consola.

```
      //Initialize SDL
      if (SDL_Init( SDL_INIT_VIDEO | SDL_INIT_TIMER ) < 0 )

      {
printf( "SDL could not initialize! SDL Error: %s\n", SDL_GetError() );
success = false;
      }
```

Se inicializa con **SDL_INIT_TIMER** para usar retrollamadas temporizadas.

```
//Set callback
SDL_TimerID timerID = SDL_AddTimer( 3 * 1000, callback, "3 seconds
waited!" );
```

```
//While application is running
                  while( !quit )
                  {
                        //Handle events on queue
                        while( SDL_PollEvent( &e ) != 0 )
                        {
                              //User requests quit
                              if( e.type == SDL_QUIT )
                              {
                                    quit = true;
                              }
                        }

//Clear screen
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render splash
gSplashTexture.render( 0, 0 );

//Update screen
SDL_RenderPresent( gRenderer );

                  }

//Remove timer in case the call back was not called
SDL_RemoveTimer( timerID );
```

La retrollamada alimentada se inicia mediante **SDL_AddTimer**.

- El primer argumento es el tiempo que la retrollamada empleará, que en este caso se establece en 3000 milisegundos (3segundos).

- El segundo argumento corresponde con la función retrollamada.

- El último argumento es el puntero de datos vacío que se está enviando.

Esta aplicación iniciará la retrollamada y a continuación se ejecuta el bucle principal. Mientras éste se está ejecutando, la retrollamada puede mandar el mensaje a la consola. En el caso de que la retrollamada no ocurra antes de que finalice el bucle principal, se eliminará empleando **SDL_RemoveTimer**. (La retrollamada temporizada es asíncrona, es decir, puede ocurrir mientras se está realizando otra tarea).

Wait for 3
seconds and
check the
console

__Figura 186:__ Resultado ejecutar aplicación Retrollamadas temporizadas

__Figura 187:__ Resultado ejecutar aplicación Retrollamadas temporizadas (II)

3.44 Multithreading

Multithreading puede utilizarse para que el programa sea capaz de ejecutar dos tareas a la vez y aprovechar las ventajas de la arquitectura "multihilo". El objetivo es realizar un programa sencillo que dé salida a la consola mientras el hilo principal se ejecuta.

```
//Using SDL, SDL Threads, SDL_image, standard IO, and, strings
#include <SDL.h>
#include <SDL_thread.h>
#include <SDL_image.h>
#include <stdio.h>
#include <string>
#include <cstring>
#include <cstdlib>
```

Cuando quieren emplearse hilos, es necesario incluir la cabecera **SDL threads**.

```
//Our test thread function
int threadFunction( void* data );
```

Al igual que con las funciones de retrollamada, las funciones de hilo se declaran de una manera concreta. Tienen que tomar un puntero nulo como argumento y devolver un número entero.

```
int threadFunction( void* data )
{
	//Print incoming data
	printf( "Running thread with value = %d\n", (int)data );

	return 0;
}
```

La función de hilo es sencilla. Todo lo que hace es tomar los datos como enteros y usarlos para imprimir un mensaje en la consola.

```
//Run the thread
int data = 101;
SDL_Thread* threadID = SDL_CreateThread( threadFunction,
"LazyThread", (void*)data );

			//While application is running
			while( !quit )
			{
				//Handle events on queue
				while( SDL_PollEvent( &e ) != 0 )
				{
					//User requests quit
					if( e.type == SDL_QUIT )
```

```
                    {
                            quit = true;
                    }
            }

//Clear screen
SDL_SetRenderDrawColor( gRenderer, 0xFF, 0xFF, 0xFF, 0xFF );
SDL_RenderClear( gRenderer );

//Render prompt
gSplashTexture.render( 0, 0 );

//Update screen
SDL_RenderPresent( gRenderer );

            }

            //Remove timer in case the call back was not called
            SDL_WaitThread( threadID, NULL );
```

Antes de entrar al bucle principal se ejecuta la función de hilo mediante **SDL_CreateThread**. Esta llamada ejecutará la función en el primer argumento, le dará el nombre en el segundo (los nombres se emplean para identificarlas para fines de depuración) y pasará los datos al tercer argumento.

A continuación, el hilo se ejecutará mientras el hilo principal todavía está en curso. En el caso de que el bucle principal termine antes que el hilo, se llama a **SDL_WaitThread** para asegurarse de que el hilo termina antes de que se cierre la aplicación.

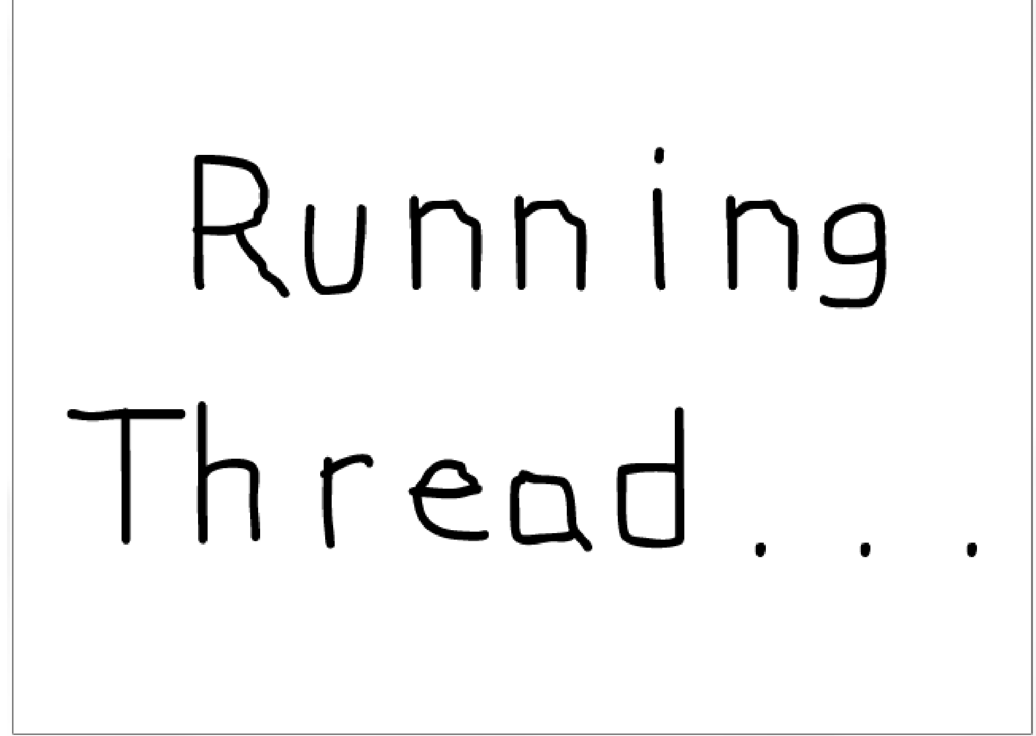


Figura 188: *Ejecución Multihilo*

3.45 Semáforos

El multithreading anterior estaba compuesto por el hilo principal y un segundo hilo, cada uno realizando su tarea. En la mayoría de los casos, los dos hilos tendrán que compartir datos y mediante los semáforos puede evitarse que ambos hilos accedan de forma accidental a los mismos datos a la vez.

```
//Our worker thread function
int worker( void* data );
```

Función de hilos. Se separarán dos hilos y cada uno de ellos ejecutará su copia de este código.

```
//Data access semaphore
SDL_sem* gDataLock = NULL;

//The "data buffer"
int gData = -1;
```

gDataLock es el semáforo que bloqueará la memoria **gData**. Un único entero no representa gran cantidad de datos de memoria que proteger, pero dado que se emplearán dos hilos para lectura y escritura, es necesario asegurarse de que sólo se accede con un hilo cada vez.

```
bool loadMedia()
{
	//Initialize semaphore
	gDataLock = SDL_CreateSemaphore( 1 );

	//Loading success flag
	bool success = true;

	//Load splash texture
	if( !gSplashTexture.loadFromFile( "47_semaphores/splash.png" ) )
	{
		printf( "Failed to load splash texture!\n" );
		success = false;
	}

	return success;
}
```

Para crear un semáforo se llama a **SDL_CreateSemaphore** con un valor inicial para el semáforo. Este valor inicial controla el número de veces que el código puede pasar a través del semáforo antes de que éste se bloquee.

En este caso sólo se quiere tener 1 hilo que acceda a la memoria de datos a la vez, por lo que el mutex se inicia con valor 1.

```
void close()
{
	//Free loaded images
	gSplashTexture.free();

	//Free semaphore
	SDL_DestroySemaphore( gDataLock );
	gDataLock = NULL;

	//Destroy window
	SDL_DestroyRenderer( gRenderer );
	SDL_DestroyWindow( gWindow );
	gWindow = NULL;
	gRenderer = NULL;

	//Quit SDL subsystems
	IMG_Quit();
	SDL_Quit();
}
```

Una vez se ha terminado con el semáforo se llama a **SDL_DestroySemaphore**.

```
int worker( void* data )
{
	printf( "%s starting...\n", data );

	//Pre thread random seeding
	srand( SDL_GetTicks() );
```

Inicio del hilo. El valor inicial del valor aleatorio se hace por hilo, por lo que hay que asegurarse de que se asignan los valores iniciales de los valores aleatorios para cada hilo que se ejecuta.

```
	//Work 5 times
	for( int i = 0; i < 5; ++i )
	{
		//Wait randomly
		SDL_Delay( 16 + rand() % 32 );

		//Lock
		SDL_SemWait( gDataLock );

		//Print pre work data
		printf( "%s gets %d\n", data, gData );

		//"Work"
		gData = rand() % 256;

		//Print post work data
		printf( "%s sets %d\n\n", data, gData );

		//Unlock
		SDL_SemPost( gDataLock );
```

```
            //Wait randomly
            SDL_Delay( 16 + rand() % 640 );
      }

      printf( "%s finished!\n\n", data );

      return 0;
}
```

Cada hilo retrasa una cantidad semialeatoria, imprime los datos que hay cuando empezó a trabajar asignados a la memoria de datos y retrasa un poco más antes de empezar a funcionar de nuevo. La razón por la que hay que bloquear los datos es que no se quieren dos hilos leyendo o escribiendo los datos compartidos al mismo tiempo.

Entre la función **SDL_SemWait** y **SDL_SemPost** hay una sección crítica de código a la que sólo puede acceder un hilo a la vez.

SDL_SemWait disminuye la cuenta del semáforo y dado que el valor inicial es uno, se bloqueará. Después de que la sección crítica se ejecute se llama a **SDL_SemPost** para incrementar el semáforo y desbloquearlo. (Si se tuviera una situación en la que el hilo A bloquea y el hilo B trata de bloquear también, el hilo B esperaría hasta que el A finalice la sección crítica y desbloquee el semáforo.)

```
//Main loop flag
bool quit = false;

//Event handler
SDL_Event e;

//Run the threads

srand( SDL_GetTicks() );
SDL_Thread* threadA = SDL_CreateThread( worker, "Thread A",
(void*)"Thread A" );
SDL_Delay( 16 + rand() % 32 );
SDL_Thread* threadB = SDL_CreateThread( worker, "Thread B",
(void*)"Thread B" );
```

En la función principal, antes de entrar al bucle principal, se ejecutan dos hilos con un pequeño retraso aleatorio entre ellos. No hay garantía de qué hilo trabajará primero, pero como los datos compartidos están protegidos, los hilos no ejecutarán la misma parte del código a la vez.

```
                  //While application is running
                  while( !quit )
                  {
                        //Handle events on queue
                        while( SDL_PollEvent( &e ) != 0 )
                        {
```

```
                        //User requests quit
                        if( e.type == SDL_QUIT )
                        {
                              quit = true;
                        }
                  }

                  //Clear screen

SDL_SetRenderDrawColor(gRenderer,0xFF,0xFF,0xFF,0xFF);
SDL_RenderClear( gRenderer );

                  //Render splash
                  gSplashTexture.render( 0, 0 );

                  //Update screen
                  SDL_RenderPresent( gRenderer );
            }

            //Wait for threads to finish
            SDL_WaitThread( threadA, NULL );
            SDL_WaitThread( threadB, NULL );
      }
```

El hilo principal se ejecuta mientras que los hilos realizan tus tareas. Si el bucle principal termina antes de que los hilos hayan terminado sus tareas, se espera hasta que terminan con **SDL_WaitThread**.

Figura 189: *Resultado ejecutar aplicación Semáforos*

```
Thread A starting...
Thread A gets -1
Thread A sets 88

Thread B starting...
Thread B gets 88
Thread B sets 204

Thread B gets 204
Thread B sets 60

Thread A gets 60
Thread A sets 223

Thread B gets 223
Thread B sets 89

Thread A gets 89
Thread A sets 77

Thread B gets 77
Thread B sets 30

Thread A gets 30
Thread A sets 167

Thread A gets 167
Thread A sets 161

Thread B gets 161
Thread B sets 92

Thread A finished!

Thread B finished!
```

Figura 190: *Resultado ejecutar aplicación Semáforos (II)*

3.46 Atomic Operations

Las operaciones atómicas son una forma de bloquear los datos a nivel de CPU eficiente. El objetivo es bloquear la sección crítica utilizando GPU spinlocks.

```
//Data access spin lock
SDL_SpinLock gDataLock = 0;

//The "data buffer"
int gData = -1;
```

Se empleará un spinlock para proteger la memoria de datos.

```
bool loadMedia()
{
	//Loading success flag
	bool success = true;

	//Load splash texture
if( !gSplashTexture.loadFromFile( "48_atomic_operations/splash.png" )
)
	{
		printf( "Failed to load splash texture!\n" );
		success = false;
	}

	return success;
}

void close()
{
	//Free loaded images
	gSplashTexture.free();

	//Destroy window
	SDL_DestroyRenderer( gRenderer );
	SDL_DestroyWindow( gWindow );
	gWindow = NULL;
	gRenderer = NULL;

	//Quit SDL subsystems
	IMG_Quit();
	SDL_Quit();
}
```

A diferencia de los semáforos, los spin locks no necesitan ser asignados ni desasignados.

```
int worker( void* data )
{
	printf( "%s starting...\n", data );
```

```
    //Pre thread random seeding
    srand( SDL_GetTicks() );

        //Work 5 times
    for( int i = 0; i < 5; ++i )
    {
        //Wait randomly
        SDL_Delay( 16 + rand() % 32 );

        //Lock
        SDL_AtomicLock( &gDataLock );

        //Print pre work data
        printf( "%s gets %d\n", data, gData );

        //"Work"
        gData = rand() % 256;

        //Print post work data
        printf( "%s sets %d\n\n", data, gData );

        //Unlock
        SDL_AtomicUnlock( &gDataLock );

        //Wait randomly
        SDL_Delay( 16 + rand() % 640 );
    }
    printf( "%s finished!\n\n", data );

    return 0;
}
```

La sección crítica está protegida por **SDL_AtomicLock** y **SDL_AtomicUnlock**.

A diferencia de las operaciones atómicas, los semáforos pueden permitir el acceso a más de un hilo mientras que las primeras son para cuando se quiere un estricto estado de bloqueo/desbloqueo.

Running

Workers...

Figura 191: Resultado ejecutar Atomic operations

```
Thread A starting...
Thread A gets -1
Thread A sets 110

Thread B starting...
Thread B gets 110
Thread B sets 52

Thread A gets 52
Thread A sets 26

Thread A gets 26
Thread A sets 202

Thread B gets 202
Thread B sets 195

Thread A gets 195
Thread A sets 199

Thread A gets 199
Thread A sets 28

Thread B gets 28
Thread B sets 214

Thread A finished!

Thread B gets 214
Thread B sets 137

Thread B gets 137
Thread B sets 65

Thread B finished!
```

Figura 192: Resultado ejecutar Atomic operations (II)

3.47 Exclusiones mutuas y Condiciones

```
//Our worker functions
int producer( void* data );
int consumer( void* data );
void produce();
void consume();
```

En este programa se tienen dos hilos: un productor que llena la memoria y un consumidor que la vacía. No sólo pueden los dos hilos no usar la misma memoria al mismo tiempo, sino que el consumidor no puede leer desde una memoria vacía y el productor no puede llenar una memoria que ya está completa.

Se utilizará mutex (exclusión mutua) para evitar que los dos hilos tomen los mismos datos y condiciones para permitir que los hilos sepan cuándo pueden consumir y producir.

```
//The protective mutex
SDL_mutex* gBufferLock = NULL;

//The conditions
SDL_cond* gCanProduce = NULL;
SDL_cond* gCanConsume = NULL;

//The "data buffer"
int gData = -1;
```

Se declaran globalmente la exclusión mutua y las condiciones que se usarán por los hilos.

```
bool loadMedia()
{
	//Create the mutex
	gBufferLock = SDL_CreateMutex();

	//Create conditions
	gCanProduce = SDL_CreateCond();
	gCanConsume = SDL_CreateCond();

	//Loading success flag
	bool success = true;

	//Load splash texture
if( !gSplashTexture.loadFromFile(
"49_mutexes_and_conditions/splash.png"))
	{
		printf( "Failed to load splash texture!\n" );
		success = false;
	}

	return success;
}
```

Se utiliza **SDL_CreateMutex** y **SDL_CreateCond** para liberar las exclusiones mutuas y condiciones, respectivamente.

```
void close()
{
	//Free loaded images
	gSplashTexture.free();

	//Destroy the mutex
	SDL_DestroyMutex( gBufferLock );
	gBufferLock = NULL;

	//Destroy conditions
	SDL_DestroyCond( gCanProduce );
	SDL_DestroyCond( gCanConsume );
	gCanProduce = NULL;
	gCanConsume = NULL;

	//Destroy window
	SDL_DestroyRenderer( gRenderer );
	SDL_DestroyWindow( gWindow );
	gWindow = NULL;
	gRenderer = NULL;

	//Quit SDL subsystems
	IMG_Quit();
	SDL_Quit();
}
```

Para desasignar exclusiones mutuas y condiciones se utiliza **SDL_DestroyMutex** y **SDL_DestroyCond**.

```
int producer( void *data )
{
	printf( "\nProducer started...\n" );
	//Seed thread random
	srand( SDL_GetTicks() );
		//Produce
	for( int i = 0; i < 5; ++i )
	{
		//Wait
		SDL_Delay( rand() % 1000 );

		//Produce
		produce();
	}

printf( "\nProducer finished!\n" );

return 0;

}
```

```
int consumer( void *data )
{
	printf( "\nConsumer started...\n" );

	//Seed thread random
	srand( SDL_GetTicks() );
	for( int i = 0; i < 5; ++i )
	{
		//Wait
		SDL_Delay( rand() % 1000 );

		//Consume
		consume();
	}

printf( "\nConsumer finished!\n" );

return 0;

}
```

El productor trata de producir 5 veces y el consumidor trata de consumir 5 veces.

```
void produce()
{
	//Lock
	SDL_LockMutex( gBufferLock );

	//If the buffer is full
	if( gData != -1 )
	{
		//Wait for buffer to be cleared
		printf( "\nProducer encountered full buffer, waiting for
consumer to empty buffer...\n" );
		SDL_CondWait( gCanProduce, gBufferLock );
	}

	//Fill and show buffer
	gData = rand() % 255;
	printf( "\nProduced %d\n", gData );

		//Unlock
	SDL_UnlockMutex( gBufferLock );

		//Signal consumer
	SDL_CondSignal( gCanConsume );
}

void consume()
{
	//Lock
	SDL_LockMutex( gBufferLock );

	//If the buffer is empty
	if( gData == -1 )

	{
```

```
            //Wait for buffer to be filled

            printf( "\nConsumer encountered empty buffer, waiting for
producer to fill buffer...\n" );
            SDL_CondWait( gCanConsume, gBufferLock );
      }

      //Show and empty buffer
      printf( "\nConsumed %d\n", gData );
      gData = -1;

      //Unlock
      SDL_UnlockMutex( gBufferLock );

            //Signal producer
      SDL_CondSignal( gCanProduce );
}
```

Funciones que producen y consumen. La producción de memoria quiere decir generar un número aleatorio, mientras que la consumición significa reiniciar el número generado.

