



Módulo de gestión gráfica

Metodología de la Programación II

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada



Este documento se ha realizado como apoyo a las prácticas de la asignatura Metodología de la Programación II de las titulaciones de Informática de la Universidad de Granada.

El objetivo del mismo es el desarrollo de un sistema de gestión de una interfaz gráfica de muy fácil uso. Dicho módulo está basado en el uso de las bibliotecas SDL, SDL_gfx, SDL_ttf y FreeType.

Antonio Garrido Carrillo (agarrido@decsai.ugr.es)
Javier Martínez Baena (jbaena@decsai.ugr.es)

Índice de contenido

1.Introducción.....	3
1.1.Un ejemplo sencillo: Hola mundo.....	4
2.Instalación del módulo.....	5
2.1.Requisitos previos.....	5
2.2.Aulas de prácticas.....	6
3.¿Cómo se usa la biblioteca gráfica?.....	7
3.1.Ejemplo: creando la aplicación Hola mundo.....	7
3.2.Creando una ventana.....	7
4.Introducción a los sistemas gráficos.....	8
4.1.Los puntos (píxeles).....	8
4.2.Los colores.....	8
4.3.Los dibujos.....	9
4.4.El texto.....	9
4.4.1.Los tipos de letra vectoriales.....	9
5.Dibujando formas sencillas.....	11
6.Trabajando con Bitmaps.....	14
7.El teclado y el ratón.....	15
8.Escribiendo texto.....	17
8.1.Formato raster.....	17
8.2.Formato vectorial.....	17
9.La gestión del tiempo.....	19
10.Listado completo de funciones y tipos disponibles.....	20
10.1.Listado de tipos y constantes.....	20
10.2.Funciones para la gestión de la interfaz gráfica.....	21
10.3.Funciones de dibujo.....	22
10.4.Funciones para la gestión del teclado y el ratón.....	24
10.5.Funciones para trabajar con bitmaps (imágenes).....	25
10.6.Funciones para trabajar con texto.....	26
10.7.Funciones para la gestión del tiempo.....	27
10.8.Otras definiciones de interés definidas en SDL.....	28
11.Referencias.....	29
Índice alfabético.....	30

1. Introducción

Generalmente, las asignaturas de programación se ciñen al estudio de algún lenguaje de programación de alto nivel, planteándose como meta el conocimiento de las herramientas que proporciona dicho lenguaje para desarrollar soluciones a problemas del mundo real.

El uso de interfaces para la comunicación con el usuario suele quedar relegado a un segundo plano o a cursos más avanzados. Esto tiene sentido, ya que primero es necesario conocer técnicas de programación básicas y familiarizarse con los lenguajes. Además, la gestión de interfaces con el usuario tiene su propia complejidad intrínseca, por lo que su estudio a niveles tempranos distraería al estudiante de los objetivos de un curso de programación básica. Por estos motivos, la comunicación con el usuario suele hacerse mediante la consola y en modo texto, simplificándola al máximo.

Obviamente las posibilidades que ofrece la consola son limitadas, por ejemplo no permite diseñar programas con interfaces gráficas.

En este documento se plantea la creación de una biblioteca que permite al usuario crear interfaces gráficas de una manera muy simple, evitando así las complejidades que conlleva el uso de la mayoría de las bibliotecas de este tipo.

Esta biblioteca hace uso de otras, en concreto hace uso de SDL¹, SDL_gfx y SDL_ttf. A su vez, estas hacen uso de otras (X11, FreeType, ...). Pero todo esto es (casi) totalmente transparente al usuario.

La elección de SDL se debe a dos motivos. El primero de ellos es la portabilidad, es decir, existen versiones de SDL en diferentes sistemas operativos tales como GNU/Linux, Windows, Mac OS X, etc.. El segundo es la simplicidad y la versatilidad de la misma: es muy completa y es fácil de usar. Esta biblioteca está pensada para la gestión de funciones multimedia tales como los gráficos, el sonido, la red, etc. permitiendo el control de las funciones gráficas, el ratón, el teclado, el joystick, etc..

La idea de este tipo de bibliotecas es proporcionar una funcionalidad básica a través de la cual podamos tener un control total sobre los dispositivos. Por ejemplo, en la parte de gráficos, SDL proporciona primitivas tan simples como pintar un punto en la pantalla. Si queremos dibujar rectas, círculos o formas más complejas, hemos de construir algoritmos que lo hagan punto a punto. Evidentemente, si podemos pintar un punto en la pantalla entonces podemos pintar cualquier cosa, pero esta segunda parte no la proporciona SDL. Es por ello que usaremos una segunda biblioteca llamada SDL_gfx. Ésta amplía las posibilidades de SDL añadiendo primitivas para dibujar algunas formas como líneas, rectángulos, círculos, texto, etc.. Las primitivas de SDL_gfx hacen uso de las primitivas de SDL y por eso diremos que SDL_gfx se desarrolla sobre -o hace uso de- SDL.

Además, para poder trabajar con fuentes (tipos de letra) TrueType (ttf), haremos uso de SDL_ttf que, a su vez, hace uso de FreeType. Con ella podemos cargar tipos de letra a partir de ficheros .ttf y usarlos para escribir en nuestra interfaz gráfica.

El objetivo de este documento no es que el lector conozca ninguna de estas bibliotecas que, aún siendo relativamente sencillas, necesitan de un conocimiento más profundo sobre el funcionamiento del hardware multimedia y sobre algunas técnicas de programación que, en nuestro nivel de conocimientos, podríamos considerar como avanzadas. Para conseguir nuestro objetivo sin complicar excesivamente el uso de interfaces gráficas, se ha desarrollado un módulo al que llamaremos "*gráficos*", que contiene un conjunto de funciones que permiten, de una forma extremadamente sencilla, hacer uso de algunas de las posibilidades que ofrecen SDL, SDL_gfx y SDL_ttf.

Si en algún momento el alumno necesita añadir alguna funcionalidad ofrecida por SDL, que no haya sido incluida en el módulo de gráficos suministrado, puede consultar el documento [MAR] en el que se resumen, en español, algunas características de SDL. También se puede usar la documentación original de SDL [SDL]. En [GFX] hay información sobre SDL_gfx. Además, existen algunos libros sobre SDL [SER, PAZ, LOK] y estructuras de datos para la programación de videojuegos usando SDL [PEN], por si se desea profundizar en el tema.

1 SDL son las siglas de Simple Directmedia Layer (<http://www.libsdl.org>).

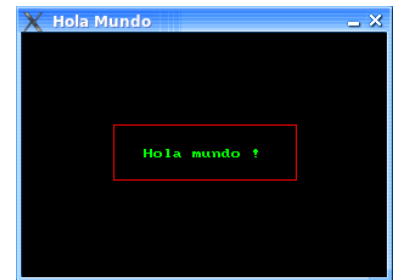
1.1. Un ejemplo sencillo: Hola mundo

La versión clásica de este conocido programa, escrita en C++, podría ser la siguiente:

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    cout << "Hola mundo" << endl;
}
```

A continuación vemos la versión de este programa haciendo uso de nuestra biblioteca gráfica:

```
#include "graficos.h"
using namespace graficos;
int main(int argc, char *argv[])
{
    CrearVentana(300,200,"Hola Mundo"); // Abrimos ventana
    Rectangulo(75,75,225,120,255,0,0); // Dibujamos rectángulo
    Texto(100,95,"Hola mundo !",0,255,0); // Mostramos un mensaje
    ObtenerTecla(); // Esperamos tecla
}
```



Podemos apreciar algunas diferencias o novedades:

- Se hace `#include` de `graficos.h` en lugar de `iostream` ya que en ese fichero están los prototipos de todas las funciones y tipos del módulo gráfico.
- Se hace uso del namespace denominado `graficos` en lugar del namespace `std`. El namespace `std` incluye funciones relacionadas con la E/S estándar y permite, por ejemplo, enviar texto a la consola (`cout`) a través del operador `<<`. En la versión gráfica no vamos a enviar texto a consola, por lo que no necesitamos ese namespace, en su lugar usaremos nuestro namespace `graficos`.
- Al comienzo de la versión gráfica debemos crear una ventana en la que poder dibujar (`CrearVentana`). Indicamos el tamaño y el título.
- Para decorar el mensaje hemos dibujado un rectángulo que envuelve al texto (`Rectangulo`).
- La escritura de texto la hacemos con la función `Texto`, en lugar de enviar datos a `cout`.
- Finalmente, ejecutamos la función `ObtenerTecla` para que no se cierre la ventana, hasta que se pulse una tecla. Si no fuera así, al finalizar el programa el sistema eliminaría la ventana y no podríamos ver el resultado.

2. Instalación del módulo

Este módulo se suministra en un fichero llamado `graficos.tgz`. Para descomprimirlo usaremos la instrucción `tar` de GNU/Linux desde la carpeta en donde queremos instalarlo:

```
tar xzvf graficos.tgz
```

Esto crea la siguiente jerarquía de carpetas:

.	
-- bin	Ejecutables
-- data	Ficheros de datos adicionales
-- fuentes	Tipos de letra de ejemplo (ficheros .ttf)
-- doc	Documentación
-- doxys	Archivos auxiliares de documentación (doxygen)
-- include	Ficheros de cabecera (ficheros .h)
-- lib	Bibliotecas (ficheros libXXX.a)
-- obj	Código objeto (ficheros .o)
`-- src	Código fuente (ficheros .cpp)

Como puede apreciarse, se mantienen en carpetas diferentes los distintos tipos de archivos: código fuente en `src/`, ficheros de cabecera en `include/`, código objeto en `obj/`, etc.

Una vez descomprimido, podemos compilar el módulo y generar la biblioteca `graficos` ejecutando `make` en el directorio raíz del módulo:

