Tecnología de la Programación

Boletín de Prácticas

Curso Académico 2013/2014

Índice

3.3 Captura de eventos del teclado

Sesión 4 - Recursividad

4.1 Ampliación de `MiSDL`

4.2 Generación de números aleatorios

4.3 Dibujo recursivo de los árboles

4.4 Dibujo de la selva

4.5 Establecimiento de la textura

Sesión 5 - Gestión de récords de tiempos

5.1 Ampliación de `MiSDL`

5.2 Ampliación de la estructura `MiJuego`

5.3 Ampliación de las funciones `MiJuegoIniciar` y `MiJuegoTerminar`

5.4 Ampliación de la función `MiJuego_Pintar`

5.5 Amplicación de las funciones `MiJuego_Bucle` y `MiJuego_EventoTeclado` y definición de la nueva función `MiJuego_EventoTexto` para realizar lectura de datos por teclado

5.6 Ampliación de las funciones `MiJuego_IniciarFase` y `MiJuego_TerminarFase`

Sesión 6 - Programación modular y TDAs

6.1 Creación de ficheros de cabecera (.h)

6.2 Creación de ficheros de implementación (.c) asociados a los ficheros de cabecera

6.3 Creación de ficheros objeto (.o) y ejecutable (.exe)

Sesión 7 - Documentación de software

Documentación requerida para la entrega de prácticas del Bloque I

Bloque II: Proyecto de Programación

Características del videojuego

Consideraciones de diseño

Ampliaciones

Documentación requerida para la entrega de prácticas del Bloque II

Apéndice A. Sugerencias para la instalación en Windows

A.1 Instalación de code::blocks

A.2 Instalación de SDL 2.0

A.3 Instalación de SDL_ttf 2.0

Apéndice B. Sugerencias para la instalación en Ubuntu 12.04

B.1 Instalación de code::blocks

B.2 Instalación de SDL 2.0

B.3 Instalación de SDL_ttf 2.0

Bloque I: Prácticas Guiadas

Este bloque de prácticas permitirá al alumno afianzar los contenidos teóricos mediante la realización guiada de pequeñas prácticas de programación orientadas cada una de ellas a cubrir uno o varios contenidos específicos, identificados como descriptores en cada práctica. Se presentará y utilizará un *Entorno de Desarrollo Integrado* (*Integrated Development Enviroment, IDE*) para las tareas de edición, compilación, depuración, ejecución y documentación de software. Las prácticas están organizadas en una sesión inicial (sesión *0*) más *7* sesiones de prácticas guiadas. La sesión *0* tiene como objetivo tomar un primer contacto con el IDE y se utilizará un código de prueba facilitado en el boletín de prácticas, y que no será necesario entregar al final. El resto de sesiones están relacionadas entre sí de forma que cada práctica amplia a la anterior mediante la introducción de un nuevo contenido práctico.

El resultado de este bloque será una única práctica la cual acumula los desarrollos realizados en las *7* sesiones, consistente en una aplicación gráfica que desarrolla procesos básicos guiados por el usuario que se requerirán para la realización de un proyecto de programación final en un segundo bloque de prácticas.

Para facilitar la comprensión y corrección de las prácticas, el alumno dispondrá de los ficheros ejecutables resultado de cada una de las sesiones de prácticas, publicados por los profesores de la asignatura a través del aula virtual.

Sesión 0 - Entorno y depuración

Descriptores: IDE, Code::Blocks, edición, compilación, construcción, ejecución y depuración de programas.

Code::Blocks (página oficial http://www.codeblocks.org/) es un IDE de código abierto multi-plataforma (Linux, Max y Windows) para los lenguajes de programación C y C++ con el que se desarrollarán las prácticas de la asignatura, las cuales se realizarán en lenguaje C. Code::Blocks es un software libre distribuido bajo los términos de la Licencia General Pública de GNU.

A continuación se mostrará un ejemplo de creación de una aplicación sencilla mediante Code::Blocks. Cuando se ejecuta Code::Blocks se abre el entorno de desarrollo.

0.1 Crear un proyecto

Lo primero que se debe hacer es crear un proyecto. Para ello habrá que seguir los siguientes pasos:

- Seleccionar la opción del menú *File – New – Project*.
- Elegir el tipo de proyecto que se requiera. Existen muchos tipos de proyectos, según el tipo de aplicación que se desee crear. En nuestro caso se elegirá un proyecto vacío (opción *Empty Project*), ya que se va a crear una aplicación a partir de cero.

- Nombre del proyecto (*Project title*): *TPSesion0*.
- Introducir la ruta del directorio donde se creará el proyecto (*Folder to create Project in*): *TPPracticas*.
- Compilador que va a utilizar el proyecto (*Compiler*): *GNU GCC Compiler*.

Se creará automáticamente un directorio con el nombre del proyecto guardado en el directorio indicado: *TPPracticas\TPSesion0*.

El proyecto se almacenará en un fichero *TPPracticas\TPSesion0\TPSesion0.cbp*, que será el fichero de proyecto en Code::Blocks.

0.2 Añadir código fuente a un proyecto

Una vez creado un proyecto, se le podrán asociar nuevos ficheros de código fuente mediante la opción *File – New – Empty file*. En este paso se pedirá el nombre del fichero. Para nuestra práctica se añadirá un fichero de nombre *TPSesion0* de tipo *C/C++ files*. Dado que el compilador elegido para el proyecto es *GNU GCC Compiler*, el fichero quedará guardado como un fichero fuente en C, *TPPracticas\TPSesion0\TPSecion0.c*. El fichero fuente *TPSesion0.c* contendrá el siguiente código C:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char ** argv)
{
    printf("Hola Mundo.\n");
    printf("Dime tu nombre: ");
    char nombre[10];
    scanf("%s",nombre);
    printf("Encantado de conocerte, %s.\n",nombre);
    printf("Estoy pensando un entero del 1 al 10.\n");
    printf("A ver si lo adivinas.\n");
    srand(time(NULL));
    int miNumero = 1+rand()%10;
    int numero;
    int nIntentos = 0;
    do
    {
        printf("El entero es el: ");
        scanf("%d",&numero);
        if (miNumero<numero) printf("Mi entero es menor.\n");
        if (miNumero>numero) printf("Mi entero es mayor.\n");
        nIntentos++;
    }
    while(numero!=miNumero);
    printf("Lo adivinaste en %d intentos!\n",nIntentos);

    system("pause");
    return 0;
}
```

Ahora ya se tiene asociado, al proyecto *TPSesion0.cbp,* el fichero fuente *TPSesion0.c* que contiene el programa principal (función *main*). Este programa simula un juego en donde un usuario debe adivinar un número entre *1* y *10* "pensado" por el programa. El programa es un proceso reiterativo en donde el usuario trata de adivinar el número y el programa da indicaciones acerca de si el número introducido es mayor o menor que el pensado. El programa termina cuando el usuario adivina el número, mostrando en pantalla el número de intentos que el usuario ha requerido para adivinarlo.

0.3 Compilar un fichero de código fuente

El siguiente paso es compilar el fichero de código fuente *TPSesion0.c*. Esto se realiza mediante la opción *Build – Compile current file*. Esto crea un fichero objeto *TPSesion0.o* en el directorio *TPPracticas\TPSesion0\obj\Debug* o en el directorio *TPPracticas\TPSesion0\obj\Release* dependiendo de la opción *Build target*. Esta opción establece el objetivo de la construcción, bien en modo *Debug*, en donde se incluyen los símbolos del depurador en el código objeto, o bien en modo *Release* en donde los objetos se crean sin información de depuración. El modo usual de trabajar es en modo *Debug* mientras se está desarrollando la aplicación y en modo *Release* para generar la versión definitiva.

0.4 Construir y ejecutar una aplicación

A continuación se construirá la aplicación mediante la opción *Build – Build*. Esto genera el fichero ejecutable *TPSesion0.exe* en el directorio *TPPracticas\TPSesion0\bin\Debug* o en el directorio *TPPracticas\TPSesion0\bin\Release* dependiendo de la opción *Build target*. Si los ficheros objeto no existen o los ficheros fuente se han modificado, la opción *Build* genera tanto los nuevos objetos como el ejecutable.

Finalmente, se ejecutará la aplicación mediante la opción *Build – Run*. El fichero ejecutable *TPSesion0.exe* también podrá ejecutarse como cualquier otro fichero ejecutable desde el propio Windows o desde el símbolo del sistema.

Nótese que los procesos de compilación, construcción y ejecución pueden realizarse todos conjuntamente con la opción *Build – Build and run*. Por último, la opción *Build – Rebuild* realiza todo el proceso de compilación y construcción desde el principio, independientemente de que los ficheros fuentes hayan sido modificados o no.

0.5 Depurar un proyecto

Las herramientas de depuración han sido creadas para ayudar a depurar y corregir un programa mientras éste se está ejecutando. Un depurador (*debugger*), básicamente ejecuta un programa mientras permite analizar sus funciones, variables e instrucciones. También permite detener la ejecución en un punto del programa determinado, al cual se le denomina punto de ruptura (*breakpoint*). Se podrán situar estos puntos de ruptura en cualquier lugar del código. Una vez alcanzado alguno de estos puntos, el depurador permite visualizar el valor actual de las variables y datos del programa.

0.5.1 Estableciendo puntos de ruptura

Se pueden usar puntos de ruptura para detener la ejecución del programa en una instrucción concreta. Para añadir un punto de ruptura, se sitúa el cursor en la línea en cuestión y se selecciona la opción *Debug – Toggle breakpoint* o simplemente se hace clic en el lateral izquierdo de la línea. Entonces la línea queda marcada con un círculo color rojo. Después de establecer los puntos de ruptura es necesario reconstruir de nuevo la aplicación mediante la opción *Rebuild.*

Cuando se ejecuta el depurador (*Debug – Start / Continue*), la ejecución del programa se detiene en dicho punto, lo que se indica con un triángulo amarillo en la línea de ruptura.

0.5.2 Ejecutando paso a paso el programa

Una vez detenida la ejecución en un punto de ruptura, el programa puede ejecutarse paso a paso de diferentes formas:

- *Next line*: El depurador ejecuta hasta la siguiente línea de código del programa.
- *Next instruction*: El depurador ejecuta hasta la siguiente instrucción del programa.
- *Step into / Step into instruction*: El depurador ejecuta hasta la siguiente instrucción del programa; si la instrucción actual es una función, el depurador ejecuta hasta la primera instrucción del código de la función.
- *Step out*: El depurador ejecuta hasta la siguiente instrucción del programa después de terminar la función actual (sube de nivel). Nótese que esta opción no debe utilizarse si la función actual es la función `main()`.
- *Run tu cursor*: El depurador continúa la ejecución del programa hasta detenerse en la línea donde se encuentra situado el cursor.

0.5.3 Visualizando el valor de variables

Se pueden visualizar las variables de dos formas diferentes:

- Seleccionando la opción *Debug – Debugging Windows - Watches* y escribir el nombre de la variable en la ventana de seguimiento *Watches*.
- Situar el cursor encima de la variable, pulsar el botón derecho del ratón y seleccionar la opción *Watch.* La variable será añadida a la lista de variables visualizadas en la ventana *Watches*.

Para concluir la práctica, ejecutar el depurador sobre el código realizando las siguientes actividades:

- Situar un punto de ruptura en la primera instrucción de la función `main()`.
- Ejecutar hasta ese punto de ruptura.
- Ejecutar el código línea a línea hasta la terminación del programa.
- Ejecutar de nuevo hasta el punto de ruptura.
- Visualizar el contenido de las variables `miNumero`, `numero` y `nIntentos`.

- Ejecutar de nuevo el código línea a línea hasta la terminación del programa observando el valor de las variables.
- Situar un segundo punto de ruptura en la última instrucción de la función `main()`.
- Ejecutar de nuevo hasta el primer punto de ruptura.
- Saltar hasta el segundo punto de ruptura.

Sesión 1 - Introducción a los gráficos en C con SDL 2.0

Descriptores: Gráficos, librería SDL 2.0, programación dirigida por eventos

Simple DirectMedia Layer (*SDL*) es un conjunto de bibliotecas desarrolladas en el lenguaje de programación C que proporcionan funciones básicas para realizar operaciones de dibujo en dos dimensiones, gestión de efectos de sonido y música, además de carga y gestión de imágenes. Aunque está programada en C, tiene wrappers a otros lenguajes de programación como C++, Ada, C#, BASIC, Erlang, Lua, Java, Python, etc. También proporciona herramientas para el desarrollo de videojuegos y aplicaciones multimedia. Una de sus grandes virtudes es el tratarse de una biblioteca multiplataforma, siendo compatible oficialmente con los sistemas Microsoft Windows, GNU/Linux, Mac OS y QNX, además de otras arquitecturas y sistemas como Sega Dreamcast, GP32, GP2X, etc. La biblioteca se distribuye bajo la licencia LGPL, que es la que ha provocado el gran avance y evolución de SDL.

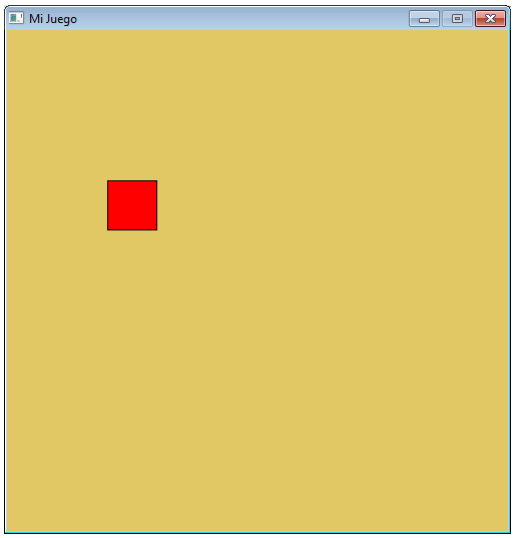
Para la realización de las prácticas utilizaremos *programación dirigida por eventos*. La programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen. Mientras que en la programación secuencial (o estructurada) es el programador el que define cuál va a ser el flujo del programa, en la programación dirigida por eventos será el propio usuario, o lo que sea que esté accionando el programa, el que dirija el flujo del programa. Aunque en la programación secuencial puede haber intervención de un agente externo al programa, estas intervenciones ocurrirán cuando el programador lo haya determinado, y no en cualquier momento como puede ser en el caso de la programación dirigida por eventos. El creador de un programa dirigido por eventos debe definir los eventos que manejarán su programa y las acciones que se realizarán al producirse cada uno de ellos, lo que se conoce como el administrador de evento. Los eventos soportados estarán determinados por el lenguaje de programación utilizado, por el sistema operativo e incluso por eventos creados por el mismo programador.

En la programación dirigida por eventos, al comenzar la ejecución del programa se llevarán a cabo las inicializaciones y demás código inicial y a continuación el programa quedará bloqueado hasta que se produzca algún evento. Cuando alguno de los eventos esperados por el programa tenga lugar, el programa pasará a ejecutar el código del correspondiente administrador de evento.

En contraposición al modelo clásico, la programación dirigida por eventos permite interactuar con el usuario en cualquier momento de la ejecución. Esto se consigue debido a que los programas creados bajo esta arquitectura se componen por un bucle

exterior permanente encargado de recoger los eventos, y distintos procesos que se encargan de tratarlos. Habitualmente, este bucle externo permanece oculto al programador que simplemente se encarga de tratar los eventos, aunque en algunos lenguajes, como ocurre con C, será necesaria su construcción.

La práctica consiste en realizar una aplicación gráfica en C dirigida por eventos que, utilizando la librería SDL 2.0, cree una ventana de dimensión *500x500* y título *"Mi Juego"*, sobre la cual se dibuje un cuadrado de lado *50* en la coordenada *(100,150)* de color rojo y borde negro sobre fondo marrón.



Para ello, realizar un proyecto *TPBloqueI.cbp* configurando *Project – Properties – Build targets – Release – Type – GUI application*, con un fichero fuente *TPBloqueI.c* con la siguiente estructura:

```
#include <SDL2/SDL.h>

int main(int argc, char ** argv)
{
  /* INICIALIZACIÓN

     - Crea una ventana con el título y tamaño especificados.
     - Recupera la superficie asociada a la ventana.
     - Crea un contexto de rendering 2D para la superficie.
  */
  ...

  // BUCLE DE EVENTOS SDL

  int terminar = 0;
  SDL_Event evento;

  // Repite mientras no se cierre la aplicación.
  while (!terminar)
  {
    // Repite mientras queden eventos en la cola.
    while (SDL_PollEvent(&evento))
    {
      switch(evento.type)
      {
```

```
        case SDL_QUIT: terminar = 1;

        // Resto de eventos.
        // case tipo_evento_1 : administrador_evento_1;
        // case tipo_evento_2 : administrador_evento_2;
        // ...
        // case tipo_evento_n : administrador_evento_n;

      }
    }

    /* PINTA EN LA PANTALLA
          - Pinta el fondo:
              - Establece el color de fondo.
              - Limpia el renderer.
          - Pinta el rectángulo:
              - Establece el color de relleno.
              - Pinta el rectángulo relleno.
              - Establece el color de borde.
              - Pinta el rectángulo borde.
          - Copia la superficie de la ventana en la pantalla.
    */
    ...

  }

  /* DESTRUCCIÓN Y SALIDA

      - Destruye el renderer para una ventana y sus texturas asociadas
      - Destruye la ventana.
      - Libera todos los subsistemas inicializados.
   */
  ...

  return 0;
}
```

Usar las siguientes funciones, ordenadas por categorías, de la librería SDL 2.0 (ver http://wiki.libsdl.org para más detalles):

SDL – Basics – Initialization and Shutdown

```
// Inicializa la librería SDL con los subsistemas especificados por
// los flags.
//
// Los subsistemas de manejo de eventos, entrada y salida de ficheros
// y hebras se inicializan por defecto.
//
// Para inicializar otros subsistemas, se debe indicar específicamente
// mediante el atributo flags.
//   flags – uno o varios (unidos por OR) de los diversos valores
//           posibles, entre los que destacamos los siguientes:
//     SDL_INIT_VIDEO – Inicializa el video y los eventos.
//     SDL_INIT_EVERYTHING – Inicializa todo.
//
// Devuelve 0 si ha tenido éxito.
// Devuelve un código de error negativo si se ha producido un fallo.
```

```
int SDL_Init(Uint32 flags)

// Libera todos los subsistemas inicializados.

void SDL_Quit(void)
```

SDL – Video – Display and Window Management

```
// Crea una ventana con el título, posición (x,y), tamaño (w,h) y
// flags especificados:
//   title – Título, en codificación UTF-8.
//   x – Posición x: SDL_WINDOWPOS_CENTERED o SDL_WINDOWPOS_UNDEFINED.
//   y – Posición y: SDL_WINDOWPOS_CENTERED o SDL_WINDOWPOS_UNDEFINED.
//   w – Anchura.
//   h – Altura.
//   flags – 0 o uno o varios (unidos por OR) de los valores
//           especificados en SDL_WindowFlags.
//
// Devuelve la ventana creada.
// Devuelve NULL si se ha producido un fallo.

SDL_Window * SDL_CreateWindow(const char * title,
                              int x,
                              int y,
                              int w,
                              int h,
                              Uint32 flags)

// Devuelve la superficie asociada a la ventana.
// Devuelve NULL si se ha producido un fallo.

SDL_Surface * SDL_GetWindowSurface(SDL_Window * window)

// Copia la superficie de una ventana en la pantalla.
//
// Devuelve 0 si ha tenido éxito.
// Devuelve un código de error negativo si se ha producido un fallo.

int SDL_UpdateWindowSurface(SDL_Window * window)

// Destruye la ventana.

void SDL_DestroyWindow(SDL_Window * window)
```

SDL – Video – 2D Accelerated Rendering

```
// Crea un contexto de rendering 2D para una superficie.
// Devuelve el contexto de rendering creado.
// Devuelve NULL si se ha producido un fallo.

SDL_Renderer * SDL_CreateSoftwareRenderer(SDL_Surface * surface)

// Establece el color actual para las operaciones de dibujo en el
// renderer.
// El color viene dado por las componenentes:
//   r - Valor del componente rojo.
//   g - Valor del componente verde.
//   b - Valor del componente azul.
//   a - Valor del componente alpha (opacidad). Normalmente se utiliza
```

```
//       para el componente alpha el valor SDL_ALPHA_OPAQUE (255).
//
// Devuelve 0 si ha tenido éxito.
// Devuelve un código de error negativo si se ha producido un fallo.

int SDL_SetRenderDrawColor(SDL_Renderer * renderer,
                           Uint8 r,
                           Uint8 g,
                           Uint8 b,
                           Uint8 a)

// Borra el renderer utilizando el color actual.
// Devuelve 0 si ha tenido éxito.
// Devuelve un código de error negativo si se ha producido un fallo.

int SDL_RenderClear(SDL_Renderer * renderer)

// Dibuja un rectángulo relleno en el renderer utilizando el color
// actual.
// EL rectángulo se especifica mediante un argumento de tipo SDL_Rect
// con los siguientes atributos:
//   x - Coordenada x de la esquina superior del rectángulo.
//   y - Coordenada y de la esquina superior del rectángulo.
//   w - Anchura del rectángulo.
//   h - Altura del rectángulo.
//
// Devuelve 0 si ha tenido éxito.
// Devuelve un código de error negativo si se ha producido un fallo.

int SDL_RenderFillRect(SDL_Renderer *  renderer,
                       const SDL_Rect * rect)

// Dibuja un rectángulo en el renderer utilizando el color actual.
// EL rectángulo se especifica mediante un argumento de tipo SDL_Rect
// con los siguientes atributos:
//   x - Coordenada x de la esquina superior del rectángulo.
//   y - Coordenada y de la esquina superior del rectángulo.
//   w - Anchura del rectángulo.
//   h - Altura del rectángulo.
//
// Devuelve 0 si ha tenido éxito.
// Devuelve un código de error negativo si se ha producido un fallo.

int SDL_RenderDrawRect(SDL_Renderer * renderer,
                       const SDL_Rect * rect)

// Destruye el renderer para una ventana y sus texturas asociadas.

void SDL_DestroyRenderer(SDL_Renderer * renderer)
```

Las funciones SDL_RenderFillRect y SDL_RenderDrawRect tienen como argumento un rectángulo, que se especifica mediante una estructura **SDL_Rect** con los siguientes atributos:

- **x** - Coordenada x de la esquina superior del rectángulo.
- **y** - Coordenada y de la esquina superior del rectángulo.
- **w** - Anchura del rectángulo.
- **h** - Altura del rectángulo.

SDL – Input Events – Event Handling

```
// Recupera el siguiente evento de la cola de eventos.
```

```
// Devuelve 0 si ha tenido éxito.
// Devuelve 1 si se ha recuperado un evento y 0 si no hay eventos.

int SDL_PollEvent(SDL_Event * event)

// El tipo SDL_Event es una unión que contiene estructuras para los
// diversos tipos de eventos con diversos atributos. Entre sus
// atributos, cabe destacar el siguiente:
//   type – Indica el tipo de evento que ha ocurrido. Entre todos
//          los valores que puede tomar, destacamos los siguientes:
//     SDL_QUIT – Cierra la ventana.
//     SDL_KEYDOWN – Pulsa una tecla.
//     SDL_TEXTINPUT – Introducción de texto.
```

Sesión 2 - Creación de funciones para manejo de gráficos y gestión de errores

Descriptores: Estructuras de datos, abstracciones funcionales, gestión de errores.

2.1 Abstracción de funciones para manejo de la librería SDL y estructura de datos asociada

Modificar el código del fichero *TPBloqueI.c* para:

1. Definir un tipo de datos **MiSDL** que contenga todas las variables del entorno gráfico**:**

```
typedef struct
{
    SDL_Window * window;
    SDL_Surface * surface;
    SDL_Renderer * renderer;
} MiSDL;
```

2. Definir las siguientes funciones propias para el manejo de gráficos:

```
// Construye una instancia MiSDL cuyos campos han
// sido inicializados mediante las operaciones de inicialización.
//
// Devuelve la instancia MiSDL creada.