- Ejemplo:

Si el productor actúa primero y bloquea la exclusión mutua mediante **SDL_LockMutex** tal y como haría un semáforo con valor de 1:

Figura 193: *Ejemplo exclusiones mutuas y condiciones*

La memoria se vacía, por lo que se produce:

```
SDL_mutexP( gBufferLock );

if( gData != -1 )
{
	SDL_CondWait( gCanProduce, gBufferLock );
}

gData = rand() % 255;

SDL_mutexV( gBufferLock );

SDL_CondSignal( gCanConsume );
```

Consumer

```
SDL_mutexP( gBufferLock );

if( gData == -1 )
{
	SDL_CondWait( gCanConsume, gBufferLock );
}

gData = -1;

SDL_mutexV( gBufferLock );

SDL_CondSignal( gCanProduce );
```

Figura 194: *Ejemplo exclusiones mutuas y condiciones (II)*

Se desbloquea la sección crítica mediante **SDL_UnlockMutex**, por lo que el consumidor puede consumir:

```
SDL_mutexP( gBufferLock );

if( gData != -1 )
{
	SDL_CondWait( gCanProduce, gBufferLock );
}

gData = rand() % 255;

SDL_mutexV( gBufferLock );

SDL_CondSignal( gCanConsume );
```

Consumer

```
SDL_mutexP( gBufferLock );

if( gData == -1 )
{
	SDL_CondWait( gCanConsume, gBufferLock );
}

gData = -1;

SDL_mutexV( gBufferLock );

SDL_CondSignal( gCanProduce );
```

Figura 195: *Ejemplo exclusiones mutuas y condiciones (III)*

Idealmente, sería el consumidor quien consumiera. En el caso de que fuera el productor el que actuara de nuevo:

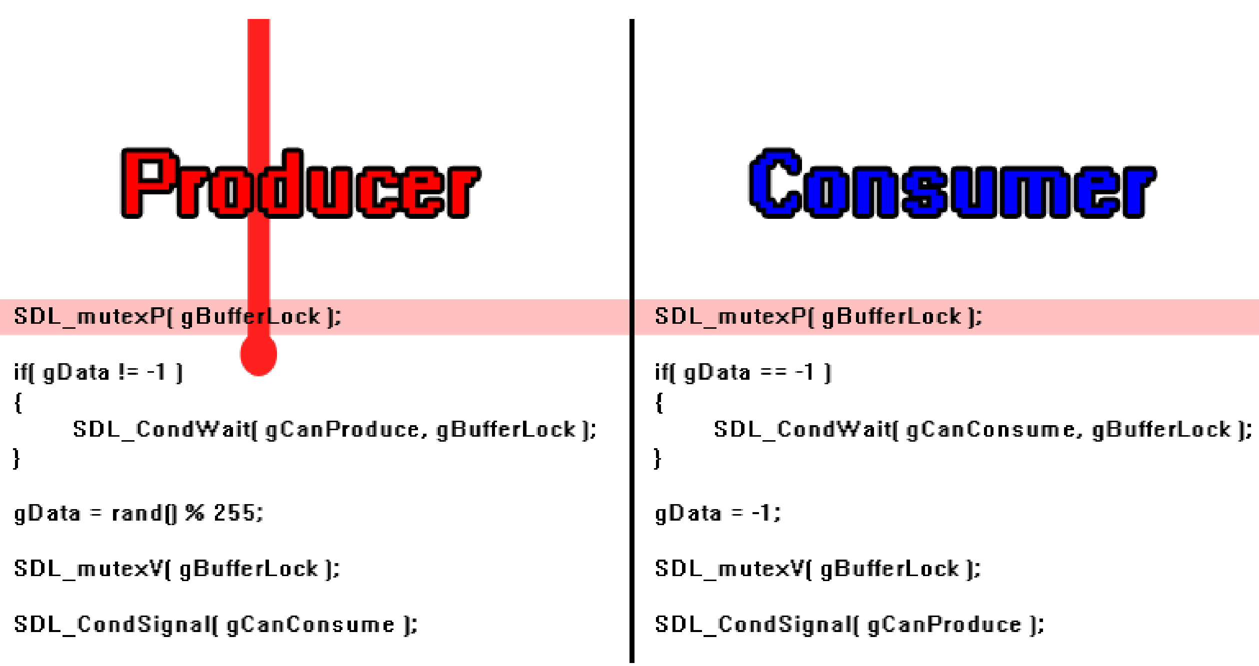


Figura 196: *Ejemplo exclusiones mutuas y condiciones (IV)*

Una vez el productor bloquea la sección crítica, el consumidor trata de consumir, pero la sección crítica ya está bloqueada:

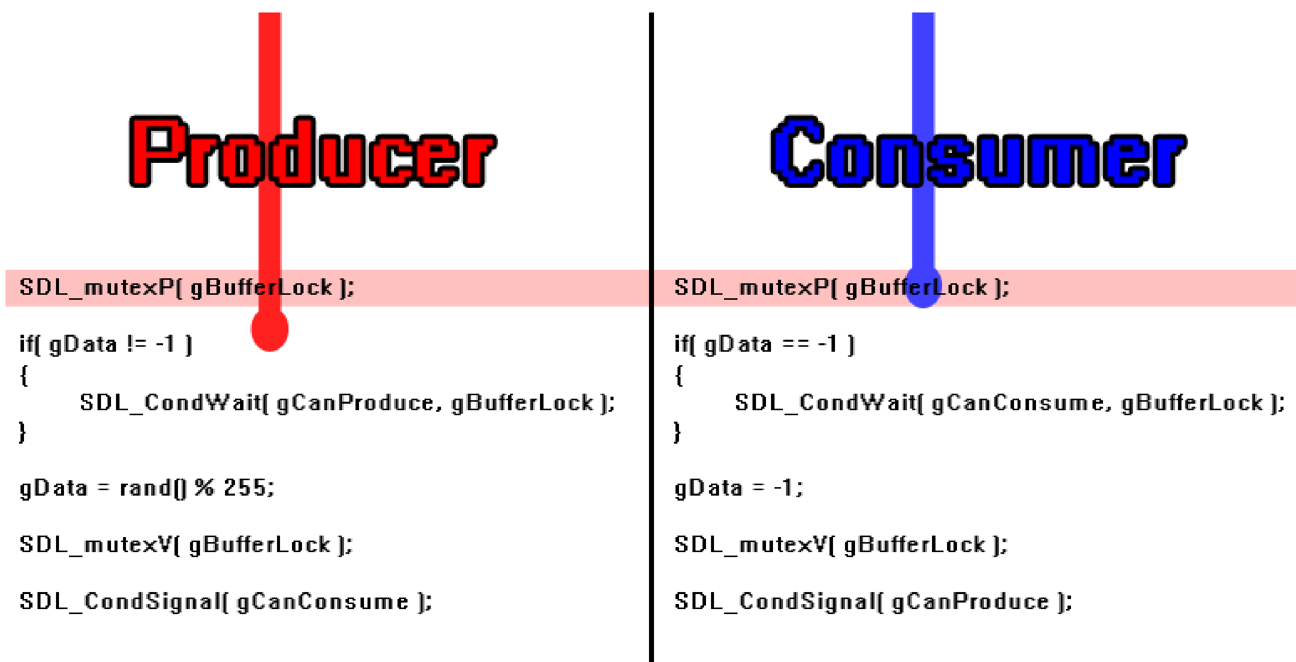


Figura 197: *Ejemplo exclusiones mutuas y condiciones (V)*

Con sólo un semáforo binario esto sería un problema, ya que el productor no puede producir en una memoria llena y el consumidor está bloqueado por la exclusión mutua. Sin embargo, la exclusión mutua tiene la capacidad de ser utilizada con condiciones.

Si la memoria ya está llena, las condiciones permiten esperar mediante **SDL_CondWait** y desbloquear la exclusión mutua para otros hilos:

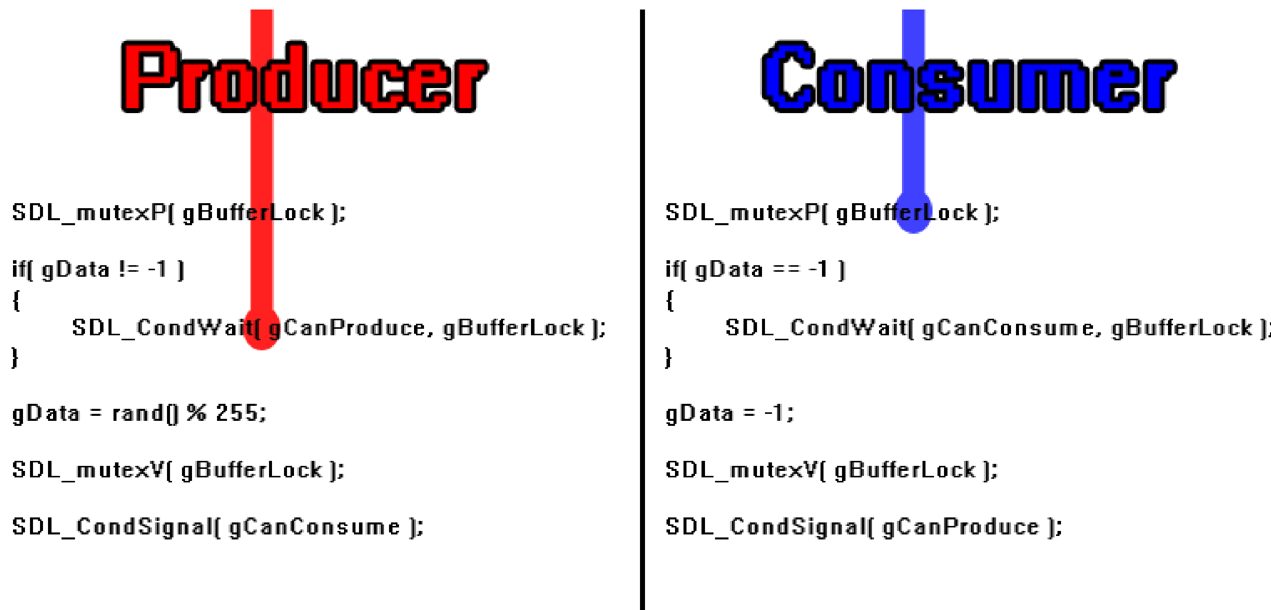


Figura 198: *Ejemplo exclusiones mutuas y condiciones (VI)*

Cuando el consumidor está desbloqueado, puede consumir:

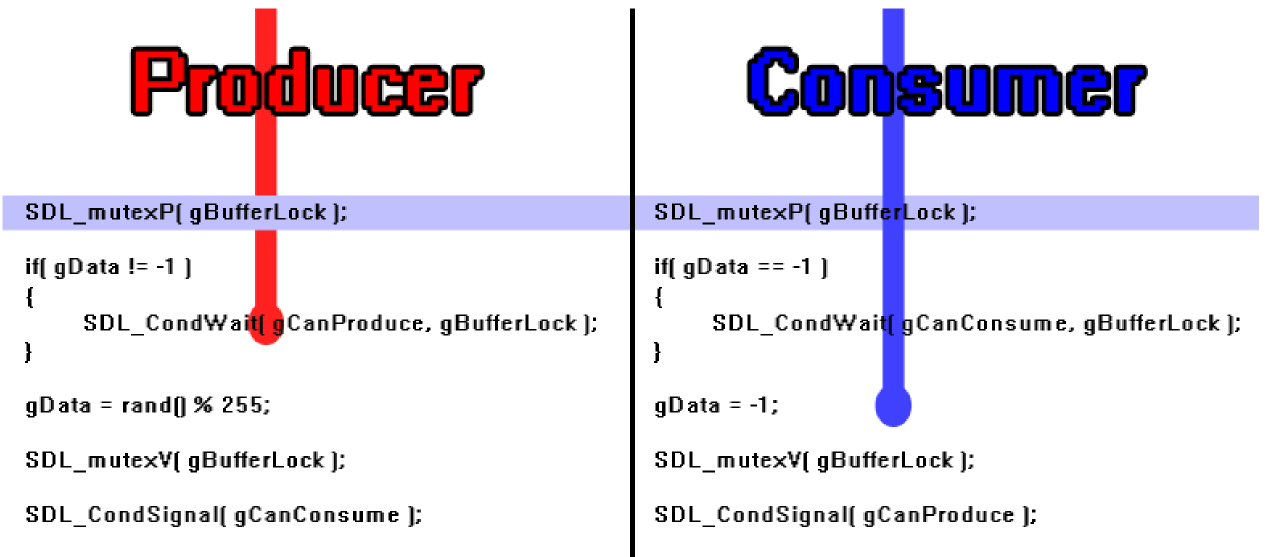


Figura 199: *Ejemplo exclusiones mutuas y condiciones (VII)*

Una vez lo ha hecho, señala al productor con **SDL_CondSignal** para producir de nuevo:

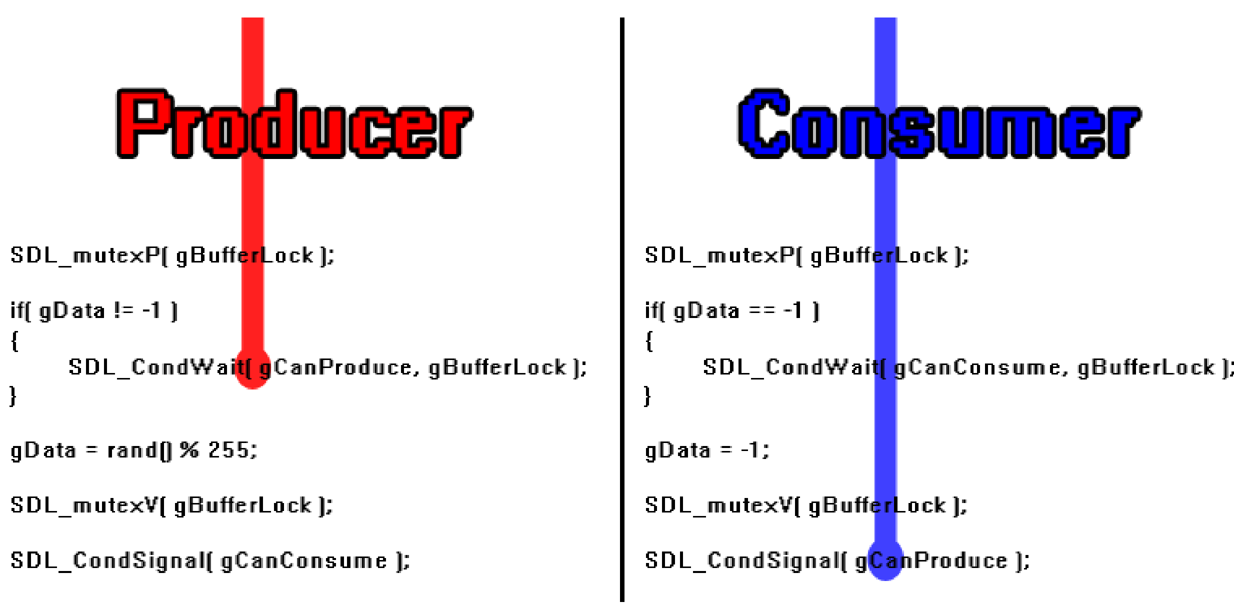


Figura 200: *Ejemplo exclusiones mutuas y condiciones (VIII)*

Con la sección crítica protegida por la exclusión mutua y la habilidad de los hilos para comunicarse entre sí, los hilos de trabajo trabajarán aunque no se sepa el orden en qué orden se ejecutarán.

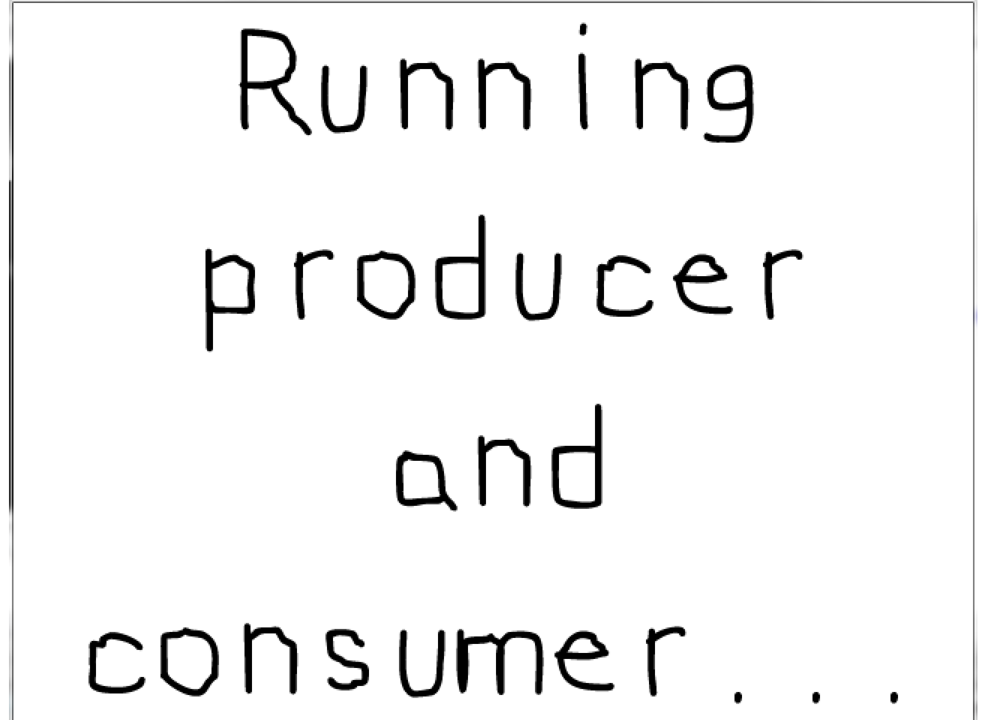


Figura 201: *Resultado ejecutar aplicación exclusiones mutuas y condiciones*

```
Producer started...

Consumer started...

Produced 43

Consumed 43

Produced 53

Consumed 53

Consumer encountered empty buffer, waiting for producer to fill buffer...

Produced 238

Consumed 238

Consumer encountered empty buffer, waiting for producer to fill buffer...

Produced 190

Consumed 190

Produced 61

Producer finished!

Consumed 61

Consumer finished!

Process returned 0 (0x0)   execution time : 30.563 s
Press any key to continue.
```

Figura 202: *Resultado ejecutar aplicación exclusiones mutuas y condiciones (II)*

3.48 SDL y OpenGL 2

Una de las características más importantes de SDL es su capacidad para combinarse con OpenGL. El objetivo es realizar una demostración básica de OpenGL.

```
//Using SDL, SDL OpenGL, standard IO, and, strings
#include <SDL.h>
#include <SDL_opengl.h>
#include <GL\GLU.h>
#include <stdio.h>
#include <string>
```

Para utilizar OpenGL con SDL hay que incluir la cabecera SDL OpenGL. También se usará GLU para este ejemplo.

```
//Starts up SDL, creates window, and initializes OpenGL
bool init();

//Initializes matrices and clear color
bool initGL();

//Input handler
void handleKeys( unsigned char key, int x, int y );

//Per frame update
void update();

//Renders quad to the screen
void render();

//Frees media and shuts down SDL
void close();

//The window we'll be rendering to
SDL_Window* gWindow = NULL;

//OpenGL context
SDL_GLContext gContext;

//Render flag
bool gRenderQuad = true;
```

Hay una función de inicialización de OpenGl con initGL, además de controladores de teclas, actualización y funciones de renderizado para el bucle principal.