```
make
```

Además de crear la biblioteca, se compilan y enlazan algunos programas de prueba, como por ejemplo el `holamundo` de la sección anterior. Los ejecutables se encuentran en la carpeta `bin/`. Para ejecutar `holamundo` nos situaremos en la carpeta raíz del proyecto y ejecutaremos:

```
bin/holamundo
```

Además, se genera la documentación automáticamente con el programa `doxygen`, situándose en el directorio `doc/html`. Así, si desea consultarla puede abrir el archivo `doc/html/index.html` en su navegador favorito.

2.1. Requisitos previos

Lo normal será que el proceso descrito anteriormente no dé ningún problema, ya que casi todas las bibliotecas adicionales requeridas suelen venir incorporadas en las distribuciones de GNU/Linux actuales. Aún así, es posible que falte alguna biblioteca en nuestro sistema por lo que, en ese caso, habría que instalarla. Lo más sencillo es utilizar la aplicación de instalación de software que se incluya en nuestra distribución (`aptget`, `drakrpm`, `YaST`, ...). Si esto no es posible por algún motivo, siempre tenemos la posibilidad de descargar las bibliotecas desde la página web correspondiente, compilarlas, e instalarlas manualmente.

Las bibliotecas que se requieren son:

- SDL. Se puede obtener en <http://www.libsdl.org>
- SDL_gfx. Se puede obtener a través de la página de SDL (<http://www.libsdl.org/libraries.php>) o bien directamente en su web: http://www2.ferzkopp.net/joomla/index.php?option=com_content&task=blogsection&id=2&Itemid=14
- SDL_ttf. Se puede obtener a través de la página de SDL (<http://www.libsdl.org/libraries.php>) o bien directamente en su web: http://www.libsdl.org/projects/SDL_ttf
- FreeType2. Se puede obtener en <http://freetype.sourceforge.net>

Todas ellas deben instalarse en su versión para desarrollo (`dev` o `devel`) para que incluyan los ficheros de cabecera que permiten compilar nuevas aplicaciones que hagan uso de ellas.

2.2. Aulas de prácticas

En las aulas de prácticas no se encuentran instaladas estas bibliotecas adicionales. Puesto que el alumno no dispone de privilegios para poder hacer una instalación de las mismas en los directorios del sistema, será necesario que haga la instalación en su cuenta de usuario. Para ello deberá descomprimir el fichero `SDL_aulas.tgz` en la misma carpeta que `graficos.tgz`, quedando la estructura de directorios de la siguiente forma:

```
|-- graficos
|   |-- bin           Ejecutables
|   |-- data          Ficheros de datos adicionales
|   |-- `-- fuentes   Tipos de letra de ejemplo (ficheros .ttf)
|   |-- doc           Documentación
|   |-- `-- doxys      Archivos auxiliares de documentación (doxygen)
|   |-- include       Ficheros de cabecera (ficheros .h)
|   |-- lib           Bibliotecas (ficheros libXXX.a)
|   |-- obj           Código objeto (ficheros .o)
|   |-- `-- src        Código fuente (ficheros .cpp)
|
|-- SDL_aulas
|   |-- include       Ficheros de cabecera (ficheros .h)
|   |-- lib           Bibliotecas (ficheros libXXX.a)
```

3. ¿Cómo se usa la biblioteca gráfica?

Para hacer una aplicación que haga uso de esta biblioteca, debemos tener presentes dos cosas:

- En la etapa de compilación, nuestro programa debe conocer los prototipos de las funciones de la biblioteca gráfica, así como los posibles nuevos tipos de datos que se definan en ella. Esa información está en el fichero de cabecera `graficos.h` y, por lo tanto, debemos hacer un `#include` del mismo desde cualquier programa que haga uso de la biblioteca.

Además, hemos de tener presente que toda la implementación de la biblioteca gráfica está dentro del espacio de nombres `graficos`.

- En la etapa de enlazado hemos de indicarle al enlazador (a través de `g++`) que use nuestra biblioteca y todas aquellas que son usadas por la nuestra (y que por defecto no se incluyan). En particular, habrá que enlazar con `libSDL.a`, `libSDL_gfx.a`, `libSDL_ttf.a` y `libfreetype.a`.

3.1. Ejemplo: creando la aplicación Hola mundo

Veamos que opciones hay que darle al compilador para que cree la aplicación `holamundo`. Supongamos que estamos situados en la carpeta raíz del módulo gráficos y que, por lo tanto, los ficheros de cabecera están en la carpeta `include/`, el código fuente en `src/` y el código objeto en `obj/`. La compilación de `holamundo.cpp` se haría con la instrucción:

```
g++ -Wall -g -Iinclude -c src/holamundo.cpp -o obj/holamundo.o
```

y el enlazado para la creación del ejecutable se haría así:

```
g++ -o bin/holamundo obj/holamundo.o -Llib -lgraficos -lSDL -lSDL_gfx -lSDL_ttf -lfreetype
```

3.2. Creando una ventana

En el ejemplo `holamundo.cpp` vemos que lo primero que se hace, antes de dibujar o escribir, es una llamada a la función `CrearVentana`. Esto es imprescindible ya que esta función es la encargada de inicializar todo el sistema gráfico. La sintaxis es la siguiente:

```
void CrearVentana(int ancho, int alto, const char *titulo);
```

ancho: ancho (en píxeles) de la ventana.

alto: altura (en píxeles) de la ventana.

titulo: el título que aparece en la barra superior de la ventana.

SDL permite abrir una única ventana en la aplicación por lo que debemos llamar a esta función, una única vez, antes de ejecutar ninguna otra función del módulo gráfico.

4. Introducción a los sistemas gráficos

Cuando estamos trabajando en modo gráfico podemos ver la pantalla como un lienzo en el que podemos pintar cualquier cosa (texto, líneas, puntos, etc). Nuestro pincel, en este caso, serán los distintos subprogramas que permiten mostrar algo en la pantalla. Habrá subprogramas que dibujen un único punto, otros que dibujen líneas, otros que dibujen formas más complejas, etc.

Con esta biblioteca nuestro lienzo consistirá en una única ventana. La primitiva gráfica que permite crear la ventana se llama `CrearVentana`.

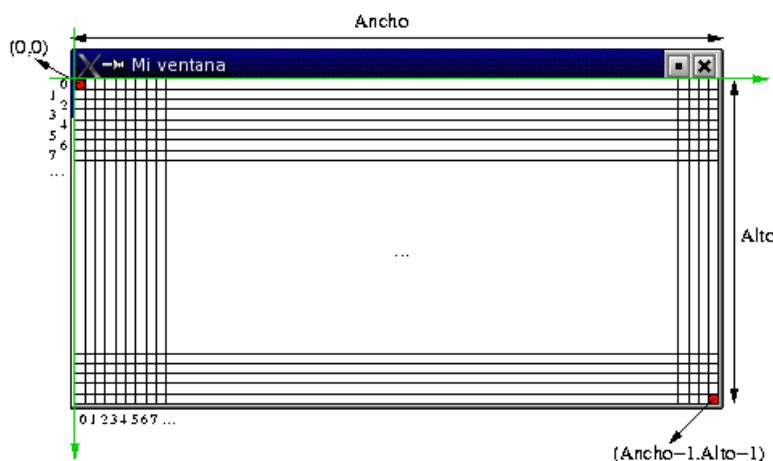
Por ejemplo, la siguiente llamada:

```
CrearVentana(400,200,"Mi ventana");
```

inicializa el sistema gráfico y crea una ventana titulada "Mi ventana" que tiene de ancho 400 puntos y de alto 200 puntos.

4.1. Los puntos (píxeles)

Este lienzo (la ventana) tiene la forma de una matriz bidimensional de manera que cada celda de esa matriz representa un punto de la pantalla (estos puntos se llaman "píxel"). Habitualmente se usa un sistema de coordenadas denominado "izquierdo"; en este sistema el (0,0) se sitúa en la esquina superior izquierda y avanzamos hacia abajo y hacia la derecha (ver figura). El tamaño del lienzo, es decir, el número total de puntos, lo decidiremos nosotros cuando inicializamos el sistema gráfico.



Puesto que en C++ los vectores y matrices comienzan a numerarse en 0, las dimensiones de una ventana de tamaño *filas* x *columnas* irán desde 0 hasta *filas-1* y *columnas-1* respectivamente.

4.2. Los colores

Para hacer un dibujo no basta con saber, únicamente, en qué posición (píxel) deseamos pintar sino que, además, hemos de indicar el color de dicho punto. El color de cada píxel se forma mediante la mezcla de tres colores básicos: rojo, verde y azul. Este sistema de color se conoce como RGB (Red, Green, Blue) y es el que habitualmente se usa para representar los colores en los monitores. Lo que hemos de hacer para decidir el color de un punto es indicar en qué proporción hemos de mezclar esos tres colores. La medida de cada color básico en esa mezcla nos la da un número que varía en el rango [0,255]. Un valor de cero indica que ese color no influye en la mezcla y un valor de 255 dice que el color influye al máximo; valores intermedios indican mayor o menor presencia del color básico en la mezcla. Por ejemplo, si deseamos obtener un píxel de color azul puro hemos de usar la combinación (rojo=0, verde=0, azul=255): con esto estamos diciendo que únicamente usaremos el color

Color	Muestra	Rojo	Verde	Azul
Amarillo		255	255	0
Naranja		255	128	0
Magenta		255	0	255
Cyan		0	255	255
Rojo		255	0	0
Verde 1		0	144	0
Verde 2		0	255	0
Verde 3		16	192	32
Violeta		176	0	176
Blanco		255	255	255
Negro		0	0	0

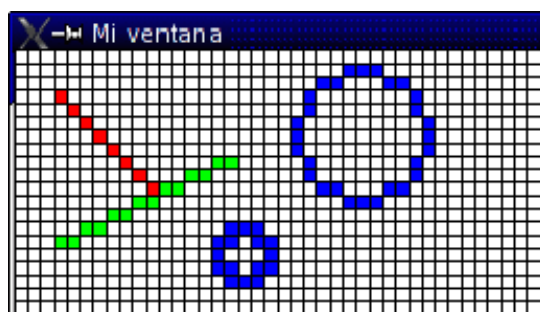
básico azul con la máxima intensidad. Si deseamos un azul menos luminoso podemos definirlo, por ejemplo, mediante la tripleta (0,0,128) indicando que sólo usaremos la componente azul pero a la mitad de intensidad. En la figura puedes ver algunos ejemplos de composiciones de color.

Observa que, usando estas tres componentes de color, cada una de ellas en el rango [0,255] podemos representar una gama de $256 \times 256 \times 256 = 16.777.216$ colores. Esta gama es más que suficiente para la visualización de cualquier imagen si tenemos en cuenta que el ojo humano tiene capacidad para distinguir aproximadamente unos 10 millones de tonos diferentes como máximo.

4.3. Los dibujos

La tarea de dibujar en una pantalla consiste, básicamente, en determinar qué píxeles deseamos encender y con qué intensidades de rojo, verde y azul (es decir, con qué color).

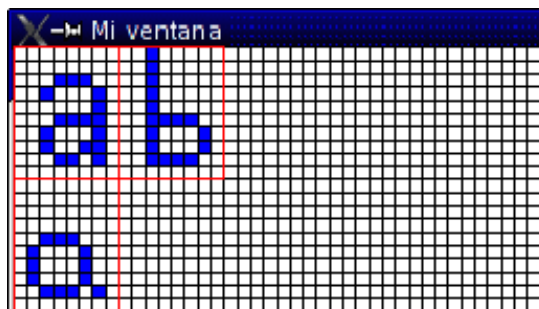
Por ejemplo, para pintar una línea existen algoritmos que determinan la secuencia de píxeles que hemos de activar, en esa matriz bidimensional, para que parezca que hay una línea. Si nos fijamos con detalle, veremos que la línea no es continua sino que, efectivamente, es una secuencia de puntos más o menos diferenciados (esto dependerá en parte de la calidad del monitor que estemos usando). Estos algoritmos de dibujo suelen ser bastante laboriosos a pesar de que la tarea pueda parecer trivial. En la figura puedes observar una



ventana en la que se han dibujado (con el tamaño de los píxeles aumentado) dos líneas y dos círculos. La línea roja va desde el punto (3,3) hasta el (10,10). La línea verde va desde el punto (3,14) hasta el (16,8). El círculo pequeño estaría centrado en (17,15) y tendría radio 2. El círculo grande estaría centrado en (26,6) y tendría radio 5.

4.4. El texto

La forma en la que se escriben textos en una ventana gráfica no difiere, en el fondo, del dibujo de cualquier otra cosa. Por ejemplo, para que podamos ver una letra 'a' en la pantalla hemos de colorear algunos de los píxeles de forma que en conjunto aparenten formar dicha letra. Lo habitual es que para construir los caracteres se utilicen pequeñas matrices bidimensionales, tal que en cada una se marcan cuáles serían los puntos que se deben encender para cada letra. Por ejemplo, en la figura de la derecha vemos varios ejemplos de letras pintadas punto a punto. Podemos ver la



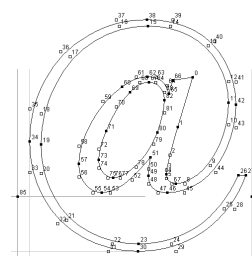
letra 'b' a la derecha y la letra 'a' con dos tipos distintos de letra (o *fuentes* de caracteres). Esta es una forma muy simple de definir nuevos conjuntos de caracteres (fuentes): tenemos una matriz para cada letra y al escribir un texto vamos pintando puntos de acuerdo a las matrices que forman cada uno de los caracteres.

El tamaño de estas matrices variará en función del tipo de letra o el tamaño. En el ejemplo anterior hemos usado una matriz de tamaño 8x10. En SDL, el tipo de letra por defecto usa matrices de tamaño 8x8.

4.4.1. Los tipos de letra vectoriales

Un inconveniente de las fuentes definidas como se explica en el apartado anterior, es que a partir del patrón original de cada letra, no es fácil cambiar el tamaño o el estilo (negrita, itálica, ...). Normalmente, la calidad de una de estas fuentes, tras cambiar su estilo o tamaño, es bastante mala. Para solucionarlo surge una nueva forma de representar las letras.

Las fuentes de tipo vectorial (también conocidas como escalables), definen cada letra mediante elementos tales como vectores, curvas de bezier, splines, polígonos, etc. En la figura de la derecha vemos cómo el carácter “@”, se define como una secuencia de puntos de control. Entre cada dos puntos de control se describe algún tipo de curva (una línea, un arco, etc.) de manera que al ir uniendo todos esos puntos aparece la forma de la letra. Así, es muy fácil cambiar el tamaño o el estilo sin perder calidad en la fuente.



Por supuesto, escribir con este tipo de fuentes es computacionalmente más costoso, ya que cada trazo (línea o curva) que se dibuja implica la ejecución de un algoritmo que determine los píxeles que deben colorearse para que se pueda ver en el monitor.

SDL proporciona una interfaz para trabajar con este tipo de fuentes a través del módulo SDL ttf. A su vez, este módulo hace uso de la biblioteca FreeType² aunque, al igual que ocurre con SDL, esto es transparente para el usuario, salvo por el detalle de que al enlazar las aplicaciones deberá indicar que se haga uso de esta biblioteca externa.

FreeType es una biblioteca de funciones especializada en trabajar con fuentes vectoriales. Soporta algunos formatos muy frecuentes como por ejemplo las fuentes True Type³ o las Type 1⁴. Esta biblioteca tiene licencia GPL por lo que se puede usar con libertad.

² <http://freetype.sourceforge.net>

³ El formato de fuente True Type es original de Apple aunque se utiliza intensivamente en sistemas Microsoft Windows. Este formato también está soportado en sistemas GNU/Linux.

⁴ El formato Type 1 fué desarrollado por Adobe y se usa mucho en los documentos con formato Postscript.

5. Dibujando formas sencillas

La biblioteca `graficos` proporciona algunas primitivas para dibujar formas geométricas sencillas tales como puntos, líneas, rectángulos, círculos y elipses. En el ejemplo “Hola Mundo” (sección 1.1) vemos un ejemplo de uso de la función que dibuja rectángulos.

Este otro ejemplo muestra el uso de otras primitivas de dibujo junto con una captura de pantalla del resultado:

```
#include "graficos.h"
using namespace graficos;
int main(int argc, char *argv[])
{
    // Abrimos la ventana
    CrearVentana(800,600,"Ejemplo Dibujo");

    // Dibujamos algunos puntos (tonos azules)
    for (int i=0; i<100; ++i)
        Punto(10+i*2,10+i*2,i*2,i*2,250);

    // Dibujamos líneas rectas (tonos violeta)
    for (int i=0; i<10; ++i)
        Linea(10,100+i*10,100,200+i*30,100+i*10,0,100+i*10);

    // Dibujamos rectángulos sin relleno (tonos rojos)
    for (int i=0; i<20; ++i)
        Rectangulo(250+i*3,20+i*3,450-i*2,300-i*2,255-i*10,0,0);

    // Dibujamos rectángulos con relleno (tonos verdes)
    RectanguloR(150,350,350,500,0,255,0);
    RectanguloR(250,310,450,400,0,200,0);
    RectanguloR(300,370,400,550,0,160,0);

    // Dibujamos círculos sin relleno (tonos amarillos)
    for (int i=0; i<20; ++i)
        Circulo(600,150,50+i*2,255-i*10,255-i*10,0);

    // Dibujamos elipses con relleno (tonos azules)
    for (int i=20; i>0; --i)
        ElipseR(600,350,50+i*4,30+i*2,0,0,255-i*10);

    // Esperamos a que se pulse una tecla
    ObtenerTecla();
}
```



En la sección 10.3 dispones de un listado completo de estas funciones en el que se explica la sintaxis de las mismas y se ilustra su comportamiento.

Todas estas funciones hacen los dibujos en la ventana que debemos haber abierto previamente. En el caso de que realicemos dibujos que se salen de las coordenadas físicas de esa ventana, el resultado es indeterminado.

Normalmente, el uso de estas funciones gráficas es bastante lento en comparación con el resto de operaciones que podemos hacer con la CPU. No vamos a entrar en detalle sobre los motivos aunque sí vamos a ver cómo es posible hacer que las tareas de dibujar sean más rápidas.

Por defecto, si ejecutamos cualquiera de las funciones de dibujo vemos el efecto de manera

inmediata en la ventana, es decir, justo tras ejecutar una llamada a la función Punto, vemos un punto dibujado en la pantalla. Si dibujamos muchos puntos, vemos que el proceso es relativamente lento. Esto se debe a que los accesos al hardware de vídeo son relativamente lentos. Podemos conseguir que dicho efecto parezca casi instantáneo si usamos las funciones ActivarDibujo y DesactivarDibujo.

Cuando llamamos a DesactivarDibujo, le indicamos a nuestra biblioteca que no haga una actualización inmediata del dibujo en la ventana. Al evitar el acceso al hardware de vídeo hacemos que el proceso de dibujo sea más rápido, pero por contra, no podemos ver el resultado de manera instantánea. Para ver el dibujo, es decir, para refrescar la ventana, llamaremos a la función ActivarDibujo.

A continuación vemos un ejemplo que ilustra el uso de estas dos funciones:

```
#include <iostream>
#include <cstdlib>
#include "graficos.h"

using namespace std;
using namespace graficos;

int main(int argc, char *argv[])
{
    if (argc!=2) {
        cout << "Sintaxis: " << argv[0] << " n" << endl;
        cout << "          n = número de círculos que hay que dibujar (por ejemplo 10000)" << endl;
        exit(0);
    }
    int ncir = atoi(argv[1]);
    srand(time(0));

    CrearVentana(800,600,"Dibujando círculos ...");

    cout << "Dibujamos " << ncir << " círculos con acceso constante al hardware de vídeo" << endl;
    CronometroInicio();
    for (int i=0; i<ncir; i++)
        CirculoR(50+rand()%690,100+rand()%440,rand()%49,rand()%256,rand()%256,rand()%256);
    cout << "Han pasado " << CronometroValor() << " milisegundo(s)" << endl;

    Texto(1,1,"Pulsa una tecla para continuar ...",255,0,0);
    ObtenerTecla();
    LimpiarVentana();

    cout << "Dibujamos " << ncir << " círculos SIN acceso constante al hardware de vídeo" << endl;
    DesactivarDibujo();
    CronometroInicio();
    for (int i=0; i<ncir; i++)
        CirculoR(50+rand()%690,50+rand()%490,rand()%49,rand()%256,rand()%256,rand()%256);
    ActivarDibujo();
    cout << "Han pasado " << CronometroValor() << " milisegundo(s)" << endl;

    Texto(1,1,"Pulsa una tecla para acabar ...",255,0,0);
    ObtenerTecla();
}
```

Este programa dibuja círculos rellenos en una ventana. Por ejemplo, al dibujar 10.000 círculos, si tenemos activado el acceso al hardware de vídeo, se tarda alrededor de 1 segundo mientras que si lo desactivamos, el tiempo se reduce aproximadamente a 0.15 segundos (en un Pentium IV a 1800GHz). Si dibujamos 100.000 círculos el tiempo pasa de unos 8 segundos a 1.4 segundos.

Además, con el dibujo desactivado, la imagen final aparece de manera instantánea y no se aprecia el proceso secuencial de dibujo de cada uno de los círculos. Por contra, con el dibujo activado, podemos ver cómo se van dibujando todos y cada uno de los círculos.

En este apartado se incluye una función que sería la recíproca de las que dibujan. La función `ObtenerPunto` permite consultar el color de un punto de la ventana.

Por último, para acabar esta sección, hemos de indicar que todas las funciones de dibujo tienen un último parámetro opcional que permite hacer los dibujos sobre un bitmap en memoria en lugar de hacerlo sobre la ventana. Más información en la sección 6.

6. Trabajando con Bitmaps

La biblioteca gráfica también permite trabajar con imágenes en memoria. Para ello disponemos del tipo `GRF_Imagen` y de primitivas para:

- Reservar memoria para una imagen nueva.
- Copiar una imagen en otra.
- Obtener información sobre el tamaño de una imagen.
- Liberar la memoria de una imagen.
- Cargar una imagen desde un fichero de disco (en formato .bmp).
- Grabar una imagen en un fichero de disco (en formato .bmp).
- Dibujar una imagen en la pantalla.

Además, es posible utilizar cualquiera de las primitivas gráficas de las secciones 5 y 8 para hacer dibujos sobre la imagen (bitmap) o para escribir texto en ella.

Ante todo, hemos de tener presente que el tipo `GRF_Imagen` es un puntero a una estructura de datos en memoria dinámica, por lo que habrá que tener especial cuidado a la hora de gestionar su memoria. Esta gestión deberá hacerse siempre a través de las primitivas proporcionadas:

- `CrearImagen`. Reserva memoria dinámica para una imagen con las dimensiones dadas y devuelve el puntero a dicha memoria.
- `ObtenerCopiaImagen`. Reciba como parámetro una imagen y devuelve un puntero a una copia, en memoria dinámica, de dicha imagen.
- `LiberarImagen`. Libera la memoria dinámica ocupada por la imagen que se pasa como parámetro.
- `LeerImagen`. Carga una imagen desde un fichero de disco en memoria dinámica y devuelve un puntero a dicha memoria.

El siguiente programa carga una imagen desde un fichero y la muestra en una ventana:

```
#include <iostream>
#include "graficos.h"

using namespace std;
using namespace graficos;

int main(int argc, char *argv[])
{
    if (argc!=2) {
        cout << "Sintaxis: " << argv[0] << " fichero.bmp" << endl;
        exit(0);
    }
    int ancho,alto;
    if (DimensionesImagenFichero(argv[1],ancho,alto)) {
        CrearVentana(ancho+20,alto+20,"Test bitmap");
        GRF_Imagen img = LeerImagen(argv[1]);
        DibujarImagen(img,10,10);
        ObtenerTecla();
        LiberarImagen(img);
    } else
        cout << "Error al abrir el fichero " << argv[1] << endl;
}
```

7. El teclado y el ratón

En cualquier aplicación que necesite un mínimo de interactividad, es necesario obtener respuestas del usuario mediante algún dispositivo. Nuestra biblioteca dispone de funciones para consultar el estado del teclado y del ratón.

- **ObtenerClick.** Esta función permite obtener información sobre el estado del ratón: qué botón hay pulsado y en qué posición está el cursor. Al llamar a la función, ésta se espera a que pulsemos un botón del ratón. Mientras no se pulse ninguno, el programa queda bloqueado a la espera. El tipo `TBotonRaton` devuelto puede ser un valor de entre estos tres para indicar el botón pulsado: `SDL_BUTTON_LEFT`, `SDL_BUTTON_MIDDLE`, `SDL_BUTTON_RIGHT`.
- **ObtenerTecla.** Esta función es similar a la anterior sólo que en este caso se espera hasta que pulsamos una tecla del teclado. Devuelve el código de la tecla pulsada. En la sección 10.8 tienes un listado con los códigos usados por SDL para las teclas.
- **ObtenerEntrada.** Las dos funciones anteriores permiten bloquear el programa hasta que se pulsa el dispositivo correspondiente. Esta nueva función permite consultar ambos dispositivos simultáneamente, de manera que responde ante el primero que se pulse. Devuelve un dato de tipo `TEventoEntrada` que indica el dispositivo que se utilizó (`GRF_RATON` o `GRF_TECLADO`). En el caso de que se haya usado el teclado, podemos averiguar la tecla pulsada mirando el parámetro `pulsa`. En el caso de hacer uso del ratón, podemos ver el botón pulsado y las coordenadas del ratón en los parámetros `boton`, `x`, `y`.

Las tres funciones anteriores son “bloqueantes”, es decir, bloquean el programa hasta que se recibe un evento de entrada, bien del teclado o bien del ratón. A veces es necesario consultar el estado de un dispositivo de entrada sin bloquear el programa. De esta forma, mientras el programa espera a que, por ejemplo, se pulse una tecla, puede estar haciendo otras tareas (dibujando cosas, haciendo cálculos, etc.). Con este objetivo disponemos de la siguiente función:

- **QueTeclaHayPulsada.** Esta función consulta si ha sido pulsada alguna tecla. En caso afirmativo devuelve el código de la misma, o el valor especial `SDLK_UNKNOWN` en caso contrario. Por tanto, evitamos el bloqueo o espera que provocan las funciones anteriores.

El siguiente programa carga una imagen desde un fichero y la muestra en una ventana. A continuación, con las teclas del cursor podemos mover la imagen por la ventana:

```
#include <iostream>
#include "graficos.h"

using namespace std;
using namespace graficos;

int main(int argc, char *argv[])
{
    if (argc!=2) {
        cout << "Sintaxis: " << argv[0] << " fichero.bmp" << endl;
        exit(0);
    }

    int ancho,alto;
    if (DimensionesImagenFichero(argv[1],ancho,alto)) {
        CrearVentana(ancho*4+20,alto*4+20,"Test bitmap 2");
        GRF_Imagen img = LeerImagen(argv[1]);
        int posx=10, posy=10;
        TTecla tec=SDLK_UNKNOWN;
```

```

while (tec!=SDLK_ESCAPE) { // Pulsar Escape para terminar
    DibujarImagen(img,posx,posy);
    tec = ObtenerTecla();
    RectanguloR(posx,posy,posx+ancho,posy+alto,0,0,0); // Borrar
    if (tec==SDLK_UP && posy>10) posy--;
    if (tec==SDLK_DOWN && posy<alto*3+10) posy++;
    if (tec==SDLK_LEFT && posX>10) posX--;
    if (tec==SDLK_RIGHT && posX<ancho*3+10) posX++;
}
LiberarImagen(img);
} else
    cout << "Error al abrir el fichero " << argv[1] << endl;
}

```

8. Escribiendo texto

Tal y como vimos en la sección 4.4, hay dos formas de representar los caracteres:

- Mediante mapas de bits (formatos *raster*).
- Mediante formatos vectoriales.

Con nuestra biblioteca tenemos la opción de usar ambos formatos. A continuación describimos las funciones que tenemos para ello.

8.1. Formato raster

Para escribir texto usando mapas de bits disponemos de la función:

```
void Texto(int x, int y, const char *c,  
           unsigned char r, unsigned char g, unsigned char b,  
           GRF_Imagen img=0);
```

Esta función escribe la cadena de caracteres *c* en las coordenadas (x,y) de la ventana y usando el color dado por (r, g, b). El último parámetro es opcional, si no se utiliza el texto se escribe en la ventana gráfica abierta por la aplicación. En caso de usarse, este último parámetro puede ser un bitmap sobre el que deseamos escribir. Por ejemplo:

```
Texto(100,95,"Hola mundo !",0,255,0);
```

Esta instrucción escribe la cadena "Hola mundo !" en la fila 95 y en la columna 100 usando el color verde.

Observa que esta función no borra lo que hubiese en la pantalla o en el bitmap al escribir sino que el texto se sobrescribe sobre el fondo que ya hubiese dibujado. Una técnica muy simple para evitar sobrescribir es borrar primero dibujando un rectángulo relleno de negro (u otro color) sobre la zona que va a ocupar el texto. Para calcular el tamaño del texto sabemos que cada letra ocupa 8x8 píxeles.

8.2. Formato vectorial

Este formato es un poco más complejo de gestionar que el anterior pero los resultados son de mucha más calidad y, por supuesto, nos da muchas más posibilidades en nuestras aplicaciones. Disponemos de las siguientes funciones:

- GRF_Fuente CargarFuente(const char *fichero, int tamano, TEstiloFuente estilo=TTF_STYLE_NORMAL);
- GRF_Fuente LiberarFuente(GRF_Fuente fuente);
- void EscribirTexto(int f, int c, const char *cadena, GRF_Fuente fuente, unsigned char r, unsigned char g, unsigned char b, GRF_Imagen imgdest=0);
- int AltoFuente(GRF_Fuente fuente);
- void TamanoTexto(GRF_Fuente fuente, const char *mensaje, int &ancho, int &alto);

La definición de las fuentes vectoriales no está incluida dentro de ninguna biblioteca (SDL o FreeType) ya que la idea es que las bibliotecas dispongan de un motor genérico para trabajar con cualquier fuente que cumpla con ciertas especificaciones. La descripción de cada fuente se almacena en un fichero que se puede cargar -en memoria dinámica- con estas bibliotecas. Para realizar estas tareas de gestión asociadas con las fuentes usaremos las funciones *CargarFuente* y *LiberarFuente*.

El tipo de dato que nos va a permitir almacenar la información de la fuente en nuestras aplicaciones será **GRF_Fuente**.

CargarFuente recibe como parámetros el nombre de un fichero en disco (*fichero*), un tamaño de fuente (*tamano*) y un estilo para la misma (*estilo*). Esta función lee la descripción de la fuente del fichero y le aplica las propiedades de tamaño y estilo, creando una estructura de datos en memoria dinámica que nos permitirá, más adelante, usarla para escribir. Como resultado, devuelve esta estructura dinámica. Las fuentes soportadas por SDL son de tipo True Type (ficheros con extensión .ttf). En internet es fácil conseguir multitud de tipos de letra gratuitos en este formato.

Al acabar de usar una fuente, y dado que ésta se encuentra en memoria dinámica, es necesario liberar los recursos que esté utilizando. Para conseguir esto haremos uso de la función `LiberarFuente`, a la que le daremos como parámetro la variable que contiene los datos de la fuente en cuestión.

Una vez que hemos cargado una fuente en memoria (y siempre antes de haberla liberado), podemos usarla para escribir texto con la función `EscribirTexto`. Esta función recibe como parámetros las coordenadas (f,c) donde comenzaremos a escribir el texto, la cadena c que queremos escribir, el tipo de letra (fuente) que queremos usar y el color (r, g, b). Al igual que con el resto de funciones, el último parámetro es opcional y permite indicar si deseamos escribir en la ventana o en un bitmap concreto.

A continuación tenemos un ejemplo de código y el resultado:

```
GRF_Fuente tletra;  
tletra = CargarFuente("Moonstar.ttf",60);  
EscribirTexto(10,20,"Hola",tletra,255,0,0);  
LiberarFuente(tletra);
```



También es posible modificar el estilo de la fuente usando el último parámetro de la función `CargarFuente`. Este parámetro puede tomar uno de los siguientes valores:

- `TTF_STYLE_NORMAL` (tipo de fuente normal)
- `TTF_STYLE_BOLD` (tipo de fuente negrita)
- `TTF_STYLE_ITALIC` (tipo de fuente itálica)

Por defecto (si no se usa) vale `TTF_STYLE_NORMAL`, pero podemos cambiarlo para usar fuentes en itálica, en negrita o en ambas.

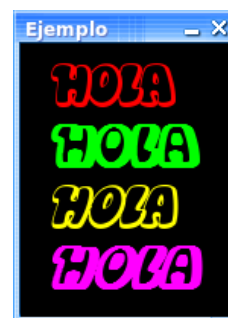
Es posible que el estilo sea una combinación de itálica y negrita. Para ello usaremos como parámetro el operador "O" bit a bit entre ambos estilos, por ejemplo:

```
tletra = CargarFuente("Moonstar.ttf",60,TTF_STYLE_BOLD|TTF_STYLE_ITALIC);
```

Las otras dos funciones que nos queda por ver permiten hacer consultas sobre el tamaño que ocupa el texto. Con los formatos vectoriales, cada letra puede tener un tamaño diferente, al contrario de lo que suele ocurrir con los formatos raster en los que todas las letras ocupan el mismo espacio independientemente de su forma (en nuestro caso 8x8). La función `AltoFuente` devuelve el número de píxeles que tiene de alto la letra más grande que haya en la fuente que le damos como parámetro.

La función `TamanoTexto` recibe una cadena de caracteres y devuelve, a través de dos parámetros pasados por referencia (ancho y alto), el número de píxeles que ocupa esa cadena concreta. Por ejemplo, en el caso del ejemplo anterior, antes de liberar la fuente o de escribir el mensaje podemos saber cuántos píxeles necesita:

```
int alto,ancho;  
TamanoTexto(tletra,"Hola",ancho,alto);  
cout << "Esta cadena ocupa " << ancho << " x " << alto << " píxeles" << endl;
```



9. La gestión del tiempo

Para ayudarnos en el problema de sincronización de tareas en nuestras aplicaciones disponemos de tres funciones:

- `void Esperar(int ms);`
- `void CronometroInicio();`
- `long int CronometroValor();`

La función `Esperar(ms)` bloquea la aplicación durante un número determinado de milisegundos (dado por el parámetro).

Las otras dos funciones sirven para medir el tiempo transcurrido entre dos puntos de nuestra aplicación. `CronometroValor()` devuelve el número de tics de reloj (milisegundos) que han pasado desde la última vez que se ejecutó `CronometroInicio()`. Por ejemplo, el siguiente trozo de código mostrará el tiempo que tardan en ejecutarse dos bucles anidados (en milisegundos):

```
CronometroInicio();
for (int i=0; i<100000; ++i)
    for (int j=0; j<1000; ++j)
        ;
cout << "Los bucles han tardado " << CronometroValor() << " milisegundo(s)" << endl;
```

Estas funciones pueden usarse para hacer que una determinada tarea se haga cada X milisegundos, con estructuras como la siguiente:

```
while (!fin) {
    Hacer tarea
    Esperar(X);
}
```

En este caso, los X milisegundos de espera no se aprovechan para hacer ningún cálculo y el tiempo que transcurre entre cada inicio de tarea sería el que tarda la tarea más los X milisegundos.

Si queremos que el comienzo de cada tarea se ejecute exactamente cada X milisegundos (salvo que la propia tarea tarde más de esos X milisegundos), podríamos usar la siguiente estructura de control:

```
while (!fin) {
    CronometroInicio();
    Hacer tarea
    Esperar(X-CronometroValor());
}
```

En los dos ejemplos anteriores el tiempo de espera se desperdicia. Para aprovecharlo se puede usar alguna estructura similar a ésta:

```
while (!fin) {
    CronometroInicio();
    Hacer tarea
    while (CronometroValor()<X) {
        Hacer algo provechoso mientras esperamos
    }
}
```

10. Listado completo de funciones y tipos disponibles

En esta sección se puede consultar un listado alfabético de todas las funciones y tipos definidos en la biblioteca `graficos`, y a continuación se presenta un índice de las mismas clasificándolas según el tipo de tarea que realizan.

- Gestión de la interfaz gráfica: `CrearVentana`, `ActivarDibujo`, `DesactivarDibujo`
- Dibujo de formas geométricas básicas: `Punto`, `Linea`, `Rectangulo`, `RectanguloR`, `Circulo`, `CirculoR`, `Elipse`, `ElipseR`, `ObtenerPunto`
- Control del teclado y el ratón: `ObtenerClick`, `ObtenerTecla`, `ObtenerEntrada`, `QueTeclaHayPulsada`
- Para la gestión específica de imágenes en memoria (bitmaps): `CrearImagen`, `ObtenerCopiaImagen`, `LimpiarImagen`, `AltoImagen`, `AnchoImagen`, `LeerImagen`, `GrabarImagen`, `LiberarImagen`, `DimensionesImagenFichero`, `DibujarImagen`
- Gestión del tiempo: `Esperar`, `CronometroInicio`, `CronometroValor`
- Escritura de texto: `Texto`, `CargarFuente`, `LiberarFuente`, `EscribirTexto`, `AltoFuente`, `TamanoTexto`

10.1. Listado de tipos y constantes

`namespace graficos`

Descripción La biblioteca completa, tanto definiciones de tipos como funciones, están incluidas dentro del espacio de nombres denominado `graficos`.
Por lo tanto, para hacer uso de ellos es necesario indicar:
`using namespace graficos`
o bien precederlos del especificador de dicho espacio de nombres (`graficos::`).

`GRF_Imagen`

Descripción Este es un tipo que permite almacenar bitmaps. Se corresponde con un puntero a un tipo de dato definido en SDL. Al ser un puntero, los datos referenciados por él se almacenan en memoria dinámica. Se deben usar funciones tales como `CrearImagen`, `ObtenerCopiaImagen` o `LiberarImagen` para reservar memoria, copiar su contenido a otro bitmap o liberar la memoria que ocupa.

`TTecla`

Descripción Tipo usado por las funciones que consultan el estado del teclado para devolver la tecla que ha sido pulsada. El listado completo de valores que puede almacenar está en la sección 10.8.

`TBotonRaton`

Descripción Tipo que codifica los distintos botones del ratón: `SDL_BUTTON_LEFT`, `SDL_BUTTON_MIDDLE`, `SDL_BUTTON_RIGHT`.

`TEventoEntrada`

Descripción Tipo usado por las funciones que consultan el estado de los dispositivos de entrada para identificar el tipo de entrada que ha ocurrido. Puede tomar los valores `GRF_RATON` o `GRF_TECLADO`.

GRF_Fuente

Descripción Este es un tipo que permite almacenar fuentes escalables. Se corresponde con un puntero a un tipo de dato definido en SDL. Al ser un puntero, los datos referenciados por él se almacenan en memoria dinámica. Se deben usar funciones tales como CargarFuente o LiberarFuente para reservar memoria o liberar la memoria que ocupa.

TEstiloFuente

Descripción Tipo que permite indicar el estilo de la fuentes cuando son creadas. Puede tomar alguno de los siguientes valores: TTF_STYLE_NORMAL, TTF_STYLE_BOLD, TTF_STYLE_ITALIC.

10.2. Funciones para la gestión de la interfaz gráfica

