

[Intro a la prog de juegos - Nacho Cabanes](#)

1.- Preparando las herramientas

[←](#) [Contenido](#) [Índice](#) [Cambios](#) [Enlaces](#) [→](#)

Versión: [0.42](#), de 08-Ago-2010

1.- Preparando las herramientas

Contenido:

Toma de contacto:

[¿Por qué este curso?](#) [Condiciones de uso.](#) [¿Cómo puedo colaborar?](#) [¿Cómo puedo preguntar dudas?](#)
[¿Qué herramientas emplearemos?](#)

Temas del curso:

1. [Preparando las herramientas \(cómo instalar los compiladores que usaremos\).](#)
2. [Entrando a modo gráfico y dibujando.](#)
3. [Leyendo del teclado y escribiendo texto.](#)
4. [Cómo generar números al azar. Un primer juego: Adivinar números.](#)
5. [Un segundo juego: Ahorcado.](#)
6. [Evitemos esperar al teclado. Tercer juego: motos de luz.](#)
7. [Mapas. Cuarto juego \(aproximación "a"\): MiniSerpiente 1.](#)
8. [Cómo crear figuras multicolor que se muevan. Cuarto juego \(aproximación "b"\): Miniserpiente 2.](#)

Completos en C o C++, pero sin adaptar a Pascal y Java:

9. [Evitemos los parpadeos. Cuarto juego \(aproximación "c"\): Miniserpiente 3.](#)
10. [Más sobre la paleta de colores.](#)
11. [Cuarto juego \(completo\): Serpiente.](#)
12. [Utilizando el ratón. Quinto Juego: Puntería. \(*\)](#)
13. [Un poco de matemáticas para juegos. Sexto Juego: TiroAlPlato. \(*\)](#)
14. [Cómo reproducir sonidos. Séptimo juego: SimeonDice. \(*\)](#)
15. [Formatos de ficheros de imágenes más habituales. Cómo leer imágenes desde ficheros.. \(*\)](#)
16. [Octavo juego \(planteamiento\): MataMarcianos. \(*\)](#)
17. [Cargar y mostrar imágenes. Octavo juego \(aproximación "a"\): Marciano 1. \(*\)](#)
18. [Mostrar y mover sólo un marciano que cambia de forma. Octavo juego \(aproximación "b"\): Marciano 2. \(*\)](#)
19. [Moviendo una nave y un enemigo a la vez con velocidades distintas. Octavo juego \(aproximación "c"\): Marciano 3. \(*\)](#)
20. [Incluyendo un disparo y comprobación de colisiones. Octavo juego \(aproximación "d"\): Marciano 4. \(*\)](#)
21. [Moviendo varios marcianos a la vez. Octavo juego\(aproximación "e"\): Marciano 5. \(*\)](#)
22. [Un doble buffer para evitar parpadeos. Octavo juego \(aproximación "e"\): Marciano 6. \(*\)](#)
23. [Enemigos que disparan: Marciano 7. \(*\)](#)
24. [Un "bucle de juego" clásico: aproximación a Columnas. \(*\)](#)
25. [Avanzando Columnas: primera parte de la lógica de juego. \(*\)](#)
26. [Un Columnas jugable: interacción con los elementos del fondo. \(*\)](#)
27. (No completo... puedes mirar más abajo para ver lo que contendrá...)
28. [La aproximación orientada a objetos \(1\). Toma de contacto con un primer "arcade": MiniMiner. \(*\)](#)
29. [La aproximación orientada a objetos \(2\). MiniMiner 2: Aislado del hardware. \(*\)](#)
30. [La aproximación orientada a objetos \(3\). MiniMiner 3: Personaje y enemigo como clases \(*\)](#)
31. [La aproximación orientada a objetos \(4\). MiniMiner 4: Una pantalla de juego real \(*\)](#)
32. [Colisiones con enemigos. Perder vidas. Aplicación a MiniMiner \(versión 5\) \(*\)](#)
33. [Volver a comenzar una partida. Una pantalla de presentación animada. Imágenes transparentes. Aplicación a MiniMiner \(versión 6\) \(*\)](#)
34. [Añadiendo funcionalidades a "MiniMiner" \(1\): chocar con el fondo, saltar. \(versión 7\) \(*\)](#)
35. [Una consola para depuración \(*\)](#)

Apartados planteados pero no resueltos (ni siquiera en C/C++):

- 27. [Completando Columnas: borrado correcto, puntuación.](#) (*)
- 36. [Añadiendo funcionalidades a "MiniMiner" \(2\): mejora del movimiento, recoger objetos.](#) (*)
- 37. [Añadiendo funcionalidades a "MiniMiner" \(3\): avanzar a otra pantalla.](#) (*)
- 38. [Una lista con las mejores puntuaciones. Cómo guardarla y recuperarla.](#) (*)
- 39. [Manejo del joystick.](#) (*)