También se cuenta con un Contexto GL. Un contexto OpenGL controla las llamadas de OpenGL y es necesario para cualquier renderizado OpenGL.

Finalmente, existe un flag booleano que alterna la renderización.

```
bool init()
{
	//Initialization flag
	bool success = true;

	//Initialize SDL
	if( SDL_Init( SDL_INIT_VIDEO ) < 0 )
	{
printf( "SDL could not initialize! SDL Error: %s\n", SDL_GetError() );
success = false;
	}

	else
	{
		//Use OpenGL 2.1
		SDL_GL_SetAttribute( SDL_GL_CONTEXT_MAJOR_VERSION, 2 );
		SDL_GL_SetAttribute( SDL_GL_CONTEXT_MINOR_VERSION, 1 );

		//Create window
gWindow = SDL_CreateWindow( "SDL Tutorial", SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT,
SDL_WINDOW_OPENGL | SDL_WINDOW_SHOWN );

		if( gWindow == NULL )
		{
printf( "Window could not be created! SDL Error: %s\n", SDL_GetError()
);
			success = false;
		}
```

Al crear una ventana SDL OpenGL hay que seguir los siguientes pasos:

- Antes de crear la ventana, es necesario especificar la versión que se quiere. En este caso es la versión OpenGL 2.1, por lo que se llama a **SDL_GL_SetAttribute** para establecer la versión principal a 2 y la menor a 1.

- Una vez se ha establecido la versión, se crea la ventana OpenGL pasando el flag **SDL_WINDOW_OPENGL flag** a **SDL_CreateWindow**.

```
else
		{
			//Create context
			gContext = SDL_GL_CreateContext( gWindow );
			if( gContext == NULL )
			{
printf( "OpenGL context could not be created! SDL Error: %s\n",
SDL_GetError() );

success = false;
			}

			else
			{
				//Use Vsync
				if( SDL_GL_SetSwapInterval( 1 ) < 0 )
				{
```

```
printf( "Warning: Unable to set VSync! SDL Error: %s\n",
SDL_GetError() );
                        }

                        //Initialize OpenGL
                        if( !initGL() )
                        {
                printf( "Unable to initialize OpenGL!\n" );
                success = false;
                        }
                }
        }
    }

    return success;
}
```

Cuando la ventana se ha creado correctamente se llama a **SDL_GL_CreateContext** para crear el contexto de renderizado OpenGL. Si se realizó correctamente, se activa la sincronización vertical con **SDL_GL_SetSwapInterval**.

Una vez se ha creado la ventana SDL OpenGL, se inicializan las variables internas de OpenGL con la propia función initGL.

```
bool initGL()
{
    bool success = true;
    GLenum error = GL_NO_ERROR;

    //Initialize Projection Matrix
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();

    //Check for error
    error = glGetError();
    if( error != GL_NO_ERROR )
    {
printf( "Error initializing OpenGL! %s\n", gluErrorString( error ) );
success = false;
    }

    //Initialize Modelview Matrix
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    //Check for error
    error = glGetError();
    if( error != GL_NO_ERROR )
    {
printf( "Error initializing OpenGL! %s\n", gluErrorString( error ) );
success = false;
    }
```

- En primer lugar se inicializa la matriz de proyección que controla cómo funciona la perspectiva en OpenGL. Se inicializa estableciéndola en la matriz de identidad.

- A continuación se comprueba si ha habido errores y se imprime en la consola.

- Finalmente se realiza lo mismo con la matriz de vista del modelo, que controla cómo se ven y sitúan los objetos renderizados.

```
        //Initialize clear color
        glClearColor( 0.f, 0.f, 0.f, 1.f );

        //Check for error
        error = glGetError();
        if( error != GL_NO_ERROR )
        {
printf( "Error initializing OpenGL! %s\n", gluErrorString( error ) );
success = false;
        }

        return success;
}
```

Por último, se establece el color claro, que es el color cue limpia la pantalla, cuando se llama a glClear.

```
void handleKeys( unsigned char key, int x, int y )
{
        //Toggle quad
        if( key == 'q' )
        {
                gRenderQuad = !gRenderQuad;
        }
}

void update()
{
        //No per frame update needed
}
```

Controladores de entrada de teclas y actualización. El controlador de entrada de teclas cambia el flag de renderizado y el controlador de actualización se emplea para la compatibilidad.

```
void render()
{
        //Clear color buffer
        glClear( GL_COLOR_BUFFER_BIT );

        //Render quad
        if( gRenderQuad )
        {
                glBegin( GL_QUADS );
                        glVertex2f( -0.5f, -0.5f );
                        glVertex2f( 0.5f, -0.5f );
```

```
                glVertex2f( 0.5f, 0.5f );
                glVertex2f( -0.5f, 0.5f );
        glEnd();
    }
}
```

OpenGL utiliza coordenadas normalizadas, es decir, van de -1 a 1:

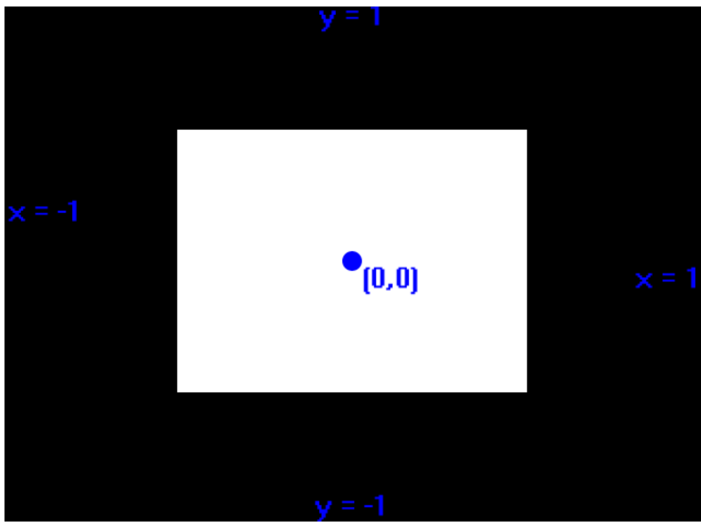


Figura 202: *Sistema de coordenadas OpenGL*

Para conseguir un sistema de coordenadas como el de SDL, hay que ajustar la matriz del proyecto a una perspectiva ortográfica.

```
        //Enable text input
        SDL_StartTextInput();

        //While application is running
        while( !quit )
        {
            //Handle events on queue
            while( SDL_PollEvent( &e ) != 0 )
            {
                //User requests quit
                if( e.type == SDL_QUIT )
                {
                    quit = true;
                }
                //Handle keypress with current mouse position
                else if( e.type == SDL_TEXTINPUT )
                {
                    int x = 0, y = 0;
                    SDL_GetMouseState( &x, &y );
                    handleKeys( e.text.text[ 0 ], x, y );
                }
            }
```

```
                //Render quad
                render();

                //Update screen
                SDL_GL_SwapWindow( gWindow );
            }

            //Disable text input
            SDL_StopTextInput();
        }
```

Cuando se utiliza la renderización de OpenGL no puede emplearse la llamada de renderizado de SDL para superficies ni texturas, sino que se necesita **SDL_GL_SwapWindow** para actualizar la pantalla.

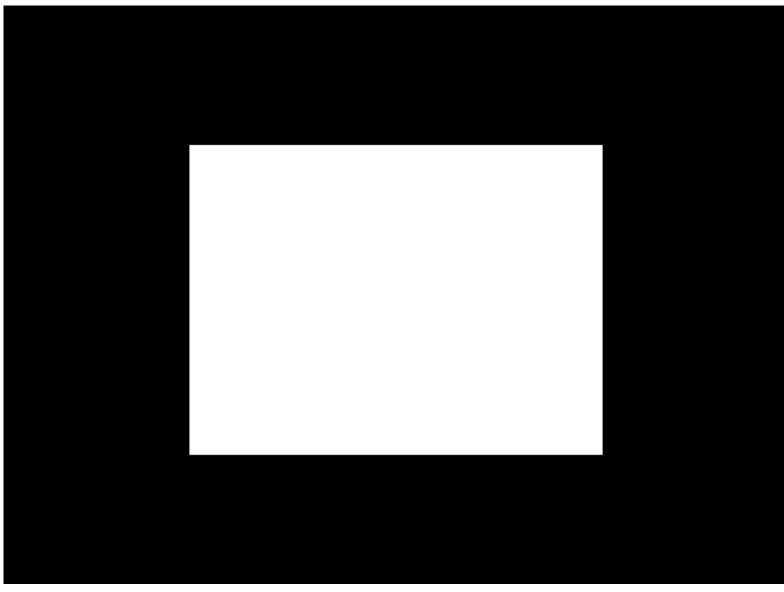


Figura 203: *Resultado tras actualizar la pantalla*

4. Estudio de la Comunicación UDP con Code::Blocks GCC

Con el fin de estudiar la comunicación UDP se realizará un programa que genere una señal sinusoidal (1 ciclo de 1000 puntos por segundo) y se enviará por UDP desde un programa a otro en el que, utilizando SDL, se represente en tiempo real los datos que te vayan llegando.

La estructura principal del programa está dividida en dos archivos denominados Signal Sender, que mandará una señal sinusoidal mediante el protocolo de comunicaciones UDP y Signal Receiver que, a su vez, recibirá dicha señal y la dibujará haciendo uso de la librería SDL.

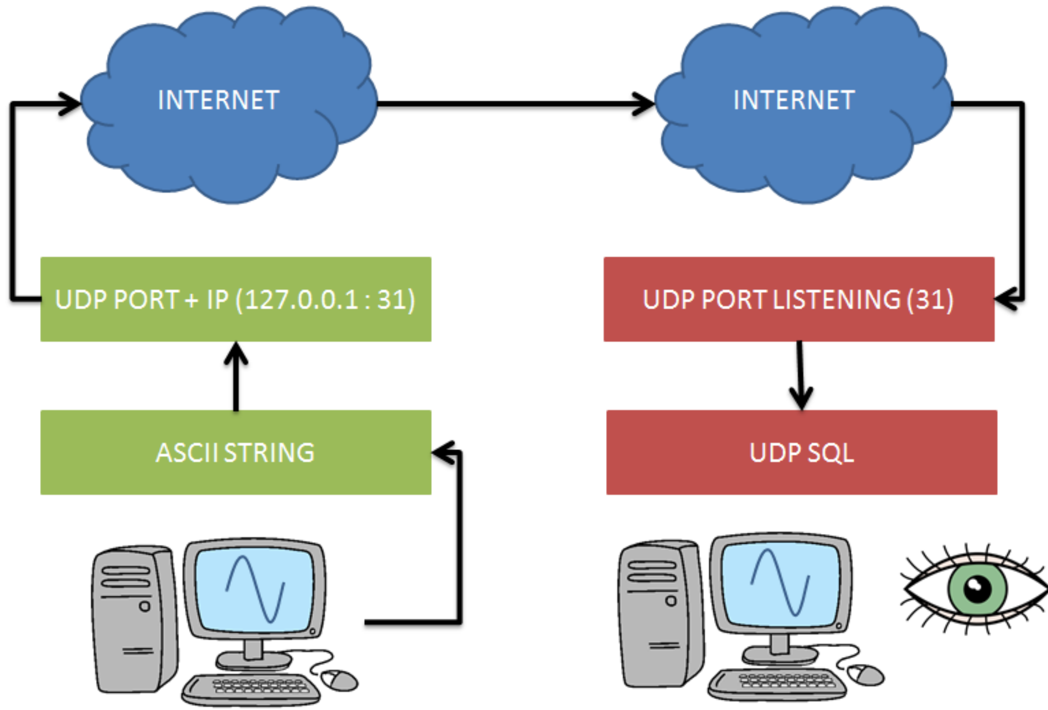


Figura 204: *Esquema visual programa*

A su vez, cada uno de estos dos archivos se encuentra dividido en distintas secciones.

A continuación se muestra el aspecto visual:

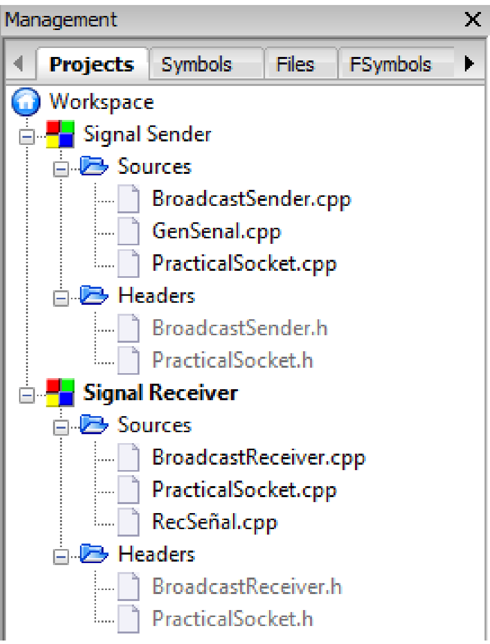


Figura 205: *Aspecto visual*

Signal Sender genera una señal sinusioidal de 1000 puntos y la envía mediante el protocolo de comunicaciones UDP. Una vez se ha enviado, Signal Receiver recibe la señal, la dibuja y la muestra en pantalla.

El cuerpo de Signal Sender se divide en dos tipos de archivos: archivos de fuente y las cabeceras. A su vez, el código está formado por los siguientes archivos fuente:

GenSenal.cpp: genera la señal sinusoidal formada por 1000 puntos.
BroadcastSender.cpp: envía la señal mediante el protocolo UDP.
PracticalSocket.cpp: en él se definen los sockets, conexiones y las funciones para mandar y recibir.

Tanto BroadcastSender como PracticalSocket llevan asociadas una cabecera, BroadcastSender.h y PracticalSocket.h respectivamente.

4.1 SIGNAL SENDER

- **GenSenal.cpp**

```
#include <stdio.h>
#include <math.h>
#define PI  3.1415926
#include <string>
#include "BroadcastSender.h"

int main( int argc, char *argv[] )
{
```

```
    int x = 0;
    int y = 0;
    double fx = 0;
    double angulo = 0;
    int senal[1000];
    char cadena[5000]="";

    PacketSender np; // Se crea el objeto

 // Creación de la señal

 for(angulo=0; angulo<2*PI; angulo= angulo + (2*PI/1000))
    {
        fx = 240 + 240* sin (-angulo ); /* - Debido al origen stma
coordenadas de SDL*/

        y = static_cast<int>( round( fx ) );

    // Guardar los valores

    if (x<1000){

        senal[x]= fx;
    }
        else
            {
            printf("Error");
            }

            x++;
    }

for(int i=0; i<1000; i++)
    {
 sprintf(cadena, "%s %d", cadena, senal[i]); // concatenar las cadenas
    }

 np.sender(5000,"127.0.0.1", 31, cadena);

 return 0;
}
```

- **BroadcastSender.cpp**

```
#include<BroadcastSender.h>
#ifdef WIN32
#include <windows.h>          // For ::Sleep()
void sleep(unsigned int seconds) {::Sleep(seconds * 1000);}
#else
#include <unistd.h>
#include <stdio.h>          // For sleep()
#endif

using namespace std;

int PacketSender::sender(int argc, string destAddress, unsigned short
destPort, char* sendString) {
```

```
     // " <Destination Address> <Destination Port> <Send String>\n";

    try {
    UDPSocket sock;

    // Repeatedly send the string (not including \0) to the server
    for (;;) {
  sock.sendTo(sendString, strlen(sendString), destAddress, destPort);
  sleep(3);

    }
  }
 catch (SocketException &e) {
    cerr << e.what() << endl;
    exit(1);
  }
return 0;

}
```