MiSDL * MiSDL_Iniciar(char titulo[],
                      int ancho,
                      int alto)
{
    //  - Crea una ventana con el título y tamaño especificados.
    //  - Recupera la superficie asociada a la ventana.
    //  - Crea un contexto de rendering 2D para la superficie.
    ...
```

```
}

// Borra el renderer asociado a miSDL utilizando el color
// especificado.

void MiSDL_LimpiarRender(MiSDL * miSDL,
                         SDL_Color color)
{ ... }

// Pinta un rectángulo en el renderer asociado a miSDL utilizando
// los parámetros especificados.
//   rectangulo - Posición y dimensiones del rectángulo.
//   colorBorde - Color del borde del cuadrado.
//   colorRelleno - Color del relleno del cuadrado.

void MiSDL_PintarRectanguloBordeRelleno(MiSDL * miSDL,
                                        SDL_Rect rectangulo,
                                        SDL_Color colorBorde,
                                        SDL_Color colorRelleno)
{ ... }

// Copia la superficie de la ventana asociada a miSDL en la
// pantalla.

void MiSDL_ActualizarVentana(MiSDL * miSDL)
{ ... }

// Realiza todas las operaciones de limpieza asociadas a miSDL.

void MiSDL_Terminar(MiSDL * miSDL)
{
    // - Destruye la ventana y el renderer asociado a miSDL.
    // - Libera todos los subsistemas inicializados.
    // - Destruye miSDL
    ...
}
```

Las funciones `MiSDL_LimpiaRender` y `MiSDL_PintarRectanguloBordeRelleno` utilizan el tipo **`SDL_Color`** (definido por la librería SDL) que es una estructura con los siguientes atributos (todos en el rango 0-255):

```
Uint8 r  // Valor del componente rojo.
Uint8 g  // Valor del componente verde.
Uint8 b  // Valor del componente azul.
Uint8 a  // Valor del componente alpha (opacidad).
```

Para el componente alpha, normalmente suele utilizarse el valor **`SDL_ALPHA_OPAQUE`** (**`255`**) definido por SDL.

Para el manejo de errores en estas funciones, cuando alguna de las llamadas a funciones de SDL devuelva un valor de error, se llamará a la siguiente función:

```
// Imprime por pantalla el mensaje de error SDL ocurrido.
// Termina la aplicación con un código de fallo.
```

```
void MiSDL_Error()
{ ... }
```

Para obtener el mensaje del error ocurrido, utilizar la función `SDL_GetError()` definida en la librería SDL

SDL – Basics – Error Handling

```
// Devuelve un mensaje con información sobre el error que ha ocurrido.
// Devuelve NULL Si no ha ocurrido ningún error.

const char * SDL_GetError(void)
```

Para terminar la aplicación con un código de fallo, realizar la función **`exit(EXIT_FAILURE)`** de la librería `<stdlib.h>`

3. Modificar la función `main` para que declare una variable `miSDL` de tipo `MiSDL *` y realice las llamadas requeridas a las funciones definidas previamente, de forma que la aplicación mantenga la misma funcionalidad que en la práctica de la sesión 1.

2.2 Abstracción de funciones para realización de aplicaciones gráficas de juegos y estructura de datos asociada

En esta práctica se realizará una reestructuración del código de la siguiente forma:

1. Definir una estructura de datos **`MiJuego`** que contenga todas las variables del entorno de la aplicación.

```
typedef struct
{
    MiSDL * miSDL;
    int ancho, alto;
    SDL_Rect cuadrado;
    SDL_Color colorFondo, colorCuadradoBorde, colorCuadradoRelleno;
} MiJuego;
```

2. Definir las siguientes funciones sobre la estructura `MiJuego`:

```
void MiJuego_Pintar(MiJuego * miJuego)
{
    //  - Borra el renderer asociado a miSDL de miJuego con el
    //    color de fondo.
    //  - Pinta el cuadrado con las características especificadas
    //    en miJuego.
    //  - Copia la superficie de la ventana asociada a miSDL de
    //    miJuego en la pantalla.
    ...
}
```

```
// Imprime por pantalla el mensaje de error indicado en el
//   argumento error.
// Termina la aplicación con un código de fallo.

void MiJuego_Error(char * error)
{ ... }

// Construye una instancia MiJuego con los valores especificados.
//
// Provoca un mensaje de error y terminación en los siguientes casos:
//   - Si ancho es menor o igual que 0.
//   - Si alto es menor o igual que 0.
//   - Si lado es mayor o igual que ancho.
//   - Si lado es mayor o igual que alto.
//
// Devuelve la instancia MiJuego creada.

MiJuego * MiJuego_Iniciar(char * titulo,
                          int ancho,
                          int alto,
                          int x,
                          int y,
                          int lado,
                          SDL_Color colorFondo,
                          SDL_Color colorCuadradoBorde,
                          SDL_Color colorCuadradoRelleno)
{ ... }

// Bucle para el control de eventos sobre miJuego.

void MiJuego_Bucle(MiJuego * miJuego)
{
  int terminar = 0;
  SDL_Event evento;

  // Repite mientras no se cierre la aplicación.
  while (!terminar)
  {
    // Repite mientras queden eventos en la cola.
    while (SDL_PollEvent(&evento))
    {
      switch(evento.type)
      {
        case SDL_QUIT: terminar = 1;

      }
    }
    MiJuego_Pintar();
  }
}

// Libera el entorno gráfico miSDL asociado a miJuego.

void MiJuego_Terminar(MiJuego * miJuego)
{ ... }
```

3. Modificar la función `main` para que declare una variable `miJuego` de tipo `MiJuego *` y realice las llamadas requeridas a las funciones definidas previamente, de forma que la aplicación mantenga la misma funcionalidad que en la práctica de la sesión 1.

Sesión 3 - Animación gráfica

Descriptores: Eventos de teclado, animación, diagrama de fases.

En esta práctica, se realizará una animación gráfica consistente en mover el cuadrado a lo largo de la ventana controlando el cambio de dirección mediante las flechas del teclado. La aplicación tendrá dos fases: una de reposo y otra de movimiento.

3.1 Ampliación de la estructura `MiJuego`

Será necesario añadir a la estructura `MiJuego` atributos para indicar la dirección y la velocidad del movimiento, así como la fase del juego. La estructura `MiJuego` queda por tanto de la siguiente forma:

```
typedef struct
{
    MiSDL * miSDL;
    int ancho, alto;
    SDL_Rect cuadrado;
    SDL_Color colorFondo, colorCuadradoBorde, colorCuadradoRelleno;
    int dx, dy;
    int retardo;
    int fase;
} MiJuego;
```

La dirección del movimiento se establecerá mediante el control de dos nuevos atributos de la estructura `MiJuego`, **`dx`** y **`dy`**, que representan la dirección en la coordenada **`x`** y en la coordenada **`y`** de la siguiente forma:

(0,0) - Sin movimiento.
(0,1) - Hacia abajo.
(0,-1) - Hacia arriba.
(1,0) - Hacia la derecha.
(-1,0) Hacia la izquierda

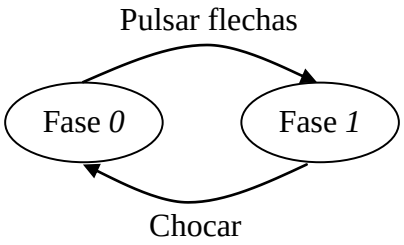
Para establecer la velocidad del movimiento, se introducirá un retardo el cual se le pasará a la función `MiJuego_Iniciar` como un nuevo argumento y que se guardará en un nuevo atributo de la estructura `MiJuego`. Para realizar el retardo se utiliza la función `SDL_Delay` con la siguiente sintaxis:

SDL – Timers – Timer Support

```
// Espera el número de milisegundos especificados

void SDL_Delay(Uint32 ms)
```

3.2 Diagrama de fases del juego y transiciones

La animación está compuesta por dos fases siguiendo el siguiente diagrama:



Las fases del programa realizan lo siguiente:

- Fase *0*: Pinta el cuadrado quieto en la posición *(0,0)*.
- Fase *1*: Cuadrado moviéndose en la dirección especificada. Si el usuario pulsa cualquiera de las flechas del teclado, la dirección del movimiento cambia a la definida por la tecla pulsada.

Las transiciones entre fases ocurrirán de la siguiente forma:

- Fase *0*: Se pasa a la fase *1* cuando se pulsa cualquiera de las flechas de teclado.
- Fase *1*: Se pasa a la fase *0* cuando el cuadrado choca con alguno de los bordes de la ventana.

Para controlar el inicio y la transición entre fases, se añadirá un nuevo atributo `fase`, de tipo entero, a la estructura `MiJuego` y se utilizarán las siguientes funciones:

```
// Inicia la fase especificada en el argumento fase.
//
// Fase 0: Inicia la posición del cuadrado a (0,0).
// Fase 1: No requiere inicialización.

void MiJuego_IniciarFase(MiJuego * miJuego, int fase)
{ ... }


// Termina la fase actual y realiza la transición a la siguiente fase.
//
// Fase 0: Inicia la fase 1.
// Fase 1: Inicia la fase 0.

void MiJuego_TerminarFase(MiJuego * miJuego)
{ ... }
```

Para llevar a cabo la fase *1*, es decir, realizar las operaciones de movimiento y comprobar si ha habido choque, se implementarán las operaciones `MiJuego_Choque` y `MiJuego_Mover`.