```
void CrearVentana(int ancho, int alto, const char *titulo);
```

Descripción Esta función dibuja tiene como objetivo crear (abrir) una ventana gráfica. Antes de utilizar cualquier función de dibujo, es necesario ejecutar esta función. De no hacerlo, el comportamiento de dichas funciones es indefinido.

Parámetros (ancho,alto) : Tamaño de la ventana (en píxeles).
titulo : Título que aparece en la barra superior de la ventana.

```
int AltoVentana();
```

Descripción Devuelve la altura de la ventana. Dicho tamaño viene dado en píxeles.

Parámetros Ninguno.

```
int AnchoVentana();
```

Descripción Devuelve la anchura de la ventana. Dicho tamaño viene dado en píxeles.

Parámetros Ninguno.

```
void LimpiarVentana();
```

Descripción Borra el contenido de la ventana. El resultado es que la ventana se queda de color negro.

Parámetros Ninguno.

```
void ActivarDibujo();
```

Descripción Esta función se usa conjuntamente con DesactivarDibujo. El objetivo de ambas funciones es proporcionar al usuario un medio para dibujar gráficos complejos de una manera eficiente y rápida. Cada operación de dibujo consume un tiempo considerable durante el acceso al hardware de vídeo, por lo que si necesitamos dibujar formas muy complejas, se puede llegar a apreciar un retraso considerable en la actualización de la ventana.

Para evitar el acceso constante al hardware de vídeo, podemos realizar el dibujo en la memoria y, una vez finalizado, acceder una única vez al hardware para que se haga el dibujo completo.

La función DesactivarDibujo permite desactivar el acceso al hardware por lo que si llamamos a alguna de las primitivas gráficas, después de haber usado esta función, el dibujo se hará en memoria y no veremos ningún efecto en la pantalla. Una vez que hayamos acabado de dibujar, podremos usar ActivarDibujo. Tras llamar a esta segunda función, la ventana se actualizará con aquello que hayamos estado dibujando en memoria.

Tras la ejecución de ActivarDibujo, cada llamada a una función que dibuja

accede al hardware y su efecto se verá reflejado en la ventana.

Tras la ejecución de DesactivarDibujo, cada llamada a una función que dibuja no accede al hardware de vídeo sino que pinta en la memoria. Hasta que no se llame a ActivarDibujo no se verá ningún efecto en la ventana.

Por defecto, al comenzar el uso del módulo graficos (tras ejecutar CrearVentana), podemos considerar que se ha ejecutado ActivarDibujo.

Para ejemplificar esta explicación puedes probar el ejemplo test_activar, suministrado junto con este módulo. Este programa dibuja círculos en una ventana (el número de círculos lo da el usuario) de las dos formas. Si el número de círculos es suficientemente elevado, podemos apreciar perfectamente los beneficios de estas dos funciones. Por ejemplo, al dibujar 10.000 círculos, si tenemos activado el acceso al hardware de vídeo, se tarda alrededor de 1 segundo mientras que si lo desactivamos, el tiempo se reduce aproximadamente a 0.15 segundos (en un Pentium IV a 1800GHz). Si dibujamos 100.000 círculos el tiempo pasa de unos 8 segundos a 1.4 segundos.

Además, con el dibujo desactivado, la imagen aparece de manera instantánea y no se aprecia el proceso secuencial de dibujo de cada uno de los círculos.

Parámetros Ninguno.

```
void DesactivarDibujo();
```

Descripción Ver ActivarDibujo.

Parámetros Ninguno.

10.3. Funciones de dibujo

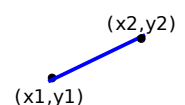
```
void Punto(int x, int y,
            unsigned char r, unsigned char g, unsigned char b,
            GRF_Imagen img=0);
```

Descripción Esta función dibuja un punto de color (r,g,b) en la posición (x,y). Si no se indica el último parámetro el punto se dibuja en la ventana. Si se indica un bitmap el punto se dibuja en él en lugar de en la ventana.

Parámetros (x,y) : Coordenadas cartesianas en las que queremos dibujar el punto.
(r,g,b) : Componentes roja, verde y azul para el color del punto.
img : Bitmap sobre el que vamos a dibujar. Si no se pone, se dibuja sobre la ventana.

```
void Linea(int x1, int y1, int x2, int y2,
            unsigned char r, unsigned char g, unsigned char b,
            GRF_Imagen img=0);
```

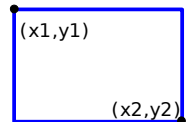
Descripción Esta función dibuja una línea de color (r,g,b) entre los puntos (x1,y1) y (x2,y2). Si no se indica el último parámetro, la línea se dibuja en la ventana. Si se indica un bitmap, la línea se dibuja en él en lugar de en la ventana.



Parámetros (x1,y1) - (x2,y2) : Coordenadas cartesianas que definen el comienzo y final de la línea que deseamos dibujar.
(r,g,b) : Componentes roja, verde y azul para el color de la línea.
img : Bitmap sobre el que vamos a dibujar. Si no se pone, se dibuja sobre la ventana.

```
void Rectangulo(int x1, int y1, int x2, int y2,
               unsigned char r, unsigned char g, unsigned char b,
               GRF_Imagen img=0);
```

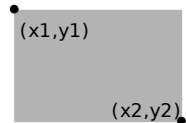
Descripción Esta función dibuja un rectángulo de color (r,g,b) definido por las posiciones de su esquina superior izquierda (x1,y1) y su esquina inferior derecha (x2,y2). Si no se indica el último parámetro, se dibuja en la ventana. Si se indica un bitmap, la



Parámetros (x1,y1) : Coordenadas cartesianas de la esquina superior izquierda.
 (x2,y2) : Coordenadas cartesianas de la esquina inferior derecha.
 (r,g,b) : Componentes roja, verde y azul para el color del rectángulo.
 img : Bitmap sobre el que vamos a dibujar. Si no se pone, se dibuja sobre la ventana.

```
void RectanguloR(int x1, int y1, int x2, int y2,
                 unsigned char r, unsigned char g, unsigned char b,
                 GRF_Imagen img=0);
```

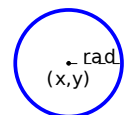
Descripción Esta función dibuja un rectángulo de color (r,g,b) definido por las posiciones de su esquina superior izquierda (x1,y1) y su esquina inferior derecha (x2,y2). Se dibuja tanto el contorno del rectángulo como el área incluida en él. Si no se indica el último parámetro, se dibuja en la ventana. Si se indica un bitmap, la línea se dibuja en él en lugar de en la ventana.



Parámetros (x1,y1) : Coordenadas cartesianas de la esquina superior izquierda.
 (x2,y2) : Coordenadas cartesianas de la esquina inferior derecha.
 (r,g,b) : Componentes roja, verde y azul para el color del rectángulo.
 img : Bitmap sobre el que vamos a dibujar. Si no se pone, se dibuja sobre la ventana.

```
void Circulo(int x, int y, int rad,
             unsigned char r, unsigned char g, unsigned char b,
             GRF_Imagen img=0);
```

Descripción Esta función dibuja la circunferencia de un círculo de color (r,g,b) centrado en el punto (x,y) y de radio rad. Si no se indica el último parámetro, se dibuja en la ventana. Si se indica un bitmap, la línea se dibuja en él en lugar de en la ventana.



Parámetros (x,y) : Coordenadas cartesianas que definen el centro del círculo.
 rad : Radio del círculo.
 (r,g,b) : Componentes roja, verde y azul para el color del círculo.
 img : Bitmap sobre el que vamos a dibujar. Si no se pone se dibuja sobre la ventana.

```
void CirculoR(int x, int y, int rad,
              unsigned char r, unsigned char g, unsigned char b,
              GRF_Imagen img=0);
```

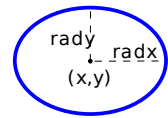
Descripción Esta función dibuja un círculo de color (r,g,b) centrado en el punto (x,y) y de radio rad. Se dibuja tanto la circunferencia como el interior del círculo. Si no se indica el último parámetro, se dibuja en la ventana. Si se indica un bitmap, la línea se dibuja en él en lugar de en la ventana.



Parámetros (x,y) : Coordenadas cartesianas que definen el centro del círculo.
 rad : Radio del círculo.
 (r,g,b) : Componentes roja, verde y azul para el color del círculo.
 img : Bitmap sobre el que vamos a dibujar. Si no se pone, se dibuja sobre la ventana.