- **PracticalSocket.cpp**

```
#include "PracticalSocket.h"

#ifdef WIN32
  #include <winsock.h>         // For socket(), connect(), send(), and
recv()
  typedef int socklen_t;
  typedef char raw_type;       // Type used for raw data on this
platform
#else
  #include <sys/types.h>       // For data types
  #include <sys/socket.h>      // For socket(), connect(), send(), and
recv()
  #include <netdb.h>           // For gethostbyname()
  #include <arpa/inet.h>       // For inet_addr()
  #include <unistd.h>          // For close()
  #include <netinet/in.h>      // For sockaddr_in
  typedef void raw_type;       // Type used for raw data on this
platform
#endif

#include <errno.h>             // For errno

using namespace std;

#ifdef WIN32
static bool initialized = false;
#endif

// SocketException Code

SocketException::SocketException(const string &message, bool
inclSysMsg)
  throw() : userMessage(message) {
  if (inclSysMsg) {
    userMessage.append(": ");
    userMessage.append(strerror(errno));
```

```
  }
}

SocketException::~SocketException() throw() {
}

const char *SocketException::what() const throw() {
  return userMessage.c_str();
}

// Function to fill in address structure given an address and port
static void fillAddr(const string &address, unsigned short port,
                     sockaddr_in &addr) {
  memset(&addr, 0, sizeof(addr));  // Zero out address structure
  addr.sin_family = AF_INET;       // Internet address

  hostent *host;  // Resolve name
  if ((host = gethostbyname(address.c_str())) == NULL) {
    // strerror() will not work for gethostbyname() and hstrerror()
    // is supposedly obsolete
    throw SocketException("Failed to resolve name (gethostbyname())");
  }
  addr.sin_addr.s_addr = *((unsigned long *) host->h_addr_list[0]);

  addr.sin_port = htons(port);     // Assign port in network byte
order
}

// Socket Code

Socket::Socket(int type, int protocol) throw(SocketException) {
  #ifdef WIN32
    if (!initialized) {
      WORD wVersionRequested;
      WSADATA wsaData;

      wVersionRequested = MAKEWORD(2, 0);              // Request
WinSock v2.0
      if (WSAStartup(wVersionRequested, &wsaData) != 0) {  // Load
WinSock DLL
        throw SocketException("Unable to load WinSock DLL");
      }
      initialized = true;
    }
  #endif

  // Make a new socket
  if ((sockDesc = socket(PF_INET, type, protocol)) < 0) {
    throw SocketException("Socket creation failed (socket())", true);
  }
}

Socket::Socket(int sockDesc) {
  this->sockDesc = sockDesc;
}

Socket::~Socket() {
  #ifdef WIN32
    ::closesocket(sockDesc);
  #else
    ::close(sockDesc);
```

```
  #endif
  sockDesc = -1;
}

string Socket::getLocalAddress() throw(SocketException) {
  sockaddr_in addr;
  unsigned int addr_len = sizeof(addr);

  if (getsockname(sockDesc, (sockaddr *) &addr, (socklen_t *)
&addr_len) < 0) {
    throw SocketException("Fetch of local address failed
(getsockname())", true);
  }
  return inet_ntoa(addr.sin_addr);
}

unsigned short Socket::getLocalPort() throw(SocketException) {
  sockaddr_in addr;
  unsigned int addr_len = sizeof(addr);

  if (getsockname(sockDesc, (sockaddr *) &addr, (socklen_t *)
&addr_len) < 0) {
    throw SocketException("Fetch of local port failed
(getsockname())", true);
  }
  return ntohs(addr.sin_port);
}

void Socket::setLocalPort(unsigned short localPort)
throw(SocketException) {
  // Bind the socket to its port
  sockaddr_in localAddr;
  memset(&localAddr, 0, sizeof(localAddr));
  localAddr.sin_family = AF_INET;
  localAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  localAddr.sin_port = htons(localPort);

  if (bind(sockDesc, (sockaddr *) &localAddr, sizeof(sockaddr_in)) <
0) {
    throw SocketException("Set of local port failed (bind())", true);
  }
}

void Socket::setLocalAddressAndPort(const string &localAddress,
    unsigned short localPort) throw(SocketException) {
  // Get the address of the requested host
  sockaddr_in localAddr;
  fillAddr(localAddress, localPort, localAddr);

  if (bind(sockDesc, (sockaddr *) &localAddr, sizeof(sockaddr_in)) <
0) {
    throw SocketException("Set of local address and port failed
(bind())", true);
  }
}

void Socket::cleanUp() throw(SocketException) {
  #ifdef WIN32
    if (WSACleanup() != 0) {
      throw SocketException("WSACleanup() failed");
    }
```

```
  #endif
}

unsigned short Socket::resolveService(const string &service,
                                      const string &protocol) {
  struct servent *serv;        /* Structure containing service
information */

  if ((serv = getservbyname(service.c_str(), protocol.c_str())) ==
NULL)
    return atoi(service.c_str());  /* Service is port number */
  else
    return ntohs(serv->s_port);    /* Found port (network byte order)
by name */
}

// CommunicatingSocket Code

CommunicatingSocket::CommunicatingSocket(int type, int protocol)
    throw(SocketException) : Socket(type, protocol) {
}

CommunicatingSocket::CommunicatingSocket(int newConnSD) :
Socket(newConnSD) {
}

void CommunicatingSocket::connect(const string &foreignAddress,
    unsigned short foreignPort) throw(SocketException) {
  // Get the address of the requested host
  sockaddr_in destAddr;
  fillAddr(foreignAddress, foreignPort, destAddr);

  // Try to connect to the given port
  if (::connect(sockDesc, (sockaddr *) &destAddr, sizeof(destAddr)) <
0) {
    throw SocketException("Connect failed (connect())", true);
  }
}

void CommunicatingSocket::send(const void *buffer, int bufferLen)
    throw(SocketException) {
  if (::send(sockDesc, (raw_type *) buffer, bufferLen, 0) < 0) {
    throw SocketException("Send failed (send())", true);
  }
}

int CommunicatingSocket::recv(void *buffer, int bufferLen)
    throw(SocketException) {
  int rtn;
  if ((rtn = ::recv(sockDesc, (raw_type *) buffer, bufferLen, 0)) < 0)
{
    throw SocketException("Received failed (recv())", true);
  }

  return rtn;
}

string CommunicatingSocket::getForeignAddress()
    throw(SocketException) {
  sockaddr_in addr;
  unsigned int addr_len = sizeof(addr);
```

```
  if (getpeername(sockDesc, (sockaddr *) &addr,(socklen_t *)
&addr_len) < 0) {
    throw SocketException("Fetch of foreign address failed
(getpeername())", true);
  }
  return inet_ntoa(addr.sin_addr);
}

unsigned short CommunicatingSocket::getForeignPort()
throw(SocketException) {
  sockaddr_in addr;
  unsigned int addr_len = sizeof(addr);

  if (getpeername(sockDesc, (sockaddr *) &addr, (socklen_t *)
&addr_len) < 0) {
    throw SocketException("Fetch of foreign port failed
(getpeername())", true);
  }
  return ntohs(addr.sin_port);
}

// TCPSocket Code

TCPSocket::TCPSocket()
    throw(SocketException) : CommunicatingSocket(SOCK_STREAM,
    IPPROTO_TCP) {
}

TCPSocket::TCPSocket(const string &foreignAddress, unsigned short
foreignPort)
    throw(SocketException) : CommunicatingSocket(SOCK_STREAM,
IPPROTO_TCP) {
  connect(foreignAddress, foreignPort);
}

TCPSocket::TCPSocket(int newConnSD) : CommunicatingSocket(newConnSD) {
}

// TCPServerSocket Code

TCPServerSocket::TCPServerSocket(unsigned short localPort, int
queueLen)
    throw(SocketException) : Socket(SOCK_STREAM, IPPROTO_TCP) {
  setLocalPort(localPort);
  setListen(queueLen);
}

TCPServerSocket::TCPServerSocket(const string &localAddress,
    unsigned short localPort, int queueLen)
    throw(SocketException) : Socket(SOCK_STREAM, IPPROTO_TCP) {
  setLocalAddressAndPort(localAddress, localPort);
  setListen(queueLen);
}

TCPSocket *TCPServerSocket::accept() throw(SocketException) {
  int newConnSD;
  if ((newConnSD = ::accept(sockDesc, NULL, 0)) < 0) {
    throw SocketException("Accept failed (accept())", true);
  }
```

```
  return new TCPSocket(newConnSD);
}

void TCPServerSocket::setListen(int queueLen) throw(SocketException) {
  if (listen(sockDesc, queueLen) < 0) {
    throw SocketException("Set listening socket failed (listen())",
true);
  }
}

// UDPSocket Code

UDPSocket::UDPSocket() throw(SocketException) :
CommunicatingSocket(SOCK_DGRAM,
    IPPROTO_UDP) {
  setBroadcast();
}

UDPSocket::UDPSocket(unsigned short localPort)  throw(SocketException)
:
    CommunicatingSocket(SOCK_DGRAM, IPPROTO_UDP) {
  setLocalPort(localPort);
  setBroadcast();
}

UDPSocket::UDPSocket(const string &localAddress, unsigned short
localPort)
     throw(SocketException) : CommunicatingSocket(SOCK_DGRAM,
IPPROTO_UDP) {
  setLocalAddressAndPort(localAddress, localPort);
  setBroadcast();
}

void UDPSocket::setBroadcast() {
  // If this fails, we'll hear about it when we try to send.  This
will allow
  // system that cannot broadcast to continue if they don't plan to
broadcast
  int broadcastPermission = 1;
  setsockopt(sockDesc, SOL_SOCKET, SO_BROADCAST,
             (raw_type *) &broadcastPermission,
sizeof(broadcastPermission));
}

void UDPSocket::disconnect() throw(SocketException) {
  sockaddr_in nullAddr;
  memset(&nullAddr, 0, sizeof(nullAddr));
  nullAddr.sin_family = AF_UNSPEC;

  // Try to disconnect
  if (::connect(sockDesc, (sockaddr *) &nullAddr, sizeof(nullAddr)) <
0) {
   #ifdef WIN32
    if (errno != WSAEAFNOSUPPORT) {
   #else
    if (errno != EAFNOSUPPORT) {
   #endif
      throw SocketException("Disconnect failed (connect())", true);
    }
  }
}
```

```
void UDPSocket::sendTo(const void *buffer, int bufferLen,
    const string &foreignAddress, unsigned short foreignPort)
    throw(SocketException) {
  sockaddr_in destAddr;
  fillAddr(foreignAddress, foreignPort, destAddr);

  // Write out the whole buffer as a single message.
  if (sendto(sockDesc, (raw_type *) buffer, bufferLen, 0,
             (sockaddr *) &destAddr, sizeof(destAddr)) != bufferLen) {
    throw SocketException("Send failed (sendto())", true);
  }
}

int UDPSocket::recvFrom(void *buffer, int bufferLen, string
&sourceAddress,
    unsigned short &sourcePort) throw(SocketException) {
  sockaddr_in clntAddr;
  socklen_t addrLen = sizeof(clntAddr);
  int rtn;
  if ((rtn = recvfrom(sockDesc, (raw_type *) buffer, bufferLen, 0,
                      (sockaddr *) &clntAddr, (socklen_t *) &addrLen))
< 0) {
    throw SocketException("Receive failed (recvfrom())", true);
  }
  sourceAddress = inet_ntoa(clntAddr.sin_addr);
  sourcePort = ntohs(clntAddr.sin_port);

  return rtn;
}

void UDPSocket::setMulticastTTL(unsigned char multicastTTL)
throw(SocketException) {
  if (setsockopt(sockDesc, IPPROTO_IP, IP_MULTICAST_TTL,
                 (raw_type *) &multicastTTL, sizeof(multicastTTL)) <
0) {
    throw SocketException("Multicast TTL set failed (setsockopt())",
true);
  }
}

void UDPSocket::joinGroup(const string &multicastGroup)
throw(SocketException) {
  struct ip_mreq multicastRequest;

  multicastRequest.imr_multiaddr.s_addr =
inet_addr(multicastGroup.c_str());
  multicastRequest.imr_interface.s_addr = htonl(INADDR_ANY);
  if (setsockopt(sockDesc, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                 (raw_type *) &multicastRequest,
                 sizeof(multicastRequest)) < 0) {
    throw SocketException("Multicast group join failed
(setsockopt())", true);
  }
}

void UDPSocket::leaveGroup(const string &multicastGroup)
throw(SocketException) {
  struct ip_mreq multicastRequest;

  multicastRequest.imr_multiaddr.s_addr =
```

```
inet_addr(multicastGroup.c_str());
  multicastRequest.imr_interface.s_addr = htonl(INADDR_ANY);
  if (setsockopt(sockDesc, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                 (raw_type *) &multicastRequest,
                 sizeof(multicastRequest)) < 0) {
    throw SocketException("Multicast group leave failed
(setsockopt())", true);
  }
}
```

- **PracticalSocket.h**

```
#ifndef __PRACTICALSOCKET_INCLUDED__
#define __PRACTICALSOCKET_INCLUDED__

#include <string>            // For string
#include <exception>         // For exception class

using namespace std;

/**
 *   Signals a problem with the execution of a socket call.
 */
class SocketException : public exception {
public:
  /**
   *   Construct a SocketException with a explanatory message.
   *   @param message explanatory message
   *   @param incSysMsg true if system message (from strerror(errno))
   *   should be postfixed to the user provided message
   */
  SocketException(const string &message, bool inclSysMsg = false)
throw();

  /**
   *   Provided just to guarantee that no exceptions are thrown.
   */
  ~SocketException() throw();

  /**
   *   Get the exception message
   *   @return exception message
   */
  const char *what() const throw();

private:
  string userMessage;  // Exception message
};

/**
 *   Base class representing basic communication endpoint
 */
class Socket {
public:
  /**
   *   Close and deallocate this socket
   */
  ~Socket();
```

```
  /**
   *   Get the local address
   *   @return local address of socket
   *   @exception SocketException thrown if fetch fails
   */
  string getLocalAddress() throw(SocketException);

  /**
   *   Get the local port
   *   @return local port of socket
   *   @exception SocketException thrown if fetch fails
   */
  unsigned short getLocalPort() throw(SocketException);

  /**
   *   Set the local port to the specified port and the local address
   *   to any interface
   *   @param localPort local port
   *   @exception SocketException thrown if setting local port fails
   */
  void setLocalPort(unsigned short localPort) throw(SocketException);

  /**
   *   Set the local port to the specified port and the local address
   *   to the specified address.  If you omit the port, a random port
   *   will be selected.
   *   @param localAddress local address
   *   @param localPort local port
   *   @exception SocketException thrown if setting local port or
address fails
   */
  void setLocalAddressAndPort(const string &localAddress,
    unsigned short localPort = 0) throw(SocketException);

  /**
   *   If WinSock, unload the WinSock DLLs; otherwise do nothing.  We
ignore
   *   this in our sample client code but include it in the library
for
   *   completeness.  If you are running on Windows and you are
concerned
   *   about DLL resource consumption, call this after you are done
with all
   *   Socket instances.  If you execute this on Windows while some
instance of
   *   Socket exists, you are toast.  For portability of client code,
this is
   *   an empty function on non-Windows platforms so you can always
include it.
   *   @param buffer buffer to receive the data
   *   @param bufferLen maximum number of bytes to read into buffer
   *   @return number of bytes read, 0 for EOF, and -1 for error
   *   @exception SocketException thrown WinSock clean up fails
   */
  static void cleanUp() throw(SocketException);

  /**
   *   Resolve the specified service for the specified protocol to the
   *   corresponding port number in host byte order
   *   @param service service to resolve (e.g., "http")
```

```
   *   @param protocol protocol of service to resolve.  Default is
"tcp".
   */
  static unsigned short resolveService(const string &service,
                                       const string &protocol =
"tcp");

private:
  // Prevent the user from trying to use value semantics on this
object
  Socket(const Socket &sock);
  void operator=(const Socket &sock);

protected:
  int sockDesc;              // Socket descriptor
  Socket(int type, int protocol) throw(SocketException);
  Socket(int sockDesc);
};

/**
 *   Socket which is able to connect, send, and receive
 */
class CommunicatingSocket : public Socket {
public:
  /**
   *   Establish a socket connection with the given foreign
   *   address and port
   *   @param foreignAddress foreign address (IP address or name)
   *   @param foreignPort foreign port
   *   @exception SocketException thrown if unable to establish
connection
   */
  void connect(const string &foreignAddress, unsigned short
foreignPort)
    throw(SocketException);

  /**
   *   Write the given buffer to this socket.  Call connect() before
   *   calling send()
   *   @param buffer buffer to be written
   *   @param bufferLen number of bytes from buffer to be written
   *   @exception SocketException thrown if unable to send data
   */
  void send(const void *buffer, int bufferLen) throw(SocketException);

  /**
   *   Read into the given buffer up to bufferLen bytes data from this
   *   socket.  Call connect() before calling recv()
   *   @param buffer buffer to receive the data
   *   @param bufferLen maximum number of bytes to read into buffer
   *   @return number of bytes read, 0 for EOF, and -1 for error
   *   @exception SocketException thrown if unable to receive data
   */
  int recv(void *buffer, int bufferLen) throw(SocketException);

  /**
   *   Get the foreign address.  Call connect() before calling recv()
   *   @return foreign address
   *   @exception SocketException thrown if unable to fetch foreign
address
   */
```

```
  string getForeignAddress() throw(SocketException);

  /**
   *   Get the foreign port.  Call connect() before calling recv()
   *   @return foreign port
   *   @exception SocketException thrown if unable to fetch foreign
port
   */
  unsigned short getForeignPort() throw(SocketException);

protected:
  CommunicatingSocket(int type, int protocol) throw(SocketException);
  CommunicatingSocket(int newConnSD);
};

/**
 *   TCP socket for communication with other TCP sockets
 */
class TCPSocket : public CommunicatingSocket {
public:
  /**
   *   Construct a TCP socket with no connection
   *   @exception SocketException thrown if unable to create TCP
socket
   */
  TCPSocket() throw(SocketException);

  /**
   *   Construct a TCP socket with a connection to the given foreign
address
   *   and port
   *   @param foreignAddress foreign address (IP address or name)
   *   @param foreignPort foreign port
   *   @exception SocketException thrown if unable to create TCP
socket
   */
  TCPSocket(const string &foreignAddress, unsigned short foreignPort)
      throw(SocketException);

private:
  // Access for TCPServerSocket::accept() connection creation
  friend class TCPServerSocket;
  TCPSocket(int newConnSD);
};

/**
 *   TCP socket class for servers
 */
class TCPServerSocket : public Socket {
public:
  /**
   *   Construct a TCP socket for use with a server, accepting
connections
   *   on the specified port on any interface
   *   @param localPort local port of server socket, a value of zero
will
   *                   give a system-assigned unused port
   *   @param queueLen maximum queue length for outstanding
   *                   connection requests (default 5)
   *   @exception SocketException thrown if unable to create TCP
server socket
```

```
   */
  TCPServerSocket(unsigned short localPort, int queueLen = 5)
      throw(SocketException);

  /**
   *   Construct a TCP socket for use with a server, accepting
connections
   *   on the specified port on the interface specified by the given
address
   *   @param localAddress local interface (address) of server socket
   *   @param localPort local port of server socket
   *   @param queueLen maximum queue length for outstanding
   *                   connection requests (default 5)
   *   @exception SocketException thrown if unable to create TCP
server socket
   */
  TCPServerSocket(const string &localAddress, unsigned short
localPort,
      int queueLen = 5) throw(SocketException);

  /**
   *   Blocks until a new connection is established on this socket or
error
   *   @return new connection socket
   *   @exception SocketException thrown if attempt to accept a new
connection fails
   */
  TCPSocket *accept() throw(SocketException);

private:
  void setListen(int queueLen) throw(SocketException);
};

/**
  *   UDP socket class
  */
class UDPSocket : public CommunicatingSocket {
public:
  /**
   *   Construct a UDP socket
   *   @exception SocketException thrown if unable to create UDP
socket
   */
  UDPSocket() throw(SocketException);

  /**
   *   Construct a UDP socket with the given local port
   *   @param localPort local port
   *   @exception SocketException thrown if unable to create UDP
socket
   */
  UDPSocket(unsigned short localPort) throw(SocketException);

  /**
   *   Construct a UDP socket with the given local port and address
   *   @param localAddress local address
   *   @param localPort local port
   *   @exception SocketException thrown if unable to create UDP
socket
   */
  UDPSocket(const string &localAddress, unsigned short localPort)
```

```
      throw(SocketException);

  /**
   *   Unset foreign address and port
   *   @return true if disassociation is successful
   *   @exception SocketException thrown if unable to disconnect UDP
socket
   */
  void disconnect() throw(SocketException);

  /**
   *   Send the given buffer as a UDP datagram to the
   *   specified address/port
   *   @param buffer buffer to be written
   *   @param bufferLen number of bytes to write
   *   @param foreignAddress address (IP address or name) to send to
   *   @param foreignPort port number to send to
   *   @return true if send is successful
   *   @exception SocketException thrown if unable to send datagram
   */
  void sendTo(const void *buffer, int bufferLen, const string
&foreignAddress,
            unsigned short foreignPort) throw(SocketException);

  /**
   *   Read read up to bufferLen bytes data from this socket.  The
given buffer
   *   is where the data will be placed
   *   @param buffer buffer to receive data
   *   @param bufferLen maximum number of bytes to receive
   *   @param sourceAddress address of datagram source
   *   @param sourcePort port of data source
   *   @return number of bytes received and -1 for error
   *   @exception SocketException thrown if unable to receive datagram
   */
  int recvFrom(void *buffer, int bufferLen, string &sourceAddress,
               unsigned short &sourcePort) throw(SocketException);

  /**
   *   Set the multicast TTL
   *   @param multicastTTL multicast TTL
   *   @exception SocketException thrown if unable to set TTL
   */
  void setMulticastTTL(unsigned char multicastTTL)
throw(SocketException);

  /**
   *   Join the specified multicast group
   *   @param multicastGroup multicast group address to join
   *   @exception SocketException thrown if unable to join group
   */
  void joinGroup(const string &multicastGroup) throw(SocketException);

  /**
   *   Leave the specified multicast group
   *   @param multicastGroup multicast group address to leave
   *   @exception SocketException thrown if unable to leave group
   */
  void leaveGroup(const string &multicastGroup)
throw(SocketException);
```

```
private:
  void setBroadcast();
};
#endif
```