```
// Devuelve 0 (falso) si no ha habido choque.
// Devuelve 1 (cierto) si ha habido choque.

int MiJuego_Choque(MiJuego * miJuego)
{ ... }

// Actualiza la posición del cuadrado en función de la dirección del
// movimiento.
//
// Si al mover el cuadrado ha habido choque, termina la fase.

void MiJuego_Mover(MiJuego * miJuego)
{ ... }
```

3.3 Captura de eventos del teclado

Cuando se pulse una tecla, ocurrirá un evento de tipo **`SDL_KEYDOWN`**. En este caso, para tratar el tipo de tecla que se ha pulsado, se llamará a la función `MiJuego_EventoTeclado` pasándole como argumento el código de la tecla pulsada, el cual se encuentra en el atributo **`key.keysym.sym`** del evento.

```
// Realiza la acción correspondiente según la tecla indicada en el
// argumento code:
//   SDLK_LEFT - Izquierda.
//   SDLK_RIGHT - Derecha.
//   SDLK_UP - Arriba.
//   SDLK_DOWN - Abajo.
//
// Si se está en la fase 0, cualquiera de estas pulsaciones provoca la
// terminación de la fase.
//
// Si está en la fase 1, se modifica la dirección del movimiento
// según la tecla pulsada.

void MiJuego_EventoTeclado(MiJuego * miJuego, SDL_Keycode code)
{ ... }
```

Para incluir todo el proceso de animación anterior, realizaremos finalmente las siguientes modificaciones en la función `MiJuego_Bucle`:

- Añadir el tipo de evento **`SDL_KEYDOWN`** que realice la llamada a la función `MiJuego_EventoTeclado` pasándole como argumento el código de la tecla pulsada, el cual se encuentra en el atributo **`key.keysym.sym`** de `evento`.
- Añadir, antes de la llamada a la función `MiJuego_Pintar`, una llamada a la función `MiJuego_Mover`, si se está en la fase *1*.
- Añadir, después de la llamada a la función `MiJuego_Pintar`, una llamada a la función `SDL_Delay`, si se está en la fase *1*.

La función `MiJuego_Bucle` quedaría por tanto con el siguiente código:

```
void MiJuego_Bucle(MiJuego * miJuego)
{
  int terminar = 0;
  SDL_Event evento;

  // Repite mientras no se cierre la aplicación.
  while (!terminar)
  {
    // Repite mientras queden eventos en la cola.
    while (SDL_PollEvent(&evento))
    {
      switch(evento.type)
      {
        case SDL_KEYDOWN:
            MiJuego_EventoTeclado(miJuego,evento.key.keysym.sym);
            break;
        case SDL_QUIT: terminar = 1;
      }
    }
    if (miJuego->fase==1) MiJuego_Mover(miJuego);
    MiJuego_Pintar(miJuego);
    if (miJuego->fase==1) SDL_Delay(miJuego->retardo);
  }
}
```

Adicionalmente, respecto a la función `MiJuego_Iniciar` deben tenerse en cuenta las siguientes consideraciones:

- Dado que la posición de inicio del cuadrado es constante *(0,0)* no será necesario pasarle los valores x e y como argumentos de la función.
- Se añadirá un nuevo argumento a la función para el `retardo`.
- Se debe comprobar que las dimensiones de la ventana sean divisibles por el lado del cuadrado y en caso contrario llamar a la función `MiJuego_Error()`
- Es necesario iniciar la fase *0*.

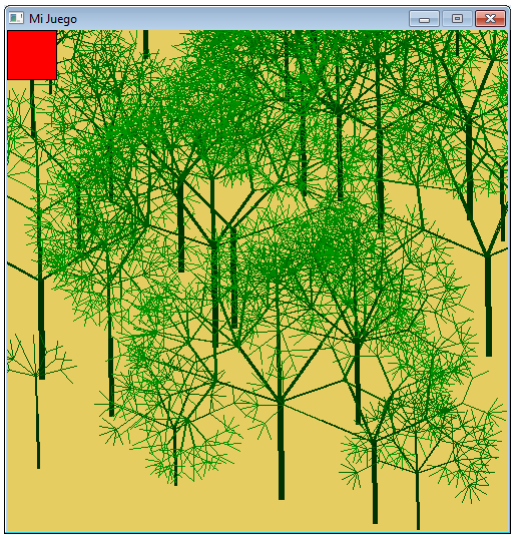
La sintaxis de la función queda por tanto de la siguiente manera:

```
// Construye una instancia MiJuego con los valores especificados.
//
// Provoca un mensaje de error y terminación en los siguientes casos:
//   - Si ancho es menor o igual que 0.
//   - Si alto es menor o igual que 0.
//   - Si ancho no es divisible por lado.
//   - Si alto no es divisible por lado.
//
// Inicia la fase 0 del juego.

// Devuelve la instancia MiJuego creada.

MiJuego * MiJuego_Iniciar(char * titulo,
                          int ancho,
                          int alto,
                          int lado,
                          SDL_Color colorFondo,
                          SDL_Color colorCuadradoBorde,
                          SDL_Color colorCuadradoRelleno,
                          int retardo)
{ ... }
```

Sesión 4 - Recursividad

Descriptores: Recursividad, generación de números aleatorios, texturas.

Esta práctica consiste en definir un fondo de ventana que simule una selva de árboles donde cada árbol se dibujará de forma recursiva.



4.1 Ampliación de `MiSDL`

En esta práctica se van a utilizar nuevas funciones de la librería SDL, por lo que habrá que ampliar la estructura `MiDSL` con un nuevo atributo que haga referencia a la textura, así como incluir las funciones necesarias sobre el tipo `MiDSL`.

La estructura `MiDSL` queda por tanto de la siguiente forma:

```
typedef struct
{
    SDL_Window * window;
    SDL_Surface * surface;
    SDL_Renderer * renderer;
    SDL_Texture * texture;
} MiSDL;
```

Las funciones que se añadirán se indican a continuación:

```
// Pinta una línea usando los siguientes parámetros:
//  - miSDL entorno.
//  - x1,y1 coordenadas de inicio.
//  - x2,y2 cordenadas de final.
//  - color de la línea.
void MiSDL_PintarLinea(MiSDL * miSDL,
                       int x1,int y1,
                       int x2,int y2,
                       SDL_Color color)
{ ... }

// Crea una textura para el entorno gráfico miSDL a partir del
// renderer de miSDL.

void MiSDL_CrearTextura(MiSDL * miSDL)
{ ... }

// Libera la textura del entorno gráfico miSDL.

void MiSDL_LiberarTextura(MiSDL * miSDL)
{ ... }

// Copia la textura de miSDL al renderer de miSDL.

void MiSDL_CopiarTextura(MiSDL * miSDL)
{ ... }
```

A continuación se muestran las funciones de la librería SDL que serán utilizadas para implementar las funciones anteriores:

SDL – Video – 2D Accelerated Rendering

```
// Dibuja una línea en el renderer utilizando el color actual.
//   x1 - Coordenada x del punto de inicio.
//   y1 - Coordenada y del punto de inicio.
//   x2 - Coordenada x del punto de fin.
//   y2 - Coordenada x del punto de fin.
//
// Devuelve 0 si ha tenido éxito.
// Devuelve un código de error negativo si se ha producido un fallo.

int SDL_RenderDrawLine(SDL_Renderer * renderer,
                       int           x1,
                       int           y1,
                       int           x2,
                       int           y2)

// Crea una textura para un renderer a partir de una superficie.
//
// Devuelve la textura creada.
// Devuelve NULL si se ha producido un fallo.

SDL_Texture * SDL_CreateTextureFromSurface(SDL_Renderer * renderer,
                                           SDL_Surface *  surface)
```

```
// Copia una porción de la textura al renderer.
//   scrrect – Rectángulo origen o NULL para la textura completa.
//   dstrect – Rectángulo destino o NULL para el renderer completo.
//
// Devuelve 0 si ha tenido éxito.
// Devuelve un código de error negativo si se ha producido un fallo.

int SDL_RenderCopy(SDL_Renderer *   renderer,
                   SDL_Texture *    texture,
                   const SDL_Rect * srcrect,
                   const SDL_Rect * dstrect)

// Destruye la textura.

void SDL_DestroyTexture(SDL_Texture * texture)
```

4.2 Generación de números aleatorios

Se pretende que cada vez que generemos una selva, ésta sea distinta, por lo que se recurrirá a la generación de números aleatorios. Con este objetivo se definirá la siguiente función:

```
// Devuelve un número entero aleatorio entre a y b.

int MiArbol_Aleatorio(int a, int b)
{
    return a+rand()%(b-a+1);
}
```

Para asegurar que la secuencia de números aleatorios generados sea distinta cada vez que se ejecuta la aplicación, se establecerá, al principio de la aplicación, una semilla de números aleatorios recurriendo al reloj del sistema, mediante la instrucción **`srand(time(NULL))`**. Será necesario incluir la librería `<time.h>`.

4.3 Dibujo recursivo de los árboles

Para dibujar los árboles, se implementará la función recursiva `MiArbol_PintarArbol` con la sintaxis indicada a continuación. Notar que el valor del número **PI** se ha incluido como una macro antes de la función. También ha sido necesario incluir la librería `<math.h>` para poder calcular cosenos utilizando la función **`cos`**. La altura mínima permitida, **`HMIN`**, también se ha definido mediante una macro igual a **`15`**.

.

```
#include <math.h>
#define PI 3.1416
#define HMIN 15

// Pinta un árbol con los siguientes parámetros:
//    miSDL – Entorno gráfico MiSDL con el que se pinta.
//    h – altura del tronco.
```

```
//    x – coordenada x de inicio del tronco.
//    y – coordenada y de inicio del tronco.
//    angulo – ángulo de salida del tronco en radianes.
//    color – color del tronco.

void MiArbol_PintarArbol(MiSDL * miSDL,
                         int h,
                         int x,
                         int y,
                         double angulo,
                         SDL_Color color)
```

La función recursiva se definirá de la siguiente forma:

- Condición base (`h<HMIN`): En este caso la función termina sin realizar ningún código.
- Recurrencia (`h>=HMIN`):
  - Establecer coordenadas del final del tronco:
    - `xFinal = x + h*cos(angulo);`
    - `yFinal = y - h*sin(angulo);`
  - Pintar el tronco.
    - Calcular el grosor del tronco en función de la altura, pintando un número de líneas `i` entre `0` y `h/HMIN-1`.
    - Para cada línea `i`, realizar una llamada a la función `MiSDL_PintarLinea` con las coordenadas de inicio (`x+i,y`) y final (`xFinal+i,yFinal`) y el color establecido.
  - Calcular `colorAclarado` sumando `20` a cada componente del color actual mayor que `0` hasta alcanzar el máximo de `255`.
  - Establecer el número de ramas `n` de forma aleatoria entre `1` y `7`.
  - Para cada rama `j`, desde `1` hasta `n`, realizar una llamada recursiva con los siguientes nuevos valores:
    - `hNueva = MiArbol_Aleatorio(h,1.1*h)*2.0/3.0;`
    - `xNueva = xFinal;`
    - `yNueva = yFinal;`
    - `anguloNuevo = angulo-PI/2-PI/(2*n)+j*PI/n;`
    - `colorNuevo = colorAclarado;`

4.4 Dibujo de la selva

Para pintar la selva se implementará la función `MiArbol_PintarSelva` que realiza lo siguiente:

- Establecer el color de fondo de la selva.
- Para cada cuadrícula, pintar un árbol realizando una llamada a la función `MiArbol_PintarArbol` con los siguientes parámetros:
  - Altura (`h`) aleatoria entre la mitad del ancho de la cuadrícula y el ancho de la cuadrícula.
  - Posición (`x`,`y`) aleatoria dentro de los límites de la cuadrícula.
  - Ángulo igual a `PI/2`.
  - Color de los árboles.