Próximos apartados previstos (el orden puede cambiar):

- ?? Enemigos "inteligentes": PicMan. (*)
- ?? Guardando y recuperando configuraciones. (*)
- ?? Empleando distintos tipos de letra. (*)
- ?? Otro clásico más: LaRana. (*)
- ?? Introducción a los Scrolls. (*)
- ?? Un primer juego de scroll horizontal, MiniMarianoBros. ?? . Cómo comprobar varias teclas a la vez. Cómo redefinir las teclas con las que jugaremos. (*)
- ?? Como saber qué modos gráficos tenemos disponibles. (*)
- ?? Distintas resoluciones sin cambiar el programa. (*)
- ?? Otro clásico: NBert. (*)
- ?? Otro clásico sencillo y adictivo: miniTetris (*)
- ?? Temporizadores con Allegro. (*)
- ?? Otro clásico: MiniAsteroides. (*)
- ?? Y otro más: Multipede. (*)
- ?? Introducción a las figuras 3D: representación, movimiento y rotación. (*)
- ?? Creando una pantalla de bienvenida animada en 3D. (*)
- ?? Figuras 3D con texturas. (*)
- ?? Un juego con figuras 3D: StarFighting. (*)
- ?? Imágenes en varios planos. (*)
- ?? Un matamarcianos con scroll vertical y varias capas: Nevious. (*)
- ?? Compactando varios ficheros de datos en uno. (*)
- (...)

(Nota: este temario puede ir cambiando, no hay ninguna garantía de que se siga exactamente ese orden en los capítulos que aún no están creados, que son los marcados con un asterisco *).

¿Por qué este curso?

Mucha gente me ha pedido que, tomando como base los cursos de Pascal y/o C/C++ que he desarrollado, creara ahora un curso de programación de juegos.

Existe más de un texto que cubre esta temática. Personalmente, a mí me gustan dos:

- PCGPE (PC Games Programmer Encyclopedia, la enciclopedia del programador de juegos de Pc) es muy completa, pero está en inglés, y hasta cierto punto está obsoleta, porque se centra en la programación bajo MsDos.
- El CPV (curso de programación de videojuegos) está en español, es algo menos completo pero es muy fácil de leer. También está algo obsoleto, porque se centra en MsDos.

¿Y qué aporta este curso? Apenas un par de cosas, que tienen sus ventajas, pero también sus inconvenientes:

- Está en español. Eso hará que llegue a menos cantidad de gente, pero también que los castellano-parlantes tengan otro texto de donde extraer información, aunque no conozcan el inglés.
- No entraré tan "a bajo nivel" como en los dos cursos que he mencionado antes. Eso puede ser malo porque quien "me siga" no dominará tanto la arquitectura del PC, pero tiene de bueno que se podrá avanzar más rápido y sin necesidad de tantos conocimientos técnicos. Hoy en día hay muchas "bibliotecas" que se encargan de la parte más cercana al hardware, sin que nosotros debamos preocuparnos tanto por ello.
- Lo que aprendamos se podrá aplicar para crear juegos que será posible compilar para MsDos, para Windows y para Linux, sin ningún cambio o casi (esperemos).

- Yo estoy [localizable](#), algo que no puedo decir (al menos en este momento) de los autores de los dos cursos anteriores, de modo que puedo intentar atender las solicitudes para dirigir el curso hacia los aspectos que despierten más interés.

Condiciones de uso.

Este texto se puede distribuir libremente a otras personas, siempre y cuando no se modifique. Tienes permiso para utilizarlo, pero pertenece a su autor, José Ignacio Cabanes.

Este texto se distribuye tal cual, sin ninguna garantía de ningún tipo. Si el uso directo o indirecto del contenido de este curso provoca cualquier problema en tu ordenador, el autor del texto no podrá ser considerado responsable en ningún caso. Este texto es para uso personal: su inclusión en cualquier otro artículo, curso o medio de cualquier tipo deberá ser [consultada previamente al autor](#) y deberá contar con su aprobación. La utilización del curso supone la aceptación total de estas condiciones.

¿Cómo puedo colaborar?

Este texto está hecho "por amor al arte", sin recibir nada a cambio. Si quieres ayudar a que se siga mejorando, tienes varias formas de hacerlo:

- La más sencilla: envíame [un mensaje](#) de apoyo, diciendo que utilizas el curso, qué es lo que te gusta y qué es lo que no te gusta.
- Si descubres algún error, [házmelo saber](#).
- Si te consideras capacitado para colaborar aportando algún artículo creado por ti, será más que bienvenido.

Insisto: la única colaboración "**imprescindible**": es la primera de todas ellas: enviar [un mensaje](#) de ánimo, para que vea que este esfuerzo sirve para algo. Eso hará que robe parte de mi (poco) tiempo libre para irlo mejorando.