Los archivos de tipo cabecera son:

- **BroadcastSender.h**

```
#include <stdio.h>
#include "PracticalSocket.h"  // For UDPSocket and SocketException
#include <iostream>           // For cout and cerr
#include <cstdlib>            // For atoi()
#include <string>            // For string
#include <exception>         // For exception class

using namespace std;

class PacketSender {

public:
int sender(int argc, string destAddress, unsigned short destPort,
char* sendString);
};
```

4.2 SIGNAL RECEIVER

Al igual que ocurría en Signal Sender, el cuerpo de Signal Receiver se divide en dos tipos de archivos: archivos de fuente y las cabeceras. A su vez, el código está formado por los siguientes archivos fuente:

RecSeñal.cpp: dibuja la señal sinusoidal recibida y la muestra en pantalla.
BroadcastReceiver.cpp: recibe la señal enviada mediante el protocolo UDP.
PracticalSocket.cpp: en él se definen los sockets, conexiones y las funciones para mandar y recibir.

Tanto BroadcastReceiver como PracticalSocket llevan asociadas una cabecera, BroadcastReceiver.h y PracticalSocket.h respectivamente.

- **RecSeñal.cpp**

```
#include <stdio.h>
#include <SDL.h>
#include <math.h>
#define PI  3.1415926
#include <string>
#include "BroadcastReceiver.h"
#include <iostream>
#include <cstring>
#include <stdlib.h>
```

```
int main( int argc, char *argv[] ) {

    int x = 0;
    int y = 0;
    int numeros [5000];
    double fx = 0;
    double angulo = 0;
    double senal[1000];
    char numbers[5000];

    Uint32 color = 0;
    string cadena;

    PacketReceiver np; //crear objeto

    // Creación Ventana

    SDL_Window* win;
    win = SDL_CreateWindow( "y=sin(x)", 100, 100, 1100, 480, 0);

    SDL_Renderer* ren;
    ren = SDL_CreateRenderer( win, -1, SDL_RENDERER_ACCELERATED |
SDL_RENDERER_PRESENTVSYNC );

    SDL_Surface* surface;
    surface = SDL_CreateRGBSurface( 0, 1100, 480, 32, 0, 0, 0, 0 );

    Uint32* pixels;
    pixels = static_cast<Uint32*>( surface->pixels );

    SDL_LockSurface(surface);
    color = SDL_MapRGB( surface-> format, 76, 153, 0 );

    cadena = np.receiver(5000,31);

for ( int i=0; i<= cadena.length(); i++)
    {
    //cout << cadena[i];
    }
    //exit (0);

strcpy(numbers, cadena.c_str());

char * ptr= numbers;

ptr= strtok(numbers, " ");
int i=0;

     while (ptr != NULL)
  {
    numeros[i]= atoi(ptr);
    i++;
    ptr = strtok (NULL, " ");
  }

for(int j=0; j<i; j++)

{
    y =  numeros[j];
```

```
    pixels[ ( y * surface-> w )+ j ] = color;

}

        SDL_UnlockSurface( surface );

        SDL_Texture *tex;
        tex = SDL_CreateTextureFromSurface( ren, surface );

        SDL_RenderCopy( ren, tex, NULL, NULL );
        SDL_RenderPresent( ren );
        SDL_Delay( 4000 );

        return 0;
}
```

- **BroadcastReceiver.cpp**

```
#include "PracticalSocket.h"   // For UDPSocket and SocketException
#include <iostream>            // For cout and cerr
#include <cstdlib>             // For atoi()
#include <BroadcastReceiver.h>

const int MAXRCVSTRING = 5192; // Longest string to receive (
//char * recvString[MAXRCVSTRING + 1]= {" "};

char* PacketReceiver::receiver(int argc, unsigned short echoServPort)
{

    try {
    UDPSocket sock(echoServPort);
    char recvString[MAXRCVSTRING + 1]; // Buffer for echo string + \0
    string sourceAddress;              // Address of datagram source
    unsigned short sourcePort;         // Port of datagram source
    int bytesRcvd = sock.recvFrom(recvString, MAXRCVSTRING,
sourceAddress,
                                  sourcePort);
    recvString[bytesRcvd] = '\0';  // Terminate string

    //cout << "Received " << recvString << " from " << sourceAddress
<< ": "
        // << sourcePort << endl;

    return recvString;
  }
    catch (SocketException &e)
    {
    cerr << e.what() << endl;
    exit(1);
    }

}
```

- **PracticalSocket.cpp**

```
#include "PracticalSocket.h"

#ifdef WIN32
  #include <winsock.h>         // For socket(), connect(), send(), and
recv()
  typedef int socklen_t;
  typedef char raw_type;       // Type used for raw data on this
platform
#else
  #include <sys/types.h>       // For data types
  #include <sys/socket.h>      // For socket(), connect(), send(), and
recv()
  #include <netdb.h>           // For gethostbyname()
  #include <arpa/inet.h>       // For inet_addr()
  #include <unistd.h>          // For close()
  #include <netinet/in.h>      // For sockaddr_in
  typedef void raw_type;       // Type used for raw data on this
platform
#endif

#include <errno.h>             // For errno

using namespace std;

#ifdef WIN32
static bool initialized = false;
#endif

// SocketException Code

SocketException::SocketException(const string &message, bool
inclSysMsg)
  throw() : userMessage(message) {
  if (inclSysMsg) {
    userMessage.append(": ");
    userMessage.append(strerror(errno));
  }
}

SocketException::~SocketException() throw() {
}

const char *SocketException::what() const throw() {
  return userMessage.c_str();
}

// Function to fill in address structure given an address and port
static void fillAddr(const string &address, unsigned short port,
                     sockaddr_in &addr) {
  memset(&addr, 0, sizeof(addr));  // Zero out address structure
  addr.sin_family = AF_INET;       // Internet address

  hostent *host;  // Resolve name
  if ((host = gethostbyname(address.c_str())) == NULL) {
    // strerror() will not work for gethostbyname() and hstrerror()
    // is supposedly obsolete
    throw SocketException("Failed to resolve name (gethostbyname())");
  }
  addr.sin_addr.s_addr = *((unsigned long *) host->h_addr_list[0]);
```

```
  addr.sin_port = htons(port);      // Assign port in network byte
order
}

// Socket Code

Socket::Socket(int type, int protocol) throw(SocketException) {
  #ifdef WIN32
    if (!initialized) {
      WORD wVersionRequested;
      WSADATA wsaData;

      wVersionRequested = MAKEWORD(2, 0);              // Request
WinSock v2.0
      if (WSAStartup(wVersionRequested, &wsaData) != 0) {  // Load
WinSock DLL
        throw SocketException("Unable to load WinSock DLL");
      }
      initialized = true;
    }
  #endif

  // Make a new socket
  if ((sockDesc = socket(PF_INET, type, protocol)) < 0) {
    throw SocketException("Socket creation failed (socket())", true);
  }
}

Socket::Socket(int sockDesc) {
  this->sockDesc = sockDesc;
}

Socket::~Socket() {
  #ifdef WIN32
    ::closesocket(sockDesc);
  #else
    ::close(sockDesc);
  #endif
  sockDesc = -1;
}

string Socket::getLocalAddress() throw(SocketException) {
  sockaddr_in addr;
  unsigned int addr_len = sizeof(addr);

  if (getsockname(sockDesc, (sockaddr *) &addr, (socklen_t *)
&addr_len) < 0) {
    throw SocketException("Fetch of local address failed
(getsockname())", true);
  }
  return inet_ntoa(addr.sin_addr);
}

unsigned short Socket::getLocalPort() throw(SocketException) {
  sockaddr_in addr;
  unsigned int addr_len = sizeof(addr);

  if (getsockname(sockDesc, (sockaddr *) &addr, (socklen_t *)
&addr_len) < 0) {
    throw SocketException("Fetch of local port failed
```

```
(getsockname())", true);
  }
  return ntohs(addr.sin_port);
}

void Socket::setLocalPort(unsigned short localPort)
throw(SocketException) {
  // Bind the socket to its port
  sockaddr_in localAddr;
  memset(&localAddr, 0, sizeof(localAddr));
  localAddr.sin_family = AF_INET;
  localAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  localAddr.sin_port = htons(localPort);

  if (bind(sockDesc, (sockaddr *) &localAddr, sizeof(sockaddr_in)) <
0) {
    throw SocketException("Set of local port failed (bind())", true);
  }
}

void Socket::setLocalAddressAndPort(const string &localAddress,
    unsigned short localPort) throw(SocketException) {
  // Get the address of the requested host
  sockaddr_in localAddr;
  fillAddr(localAddress, localPort, localAddr);

  if (bind(sockDesc, (sockaddr *) &localAddr, sizeof(sockaddr_in)) <
0) {
    throw SocketException("Set of local address and port failed
(bind())", true);
  }
}

void Socket::cleanUp() throw(SocketException) {
  #ifdef WIN32
    if (WSACleanup() != 0) {
      throw SocketException("WSACleanup() failed");
    }
  #endif
}

unsigned short Socket::resolveService(const string &service,
                                      const string &protocol) {
  struct servent *serv;        /* Structure containing service
information */

  if ((serv = getservbyname(service.c_str(), protocol.c_str())) ==
NULL)
    return atoi(service.c_str());  /* Service is port number */
  else
    return ntohs(serv->s_port);    /* Found port (network byte order)
by name */
}

// CommunicatingSocket Code

CommunicatingSocket::CommunicatingSocket(int type, int protocol)
    throw(SocketException) : Socket(type, protocol) {
}

CommunicatingSocket::CommunicatingSocket(int newConnSD) :
```

```
Socket(newConnSD) {
}

void CommunicatingSocket::connect(const string &foreignAddress,
    unsigned short foreignPort) throw(SocketException) {
  // Get the address of the requested host
  sockaddr_in destAddr;
  fillAddr(foreignAddress, foreignPort, destAddr);

  // Try to connect to the given port
  if (::connect(sockDesc, (sockaddr *) &destAddr, sizeof(destAddr)) <
0) {
    throw SocketException("Connect failed (connect())", true);
  }
}

void CommunicatingSocket::send(const void *buffer, int bufferLen)
    throw(SocketException) {
  if (::send(sockDesc, (raw_type *) buffer, bufferLen, 0) < 0) {
    throw SocketException("Send failed (send())", true);
  }
}

int CommunicatingSocket::recv(void *buffer, int bufferLen)
    throw(SocketException) {
  int rtn;
  if ((rtn = ::recv(sockDesc, (raw_type *) buffer, bufferLen, 0)) < 0)
{
    throw SocketException("Received failed (recv())", true);
  }

  return rtn;
}

string CommunicatingSocket::getForeignAddress()
    throw(SocketException) {
  sockaddr_in addr;
  unsigned int addr_len = sizeof(addr);

  if (getpeername(sockDesc, (sockaddr *) &addr,(socklen_t *)
&addr_len) < 0) {
    throw SocketException("Fetch of foreign address failed
(getpeername())", true);
  }
  return inet_ntoa(addr.sin_addr);
}

unsigned short CommunicatingSocket::getForeignPort()
throw(SocketException) {
  sockaddr_in addr;
  unsigned int addr_len = sizeof(addr);

  if (getpeername(sockDesc, (sockaddr *) &addr, (socklen_t *)
&addr_len) < 0) {
    throw SocketException("Fetch of foreign port failed
(getpeername())", true);
  }
  return ntohs(addr.sin_port);
}

// TCPSocket Code
```

```
TCPSocket::TCPSocket()
    throw(SocketException) : CommunicatingSocket(SOCK_STREAM,
    IPPROTO_TCP) {
}

TCPSocket::TCPSocket(const string &foreignAddress, unsigned short
foreignPort)
    throw(SocketException) : CommunicatingSocket(SOCK_STREAM,
IPPROTO_TCP) {
  connect(foreignAddress, foreignPort);
}

TCPSocket::TCPSocket(int newConnSD) : CommunicatingSocket(newConnSD) {
}

// TCPServerSocket Code

TCPServerSocket::TCPServerSocket(unsigned short localPort, int
queueLen)
    throw(SocketException) : Socket(SOCK_STREAM, IPPROTO_TCP) {
  setLocalPort(localPort);
  setListen(queueLen);
}

TCPServerSocket::TCPServerSocket(const string &localAddress,
    unsigned short localPort, int queueLen)
    throw(SocketException) : Socket(SOCK_STREAM, IPPROTO_TCP) {
  setLocalAddressAndPort(localAddress, localPort);
  setListen(queueLen);
}

TCPSocket *TCPServerSocket::accept() throw(SocketException) {
  int newConnSD;
  if ((newConnSD = ::accept(sockDesc, NULL, 0)) < 0) {
    throw SocketException("Accept failed (accept())", true);
  }

  return new TCPSocket(newConnSD);
}

void TCPServerSocket::setListen(int queueLen) throw(SocketException) {
  if (listen(sockDesc, queueLen) < 0) {
    throw SocketException("Set listening socket failed (listen())",
true);
  }
}

// UDPSocket Code

UDPSocket::UDPSocket() throw(SocketException) :
CommunicatingSocket(SOCK_DGRAM,
    IPPROTO_UDP) {
  setBroadcast();
}

UDPSocket::UDPSocket(unsigned short localPort)  throw(SocketException)
:
    CommunicatingSocket(SOCK_DGRAM, IPPROTO_UDP) {
  setLocalPort(localPort);
  setBroadcast();
```

```
}

UDPSocket::UDPSocket(const string &localAddress, unsigned short
localPort)
     throw(SocketException) : CommunicatingSocket(SOCK_DGRAM,
IPPROTO_UDP) {
  setLocalAddressAndPort(localAddress, localPort);
  setBroadcast();
}

void UDPSocket::setBroadcast() {
  // If this fails, we'll hear about it when we try to send.  This
will allow
  // system that cannot broadcast to continue if they don't plan to
broadcast
  int broadcastPermission = 1;
  setsockopt(sockDesc, SOL_SOCKET, SO_BROADCAST,
             (raw_type *) &broadcastPermission,
sizeof(broadcastPermission));
}

void UDPSocket::disconnect() throw(SocketException) {
  sockaddr_in nullAddr;
  memset(&nullAddr, 0, sizeof(nullAddr));
  nullAddr.sin_family = AF_UNSPEC;

  // Try to disconnect
  if (::connect(sockDesc, (sockaddr *) &nullAddr, sizeof(nullAddr)) <
0) {
   #ifdef WIN32
    if (errno != WSAEAFNOSUPPORT) {
   #else
    if (errno != EAFNOSUPPORT) {
   #endif
      throw SocketException("Disconnect failed (connect())", true);
    }
  }
}

void UDPSocket::sendTo(const void *buffer, int bufferLen,
    const string &foreignAddress, unsigned short foreignPort)
    throw(SocketException) {
  sockaddr_in destAddr;
  fillAddr(foreignAddress, foreignPort, destAddr);

  // Write out the whole buffer as a single message.
  if (sendto(sockDesc, (raw_type *) buffer, bufferLen, 0,
             (sockaddr *) &destAddr, sizeof(destAddr)) != bufferLen) {
    throw SocketException("Send failed (sendto())", true);
  }
}

int UDPSocket::recvFrom(void *buffer, int bufferLen, string
&sourceAddress,
    unsigned short &sourcePort) throw(SocketException) {
  sockaddr_in clntAddr;
  socklen_t addrLen = sizeof(clntAddr);
  int rtn;
  if ((rtn = recvfrom(sockDesc, (raw_type *) buffer, bufferLen, 0,
                      (sockaddr *) &clntAddr, (socklen_t *) &addrLen))
< 0) {
```

```
    throw SocketException("Receive failed (recvfrom())", true);
  }
  sourceAddress = inet_ntoa(clntAddr.sin_addr);
  sourcePort = ntohs(clntAddr.sin_port);

  return rtn;
}

void UDPSocket::setMulticastTTL(unsigned char multicastTTL)
throw(SocketException) {
  if (setsockopt(sockDesc, IPPROTO_IP, IP_MULTICAST_TTL,
                 (raw_type *) &multicastTTL, sizeof(multicastTTL)) <
0) {
    throw SocketException("Multicast TTL set failed (setsockopt())",
true);
  }
}

void UDPSocket::joinGroup(const string &multicastGroup)
throw(SocketException) {
  struct ip_mreq multicastRequest;

  multicastRequest.imr_multiaddr.s_addr =
inet_addr(multicastGroup.c_str());
  multicastRequest.imr_interface.s_addr = htonl(INADDR_ANY);
  if (setsockopt(sockDesc, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                 (raw_type *) &multicastRequest,
                 sizeof(multicastRequest)) < 0) {
    throw SocketException("Multicast group join failed
(setsockopt())", true);
  }
}

void UDPSocket::leaveGroup(const string &multicastGroup)
throw(SocketException) {
  struct ip_mreq multicastRequest;

  multicastRequest.imr_multiaddr.s_addr =
inet_addr(multicastGroup.c_str());
  multicastRequest.imr_interface.s_addr = htonl(INADDR_ANY);
  if (setsockopt(sockDesc, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                 (raw_type *) &multicastRequest,
                 sizeof(multicastRequest)) < 0) {
    throw SocketException("Multicast group leave failed
(setsockopt())", true);
  }
}
```