```
// Pinta una selva de árboles aleatorios con los siguientes valores:
//   ancho – ancho de la ventana.
//   alto  - alto de la ventana.
//   tramos_x – número de cuadrículas horizontales.
//   tramos_y – número de cuadrículas verticales.
//   colorArbol – color inicial de los árboles.
//   colorFondo – color del fondo.

void MiArbol_PintarSelva(MiSDL * miSDL,
                         int ancho,
                         int alto,
                         int tramos_x,
                         int tramos_y,
                         SDL_Color colorArbol,
                         SDL_Color colorFondo)
{...}
```

4.5 Establecimiento de la textura

Finalmente, dado que el proceso de pintado de la selva es recursivo, el tiempo requerido resulta excesivo. Por lo tanto, una vez calculada la imagen, ésta se establecerá como textura y cada vez que se pinte la pantalla simplemente será necesario copiar la textura a la pantalla.

Como la textura lleva ya incluido su propio color de fondo ya no resulta necesario el atributo `colorFondo` dentro de la estructura de `MiJuego`, por lo que podemos eliminar este atributo, quedando la estructura `MiJuego` de la siguiente forma:

```
typedef struct
{
    MiSDL * miSDL;
    int ancho, alto;
    SDL_Rect cuadrado;
    SDL_Color colorCuadradoBorde, colorCuadradoRelleno;
    int dx, dy;
    int retardo;
    int fase;
} MiJuego;
```

Para establecer el número de árboles en la selva se añadirán dos nuevos argumentos, `tramos_x` y `tramos_y`, a la función `MiJuego_Iniciar`, que indican el número de cuadrículas por ancho y por alto en la que se dividirá la ventana para pintar la selva.
En la versión facilitada por los profesores de la asignatura, se utilizan los valores `tramos_x=5` y `tramos_y=5`. Adicionalmente, se incluye otro argumento `colorArbol` que indican el color inicial de los árboles. Notar que el argumento `colorFondo` no se ha eliminado de la función `MiJuego_Iniciar` porque se requiere para establecer el color de fondo de la selva, aunque ahora ya no será necesario almacenar su valor como atributo de `MiJuego`.

La función `MiJuego_Iniciar` debe realizar ahora las operaciones para pintar la selva y establecer la textura. Por otra parte, la función `MiJuego_Pintar` ya no se iniciará borrando el renderer con el color de fondo, sino copiando al renderer la superficie establecida como textura.

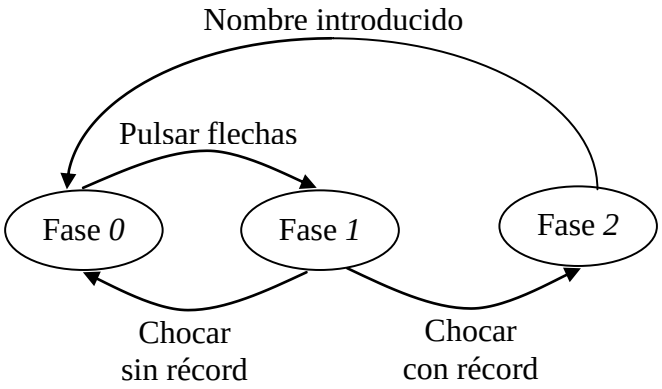
Además de implementar las nuevas funciones, será necesario ampliar la función `MiJuego_Terminar` con las operaciones requeridas para destruir la textura asociada. Las nuevas operaciones quedan como siguen:

```
// Construye una instancia MiJuego con los valores especificados.
//
// Provoca un mensaje de error y terminación en los siguientes casos:
//   - Si ancho es menor o igual que 0.
//   - Si alto es menor o igual que 0.
//   - Si ancho no es divisible por lado.
//   - Si alto no es divisible por lado.
//
// Inicia la fase 0 del juego.
//
// PINTA LA SELVA
//   - Pinta la selva aleatoria para los valores de tramos_x y
//     tramos_y establecidos.
//   - Crea la textura.
//
// Devuelve la instancia MiJuego creada.

MiJuego * MiJuego_Iniciar(char * titulo,
                          int ancho,
                          int alto,
                          int lado,
                          SDL_Color colorFondo,
                          SDL_Color colorCuadradoBorde,
                          SDL_Color colorCuadradoRelleno,
                          int retardo,
                          int tramos_x,
                          int tramos_y,
                          SDL_Color colorArbol)
{ ... }

// Pinta los gráficos asociados a miJuego en la pantalla.

void MiJuego_Pintar(MiJuego * miJuego)
{
    //  - Copia la textura de miSDL al renderer de miSDL.
    //  - Pinta el cuadrado con las características especificadas
    //    en miJuego.
    //  - Copia la superficie de la ventana asociada a miSDL de
    //    miJuego en la pantalla.
    ...
}

// Libera el entorno gráfico miSDL asociado a miJuego
// y las texturas asociadas.

void MiJuego_Terminar(MiJuego * miJuego)
{ ... }
```

Sesión 5 - Gestión de récords de tiempos

Descriptores: Librería SDL_ttf, escritura de datos en pantalla, lectura de datos por teclado, lectura y escritura de datos en ficheros, control del tiempo de ejecución, arrays.

La práctica consiste en ampliar la aplicación de forma que se mantenga un registro con los récords establecidos por los usuarios. Esta ampliación requiere una nueva fase (fase *2*) de ejecución siguiendo el siguiente diagrama:



En este diagrama cuando estando en la fase *1* se choca, se pasará a la fase *0* si el usuario no ha producido un nuevo récord de tiempo, y a la fase *2* si el usuario ha establecido un nuevo récord. Un récord se establece cuando el usuario ha conseguido jugar durante un tiempo mayor que alguno de los ya registrados. Una vez en la fase *2*, se pedirá el nombre del usuario para actualizar un fichero que almacena un máximo de *5* récords. Una vez introducido el nombre y actualizado el fichero, se pasa a la fase *0* donde se podrá iniciar de nuevo el juego. La fase *0* también se modificará para que muestre por pantalla los récords establecidos hasta el momento. Por último, al iniciar la aplicación y antes de iniciar la fase *0*, deben leerse los récords de un fichero para cargarlos en memoria.

5.1 Ampliación de `MiSDL`

Para escritura en pantalla en modo gráfico es necesario incluir la librería `<SDL_ttf.h>` para manejo de las fuentes (http://www.libsdl.org/projects/SDL_ttf/). SDL_ttf es una librería de rendering de fuentes *TrueType* que se usa como extensión de la librería SDL para mostrar salida de texto de alta calidad en entorno gráfico. Adicionalmente también se incluye la librería `<string.h>` para manejo de cadenas de caracteres. En este apartado se añadirán nuevas funciones y se modificarán algunas de las existentes. Es necesario también disponer de un fichero de fuentes de texto (por ejemplo, "arial.ttf") que se ubicará en el directorio donde se genere el fichero ejecutable. El código a incluir se muestra a continuación:

```
#include <SDL2/SDL_ttf.h>
#include <string.h>

void MiSDL_TTF_Error()
```

```
{
    printf("%s\n", TTF_GetError());
    exit(EXIT_FAILURE);
}

MiSDL * MiSDL_Iniciar(char titulo[], int ancho, int alto)
{
    if (TTF_Init()<0) MiSDL_TTF_Error();
    ...
}

void MiSDL_Terminar(MiSDL * miSDL)
{
    TTF_Quit();
    ...
}

// Crea y devuelve una fuente a partir del fichero y tamaño especificados.

TTF_Font * MiSDL_CrearFuente(const char * ficheroFuente, int tamFuente)
{
    TTF_Font * fuente = TTF_OpenFont(ficheroFuente,tamFuente);
    if (fuente == NULL) MiSDL_TTF_Error();
    return fuente;
}

// Destruye la fuente

void MiSDL_LiberarFuente(TTF_Font * fuente)
{
    TTF_CloseFont(fuente);
}

// Pinta el texto centrado en la coordenada y de pantalla.
// El rectángulo que rodea al texto se ajusta al tamaño del texto.

void MiSDL_PintarTexto(MiSDL * miSDL,
                       int y,
                       const char * texto,
                       SDL_Color colorTexto,
                       SDL_Color colorBorde,
                       SDL_Color colorRelleno,
                       TTF_Font * fuente)
{
    char textoImprimir[100];
    sprintf(textoImprimir," %s ",texto);
    SDL_Surface * surface = TTF_RenderText_Blended(fuente,textoImprimir,colorTexto);
    if (surface == NULL) MiSDL_TTF_Error();
    SDL_Rect rectangulo;
    rectangulo.w = surface->w;
    rectangulo.h = surface->h;
    rectangulo.x = miSDL->surface->w/2-rectangulo.w/2;
    rectangulo.y = y;
    MiSDL_PintarRectanguloBordeRelleno(miSDL,rectangulo,colorBorde,colorRelleno);
    if (SDL_BlitSurface(surface,NULL,miSDL->surface,&rectangulo) < 0) MiSDL_Error();
    SDL_FreeSurface(surface);
}

// Pinta el texto centrado en la coordenada y de pantalla.
// El rectángulo que rodea al texto es de tamaño fijo.

void MiSDL_PintarTextoEntrada(MiSDL * miSDL,
                       int y,
                       int w,
                       int h,
                       const char * texto,
                       SDL_Color colorTexto,
                       SDL_Color colorBorde,
                       SDL_Color colorRelleno,
                       TTF_Font * fuente)
{
    char textoImprimir[100];
    sprintf(textoImprimir," %s ",texto);
    SDL_Rect rectangulo;
    rectangulo.w = w;
    rectangulo.h = h;
    rectangulo.x = miSDL->surface->w/2-rectangulo.w/2;
```

```
    rectangulo.y = y;
    SDL_Surface * surface = TTF_RenderText_Blended(fuente,textoImprimir,colorTexto);
    if (surface == NULL) MiSDL_TTF_Error();
    MiSDL_PintarRectanguloBordeRelleno(miSDL,rectangulo,colorBorde,colorRelleno);
    if (SDL_BlitSurface(surface,NULL,miSDL->surface,&rectangulo) < 0) MiSDL_Error();
    SDL_FreeSurface(surface);
}
```

5.2 Ampliación de la estructura MiJuego

Se deberá ampliar la estructura `MiJuego` de forma que contenga los siguientes nuevos atributos:

- `tiempo`: tiempo conseguido por el usuario en la partida.
- `nombre`: nombre del último usuario que ha establecido un récord.
- `usuarios`: array de los últimos *5* usuarios con récords y sus tiempos.
- `ficheroFuente1,ficheroFuente2,ficheroFuente3`: nombres de los ficheros con las fuentes de letras utilizadas por el juego para los distintos tipos de mensaje.
- `colorTexto1,colorRectanguloBorde1,colorRectanguloRelleno1`: colores del texto, borde y relleno del rectángulo para los mensajes del tipo *1* (información al usuario).
- `colorTexto2,colorRectanguloBorde2,colorRectanguloRelleno2`: colores del texto, borde y relleno del rectángulo para los mensajes del tipo *2* (petición de acción al usuario).
- `colorTexto3,colorRectanguloBorde3,colorRectanguloRelleno3`: colores del texto, borde y relleno del rectángulo para los mensajes del tipo *3* (eco de la lectura de datos desde teclado).
- `nombreFichero`: El nombre del fichero de récords.

La estructura `MiJuego`, junto con las macros requeridas, quedan de la siguiente forma:

```
#define TAMNOMBRE 20
#define TAMUSUARIO 5

typedef struct
{
  char nombre[TAMNOMBRE];
  int tiempo;
} Usuario;

typedef struct
{
  MiSDL * miSDL;
  int ancho, alto;
  SDL_Rect cuadrado;
  SDL_Color colorFondo, colorCuadradoBorde, colorCuadradoRelleno;
  int dx, dy;
  int retardo;
  int fase;
  int tiempo;
  char nombre[TAMNOMBRE];
  Usuario usuarios[TAMUSUARIO];
  TTF_Font * fuente1, * fuente2, * fuente3;
```

```
  SDL_Color colorTexto1,colorRectanguloBorde1,colorRectanguloRelleno1;
  SDL_Color colorTexto2,colorRectanguloBorde2,colorRectanguloRelleno2;
  SDL_Color colorTexto3,colorRectanguloBorde3,colorRectanguloRelleno3;
  char nombreFichero[40];
} MiJuego;
```

5.3 Ampliación de las funciones `MiJuegoIniciar` y `MiJuegoTerminar`

Se ampliará la función `MiJuegoIniciar` para que inicialice los parámetros adicionales.