```
void Elipse(int x, int y, int radx, int rady,
            unsigned char r, unsigned char g, unsigned char b,
            GRF_Imagen img=0);
```

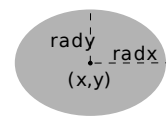
Descripción Esta función dibuja una elipse de color (r,g,b) centrada en el punto (x,y). El radio horizontal lo da radx y el radio vertical viene dado por rady. Si no se indica el último parámetro, se dibuja en la ventana. Si se indica un bitmap, la línea se dibuja en él en lugar de



Parámetros (x,y) : Coordenadas cartesianas del centro de la elipse.
 radx : Radio horizontal.
 rady : Radio vertical.
 (r,g,b) : Componentes roja, verde y azul para el color del rectángulo.
 img : Bitmap sobre el que vamos a dibujar. Si no se pone, se dibuja sobre la ventana.

```
void ElipseR(int x, int y, int radx, int rady,
             unsigned char r, unsigned char g, unsigned char b,
             GRF_Imagen img=0);
```

Descripción Esta función dibuja una elipse de color (r,g,b) centrada en el punto (x,y). Se dibuja tanto la circunferencia como el interior de la elipse. El radio horizontal lo da radx y el radio vertical viene dado por rady. Si no se indica el último parámetro, se dibuja en la ventana. Si se indica un bitmap, la línea se dibuja en él en lugar de en la ventana.



Parámetros (x,y) : Coordenadas cartesianas del centro de la elipse.
 radx : Radio horizontal.
 rady : Radio vertical.
 (r,g,b) : Componentes roja, verde y azul para el color del rectángulo.
 img : Bitmap sobre el que vamos a dibujar. Si no se pone, se dibuja sobre la ventana.

```
void ObtenerPunto(int x, int y,
                  unsigned char &r, unsigned char &g, unsigned char &b,
                  GRF_Imagen img=0);
```

Descripción Esta función consulta el color del pixel situado en las coordenadas (x,y) y lo devuelve a través de los parámetros (r, g, b). Si no se indica el último parámetro, la consulta se hace en la ventana. Si se indica un bitmap, la consulta es sobre ese bitmap concreto.

Parámetros (x,y) : Coordenadas cartesianas del punto.
 (r,g,b) : Componentes roja, verde y azul del color del punto.
 img : Bitmap sobre el que hacemos la consulta. Si no se pone, la consulta se hace en la ventana.

10.4. Funciones para la gestión del teclado y el ratón

```
TBotonRaton ObtenerClick(int &x, int &y);
```

Descripción Espera hasta que se pulsa alguno de los botones del ratón sobre la ventana. Esta función devuelve el botón que ha sido pulsado (SDL_BUTTON_LEFT, SDL_BUTTON_MIDDLE, SDL_BUTTON_RIGHT) y, además, las coordenadas (x,y) en donde está situado el cursor en el momento de la pulsación.

Parámetros (x,y) : Contiene las coordenadas del cursor al pulsar el ratón.

```
TTeclea ObtenerTecla();
```

Descripción Espera hasta que se pulsa una tecla y devuelve el código de la tecla pulsada (el listado de códigos disponibles está en la sección 10.8).

Parámetros ninguno

TEventoEntrada ObtenerEntrada(TTecla &pulsa, TBotonRaton &boton, int &x, int &y);

Descripción Espera hasta que, o bien se pulsa alguno de los botones del ratón sobre la ventana, o bien se pulsa una tecla. Si se ha usado el teclado, entonces devuelve GRF_TECLADO y la tecla pulsada se devuelve a través del parámetro pulsa (el listado de códigos de tecla está en la sección 10.8). Si se ha hecho uso del ratón, entonces devuelve GRF_RATON y podemos saber el botón que se pulsó a través del parámetro boton (SDL_BUTTON_LEFT, SDL_BUTTON_MIDDLE, SDL_BUTTON_RIGHT) así como las coordenadas del cursor a través de los parámetros x e y.

Parámetros pulsa : Tecla pulsada (si devuelve GRF_TECLADO).
boton : Botón del ratón pulsado (si devuelve GRF_RATON).
(x,y) : Coordenadas del cursor (si devuelve GRF_RATON).

TTecla QueTeclaHayPulsada();

Descripción Consulta si hay una tecla pulsada para devolver su código (el listado de códigos disponibles está en la sección 10.8). La función no bloquea la ejecución en espera de la pulsación, sino que devuelve el valor SDLK_UNKNOWN.

Parámetros Ninguno.

10.5. Funciones para trabajar con bitmaps (imágenes)

GRF_Imagen CrearImagen(int ancho, int alto);

Descripción Crea un bitmap vacío (negro) del tamaño indicado (en píxeles). Devuelve el bitmap creado. Recuerda que el tipo GRF_Imagen es un puntero (ver sección 10.1).

Parámetros ancho : Tamaño horizontal del bitmap en píxeles.
alto : Tamaño vertical del bitmap en píxeles.

GRF_Imagen ObtenerCopiaImagen(GRF_Imagen orig);

Descripción Devuelve una copia del bitmap orig. Recuerda que el tipo GRF_Imagen es un puntero (ver sección 10.1).

Parámetros orig : Bitmap original que deseamos duplicar.

void LiberarImagen(GRF_Imagen img);

Descripción Libera la memoria de la imagen img. Obviamente, esta imagen debe haber sido previamente creada o leída desde disco.

Parámetros img : Imagen que vamos liberar.

int AltoImagen(GRF_Imagen img);

Descripción Devuelve el tamaño vertical del bitmap img, expresado en píxeles.

Parámetros img : Bitmap del que queremos averiguar su altura.

int AnchoImagen(GRF_Imagen img);

Descripción Devuelve el tamaño horizontal del bitmap img, expresado en píxeles.

Parámetros img : Bitmap del que queremos averiguar su anchura.

void LimpiarImagen(GRF_Imagen img);

Descripción Borra el contenido de la imagen img, poniendo a cero el valor de todos sus píxeles (negro). Esta función no libera ninguna memoria ni modifica el tamaño de la imagen.

Cuidado: no confundir con `LiberarImagen`.

Parámetros `img` : Bitmap que deseamos borrar.

```
void DibujarImagen(GRF_Imagen img, int x, int y, GRF_Imagen imgdest=0);
```

Descripción Dibuja, en la imagen `imgdest`, la imagen almacenada en `img`. Lo hace en las coordenadas `(x,y)`. Si no se indica este último parámetro, la imagen `img` se dibuja en la ventana gráfica.

Parámetros `img` : Imagen que vamos a dibujar.
 `(x,y)` : Posición donde vamos a dibujar la imagen.
 `imgdest` : Imagen en donde vamos a dibujar. Si no se indica este parámetro, se dibuja en la ventana gráfica.

```
GRF_Imagen LeerImagen(const char *fich);
```

Descripción Carga una imagen desde un fichero. Si no ha sido posible la carga de la imagen devolverá 0 (puntero nulo). Recuerda que el tipo `GRF_Imagen` es un puntero (ver sección 10.1). El formato del fichero en disco debe ser BMP.

Parámetros `fich` : Nombre del fichero que contiene la imagen en disco (formato BMP).

```
bool GrabarImagen(const char *fich, GRF_Imagen img);
```

Descripción Graba la imagen `img` en el fichero `fich`. Devuelve un valor lógico que será `true` si la operación ha sido exitosa y `false` en caso contrario.

Parámetros `fich` : Nombre del fichero en el que vamos a grabar la imagen (formato BMP).
 `img` : Imagen que vamos a guardar en disco.

```
bool DimensionesImagenFichero(const char *fich, int &ancho, int &alto);
```

Descripción Lee, del fichero `fich`, las dimensiones de la imagen que almacena (en formato BMP). En el caso de que se produzca algún error al acceder al fichero, devolverá `false` y las dimensiones serán cero. Si todo va bien, devuelve `true`.

Parámetros `fich` : Nombre del fichero (en formato BMP).
 `ancho` : Aquí devuelve el ancho de la imagen.
 `alto` : Aquí devuelve el alto de la imagen.

10.6. Funciones para trabajar con texto

```
GRF_Fuente CargarFuente(const char *fichero, int tamano,  
                          TEstiloFuente estilo=TTF_STYLE_NORMAL);
```

Descripción Carga la fuente desde el fichero `fichero` (en formato .ttf). Al cargar la fuente se establece tanto el tamaño como el estilo (`TTF_STYLE_NORMAL`, `TTF_STYLE_ITALIC`, `TTF_STYLE_BOLD`). Esta función lee desde el fichero la fuente, la almacena en memoria dinámica, y devuelve un puntero a dicha estructura dinámica. Para liberar la memoria usaremos la función `LiberarFuente`.

Parámetros `fichero` : Nombre del fichero (en formato ttf).
 `tamano` : Tamaño de la fuente.
 `estilo` : Estilo de la fuente.

```
void LiberarFuente(GRF_Fuente fuente);
```

Descripción Libera la memoria dinámica de la fuente indicada. Previamente, ésta habrá sido cargada mediante `CargarFuente`.

Parámetros `fuente` : Fuente que deseamos liberar.