Los archivos de tipo cabecera son:

- **PracticalSocket.h**

```
#ifndef __PRACTICALSOCKET_INCLUDED__
#define __PRACTICALSOCKET_INCLUDED__

#include <string>            // For string
#include <exception>         // For exception class
```

```
using namespace std;

/**
 *   Signals a problem with the execution of a socket call.
 */
class SocketException : public exception {
public:
  /**
   *   Construct a SocketException with a explanatory message.
   *   @param message explanatory message
   *   @param incSysMsg true if system message (from strerror(errno))
   *   should be postfixed to the user provided message
   */
  SocketException(const string &message, bool inclSysMsg = false)
throw();

  /**
   *   Provided just to guarantee that no exceptions are thrown.
   */
  ~SocketException() throw();

  /**
   *   Get the exception message
   *   @return exception message
   */
  const char *what() const throw();

private:
  string userMessage;  // Exception message
};

/**
 *   Base class representing basic communication endpoint
 */
class Socket {
public:
  /**
   *   Close and deallocate this socket
   */
  ~Socket();

  /**
   *   Get the local address
   *   @return local address of socket
   *   @exception SocketException thrown if fetch fails
   */
  string getLocalAddress() throw(SocketException);

  /**
   *   Get the local port
   *   @return local port of socket
   *   @exception SocketException thrown if fetch fails
   */
  unsigned short getLocalPort() throw(SocketException);

  /**
   *   Set the local port to the specified port and the local address
   *   to any interface
   *   @param localPort local port
   *   @exception SocketException thrown if setting local port fails
```

```
   */
  void setLocalPort(unsigned short localPort) throw(SocketException);

  /**
   *   Set the local port to the specified port and the local address
   *   to the specified address.  If you omit the port, a random port
   *   will be selected.
   *   @param localAddress local address
   *   @param localPort local port
   *   @exception SocketException thrown if setting local port or
address fails
   */
  void setLocalAddressAndPort(const string &localAddress,
    unsigned short localPort = 0) throw(SocketException);

  /**
   *   If WinSock, unload the WinSock DLLs; otherwise do nothing.  We
ignore
   *   this in our sample client code but include it in the library
for
   *   completeness.  If you are running on Windows and you are
concerned
   *   about DLL resource consumption, call this after you are done
with all
   *   Socket instances.  If you execute this on Windows while some
instance of
   *   Socket exists, you are toast.  For portability of client code,
this is
   *   an empty function on non-Windows platforms so you can always
include it.
   *   @param buffer buffer to receive the data
   *   @param bufferLen maximum number of bytes to read into buffer
   *   @return number of bytes read, 0 for EOF, and -1 for error
   *   @exception SocketException thrown WinSock clean up fails
   */
  static void cleanUp() throw(SocketException);

  /**
   *   Resolve the specified service for the specified protocol to the
   *   corresponding port number in host byte order
   *   @param service service to resolve (e.g., "http")
   *   @param protocol protocol of service to resolve.  Default is
"tcp".
   */
  static unsigned short resolveService(const string &service,
                                       const string &protocol =
"tcp");

private:
  // Prevent the user from trying to use value semantics on this
object
  Socket(const Socket &sock);
  void operator=(const Socket &sock);

protected:
  int sockDesc;              // Socket descriptor
  Socket(int type, int protocol) throw(SocketException);
  Socket(int sockDesc);
};

/**
```

```
 *   Socket which is able to connect, send, and receive
 */
class CommunicatingSocket : public Socket {
public:
  /**
   *   Establish a socket connection with the given foreign
   *   address and port
   *   @param foreignAddress foreign address (IP address or name)
   *   @param foreignPort foreign port
   *   @exception SocketException thrown if unable to establish
connection
   */
  void connect(const string &foreignAddress, unsigned short
foreignPort)
    throw(SocketException);

  /**
   *   Write the given buffer to this socket.  Call connect() before
   *   calling send()
   *   @param buffer buffer to be written
   *   @param bufferLen number of bytes from buffer to be written
   *   @exception SocketException thrown if unable to send data
   */
  void send(const void *buffer, int bufferLen) throw(SocketException);

  /**
   *   Read into the given buffer up to bufferLen bytes data from this
   *   socket.  Call connect() before calling recv()
   *   @param buffer buffer to receive the data
   *   @param bufferLen maximum number of bytes to read into buffer
   *   @return number of bytes read, 0 for EOF, and -1 for error
   *   @exception SocketException thrown if unable to receive data
   */
  int recv(void *buffer, int bufferLen) throw(SocketException);

  /**
   *   Get the foreign address.  Call connect() before calling recv()
   *   @return foreign address
   *   @exception SocketException thrown if unable to fetch foreign
address
   */
  string getForeignAddress() throw(SocketException);

  /**
   *   Get the foreign port.  Call connect() before calling recv()
   *   @return foreign port
   *   @exception SocketException thrown if unable to fetch foreign
port
   */
  unsigned short getForeignPort() throw(SocketException);

protected:
  CommunicatingSocket(int type, int protocol) throw(SocketException);
  CommunicatingSocket(int newConnSD);
};

/**
 *   TCP socket for communication with other TCP sockets
 */
class TCPSocket : public CommunicatingSocket {
public:
```

```
  /**
   *   Construct a TCP socket with no connection
   *   @exception SocketException thrown if unable to create TCP
socket
   */
  TCPSocket() throw(SocketException);

  /**
   *   Construct a TCP socket with a connection to the given foreign
address
   *   and port
   *   @param foreignAddress foreign address (IP address or name)
   *   @param foreignPort foreign port
   *   @exception SocketException thrown if unable to create TCP
socket
   */
  TCPSocket(const string &foreignAddress, unsigned short foreignPort)
      throw(SocketException);

private:
  // Access for TCPServerSocket::accept() connection creation
  friend class TCPServerSocket;
  TCPSocket(int newConnSD);
};

/**
 *   TCP socket class for servers
 */
class TCPServerSocket : public Socket {
public:
  /**
   *   Construct a TCP socket for use with a server, accepting
connections
   *   on the specified port on any interface
   *   @param localPort local port of server socket, a value of zero
will
   *                    give a system-assigned unused port
   *   @param queueLen maximum queue length for outstanding
   *                    connection requests (default 5)
   *   @exception SocketException thrown if unable to create TCP
server socket
   */
  TCPServerSocket(unsigned short localPort, int queueLen = 5)
      throw(SocketException);

  /**
   *   Construct a TCP socket for use with a server, accepting
connections
   *   on the specified port on the interface specified by the given
address
   *   @param localAddress local interface (address) of server socket
   *   @param localPort local port of server socket
   *   @param queueLen maximum queue length for outstanding
   *                    connection requests (default 5)
   *   @exception SocketException thrown if unable to create TCP
server socket
   */
  TCPServerSocket(const string &localAddress, unsigned short
localPort,
      int queueLen = 5) throw(SocketException);
```

```
  /**
   *   Blocks until a new connection is established on this socket or
error
   *   @return new connection socket
   *   @exception SocketException thrown if attempt to accept a new
connection fails
   */
  TCPSocket *accept() throw(SocketException);

private:
  void setListen(int queueLen) throw(SocketException);
};

/**
  *   UDP socket class
  */
class UDPSocket : public CommunicatingSocket {
public:
  /**
   *   Construct a UDP socket
   *   @exception SocketException thrown if unable to create UDP
socket
   */
  UDPSocket() throw(SocketException);

  /**
   *   Construct a UDP socket with the given local port
   *   @param localPort local port
   *   @exception SocketException thrown if unable to create UDP
socket
   */
  UDPSocket(unsigned short localPort) throw(SocketException);

  /**
   *   Construct a UDP socket with the given local port and address
   *   @param localAddress local address
   *   @param localPort local port
   *   @exception SocketException thrown if unable to create UDP
socket
   */
  UDPSocket(const string &localAddress, unsigned short localPort)
      throw(SocketException);

  /**
   *   Unset foreign address and port
   *   @return true if disassociation is successful
   *   @exception SocketException thrown if unable to disconnect UDP
socket
   */
  void disconnect() throw(SocketException);

  /**
   *   Send the given buffer as a UDP datagram to the
   *   specified address/port
   *   @param buffer buffer to be written
   *   @param bufferLen number of bytes to write
   *   @param foreignAddress address (IP address or name) to send to
   *   @param foreignPort port number to send to
   *   @return true if send is successful
   *   @exception SocketException thrown if unable to send datagram
   */
```

```
  void sendTo(const void *buffer, int bufferLen, const string
&foreignAddress,
            unsigned short foreignPort) throw(SocketException);

  /**
   *   Read read up to bufferLen bytes data from this socket.  The
given buffer
   *   is where the data will be placed
   *   @param buffer buffer to receive data
   *   @param bufferLen maximum number of bytes to receive
   *   @param sourceAddress address of datagram source
   *   @param sourcePort port of data source
   *   @return number of bytes received and -1 for error
   *   @exception SocketException thrown if unable to receive datagram
   */
  int recvFrom(void *buffer, int bufferLen, string &sourceAddress,
               unsigned short &sourcePort) throw(SocketException);

  /**
   *   Set the multicast TTL
   *   @param multicastTTL multicast TTL
   *   @exception SocketException thrown if unable to set TTL
   */
  void setMulticastTTL(unsigned char multicastTTL)
throw(SocketException);

  /**
   *   Join the specified multicast group
   *   @param multicastGroup multicast group address to join
   *   @exception SocketException thrown if unable to join group
   */
  void joinGroup(const string &multicastGroup) throw(SocketException);

  /**
   *   Leave the specified multicast group
   *   @param multicastGroup multicast group address to leave
   *   @exception SocketException thrown if unable to leave group
   */
  void leaveGroup(const string &multicastGroup)
throw(SocketException);

private:
  void setBroadcast();
};

#endif
```

- **BroadcastReceiver.h**

```
#include "PracticalSocket.h"  // For UDPSocket and SocketException
#include <iostream>           // For cout and cerr
#include <cstdlib>            // For atoi()
#include <stdio.h>
#include <string>           // For string
#include <exception>        // For exception class

class PacketReceiver

{
```

```
public:
char* receiver(int argc, unsigned short echoServPort);

};
```

La ejecución se hará a través de la interfaz en línea de comandos.

- **SignalSender:**

1. En primer lugar se compila el proyecto SignalSender, que es el encargado de enviar la señal.
2. Una vez compilado, se abre el símbolo del sistema y se accede a la ruta del archivo:

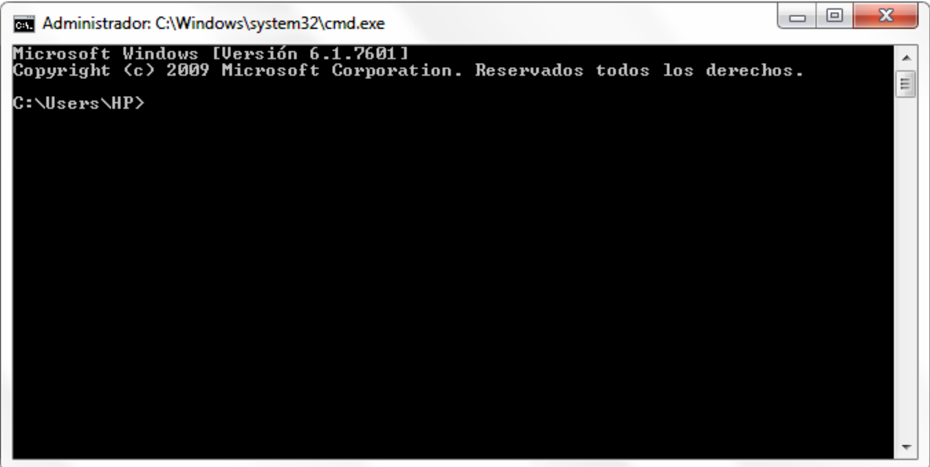


Figura 206: *Ejecución SignalSender*

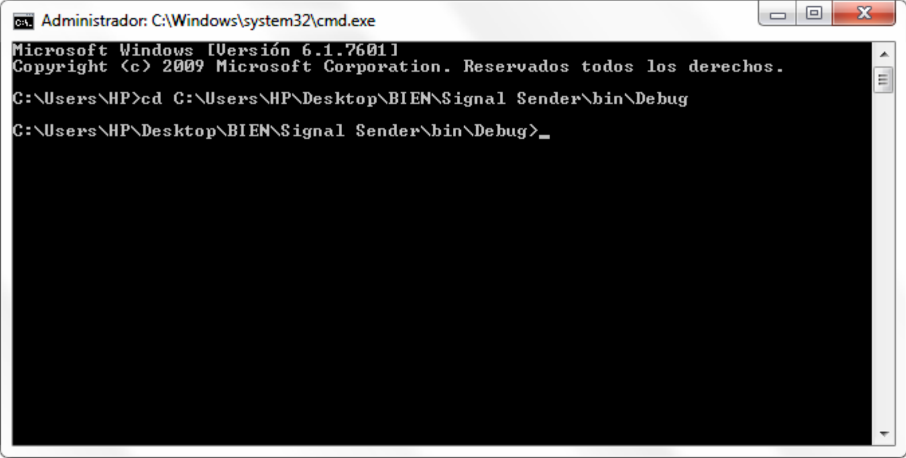


Figura 207: *Ejecución SignalSender (II)*

3. Cuando se ha accedido a la ruta del archivo, se accede a sus directorios donde se encontrará el archivo ejecutable:

```
Administrador: C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\HP>cd C:\Users\HP\Desktop\BIEN\Signal Sender\bin\Debug

C:\Users\HP\Desktop\BIEN\Signal Sender\bin\Debug>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 02FF-01FE

 Directorio de C:\Users\HP\Desktop\BIEN\Signal Sender\bin\Debug

02/07/2015  17:08    <DIR>          .
02/07/2015  17:08    <DIR>          ..
02/07/2015  17:08         1.064.558 Signal Sender.exe
               1 archivos      1.064.558 bytes
               2 dirs  359.371.976.704 bytes libres

C:\Users\HP\Desktop\BIEN\Signal Sender\bin\Debug>_
```

Figura 208: *Ejecución SignalSender (III)*

4. Finalmente, se ejecuta escogiendo como dirección IP “127.0.0.1” y el puerto 31:

```
Administrador: C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\HP>cd C:\Users\HP\Desktop\BIEN\Signal Sender\bin\Debug

C:\Users\HP\Desktop\BIEN\Signal Sender\bin\Debug>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 02FF-01FE

 Directorio de C:\Users\HP\Desktop\BIEN\Signal Sender\bin\Debug

02/07/2015  17:08    <DIR>          .
02/07/2015  17:08    <DIR>          ..
02/07/2015  17:08         1.064.558 Signal Sender.exe
               1 archivos      1.064.558 bytes
               2 dirs  359.371.976.704 bytes libres

C:\Users\HP\Desktop\BIEN\Signal Sender\bin\Debug>"Signal Sender.exe" 127.0.0.1 3
1_
```

Figura 209: *Ejecución SignalSender (IV)*

- **SignalReceiver:**

1. En primer lugar se compila el proyecto SignalReceiver, que es el encargado de recibir la señal y mostrarla por pantalla.

2. Una vez compilado, se abre el símbolo del sistema y se accede a la ruta del archivo:

```
Administrador: Símbolo del sistema
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\HP>cd C:\Users\HP\Desktop\BIEN\Signal Receiver\bin\Debug_
```

Figura 210: *Ejecución SignalReceiver*

3. Cuando se ha accedido a la ruta del archivo, se accede a sus directorios donde se encontrará el archivo ejecutable:

```
Administrador: Símbolo del sistema
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\HP>cd C:\Users\HP\Desktop\BIEN\Signal Receiver\bin\Debug

C:\Users\HP\Desktop\BIEN\Signal Receiver\bin\Debug>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 02FF-01FE

 Directorio de C:\Users\HP\Desktop\BIEN\Signal Receiver\bin\Debug

09/09/2015  19:12    <DIR>          .
09/09/2015  19:12    <DIR>          ..
09/09/2015  19:12         1.077.393 Signal Receiver.exe
               1 archivos      1.077.393 bytes
               2 dirs  359.367.405.568 bytes libres

C:\Users\HP\Desktop\BIEN\Signal Receiver\bin\Debug>
```

Figura 211: *Ejecución SignalReceiver (II)*

5. Finalmente, se ejecuta escogiend la misma dirección IP y puerto que en el programa anterior:

```
Administrador: Símbolo del sistema
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\HP>cd C:\Users\HP\Desktop\BIEN\Signal Receiver\bin\Debug

C:\Users\HP\Desktop\BIEN\Signal Receiver\bin\Debug>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 02FF-01FE

 Directorio de C:\Users\HP\Desktop\BIEN\Signal Receiver\bin\Debug

09/09/2015  19:12    <DIR>          .
09/09/2015  19:12    <DIR>          ..
09/09/2015  19:12         1.077.393 Signal Receiver.exe
               1 archivos      1.077.393 bytes
               2 dirs  359.360.704.512 bytes libres

C:\Users\HP\Desktop\BIEN\Signal Receiver\bin\Debug>"Signal Receiver.exe" 127.0.0
.1 31

C:\Users\HP\Desktop\BIEN\Signal Receiver\bin\Debug>
```

Figura 212: *Ejecución SignalReceiver (III)*

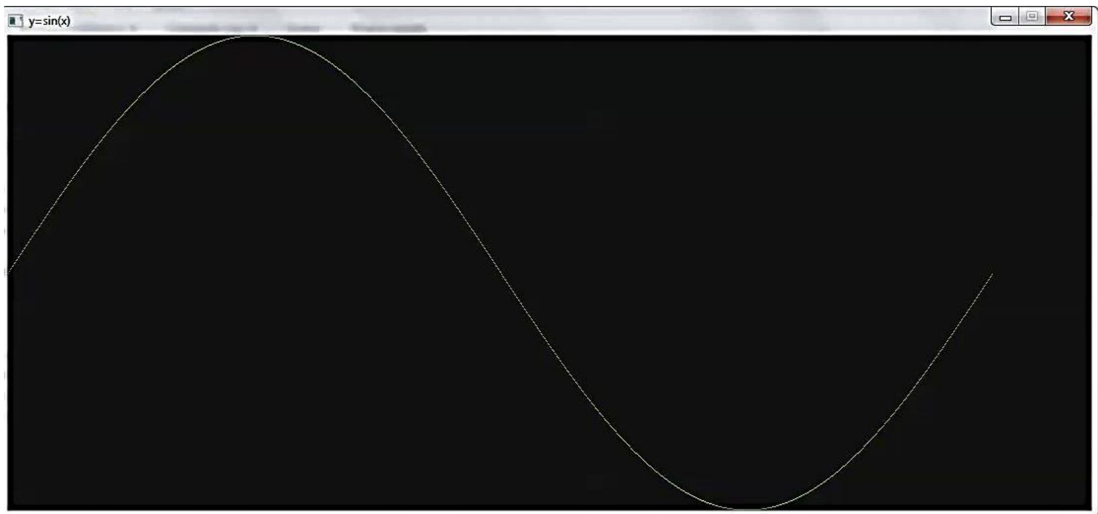


Figura 213: *Resultado mostrar imagen por pantalla*

(*)NOTA: Se ha utilizado como base para el código que establece la comunicación el ejemplo presente en la siguiente página: http://cs.ecs.baylor.edu/~donahoo/practical/CSockets/practical/

5. SDL y Raspberry PI

A continuación se estudiará SDL para Linux. Para ello se empleará una Raspberry Pi 2 Modelo B.