¿Cómo puedo preguntar dudas?

Si tienes dudas, posiblemente tu mejor alternativa será acudir al foro específico para el curso, creado en [AprendeAProgramar.com](#)

¿Qué herramientas emplearemos en este curso?

El lenguaje que emplearemos va a ser C++(aunque traduciremos muchas cosas a Pascal, lee más abajo) Los primeros ejercicios serán más simples y usaremos un lenguaje muy cercano al C, sin sacar partido de la Programación Orientada a Objetos, pero más adelante sí la utilizaremos, cuando nuestros proyectos vayan siendo de mayor envergadura.

En concreto, comenzaremos por usar el compilador **DJGPP** y la librería (perdón, quería decir "biblioteca") **Allegro** .

Esta biblioteca se distribuye como código fuente, de modo que podemos recompilarla para que nuestros programas la aprovechen aunque usemos otros compiladores. Esto nos permitirá que nuestros programas estén diseñados inicialmente para DOS, pero después podremos hacer fácilmente que sean programas de Windows, sólo con usar otro compilador como **MinGW**.

Eso sí, tampoco necesitarás dominar C++ para seguir este curso (espero). Las órdenes relativas a gráficos que necesitemos las iremos viendo conforme nos hagan falta.

¿Y si no sé programar en C++?

Es muy posible que seas capaz de seguir el curso sin problemas si tienes conocimientos de **otro(s) lenguajes** de programación, como C (por supuesto), Pascal (te costará algo más) o Basic (más difícil todavía, pero posible).

En cualquier caso, tienes muchos cursos gratuitos de C++ a tu disposición en Internet. En mi propia [página Web](#) tendrás a tu disposición más de un curso que te puede resultar útil. Entre ellos estará mi curso (claro) y algún otro que yo haya considerado interesante. Por cierto, si encuentras por Internet algún curso que esté bien y que se pueda distribuir libremente, te agradecería que me lo indicaras para que lo incluya en mi Web.

¿Y si prefiero el lenguaje Pascal?

Tienes tu parte de razón. El lenguaje Pascal es más legible que C (y C++), y también existen compiladores para Windows, Dos, Linux y otros sistemas operativos. Así que se aprueba la moción: haremos las cosas también en lenguaje Pascal, utilizando el compilador de libre distribución **Free Pascal**. Eso sí, nos limitaremos a las posibilidades "estándar" que este compilador incorpora, de modo que no podremos hacer tantas cosas avanzadas como con C++ y Allegro (existen bibliotecas análogas para Free Pascal, como Graphix, pero no entraremos a ellas... al menos por ahora).

¿Y qué tal el lenguaje Java?

Tampoco es una mala idea:

- El lenguaje Java tiene una sintaxis parecida a la de C++, de modo que los cambios no serán exageradamente grandes.
- Existen versiones de Java para Windows, Linux y otros sistemas operativos (incluso algunos teléfonos móviles actuales incorporan una versión reducida del lenguaje Java).
- Las órdenes de gráficos, manejo de ratón, etc. son parte del propio lenguaje, de modo que no dependemos de "bibliotecas externas" como Allegro.

Entonces... ¿qué usaremos?

Nos centraremos en C (y más adelante en C++) con la ayuda de la biblioteca Allegro, pero indicaremos los cambios que serían necesarios para que cada juego funcione en Pascal o en Java, aunque no siempre sea posible o a veces suponga que el juego tenga alguna limitación (como no tener sonido, por ejemplo).

Preparando las herramientas...

Vamos a ver cómo instalar los distintos compiladores que usaremos durante el curso. Elige un lenguaje (C, Pascal o Java) o varios, un sistema operativo (Windows, Dos o Linux) o varios, y sigue las instrucciones para instalar el entorno de desarrollo que necesites:

- C, Para Windows (Recomendado): [Instalación simplificada del compilador GCC - MinGW \(con el entorno Dev-C++\) y Allegro.](#)
- C, Para Linux: [Instalación de Allegro en Linux.](#)
- C, Para Dos: [Instalación del compilador DJGPP y Allegro.](#)
- Pascal para Windows: [Instalación de Free Pascal.](#)
- Pascal para Linux: [Instalación de Free Pascal para Linux.](#)
- Pascal para Dos: [Instalación de Free Pascal para DOS.](#)
- Java para Windows: [Instalación del Kit de Desarrollo en Java \(JDK\).](#)
- Java para Linux: [Instalación del Kit de Desarrollo en Java \(JDK\).](#)

Otras herramientas alternativas:

- C, Para Windows: [Instalación manual de Dev-C++ y Allegro.](#)
- C, Para Windows: [MinGW Developer Studio y Allegro.](#)
- C, Para Windows: [Instalación del compilador MinGW \(sin entorno\) y Allegro.](#)

Puedes consultar el apartado que te interese, o si ya tienes instalado el entorno que quieres utilizar, pasar al [siguiente tema](#).