```
MiJuego * MiJuego_Iniciar(char * titulo,
                          int ancho,
                          int alto,
                          int lado,
                          SDL_Color colorFondo,
                          SDL_Color colorCuadradoBorde,
                          SDL_Color colorCuadradoRelleno,
                          int retardo,
                          int tramos_x,
                          int tramos_y,
                          SDL_Color colorArbol,
                          char * ficheroFuente1,
                          char * ficheroFuente2,
                          char * ficheroFuente3,
                          int tamFuente1,
                          int tamFuente2,
                          int tamFuente3,
                          SDL_Color colorTexto1,
                          SDL_Color colorRectanguloBorde1,
                          SDL_Color colorRectanguloRelleno1,
                          SDL_Color colorTexto2,
                          SDL_Color colorRectanguloBorde2,
                          SDL_Color colorRectanguloRelleno2,
                          SDL_Color colorTexto3,
                          SDL_Color colorRectanguloBorde3,
                          SDL_Color colorRectanguloRelleno3,
                          char * nombreFichero)
{
    ...

    // Crear fuentes.
    ...

    // Transferir el valor de los parámetros adicionales
    // (colores de los rectángulos y nombre del fichero)
    // a los campos de la estructura MiJuego.
    ...

    // Leer los datos desde el fichero y los transfiere al array
    // usuarios de la estructura MiJuego.
    ...

    return miJuego;
}
```

Se deberá controlar las situaciones de error al abrir y cerrar el fichero, así como errores de formato en los datos del fichero. El formato del fichero deberá ser el siguiente:

```
nombreUsuario1 tiempoUsuario1
nombreUsuario2 tiempoUsuario2
nombreUsuario3 tiempoUsuario3
nombreUsuario4 tiempoUsuario4
nombreUsuario5 tiempoUsuario5
```

Los nombres de usuarios están formados por una secuencia de caracteres alfanuméricos y/o especiales (sin incluir el carácter espaciador) y los tiempos de los usuarios consisten en una secuencia de caracteres numéricos que representan un tiempo en segundos.

Para obtener información sobre el error ocurrido, se podrá utilizar la variable externa `errno` junto con la función **`strerror(errno)`** de la librería `<errno.h>`.

Finalmente, deberá también ampliarse la función **`MiJuegoTerminar`** para que realice la destrucción de las fuentes.

```
// Libera el entorno gráfico miSDL asociado a miJuego, las texturas
// y las fuentes.

void MiJuego_Terminar(MiJuego * miJuego)
{ ... }
```

5.4 Ampliación de la función `MiJuego_Pintar`

En este apartado se ampliará la función `MiJuegoPintar` para que pinte en pantalla lo requerido en cada una de las fases:

Fase *0*:

- Dibuja el cuadrado.
- Imprime en pantalla mensajes informativos sobre el nombre y tiempo de cada uno de los récords.
- Imprime en pantalla un mensaje requiriendo al usuario pulsar una flecha para iniciar el juego.

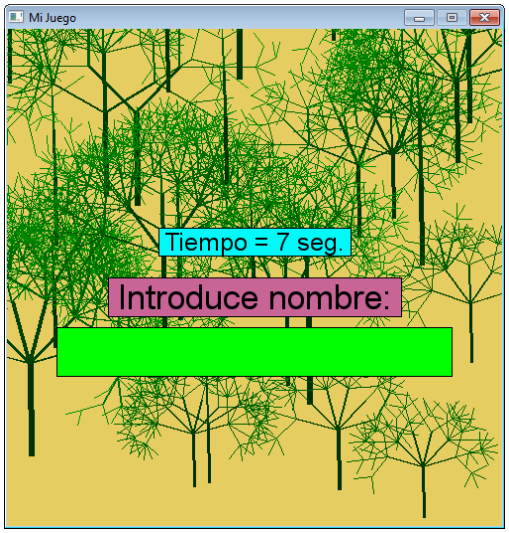
Fase *1*:

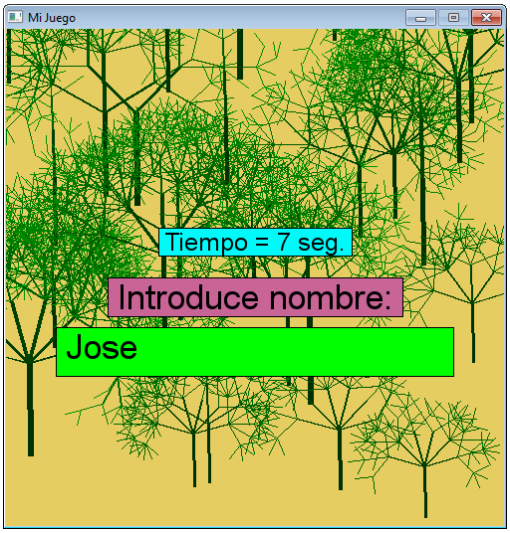
- Dibuja el cuadrado.

Fase *2*:

- Imprime en pantalla un mensaje informativo sobre el tiempo obtenido.
- Imprime en pantalla un mensaje requiriendo al usuario introducir su nombre.
- Imprime en pantalla un mensaje con el eco del nombre introducido.

A continuación se muestra un ejemplo de pantalla en fase *0* y dos ejemplos de pantalla en fase *2* (antes de empezar a introducir el nombre y con el nombre ya escrito):





5.5 Amplicación de las funciones `MiJuego_Bucle` y `MiJuego_EventoTeclado` y definición de la nueva función `MiJuego_EventoTexto` para realizar lectura de datos por teclado

Ampliar la función `MiJuego_Bucle` para que añada un nuevo evento **`SDL_TEXTINPUT`** que realice una llamada a la función **`MiJuego_EventoTexto`**. La función `MiJuego_Bucle` quedaría por tanto con el siguiente código:

```
void MiJuego_Bucle(MiJuego * miJuego)
{
  int terminar = 0;
  SDL_Event evento;

  // Repite mientras no se cierre la aplicación.
```

```
  while (!terminar)
  {
    // Repite mientras queden eventos en la cola.
    while (SDL_PollEvent(&evento))
    {
      switch(evento.type)
      {
        case SDL_KEYDOWN:
          MiJuego_EventoTeclado(miJuego,evento.key.keysym.sym);
          break;
        case SDL_TEXTINPUT:
          MiJuego_EventoTexto(miJuego,evento.edit.text);
          break;
        case SDL_QUIT: terminar = 1;
      }
    }
    if (miJuego->fase==1) MiJuego_Mover(miJuego);
    MiJuego_Pintar(miJuego);
    if (miJuego->fase==1) SDL_Delay(miJuego->retardo);
 }
}
```

La función `MiJuego_EventoTeclado` debe ampliarse para incluir las teclas **`RETURN`** (fin de introducción del texto) y **`BACKSPACE`** (borrado del último carácter) en la fase *2* de la siguiente forma:

```
void MiJuego_EventoTeclado(MiJuegoAp miJuego, SDL_Keycode code)
{
  switch(miJuego->fase)
  {
    case 0: ...
    case 1: ...
    case 2: if ((code==SDLK_RETURN)&&(strlen(miJuego->nombre)>0))
               MiJuego_TerminarFase(miJuego);
            if ((code==SDLK_BACKSPACE)&&(strlen(miJuego->nombre)>0))
               miJuego->nombre[strlen(miJuego->nombre)-1] = '\0';
  }
}
```

La función `MiJuego_EventoTexto` es la encargada de copiar en el atributo `nombre` de `miJuego` el texto que el usuario va introduciendo por teclado. Por simplicidad del código no se han tenido en cuenta las posibilidades de edición completa del texto, solo la posibilidad de borrar el último carácter con la tecla `BACKSPACE`. El código quedaría de la siguiente forma:

```
void MiJuego_EventoTexto(MiJuego * miJuego, char * texto)
{
    // Copia en el atributo nombre de miJuego el texto introducido
    if (strlen(miJuego->nombre)+strlen(texto)<TAMNOMBRE)
        strcat(miJuego->nombre,texto);
}
```

5.6 Ampliación de las funciones `MiJuego_IniciarFase` y `MiJuego_TerminarFase`

Las funciones `MiJuego_IniciarFase` y `MiJuegoTerminarFase` deberán ampliarse para realizar el control del nuevo diagrama de fases. Esto implica cambios en la fase *1* e incluir la nueva fase *2* como se muestra a continuación:

```
// Inicia la fase especificada en el argumento fase.
//
// Fase 0: Inicia la posición del cuadrado a (0,0).
// Fase 1: Inicia el contador de tiempo.
// Fase 2: Inicia el atributo nombre con la cadena vacía.
//         Inicia la entrada de texto.

void MiJuego_IniciarFase(MiJuego * miJuego, int fase)
{ ... }


// Termina la fase actual y realiza la transición a la siguiente fase.
//
// Fase 0: Inicia la fase 1.
// Fase 1: Calcula el tiempo.
//         Si se ha establecido un récord, inicia la fase 2.
//         Si no se ha establecido un récord, inicia la fase 0.
// Fase 2: Termina la entrada de texto.
//         Actualiza el array de usuarios con el nuevo récord.
//         cuyo nombre de usuario está en el atributo nombre y su
//         tiempo en el atributo tiempo.
//         Escribe el array de usuarios en el fichero.
//         Inicia la fase 0.