5.1 Raspberry PI

Raspberry Pi es un pequeño ordenador de bajo coste desarrollado en Reino Unido por la Fundación Raspberry Pi con el objetivo de estimular la enseñanza de ciencias de la computación en las escuelas.

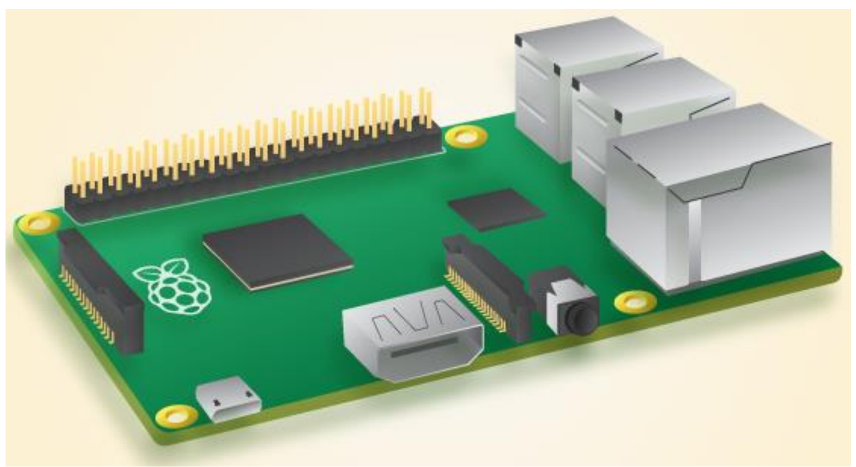


Figura 214: *Raspberry Pi 2*

Se trata de una pequeña placa base de 85 x 54 milímetros en el que se aloja un chip Broadcom BCM2835 con procesador ARM de hasta 1 GHz de velocidad (modo Turbo haciendo overclock), GPU VideoCore IV y 512 Mbytes de memoria RAM (Las primeras placas contaban con sólo 256MB de RAM).

Cuenta con entradas para teclado y ratón, así como conexión a Internet y pines GPIO (General Purpose Input/Output), de tal manera que se puede interactuar con la placa mediante sensores o botones. Por otra parte, reproduce vídeo en alta definición y también sirve como procesador de textos o para jugar, pero realmente está pensada para programar.

En la placa se cuenta además con una salida de vídeo y audio a través de un conector HDMI, con lo que se conseguirá conectar la tarjeta tanto a televisores como a monitores que cuenten con dicha conexión. En cuanto a vídeo se refiere, también cuenta con una salida de vídeo compuesto y una salida de audio a través de un minijack. Posee una conexión ethernet 10/100 y, si bien es cierto que podría echarse en falta una conexión Wi-Fi, gracias a los dos puertos USB incluidos puede suplirse dicha carencia con un adaptador WIFI si fuera necesario.

Los puertos tienen una limitación de corriente, por lo que si quiere conectarse discos duros u otros dispositivos habrá que hacerlo a través de un hub USB con alimentación. En su parte

inferior cuenta con un lector de tarjetas SD (microSD para los modelos A+, B+ y Pi 2), lo que abarata enormemente su precio y da la posibilidad de instalar un sistema operativo en una tarjeta de memoria de 4 GB o más (clase 4 o clase 10).

Así, también es posible minimizar el espacio necesario para tener todo un ordenador en un volumen mínimo.

La placa Raspberry Pi se entrega sin ningún Sistema Operativo; hay que descargarlo e instalarlo sobre una tarjeta SD / microSD que se introducirá en la ranura de la Raspberry Pi. Entre estos programas se encuentra BerryBoot, que se encarga de todo el trabajo de instalación del software desde la propia Raspberry Pi. Una vez copiados los archivos a la tarjeta SD / microSD, se introduce en la Raspberry Pi y BerryBoot permite elegir el Sistema Operativo descargándolo desde internet.

Otra opción interesante es Noobs, una aplicación que facilita la instalación de diversas distribuciones Linux. Noobs hace innecesario el acceso a Internet durante la instalación en su versión Full. Tan sólo habrá que descargar Noobs y descomprimirlo en una tarjeta SD / MicroSD de al menos 4GB de capacidad. Al hacerlo se dará la opción de instalar soluciones como Raspbian, Arch Linux, RaspBMC, Pidora u OpenELEC sin problemas.

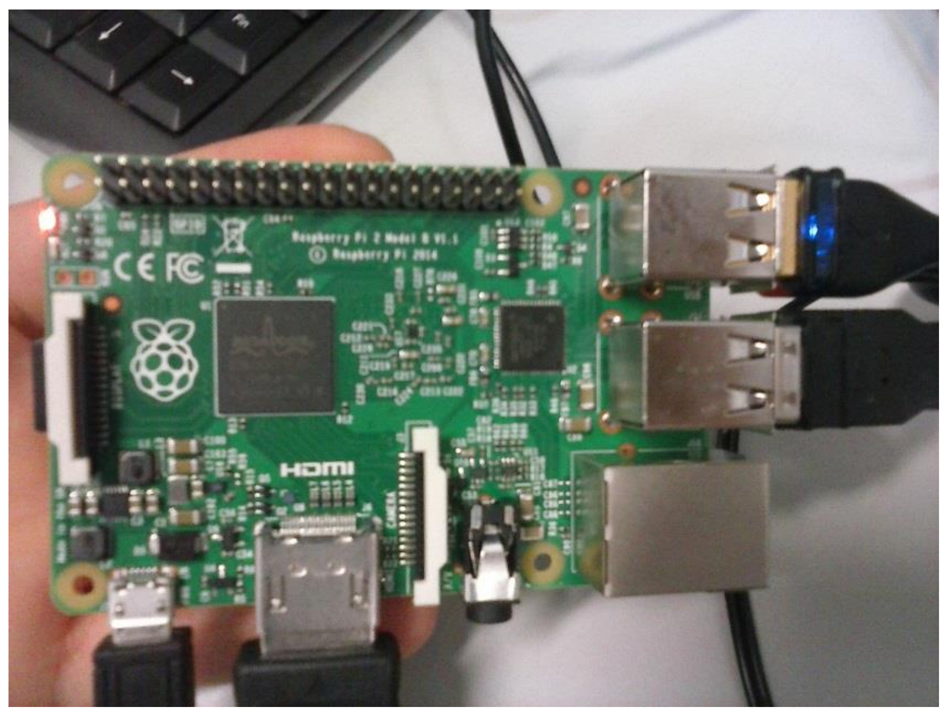


***Figura 215:** Raspberry Pi 2 empleada*

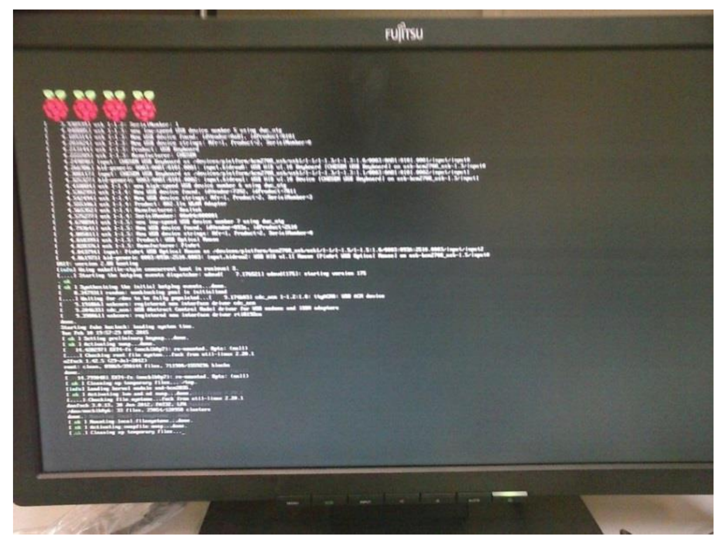


Figura 216: *Encendido Raspberry Pi 2*

```
[ ok ] Configuring network interfaces...done.
[ ok ] Cleaning up temporary files....
[ ok ] Setting up ALSA...done.
[info] Setting console screen modes.
[info] Skipping font and keymap setup (handled by console-setup).
[ ok ] Setting up console font and keymap...done.
[ ok ] Checking if shift key is held down: No. Switching to ondemand sca
[ ok ] Setting up X socket directories... /tmp/.X11-unix /tmp/.ICE-unix.
INIT: Entering runlevel: 2
[info] Using makefile-style concurrent boot in runlevel 2.
[ ok ] Network Interface Plugging Daemon...skip eth0...skip wlan0...done.
[ ok ] Starting enhanced syslogd: rsyslogd.
[info] Initializing cgroups.
[warn] Kernel lacks cgroups or memory controller not available, not starti
Starting dphys-swapfile swapfile setup ...
want /var/swap=100MByte, checking existing: keeping it
done.
[ ok ] Starting periodic command scheduler: cron.
[ ok ] Starting NTP server: ntpd.
[ ok ] Starting system message bus: dbus.
[ ok ] Starting OpenBSD Secure Shell server: sshd.

Raspbian GNU/Linux 7 raspberrypi tty1

raspberrypi login: _
```

Figura 217: *Encendido Raspberry Pi 2 (II)*

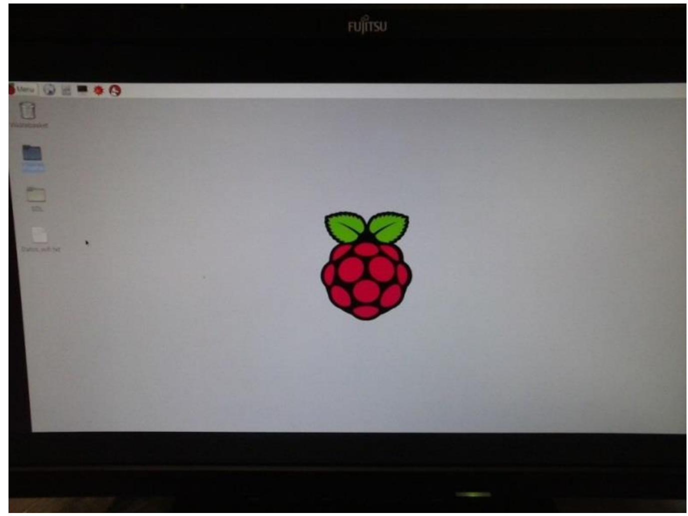


Figura 218: *Encendido Raspberry Pi 2 (III)*

5.2 Configuración SDL2.0 para Linux

- En primer lugar, se inicia Code::Blocks y se crea un nuevo proyecto vacío:

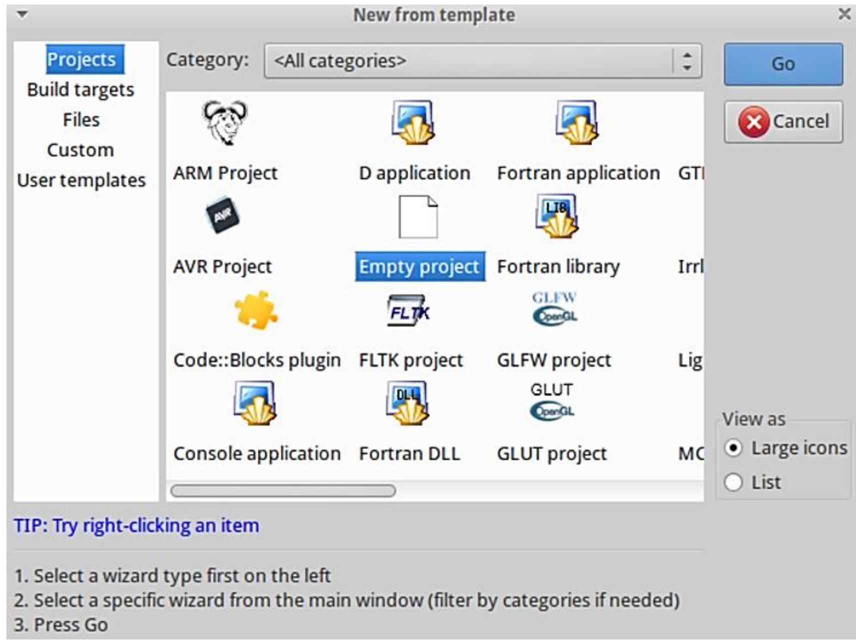


Figura 219: *Configuración SDL Linux*

- En segundo lugar, se accede a sus propiedades:

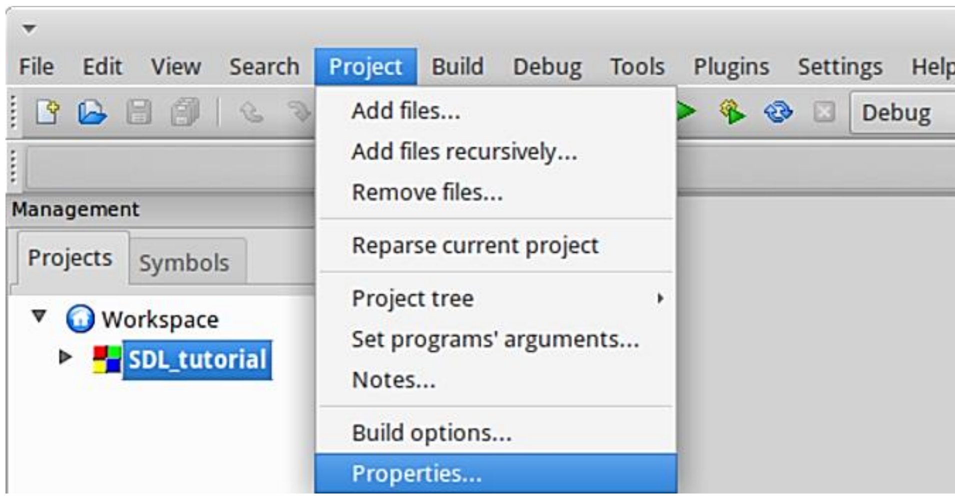


Figura 220: *Configuración SDL Linux (II)*

- A continuación, en las propiedades del proyecto, hacer click en Build Options:

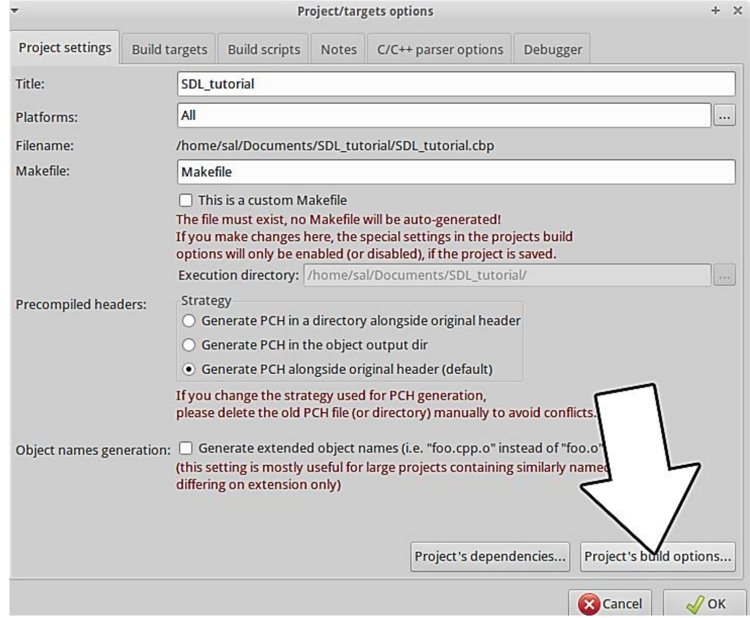


Figura 221: *Configuración SDL Linux (III)*

- Para compilar el código hay que especificar al compilador las librerías a enlazar. Para ello se copia la frase <lSDL2> en Linker Settings.