```
void EscribirTexto(int x, int y, const char *cadena, GRF_Fuente fuente,
                  unsigned char r, unsigned char g, unsigned char b,
                  GRF_Imagen imgdest=0);
```

Descripción Escribe un mensaje de texto en las coordenadas (x,y) con la fuente especificada y de color (r,g,b). El último parámetro es opcional: si no se pone, el mensaje se escribe en la ventana gráfica en la que estamos trabajando. Si por el contrario, se indica un bitmap, entonces el mensaje se escribe en dicho bitmap.

Parámetros (x,y) : Coordenadas donde vamos a escribir el mensaje.
cadena : El mensaje.
fuente : El tipo de letra elegido.
(r,g,b) : Color del texto.
img : Bitmap sobre el que vamos a escribir. Si no se pone se escribe sobre la ventana.

```
int AltoFuente(GRF_Fuente fuente);
```

Descripción Devuelve la altura máxima, en píxeles, que pueden ocupar los caracteres de la fuente indicada.

Parámetros fuente : Fuente sobre la que hacemos la consulta.

```
void TamanoTexto(GRF_Fuente fuente, const char *mensaje, int &ancho, int &alto);
```

Descripción Calcula el tamaño, en píxeles, que ocupa un mensaje de texto.

Parámetros fuente : Fuente sobre la que hacemos la consulta.
mensaje : El mensaje del que deseamos conocer los píxeles que ocupa.
ancho : Aquí se devuelve el número de píxeles horizontales que ocupa el mensaje.
alto : Aquí se devuelve el número de píxeles verticales que ocupa el mensaje.

10.7. Funciones para la gestión del tiempo

```
void Esperar(int ms);
```

Descripción Bloquea el programa durante el tiempo indicado (en milisegundos).

Parámetros ms : Milisegundos de espera.

```
void CronometroInicio();
```

Descripción Pone a cero un cronómetro para medir tiempos.

Parámetros Ninguno.

```
long int CronometroValor();
```

Descripción Devuelve el tiempo transcurrido (en milisegundos) desde la última puesta a cero del cronómetro de la biblioteca.

Parámetros Ninguno.

10.8. Otras definiciones de interés definidas en SDL

Constantes utilizadas para la identificación de las teclas (valores devueltos por `ObtenerTecla` o `QueTeclaHayPulsada`). Estas definiciones están en el fichero de cabecera `SDL_keysym.h` de la biblioteca SDL.

SDLK_UNKNOWN	SDLK_BACKSPACE	SDLK_TAB	SDLK_CLEAR	SDLK_RETURN
SDLK_PAUSE	SDLK_ESCAPE	SDLK_SPACE	SDLK_EXCLAIM	SDLK_QUOTEDBL
SDLK_HASH	SDLK_DOLLAR	SDLK_AMPERSAND	SDLK_QUOTE	SDLK_LEFTPAREN
SDLK_RIGHTPAREN	SDLK_ASTERISK	SDLK_PLUS	SDLK_COMMA	SDLK_MINUS
SDLK_PERIOD	SDLK_SLASH	SDLK_COLON	SDLK_SEMICOLON	SDLK_LESS
SDLK_EQUALS	SDLK_GREATER	SDLK_QUESTION	SDLK_AT	SDLK_LEFTBRACKET
SDLK_BACKSLASH	SDLK_RIGHTBRACKET	SDLK_CARET	SDLK_UNDERSCORE	SDLK_BACKQUOTE
SDLK_DELETE				
SDLK_0	SDLK_1	SDLK_2	SDLK_3	SDLK_4
SDLK_5	SDLK_6	SDLK_7	SDLK_8	SDLK_9
SDLK_a	SDLK_b	SDLK_c	SDLK_d	SDLK_e
SDLK_f	SDLK_g	SDLK_h	SDLK_i	SDLK_j
SDLK_k	SDLK_l	SDLK_m	SDLK_n	SDLK_o
SDLK_p	SDLK_q	SDLK_r	SDLK_s	SDLK_t
SDLK_u	SDLK_v	SDLK_w	SDLK_x	SDLK_y
SDLK_z				
SDLK_WORLD_0	SDLK_WORLD_1	SDLK_WORLD_2	...	SDLK_WORLD_95
SDLK_KP0	SDLK_KP1	SDLK_KP2	SDLK_KP3	SDLK_KP4
SDLK_KP5	SDLK_KP6	SDLK_KP7	SDLK_KP8	SDLK_KP9
SDLK_KP_PERIOD	SDLK_KP_DIVIDE	SDLK_KP_MULTIPLY	SDLK_KP_MINUS	SDLK_KP_PLUS
SDLK_KP_ENTER	SDLK_KP_EQUALS			
SDLK_UP	SDLK_DOWN	SDLK_RIGHT	SDLK_LEFT	
SDLK_INSERT	SDLK_HOME	SDLK_END	SDLK_PAGEUP	SDLK_PAGEDOWN
SDLK_F1	SDLK_F2	SDLK_F3	SDLK_F4	SDLK_F5
SDLK_F6	SDLK_F7	SDLK_F8	SDLK_F9	SDLK_F10
SDLK_F11	SDLK_F12	SDLK_F13	SDLK_F14	SDLK_F15
SDLK_NUMLOCK	SDLK_CAPSLOCK	SDLK_SCROLLLOCK	SDLK_RSHIFT	SDLK_LSHIFT
SDLK_RCTRL	SDLK_LCTRL	SDLK_RALT	SDLK_LALT	SDLK_RMETA
SDLK_LMETA	SDLK_LSUPER	SDLK_RSUPER	SDLK_MODE	SDLK_COMPOSE
SDLK_HELP	SDLK_PRINT	SDLK_SYSREQ	SDLK_BREAK	SDLK_MENU
SDLK_POWER	SDLK_EURO	SDLK_UNDO		

11. Referencias

- [MAR] Martínez Baena, Javier. "Introducción a SDL (módulo gráfico)". (pdf disponible en la página web de la asignatura)
- [SDL] Información sobre SDL en <http://www.libsdl.org>
- [GFX] Información sobre SDL_gfx en http://www.ferzkopp.net/~aschiffler/Software/SDL_gfx-2.0/
- [SER] Alberto García Serrano. "Programación de videojuegos con SDL". Ediversitas Multimedia. 2003.
- [PAZ] Ernest Pazera, André LaMothe. "Focus on SDL". Premier Press. 2003.
- [LOK] Loki Software Inc. with John R. Hall. "Programming Linux games". Linux Journal Press. 2001.
- [PEN] Ron Penton, André LaMothe. "Data structures for game programmers". Premier Press. 2003.
- <http://www.answers.com/topic/list-of-games-using-sdl>
Lista de juegos hechos con SDL. En el listado se incluyen tanto juegos comerciales y con licencia de tipo GNU.
- <http://cs-sdl.sourceforge.net/>
Biblioteca construida sobre SDL y orientada a objetos compatible con .NET (CLS compliant).
- http://gpwiki.org/index.php/SDL_tutorials
Tutoriales sobre SDL.

Índice alfabético

ActivarDibujo.....	12, 21
AltoFuente.....	17, 27
AltoImagen.....	25
AltoVentana.....	21
AnchoImagen.....	25
AnchoVentana.....	21
bitmap.....	13
Bitmaps.....	14
CargarFuente.....	17, 26
Circulo.....	23
CirculoR.....	23
colores.....	8
CrearImagen.....	14, 25
CrearVentana.....	4, 7 s., 21
CronometroInicio.....	19, 27
CronometroValor.....	19, 27
definiciones.....	28
DesactivarDibujo.....	12, 22
DibujarImagen.....	26
dibujos.....	9
DimensionesImagenFichero.....	26
Elipse.....	24
ElipseR.....	24
EscribirTexto.....	17, 27
Esperar.....	19, 27
estilo.....	17
FreeType.....	3, 10
FreeType2.....	5
fuentes.....	9
fuentes de tipo vectorial.....	10
g++.....	7
gestión del tiempo.....	19, 27
GrabarImagen.....	26
graficos.h.....	4
GRF_Fuente.....	17, 21, 26
GRF_Imagen.....	14, 20, 25
GRF_RATON.....	15, 20, 25
GRF_TECLADO.....	15, 20, 25
Hola mundo.....	4
Instalación.....	5
LeerImagen.....	14, 26
LiberarFuente.....	17, 26
LiberarImagen.....	14, 25
LimpiarImagen.....	25
LimpiarVentana.....	21
Linea.....	22
namespace graficos.....	4, 20
ObtenerClick.....	15, 24

ObtenerCopiaImagen.....	14, 25
ObtenerEntrada.....	15, 25
ObtenerPunto.....	24
ObtenerTecla.....	15, 24, 28
píxel.....	8
primitivas para dibujar.....	11
Punto.....	22
QueTeclaHayPulsada.....	15, 25, 28
raster.....	17
ratón.....	15
Rectangulo.....	23
RectanguloR.....	23
refrescar la ventana.....	12
RGB.....	8
SDL.....	3, 5
SDL_BUTTON_LEFT.....	15, 20, 24
SDL_BUTTON_MIDDLE.....	15, 20, 24
SDL_BUTTON_RIGHT.....	15, 20, 24
SDL_gfx.....	3, 5
SDL_keysym.h.....	28
SDL_ttf.....	3, 5, 10
sincronización.....	19
sistemas gráficos.....	8
TamanoTexto.....	17, 27
TBotonRaton.....	15, 20, 24 s.
teclado.....	15
TEstiloFuente.....	17, 21, 26
TEventoEntrada.....	15, 20, 25
texto.....	9, 17
TTecla.....	20, 24 s.
TTF_STYLE_BOLD.....	18, 21, 26
TTF_STYLE_ITALIC.....	18, 21, 26
TTF_STYLE_NORMAL.....	18, 21, 26
vectorial.....	17
ventana.....	8