void MiJuego_TerminarFase(MiJuego * miJuego)
{ ... }
```

Deben tenerse en cuenta las siguientes consideraciones:

- Al iniciar la fase *2*, se inicia la entrada de texto mediante la instrucción **`SDL_StartTextInput()`**.
- Al terminar la fase *2*, se termina la entrada de texto mediante la instrucción **`SDL_StopTextInput()`**.
- Al terminar la fase *2*, el nombre leído por el teclado se encuentra en el atributo `nombre` de `miJuego`.

Sesión 6 - Programación modular y TDAs

Descriptores: Programación modular, ficheros cabecera, prototipo de funciones, tipos de datos abstractos, protección y privacidad.

El lenguaje C cuenta con recursos para dividir en partes un programa las cuales se guardan en ficheros individuales, se compilan por separado y posteriormente se enlazan (link) antes de ejecutar el programa. La práctica consiste en organizar el código fuente

en distintos componentes (módulos), cada uno de ellos con una funcionalidad propia. En nuestro caso, el código se dividirá en cuatro componentes:

- Módulo MiSDL, para manejo de la librería SDL y SDL_ttf.
- Módulo MiArbol, para el dibujo de árboles aleatorios.
- Módulo MiJuego, para el desarrollo de las fases del juego.
- Módulo TPBloqueI, para el programa principal (función `main`).

A continuación se describirá cómo disponer los ficheros y proyectos para realizar la compilación por separado.

6.1 Creación de ficheros de cabecera (*.h*)

Un *fichero cabecera* (*header file*) es un fichero que contiene el *prototipo de las funciones* que podrán ser utilizadas en otras unidades de programa. Un fichero de cabecera contiene usualmente declaraciones adelantadas de funciones, variables y otros identificadores. Un fichero de cabecera permite a los programadores separar ciertos elementos del código fuente de un programa en ficheros reusables. De esta forma se mantienen separados los prototipos de las funciones (*interfaces*), que se ofrecen a otras unidades de programa, de su implementación.

El compilador puede incluir un fichero en cualquier otro fichero, mediante la directiva del preprocesador `#include`, y particularmente utilizaremos esta directiva para incluir los ficheros de cabecera.

Por razones de organización es aconsejable que todos los ficheros cabecera de un proyecto se ubiquen en un mismo directorio. De esta forma, cualquier módulo del proyecto que utilice funciones con prototipos contenidos en los módulos de cabecera podrá encontrar en ese directorio los prototipos de la funciones. Para ello será necesario indicar en el entorno de desarrollo Code::Block, opción *Project – Build options – Search Directories – Compiler*, el directorio de includes (por ejemplo *TPBloqueI\MisIncludes*\). Los ficheros cabecera se deberán guardarán en el directorio de includes dándole en su nombre la extensión *.h*

Los siguientes ficheros de cabecera deben ser creados:

```
//
// Fichero MiSDL.h
//

#ifndef __MISDL_H
#define __MISDL_H

#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>

typedef void * MiSDL;

MiSDL MiSDL_Iniciar(char titulo[], int ancho, int alto);
void MiSDL_Terminar(MiSDL miSDL);
void MiSDL_LimpiarRender(MiSDL miSDL, SDL_Color color);
void MiSDL_ActualizarVentana(MiSDL miSDL);
```

```
void MiSDL_PintarRectanguloBordeRelleno(MiSDL miSDL,
                                        SDL_Rect rectangulo,
                                        SDL_Color colorBorde,
                                        SDL_Color colorRelleno);
void MiSDL_PintarLinea(MiSDL miSDL,
                       int x1,
                       int y1,
                       int x2,
                       int y2,
                       SDL_Color color);
void MiSDL_CrearTextura(MiSDL miSDL);
void MiSDL_LiberarTextura(MiSDL miSDL);
void MiSDL_CopiarTextura(MiSDL miSDL);
TTF_Font * MiSDL_CrearFuente(const char * ficheroFuente,
                             int tamFuente);
void MiSDL_LiberarFuente(TTF_Font * fuente);
void MiSDL_PintarTexto(MiSDL miSDL,
                       int y,
                       const char * texto,
                       SDL_Color colorTexto,
                       SDL_Color colorBorde,
                       SDL_Color colorRelleno,
                       TTF_Font * fuente);
void MiSDL_PintarTextoEntrada(MiSDL miSDL,
                       int y,
                       int w,
                       int h,
                       const char * texto,
                       SDL_Color colorTexto,
                       SDL_Color colorBorde,
                       SDL_Color colorRelleno,
                       TTF_Font * fuente);

#endif // __MISDL_H
```

```
//
// Fichero MiArbol.h
//

#ifndef __MIARBOL_H
#define __MIARBOL_H

#include "MiSDL.h"

void MiArbol_PintarSelva(MiSDL miSDL,
                         int ancho,
                         int alto,
                         int tramos_x,
                         int tramos_y,
                         SDL_Color colorArbol,
                         SDL_Color colorFondo);

#endif // __MIARBOL_H
```

```
//
// Fichero MiJuego.h
//

#ifndef __MIJUEGO_H
#define __MIJUEGO_H

#include "MiSDL.h"

typedef void * MiJuego;

MiJuego MiJuego_Iniciar(char * titulo,
                        int ancho,
                        int alto,
                        int lado,
                        SDL_Color colorFondo,
                        SDL_Color colorCuadradoBorde,
                        SDL_Color colorCuadradoRelleno,
                        int retardo,
                        int tramos_x,
                        int tramos_y,
                        SDL_Color colorArbol,
                        char * ficheroFuente1,
                        char * ficheroFuente2,
                        char * ficheroFuente3,
                        int tamFuente1,
                        int tamFuente2,
                        int tamFuente3,
                        SDL_Color colorTexto1,
                        SDL_Color colorRectanguloBorde1,
                        SDL_Color colorRectanguloRelleno1,
                        SDL_Color colorTexto2,
                        SDL_Color colorRectanguloBorde2,
                        SDL_Color colorRectanguloRelleno2,
                        SDL_Color colorTexto3,
                        SDL_Color colorRectanguloBorde3,
                        SDL_Color colorRectanguloRelleno3,
                        char * nombreFichero);

void MiJuego_Terminar(MiJuego miJuego);
void MiJuego_Bucle(MiJuego miJuego);