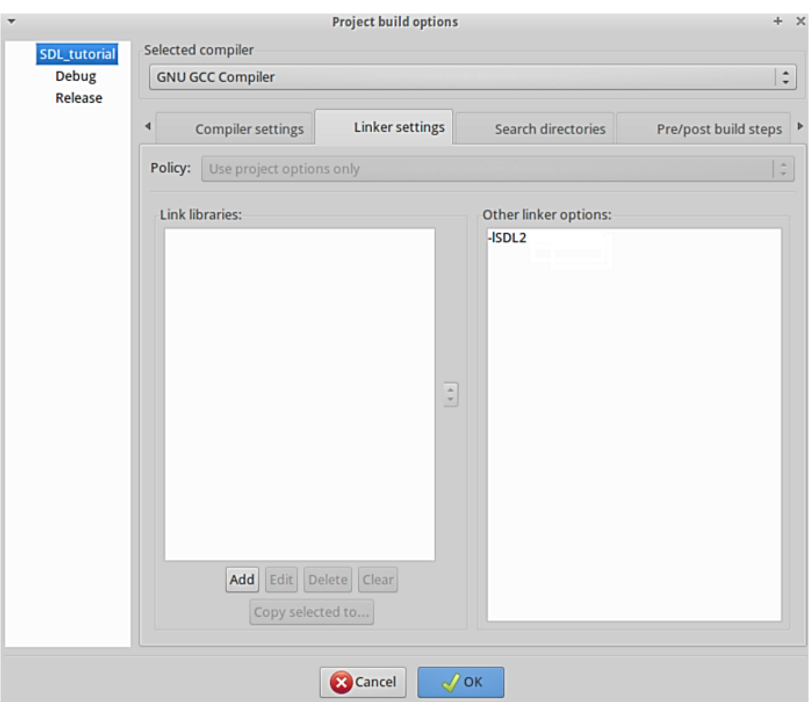


Figura 222: *Configuración SDL Linux (IV)*

5.3 Ejemplos SDL para Linux

5.3.1 Vibración Joystick

Una vez conocido cómo usar los joysticks con SDL para Windows, se empleará dicha librería para hacer temblar el controlador.

```
//Game Controller 1 handler with force feedback
SDL_Joystick* gGameController = NULL;
SDL_Haptic* gControllerHaptic = NULL;
```

Un dispositivo de tecnología de tacto proporciona un tipo de retroalimentación física. En este caso, hace que el controlador tiemble. El tipo de datos para dispositivos de tecnología de tacto se denomina **SDL_haptic**.

```
	//Initialize SDL
	if( SDL_Init( SDL_INIT_VIDEO | SDL_INIT_JOYSTICK |
SDL_INIT_HAPTIC ) < 0 )
	{
		printf( "SDL could not initialize! SDL Error: %s\n",
SDL_GetError() );
		success = false;
	}
```

Al igual que el subsistema joystick, es necesario asegurarse de inicializar el subsistema específico de la tecnología de tacto para poder usar dicha tecnología.

```
		//Check for joysticks
		if( SDL_NumJoysticks() < 1 )
		{
			printf( "Warning: No joysticks connected!\n" );
		}
		else
		{
			//Load joystick
			gGameController = SDL_JoystickOpen( 0 );
			if( gGameController == NULL )
			{
				printf( "Warning: Unable to open game
controller! SDL Error: %s\n", SDL_GetError() );
			}
			else
			{
				//Get controller haptic device
				gControllerHaptic =
SDL_HapticOpenFromJoystick( gGameController );
				if( gControllerHaptic == NULL )
				{
					printf( "Warning: Controller does not
support haptics! SDL Error: %s\n", SDL_GetError() );
				}
```

```
                        else
                        {
                                //Get initialize rumble
                                if( SDL_HapticRumbleInit(
gControllerHaptic ) < 0 )
                                {
                                        printf( "Warning: Unable to
initialize rumble! SDL Error: %s\n", SDL_GetError() );
                                }
                        }
                }
        }
```

Una vez inicializado el joystick, se obtiene el dispositivo de tecnología de tacto para el joystick empleando **SDL_HapticOpenFromJoystick** en un joystick abierto. Si el dispositivo de tecnología de tacto se obtiene del controlador, es necesario inicializar la vibración mediante **SDL_HapticRumbleInit**.

```
void close()
{
        //Free loaded images
        gSplashTexture.free();

        //Close game controller with haptics
        SDL_HapticClose( gControllerHaptic );
        SDL_JoystickClose( gGameController );
        gGameController = NULL;
        gControllerHaptic = NULL;

        //Destroy window
        SDL_DestroyRenderer( gRenderer );
        SDL_DestroyWindow( gWindow );
        gWindow = NULL;
        gRenderer = NULL;

        //Quit SDL subsystems
        IMG_Quit();
        SDL_Quit();
}
```

Cuando se ha terminado con el dispositivo de tecnología de tacto, se llama a **SDL_HapticClose**.

```
                        //Handle events on queue
                        while( SDL_PollEvent( &e ) != 0 )
                        {
                                //User requests quit
                                if( e.type == SDL_QUIT )
                                {
                                        quit = true;
                                }
                                //Joystick button press
                                else if( e.type == SDL_JOYBUTTONDOWN )
                                {
                                        //Play rumble at 75% strenght for
```

```
500 milliseconds
                                    if( SDL_HapticRumblePlay(
gControllerHaptic, 0.75, 500 ) != 0 )
                                    {
                                          printf( "Warning: Unable to
play rumble! %s\n", SDL_GetError() );
                                    }
                              }
                        }
```

Para que realmente vibre el controlador es necesario hacerle reproducir algún tipo de vibración. La manera más sencilla de que vibre el controlador es llamando a **SDL_HapticRumblePlay**, que toma del dispositivo de tecnología de tacto la fuerza en porcentaje y la duración de la vibración. En este caso se controlará la vibración al 75% de fuerza durante medio segundo siempre que ocurra **SDL_JoyButtonEvent** (evento de pulsación de botón)

Figura 223: *Vibración joystick*

5.3.2 SDL y Modern OpenGL

```
//Using SDL, SDL OpenGL, GLEW, standard IO, and strings
#include <SDL.h>
#include <gl\glew.h>
#include <SDL_opengl.h>
#include <gl\glu.h>
#include <stdio.h>
#include <string>
```

En este ejemplo se utilizará la extensión Wrangle OpenGL. Ciertos sistemas operativos como Windows admiten una cantidad determinada de OpenGL por defecto. Mediante Glew, una librería de la extensión, puede obtenerse la última funcionalidad.

```
//Shader loading utility programs
void printProgramLog( GLuint program );
void printShaderLog( GLuint shader );
```

Éstas son algunas funciones personalizadas que se realizan para informar de cualquier erros al realizar los programas de sombreado.

```
//Graphics program
GLuint gProgramID = 0;
GLint gVertexPos2DLocation = -1;
GLuint gVBO = 0;
GLuint gIBO = 0;
```

La forma en la que funciona OpenGL moderno es creando programas de sombreado (gProgramID) que procesan los vértices de los atributos como posiciones (gVertexPos2DLocation). Los vértices se ponen en Vertex Buffer Objects (gVBO) y se especifica el orden en el que se dibujarán utilizando el buffer de índice de objetos.

```
//Use OpenGL 3.1 core
SDL_GL_SetAttribute( SDL_GL_CONTEXT_MAJOR_VERSION, 3 );
SDL_GL_SetAttribute( SDL_GL_CONTEXT_MINOR_VERSION, 1 );
SDL_GL_SetAttribute( SDL_GL_CONTEXT_PROFILE_MASK,
SDL_GL_CONTEXT_PROFILE_CORE );
```

Inicialización para un contexto de la versión núcleo 3.1. El núcleo 3.1 se deshace de la antigua funcionalidad. Se especifica la mayor y la menor versión como anteriormente y lo convierte en un contexto núcleo ajustando la máscara del perfil al núcleo.

```
//Create context
                    gContext = SDL_GL_CreateContext( gWindow );
                    if( gContext == NULL )
                    {
                          printf( "OpenGL context could not be created!
SDL Error: %s\n", SDL_GetError() );
                          success = false;
                    }
                    else
                    {
                          //Initialize GLEW
                          glewExperimental = GL_TRUE;
                          GLenum glewError = glewInit();
                          if( glewError != GLEW_OK )
                          {
                                printf( "Error initializing GLEW! %s\n",
glewGetErrorString( glewError ) );
                          }

                          //Use Vsync
                          if( SDL_GL_SetSwapInterval( 1 ) < 0 )
                          {
                                printf( "Warning: Unable to set VSync!
SDL Error: %s\n", SDL_GetError() );
                          }

                          //Initialize OpenGL
                          if( !initGL() )
                          {
                                printf( "Unable to initialize OpenGL!\n"
);
                                success = false;
                          }
                    }
              }
        }
```

Tras crear el context, se inicializa Glew. Dado que se quieren las últimas características, hay que establecer glewExperimental a true. Una vez se ha realizado, se llama a glewInit() para inicializar GLEW.

```
bool initGL()
{
      //Success flag
      bool success = true;

      //Generate program
      gProgramID = glCreateProgram();
```

En la función de inicialización se creará el programa de sombreado para renderizar con los datos VBO e IBO.

```
      //Create vertex shader
      GLuint vertexShader = glCreateShader( GL_VERTEX_SHADER );

      //Get vertex source
```

```
	const GLchar* vertexShaderSource[] =
	{
		"#version 140\nin vec2 LVertexPos2D; void main() {
gl_Position = vec4( LVertexPos2D.x, LVertexPos2D.y, 0, 1 ); }"
	};

	//Set vertex source
	glShaderSource( vertexShader, 1, vertexShaderSource, NULL );

	//Compile vertex source
	glCompileShader( vertexShader );

	//Check vertex shader for errors
	GLint vShaderCompiled = GL_FALSE;
	glGetShaderiv( vertexShader, GL_COMPILE_STATUS, &vShaderCompiled
);
	if( vShaderCompiled != GL_TRUE )
	{
		printf( "Unable to compile vertex shader %d!\n",
vertexShader );
		printShaderLog( vertexShader );
	  success = false;
	}
```

Aquí se está cargando un vértice de sombreado. Si el vértice no se consigue cargar y compilar, se utilizará la función de impresión de registro para comunicar el error.

```
else
	{
		//Attach vertex shader to program
		glAttachShader( gProgramID, vertexShader );


		//Create fragment shader
		GLuint fragmentShader = glCreateShader( GL_FRAGMENT_SHADER
);

		//Get fragment source
		const GLchar* fragmentShaderSource[] =
		{
			"#version 140\nout vec4 LFragment; void main() {
LFragment = vec4( 1.0, 1.0, 1.0, 1.0 ); }"
		};

		//Set fragment source
		glShaderSource( fragmentShader, 1, fragmentShaderSource,
NULL );

		//Compile fragment source
		glCompileShader( fragmentShader );

		//Check fragment shader for errors
		GLint fShaderCompiled = GL_FALSE;
		glGetShaderiv( fragmentShader, GL_COMPILE_STATUS,
&fShaderCompiled );
		if( fShaderCompiled != GL_TRUE )
		{
			printf( "Unable to compile fragment shader %d!\n",
```

```
fragmentShader );
                        printShaderLog( fragmentShader );
                        success = false;
```

Si el vértice de sombreado se carga correctamente, se añade al programa y se compila el fragment shader.

```
                else
                {
                        //Attach fragment shader to program
                        glAttachShader( gProgramID, fragmentShader );


                        //Link program
                        glLinkProgram( gProgramID );

                        //Check for errors
                        GLint programSuccess = GL_TRUE;
                        glGetProgramiv( gProgramID, GL_LINK_STATUS,
&programSuccess );
                        if( programSuccess != GL_TRUE )
                        {
                                printf( "Error linking program %d!\n",
gProgramID );
                                printProgramLog( gProgramID );
                                success = false;
```

Si se compila el fragment shader, se añade al programa de sombreado y se vincula.

```
                        else
                        {
                                //Get vertex attribute location
                                gVertexPos2DLocation = glGetAttribLocation(
gProgramID, "LVertexPos2D" );
                                if( gVertexPos2DLocation == -1 )
                                {
                                        printf( "LVertexPos2D is not a valid
glsl program variable!\n" );
                                        success = false;
                                }
                                else
                                {
                                        //Initialize clear color
                                        glClearColor( 0.f, 0.f, 0.f, 1.f );

                                        //VBO data
                                        GLfloat vertexData[] =
                                        {
                                                -0.5f, -0.5f,
                                                 0.5f, -0.5f,
                                                 0.5f,  0.5f,
                                                -0.5f,  0.5f
                                        };
```

```
                              //IBO data
                              GLuint indexData[] = { 0, 1, 2, 3 };

                              //Create VBO
                              glGenBuffers( 1, &gVBO );
                              glBindBuffer( GL_ARRAY_BUFFER, gVBO );
                              glBufferData( GL_ARRAY_BUFFER, 2 * 4 *
sizeof(GLfloat), vertexData, GL_STATIC_DRAW );

                              //Create IBO
                              glGenBuffers( 1, &gIBO );
                              glBindBuffer( GL_ELEMENT_ARRAY_BUFFER,
gIBO );
                              glBufferData( GL_ELEMENT_ARRAY_BUFFER, 4
* sizeof(GLuint), indexData, GL_STATIC_DRAW );
                        }
                  }
            }
      }

      return success;
```

Una vez está funcionando el programa de sombreado, se crea VBO e IBO.

```
void printProgramLog( GLuint program )
{
      //Make sure name is shader
      if( glIsProgram( program ) )
      {
            //Program log length
            int infoLogLength = 0;
            int maxLength = infoLogLength;

            //Get info string length
            glGetProgramiv( program, GL_INFO_LOG_LENGTH, &maxLength );

            //Allocate string
            char* infoLog = new char[ maxLength ];

            //Get info log
            glGetProgramInfoLog( program, maxLength, &infoLogLength,
infoLog );
            if( infoLogLength > 0 )
            {
                  //Print Log
                  printf( "%s\n", infoLog );
            }

            //Deallocate string
            delete[] infoLog;
      }
      else
      {
            printf( "Name %d is not a program\n", program );
      }
}
```

```
void printShaderLog( GLuint shader )
{
	//Make sure name is shader
	if( glIsShader( shader ) )
	{
		//Shader log length
		int infoLogLength = 0;
		int maxLength = infoLogLength;

		//Get info string length
		glGetShaderiv( shader, GL_INFO_LOG_LENGTH, &maxLength );

		//Allocate string
		char* infoLog = new char[ maxLength ];

		//Get info log
		glGetShaderInfoLog( shader, maxLength, &infoLogLength,
infoLog );
		if( infoLogLength > 0 )
		{
			//Print Log
			printf( "%s\n", infoLog );
		}

		//Deallocate string
		delete[] infoLog;
	}
	else
	{
		printf( "Name %d is not a shader\n", shader );
	}
}
```

Funciones de impresión. Cogen el registro de sombreado del programa de sombreado dado y lo imprimen en la consola.

```
void render()
{
	//Clear color buffer
	glClear( GL_COLOR_BUFFER_BIT );

	//Render quad
	if( gRenderQuad )
	{
		//Bind program
		glUseProgram( gProgramID );

		//Enable vertex position
		glEnableVertexAttribArray( gVertexPos2DLocation );

		//Set vertex data
		glBindBuffer( GL_ARRAY_BUFFER, gVBO );
		glVertexAttribPointer( gVertexPos2DLocation, 2, GL_FLOAT,
GL_FALSE, 2 * sizeof(GLfloat), NULL );

		//Set index data and render
		glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, gIBO );
		glDrawElements( GL_TRIANGLE_FAN, 4, GL_UNSIGNED_INT, NULL
```

```
);

            //Disable vertex position
            glDisableVertexAttribArray( gVertexPos2DLocation );

            //Unbind program
            glUseProgram( NULL );
      }
}
```

En la función de renderizado, se ata el programa de sombreado y se activan las posiciones de los vértices. A continuación se ata el VBO e IBO. Finalmente se desactiva el atributo del vértice y se desata el programa.

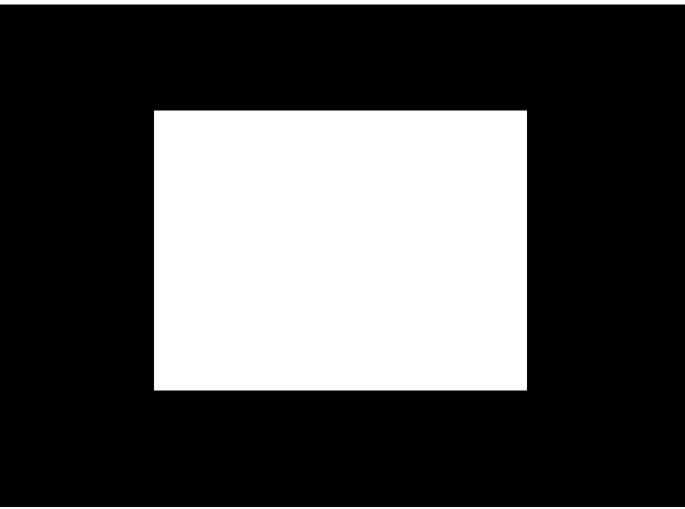


Figura 224: *Shader program*

6. Alternativas a SDL

Tal y como se comentó en la introducción, entre los principales motivos que llevaron a escoger la librería SDL se encuentra su licencia y, sobre todo, que se trata de una librería libre.

Además de ser multiplataforma, una cualidad importante de SDL es su, relativamente, poca complejidad. Proporciona las herramientas básicas sobre todo lo que tiene que hacer la librería para funcionar en cualquier máquina para programar un videojuego.

DirectX y OpenGL son dos de las grandes alternativas a SDL para realizar tareas parecidas. Sin embargo, no son soluciones definitivas ya que presentan diferentes problemas.

Figura 225: *Logo DirectX y Logo OpenGL*

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 mientras que DirectX es bastante más completo al proporcionar gráficos en 2D y 3D, entrada y salida, comunicación entre redes. El problema es que sólo es compatible con sistemas Windows.

SDL no termina aquí, sino que presenta un amplio futuro. Existen numerosos proyectos para mejorar la librería a los que cualquier usuario puede contribuir. Actualmente 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.

Para programadores no experimentados, SDL tiene mucho que aportar, ya que permite que el usuario se centre en la tarea de desarrollo del juego, obviando los detalles del sistema operativo y, además, puede utilizarla de manera gratuita. Para programadores experimentados, SDL proporciona una biblioteca clara que ahorra al usuario mucho tiempo en el desarrollo de la aplicación sin perder estabilidad y que, además, es portable entre varias máquinas y sistemas operativos.

Además de OpenGL y DirectX también hay otras posibilidades, como pueden ser Pygame y Box2D.

Pygame es un conjunto de módulos del lenguaje Python diseñados para la creación de videojuegos en dos dimensiones de manera sencilla. Orientado al manejo de sprites, Pygame es altamente portable y puede ejecutarse en casi cualquier plataforma y sistema operativo. Pygame se distribuye bajo licencia LGPL 2.1, por tanto permite la distribución, tanto libre como comercial, del contenido creado con ella.

Figura 226: *Logo pygame*

Box2D es un motor en C++ para la simulación de sólidos rígidos en 2D. Se distribuye bajo licencia zlib. La librería incluye numerosos módulos que permiten simular multitud de comportamientos tales como la detección de colisiones continuas, sistemas de sólidos con diversos tipos de ligaduras, apilamientos de diferentes formas geométricas, sistemas en árbol para la gestión de cuerpos, etc.

Toda la funcionalidad de la librería esta orientada a la simulación física y numérica y, por tanto, es necesario el uso de otras herramientas para visualizar los elementos en pantalla.

Figura 227: *Logo Box2D*

7. Conclusiones y Líneas futuras de trabajo

Una vez finalizado este estudio, puede decirse que:

- Se ha analizado la librería SDL tanto en dos sistemas operativos: Window y Linux.
- Se han realizado 48 ejemplos para Windows.
- Se han realizado 2 ejemplos para Linux.
- Se han empleado 116 funciones en total.
- Se ha estudiado la comunicación UDP mediante la realización de una aplicación consistente en el envío de una señal y tras su recepción mostrarla en pantalla mediante la librería SDL.

Partiendo de este estudio, podría realizarse un futuro trabajo en el que tratase información de señales biomédicas. En este caso se implementaría una aplicación SCADA libre para utilizar en equipos médicos o para la visualización de la información.

8. Bibliografía

"SDL Game Development" Shaun Mitchell 256 pp. Packt Publishing (June 24, 2013) ISBN-10: 1849696829 ISBN-13: 978-1849696821

Beginning Game Programming v2.0 Lazy Foo Production

geekytheory.com/tutorial-raspberry-pi-1-el-primer-encendido/

pygame.org/hifi.html

box2d.org/