1.1. Instalando Dev-C++ y Allegro para Windows.

Contenido de este apartado:

- [¿Dónde encontrar el compilador?](#)
- [¿Cómo instalarlo?](#)
- [Probando el compilador con un ejemplo básico y con un juego.](#)

1.1.1. ¿Dónde encontrar el compilador?

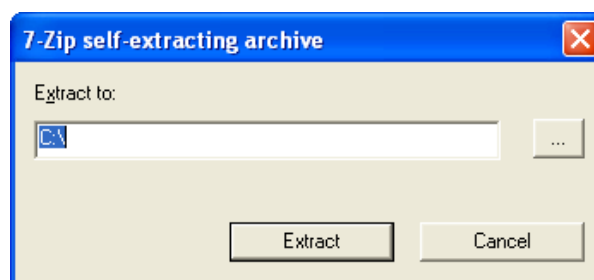
El entorno Dev-C++ es uno de los más extendidos para desarrollar programas en C y C++ desde Windows. Incorpora el compilador GCC, en su versión para Windows (conocida como MinGW).

Si sólo quieres el compilador, puedes acudir a su página oficial en www.bloodshed.net.

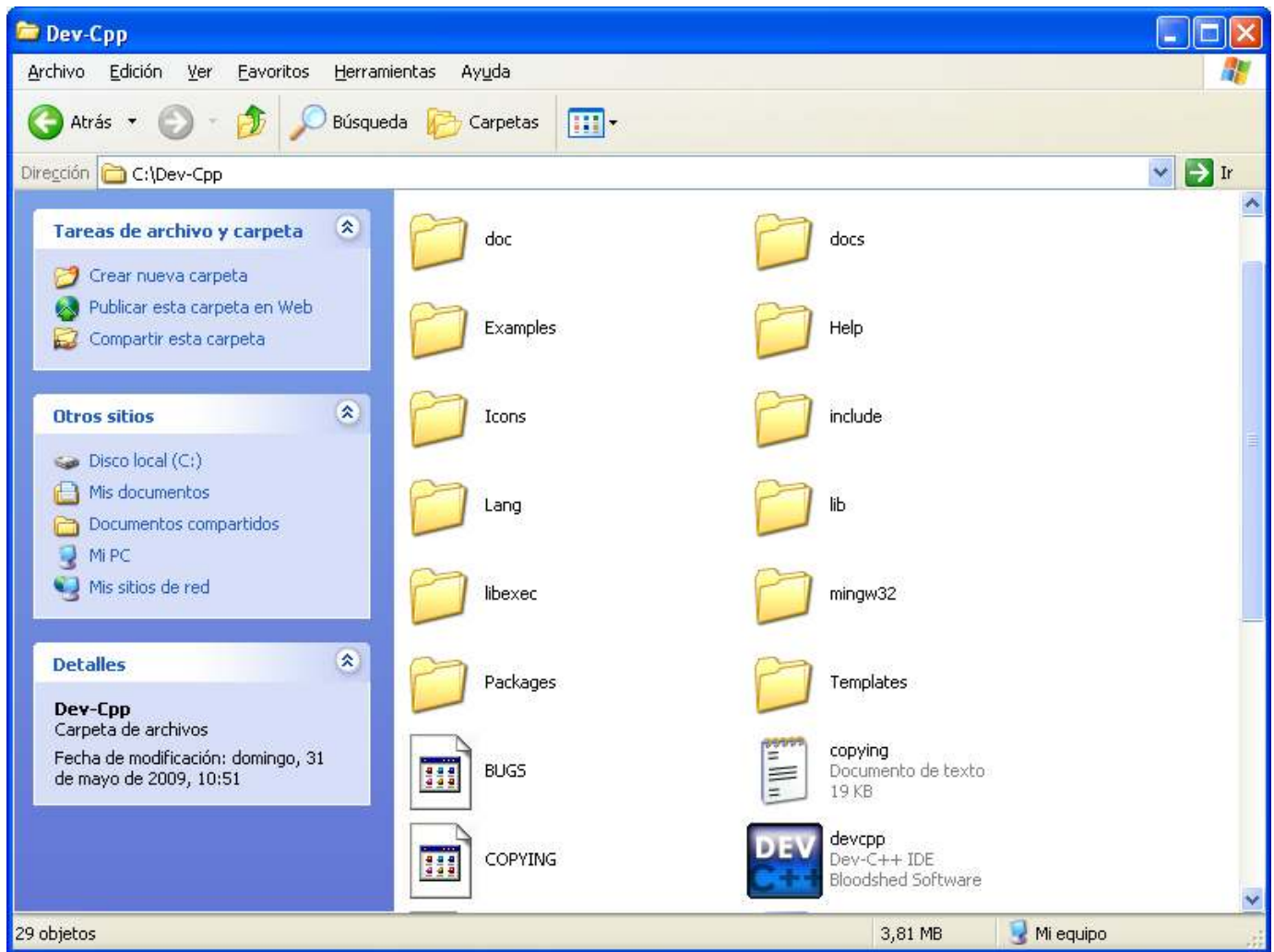
Si lo quieres usar para el curso, tendrás que añadir la biblioteca Allegro. Para simplificarte el proceso, tienes el compilador ya listo para instalar con todas las librerías habituales, en una [descarga local desde mi propia página](#) (cerca de 18 Mb de tamaño).