#endif // __MIJUEGO_H
```

6.2 Creación de ficheros de implementación (*.c*) asociados a los ficheros de cabecera

Para cada fichero de cabecera *.h* se creará un fichero de implementación *.c*, aconsejablemente con el mismo nombre que su fichero de cabecera pero con extensión *.c*, el cual contiene la implementación de las funciones cuyo prototipo muestra el fichero de cabecera. Por razones de organización, los ficheros de implementación *.c* se guardarán en el directorio de trabajo del proyecto.

Es importante indicar que los módulos MiSDL y MiJuego manejan las estructuras de datos `MiSDL` y `MiJuego` respectivamente. Para estos casos, la práctica debe llevar a

cabo mecanismos de *protección* y *privacidad* sobre estas estructuras de datos. Para ello, estos tipos se definen como un apuntador a `void` en los ficheros de cabecera y como un apuntador a la estructura concreta en el fichero de implementación, haciendo necesarias operaciones de casting entre ambos tipos.

6.3 Creación de ficheros objeto (*.o*) y ejecutable (*.exe*)

A continuación, se pueden compilar de forma separada cada uno de los ficheros de implementación *.c* para generar sus correspondientes ficheros objeto *.o* (por defecto, en el directorio *\obj\Debug* (versión debug) y *\obj\Release* (versión release).

Finalmente, se construirá el ejecutable (*.exe*) mediante las opciones de construcción.

Sesión 7 - Documentación de software

Descriptores: Doxygen, Doxywizard, documentación de software.

Al compilar un fichero fuente *.c* se ha creado el fichero objeto *.o* que puede ser utilizado por cualquier otro módulo. Para que otro módulo utilice las funciones definidas en este fichero necesitará los ficheros *.o* y *.h*, así como la documentación de todas las funciones. En este apartado se describirá una herramienta, denominada *Doxygen* (página oficial: http://www.stack.nl/~dimitri/doxygen/), que proporciona una forma muy sencilla de generar automáticamente la documentación. Doxygen es una herramienta de documentación para C y C++, y otros lenguajes de programación.

Para obtener una documentación con esta herramienta, hay que incluir comentarios en el propio código. Cuando se realizan comentarios que empiezan por `/**`, Doxygen entiende que hay que añadirlos en la documentación. Lo mismo ocurre con los comandos de documentación, como por ejemplo `\brief`, `\author`, `\version`, `\pre`, `\post`, `\param` y `\return` que se pueden utilizar para dar una breve descripción de una entidad (fichero, función, etc.), informar sobre los autores, versión, significado de los parámetros, y los valores devueltos por cada una de las funciones.

Doxywizard es una aplicación que permite editar de forma interactiva un fichero de configuración, el cual le indica a Doxygen las diversas opciones que definen la documentación a generar. Code::Blocks permite ejecutar Doxywizard mediante la opción *DoxyBlocks – Doxywizard.* El fichero de configuración se llama *doxyfile* y Code::Blocks lo guarda por defecto en el directorio *\doxygen* que cuelga del directorio del proyecto (*Step 1* de Doxywizard).

Con el paso 2 (*Step 2)* se configuran las diferentes opciones y se permite la ejecución de Doxygen para generar la documentación.

- Opciones básicas (pestaña *Wizard*). Algunas de las opciones básicas que podremos considerar para estas prácticas son:
  - Opción *Project:*
    - Título que se le asignará a la documentación generada (*Project name*).

    - Versión de la documentación (*Project version or id*).
    - Directorio donde se encuentra el código fuente a documentar indicando el path completo (*Source code directory*).
    - Directorio donde *Doxygen* generará la documentación indicando de nuevo el path completo (*Destination directory*).
- Opción *Mode*:
    - Habilitar *Documented entities only*.
    - Habilitar *Optimize for C or PHP output*.
- Opción *Output*:
    - Habilitar *HTML – plain HTML- With search function*.
    - Deshabilitar *LaTex*.
- Opción *Diagrams:*
    - Habilitar *No diagrams*.

➢ Opciones avanzadas (pestaña *Expert*). Algunas de las opciones avanzadas que podremos considerar para estas prácticas son:
- Opción *Project:*
    - *OUTPUT_LANGUAGE*: *Spanish*.
    - Deshabilitar *FULL_PATH_NAMES*.
- Opción *Input:*
    - *INPUT:* Para añadir ficheros adicionales a considerar.
    - *INPUT_ENCODING: ISO-8859-1* (codificación del alfabeto latino).

➢ Ejecutar *Doxygen* (pestaña *Run*). Algunas de las opciones avanzadas que consideraremos para estas prácticas son:
- Opción *Project:*
    - *Run doxygen*: Ejecutar doxygen.
    - *Show HTML output*: Mostrar la documentación resultante.
    - *Show configuración*: Mostrar el fichero de configuración doxyfile.

Notar que también puede visualizarse la documentación directamente desde Code::Blocks mediante la opción *DoxyBlocks – Run HTML*

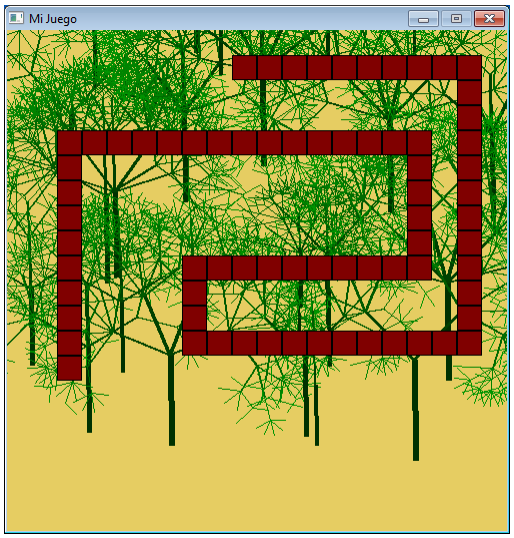
Documentación requerida para la entrega de prácticas del Bloque I

Para la entrega de las prácticas del Bloque I deberá incluirse la siguiente documentación:

- Ficheros fuente (*TPBloqueI\MisIncludes*.h* y *TPBloqueI*.c*).
- Ficheros objeto (*TPBloqueI\obj\Debug*.o y TPBloqueI\obj\Release*.o*).
- Ficheros ejecutables (*TPBloqueI\bin\Debug\TPBloqueI.exe y TPBloqueI\bin\Release\TPBloqueI.exe*).
- Ficheros de proyecto (*TPBloqueI\TPBloqueI.cbp*).
- Ficheros de documentación (*TPBloqueI\doxygen\html*.html* y asociados).

Bloque II: Proyecto de Programación

Este bloque de prácticas consiste en la realización de un proyecto de programación que permitirá al alumno aplicar y desarrollar las metodologías y técnicas estudiadas en teoría y reforzadas en el Bloque I de prácticas. Concretamente, la práctica consiste en la realización de una aplicación gráfica en C para una versión simple del *videojuego de la serpiente* (*Snake Game* http://es.wikipedia.org/wiki/Snake_(videojuego)).



Características del videojuego

- Inicialmente la serpiente está formada sólo por la cabeza (un cuadrado u otro objeto) situada en la parte superior izquierda de la ventana.
- El usuario controla la dirección de la cabeza de la serpiente (arriba, abajo, izquierda o derecha) y el cuerpo de la serpiente la sigue, tratando de evitar chocar con ella misma o con los bordes que rodean el área de la ventana del videojuego.
- Cada vez que la serpiente realiza *n* movimientos, la serpiente crece más, aumentando su cuerpo con un nuevo segmento de igual longitud que la cabeza, y aumentando su velocidad (o lo que es equivalente, disminuyendo el retardo de visualización), provocando que aumente la dificultad del videojuego. En la versión facilitada por los profesores de la asignatura, *n=3*, el retardo inicial es de *200 ms*, el retardo mínimo final es de *100 ms*, y la disminución de retardo es de *1 ms*.
- El jugador no puede detener el movimiento de la serpiente, mientras que el videojuego está en marcha.
- El mejor jugador es el que mantiene más tiempo activa la serpiente, registrándose los *5* mejores jugadores y sus tiempos, que se visualizan al final de cada juego.

Consideraciones de diseño

- La aplicación deberá realizarse bajo el mismo diseño que la aplicación realizada en el Bloque I de prácticas. Únicamente será necesario modificar el fichero *MiJuego.c* y personalizar el videojuego mediante los parámetros de entrada establecidos en la función `main` (dimensiones, colores, retardos, etc.). Se deberá crear un nuevo proyecto *TPBloqueII.cbp* que contenga los siguientes ficheros:
    - *MiJuego.c*
    - *MiJuego.h*
    - *TPBloqueII.c*
- La aplicación se construirá enlazando el código objeto *MiSDL.o,* y *MiArbol.o* añadiendo estos archivos con su ruta completa mediante la opción *Project – Build options – Linker settings – Other linker options* diferenciando las rutas según la opción debug o release. También debe añadirse la ruta de los ficheros de cabecera *MiSDL.h* y *MiArbol.h.*
- Se deberá elegir la estructura de datos más eficiente posible para representar la serpiente y su movimiento.

Ampliaciones

Se podrán realizar ampliaciones del videojuego para obtener una mejor calificación, tales como:

- Distintos niveles de juego (principiante, medio, avanzado) utilizando para cada uno de ellos distintos valores de *n*, retardo inicial, retardo mínimo final y/o disminución de retardo.
- Imágenes gráficas para dibujar cabeza, cuerpo y cola de la serpiente.
- Audio.
- Ratón.
- Objetos que aparecen cada cierto tiempo en sitios aleatorios de la ventana, los cuales, si se "comen", disminuyen la velocidad y/o longitud de la serpiente (objetos "buenos") o aumentando la velocidad y/o longitud de la serpiente (objetos "malos").
- Edición online completa de la entrada de texto.
- Permitir nombres y apellidos de usuarios (posibilidad de incluir espacios en el nombre de usuario, lo que implicaría cambios en el formato del fichero de récords).

Documentación requerida para la entrega de prácticas del Bloque II

Para la entrega de las prácticas del Bloque II deberá incluirse la siguiente documentación:

- Ficheros fuente (*TPBloqueII\MisIncludes*.h* y *TPBloqueII*.c*).
- Ficheros objeto (*TPBloqueII\obj\Debug*.o y TPBloqueII\obj\Release*.o*)
- Ficheros ejecutables (*TPBloqueII\bin\Debug\TPBloqueII.exe y TPBloqueII\bin\Release\TPBloqueII.exe*).

- Ficheros de proyecto (*TPBloqueII\TPBloqueII.cbp*).
- Ficheros de documentación (*TPBloqueII\doxygen\html*.html* y asociados).
- Breve informe, en formato pdf, que describa brevemente la organización del trabajo realizado por el alumno. Se deberá indicar:
    - Nombre del alumno/s.
    - Descripción breve del trabajo realizado y de la elección de la estructura de datos utilizada para representar la serpiente. Si las prácticas se han realizado en equipo, se deberán indicar las tareas realizadas por cada uno de los alumnos y los mecanismos de coordinación llevados a cabo.
    - Dificultades encontradas en la realización de las prácticas.
    - Grado de satisfacción del alumno/s en relación a la formación adquirida con la realización de las prácticas, y sugerencias de mejora, si las hubiese.

  El informe tiene como objetivo evaluar al alumno en las competencias transversales de la Universidad de Murcia para la asignatura, que son:
    - Ser capaz de expresarse correctamente en español en su ámbito disciplinar (*Transversal 1*).
    - Ser capaz de gestionar la información y el conocimiento en su ámbito disciplinar, incluyendo saber utilizar como usuario las herramientas básicas en TIC (*Transversal 3*).
    - Ser capaz de trabajar en equipo y para relacionarse con otras personas del mismo o distinto ámbito profesional (*Transversal 6*).

Apéndice A. Sugerencias para la instalación en Windows

A.1 Instalación de code::blocks

Web oficial

http://www.codeblocks.org/

Downloads

http://www.codeblocks.org/downloads/26 codeblocks-12.11mingw-setup.exe (versión para instalación del administrador con mingw incluido).

Configuración

Para que el compilador utilice el estándar actual para el lenguaje C conocido como C99, lo que permitirá usar ciertas características añadidas sobre el estándar original, en el menú *Settings – Compiler Settimgs – Other options* se añadirá el comando ***–std=c99***.

A.2 Instalación de SDL 2.0

Web oficial

http://www.libsdl.org/

Download

http://www.libsdl.org/download-2.0.php SDL2-devel-2.0.0-mingw.tar.gz

Pasos para la instalación

- Descomprimir
- Copiar el fichero SDL2.dll del directorio *SDL2-2.0.0\i686-w64-mingw32\bin* al directorio *C:\WINDOWS\system32*
- Copiar el contenido del directorio *SDL2-2.0.0\i686-w64-mingw32\lib* al directorio *C:\Program Files\CodeBlocks\MinGW\lib*
- Copiar el directorio completo *SDL2-2.0.0\i686-w64-mingw32\include\SDL2* al directorio *C:\Program Files\CodeBlocks\MinGW\include*
- Configurar Code::Blocks en *Compiler Settings - Linker Settings - Other linker options:* con:

-l mingw32 -l SDL2main -l SDL2

Uso de la librería

```
#include "SDL2\SDL.h"
```

A.3 Instalación de SDL_ttf 2.0

Web oficial

http://www.libsdl.org/projects/SDL_ttf/

Download

SDL2_ttf-devel-2.0.12-mingw.tar.gz (MinGW 32/64-bit)

Pasos para la instalación

- -Descomprimir
- Copiar los ficheros SDL2_ttf.dll, libfreetype-6.dll y zlib1.dll del directorio *SDL2_ttf-2.0.12\i686-w64-mingw32\bin* al directorio *C:\WINDOWS\system32*
- Copiar el contenido del directorio *SDL2_ttf-2.0.12\i686-w64-mingw32\lib* al directorio *C:\Program Files\CodeBlocks\MinGW\lib*
- Copiar el fichero SDL_ttf.h del directorio *SDL2_ttf-2.0.12\i686-w64-mingw32\include\SDL2* al directorio *C:\Program Files\CodeBlocks\MinGW\include\SDL2*
- Configurar Code::Blocks en *Compiler Settings - Linker Settings - Other linker options:* con:

-l mingw32 -l SDL2main -l SDL2 -l SDL2_ttf

Uso de la librería

```
#include "SDL2\SDL_ttf.h"
```

Apéndice B. Sugerencias para la instalación en Ubuntu 12.04

B.1 Instalación de code::blocks

Ejecutar el Centro de Software de Ubuntu y teclear Code::Blocks. Una vez que aparece code::block como no instalado, pulsar el botón Instalar.

Configuración

Para hacer que este entorno sea compatible con el estándar actual para el lenguaje C conocido como C99, lo que permitirá usar ciertas características añadidas sobre el estándar original, en el menú *Settings – Compiler* and *Debugger Settings*, en la pestaña *Compiler Settings*, subpestaña *Other Options* se tecleará el parámetro **–std=c99** a la línea de comandos que llama al compilador.

B.2 Instalación de SDL 2.0

Resolución de dependencias

Ejecutar la siguiente orden en un terminal:

```
sudo apt-get install build-essential xorg-dev libudev-dev
libts-dev libgl1-mesa-dev libglu1-mesa-dev libasound2-dev
libpulse-dev libopenal-dev libogg-dev libvorbis-dev
libaudiofile-dev libpng12-dev libfreetype6-dev libusb-dev
libdbus-1-dev zlib1g-dev libdirectfb-dev
```

Instalación de SDL 2.0

- Abrir un navegador e ir a http://www.libsdl.org/download-2.0.php
- Descargar SDL2-2.0.0.tar.gz.
- Extraer el archivo. Para ello ejecuta la siguiente orden en el terminal:
  ```
  tar -xvzf SDL2-2.0.0.tar.gz
  ```
- Ejecutar en el terminal los siguientes comandos:
  ```
  ./configure
  Make
  sudo make install
  ```
- Actualizar los enlaces a las librerías y de la caché del linker. Ejecutar en un terminal:
  ```
  sudo ldconfig
  ```

Configuración de Code::Blocks.

- Ir a *Project* - *Build options...* - *Compiler settings* - *Other options* y añadir -lSDL2

- Ir a *Project* - *Build options...* - *Linker settings* - *Other linker options* y añadir -lSDL2

Uso de la librería

```
New project-> SDL Project
#include "SDL2/SDL.h"
```

B.3 Instalación de SDL_ttf 2.0

Pasos

- Abrir un terminal.
- Escribir en la línea de comandos:
  - `sudo add-apt-repository ppa:zoogie/sdl2-snapshots`
  - `sudo apt-get update`
  - `sudo apt-get install libsdl2-ttf-dev`

Configuración de Code::Blocks.

- Ir a *Project* - *Build options...* - *Compiler settings* - *Other options* y añadir -l SDL2_ttf
- Ir a *Project* - *Build options...* - *Linker settings* - *Other linker options* y añadir -l SDL2_ttf

Uso de la librería

```
New project-> SDL Project
#include "SDL2/SDL_ttf.h"
```