1.1.2. ¿Cómo instalarlo?

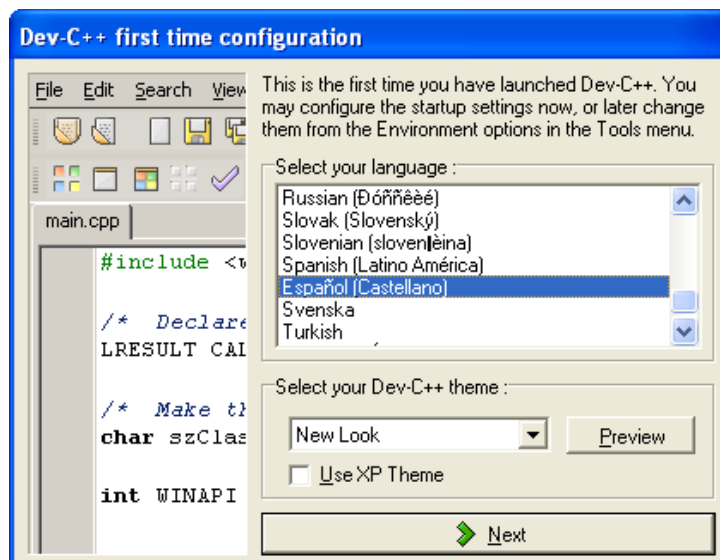
La instalación es sencilla. Basta con descargar ese fichero y hacer doble clic. Si usamos Windows Vista, pedirá permiso para continuar, preguntando si procede de una fuente fiable. A continuación, o como primer paso si usamos Windows XP o anterior, nos preguntará dónde queremos descomprimirlo (por ejemplo, en c:\).



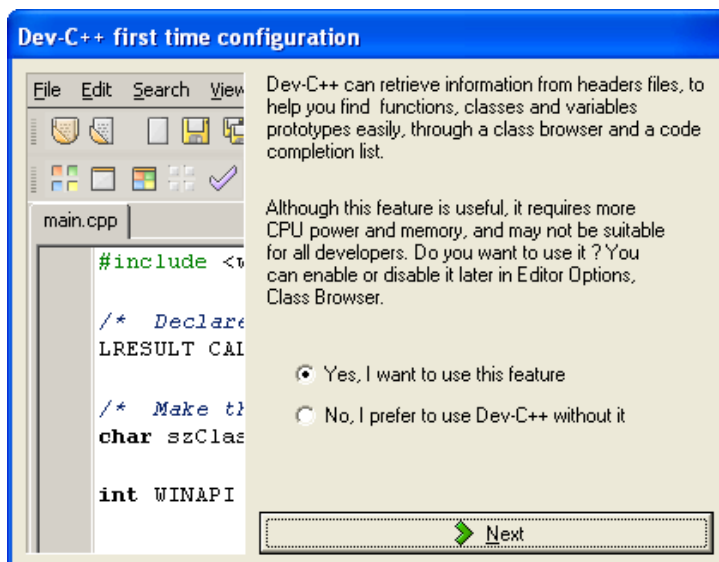
En esa carpeta, se creará una subcarpeta llamada Dev-Cpp. Dentro de ella, está el fichero "devcpp" que nos permite acceder al entorno de programación



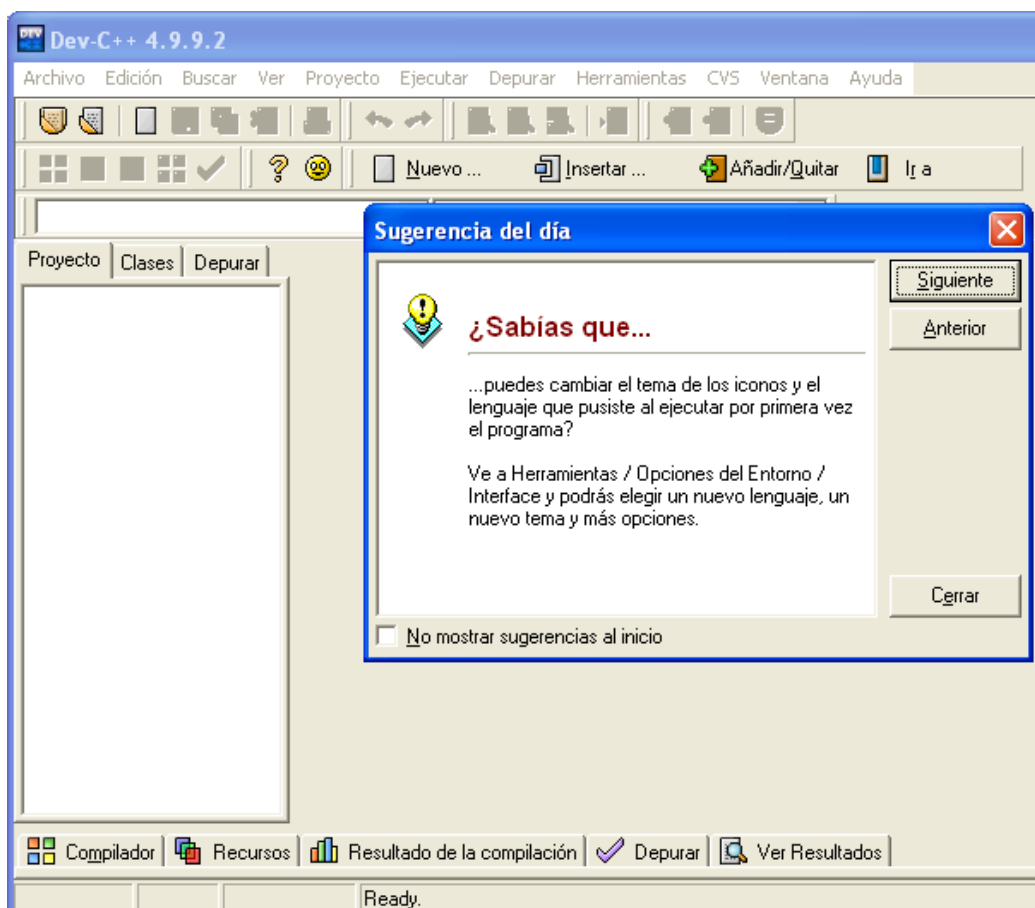
Se nos preguntará en qué idioma quieres usar el entorno (incluye el español, aunque aparece descolocado, en la letra S de "Spanish"),



Después se nos consultará si queremos que se cree un "caché" de las funciones existentes, para ayudarnos a teclear más rápido, autocompletándolas. Como no tarda mucho tiempo, y los ordenadores actuales tiene mucha capacidad, puede ser razonable decir que sí (Yes).

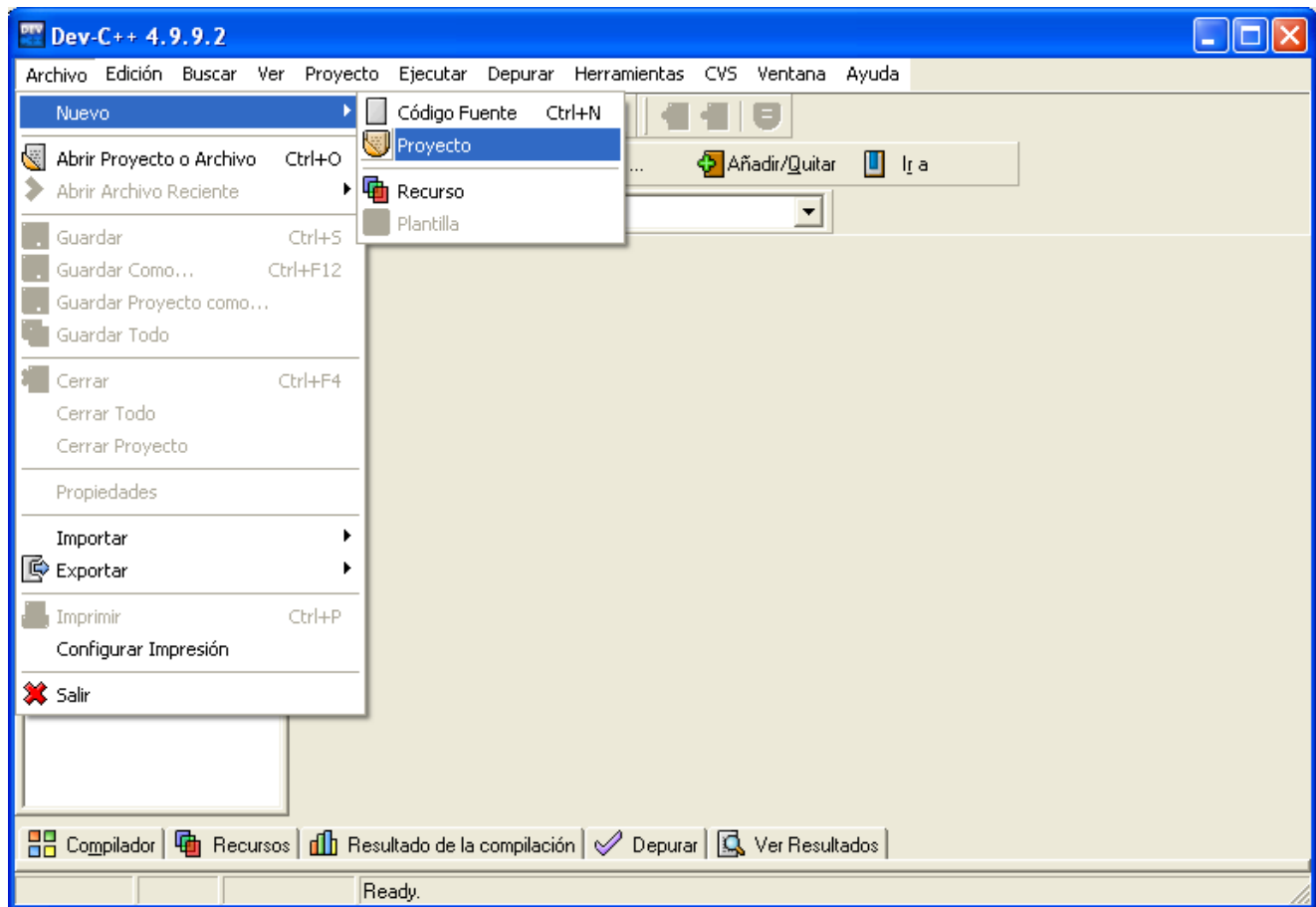


Finalmente aparecerá la pantalla de trabajo de Dev-C++, junto con su pantalla de Bienvenida/Sugerencias:

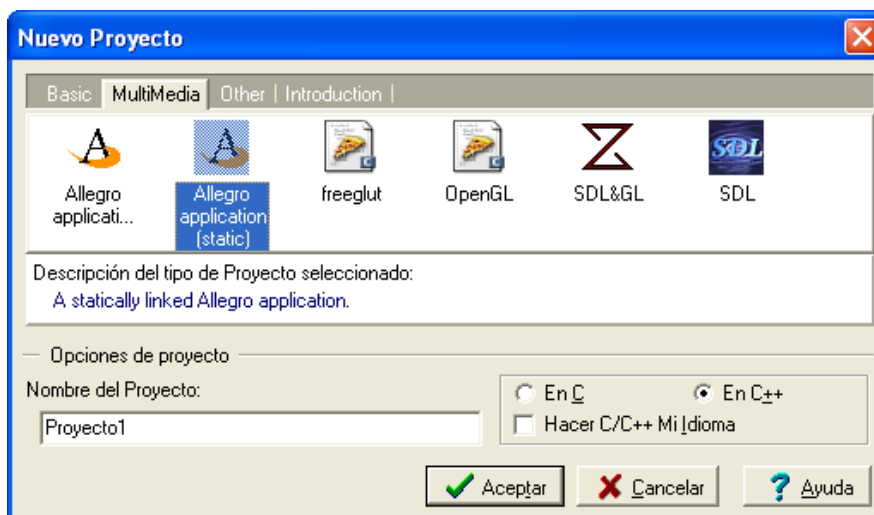


1.1.3. Probando el compilador con un ejemplo básico y con un juego.

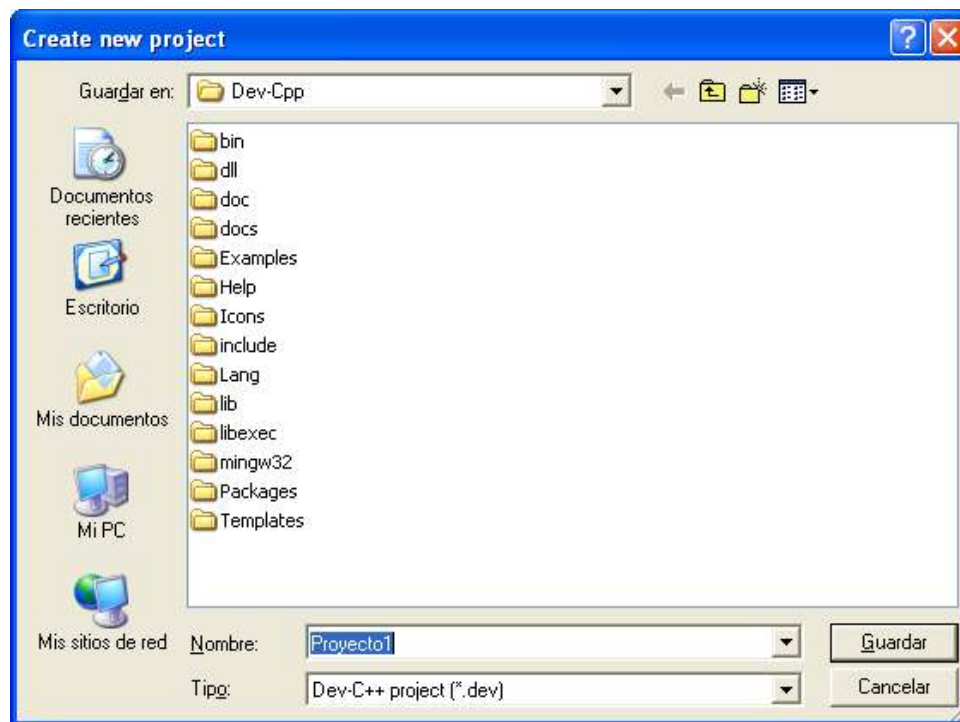
Basta con crear un "Nuevo proyecto", desde el menú Archivo / Nuevo / Proyecto:



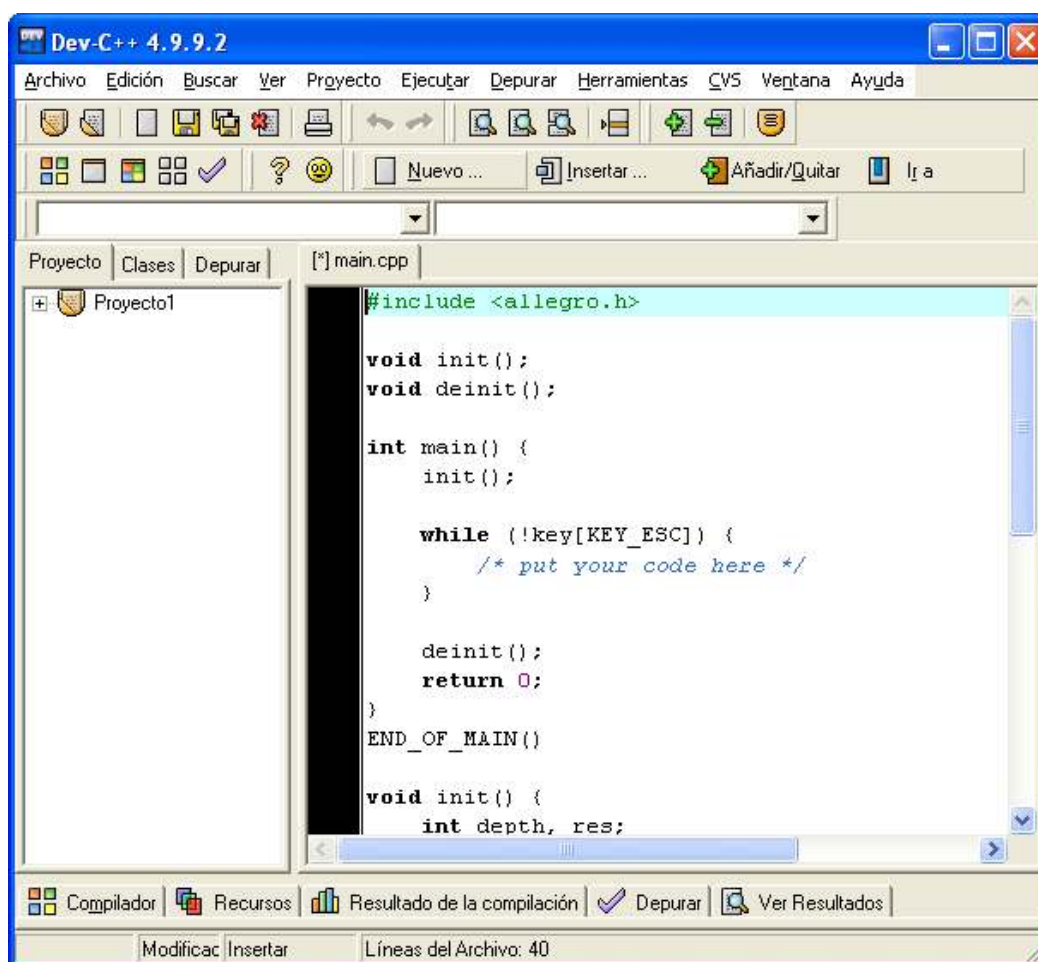
El asistente muestra varias pestañas, entre las que está "Multimedia", y dentro de ella tenemos "Allegro Application (static)", para crear una aplicación que use la librería Allegro. Podremos indicar el nombre del proyecto (nos propone "Proyecto1"), y si va a estar creado en C++ o en C:



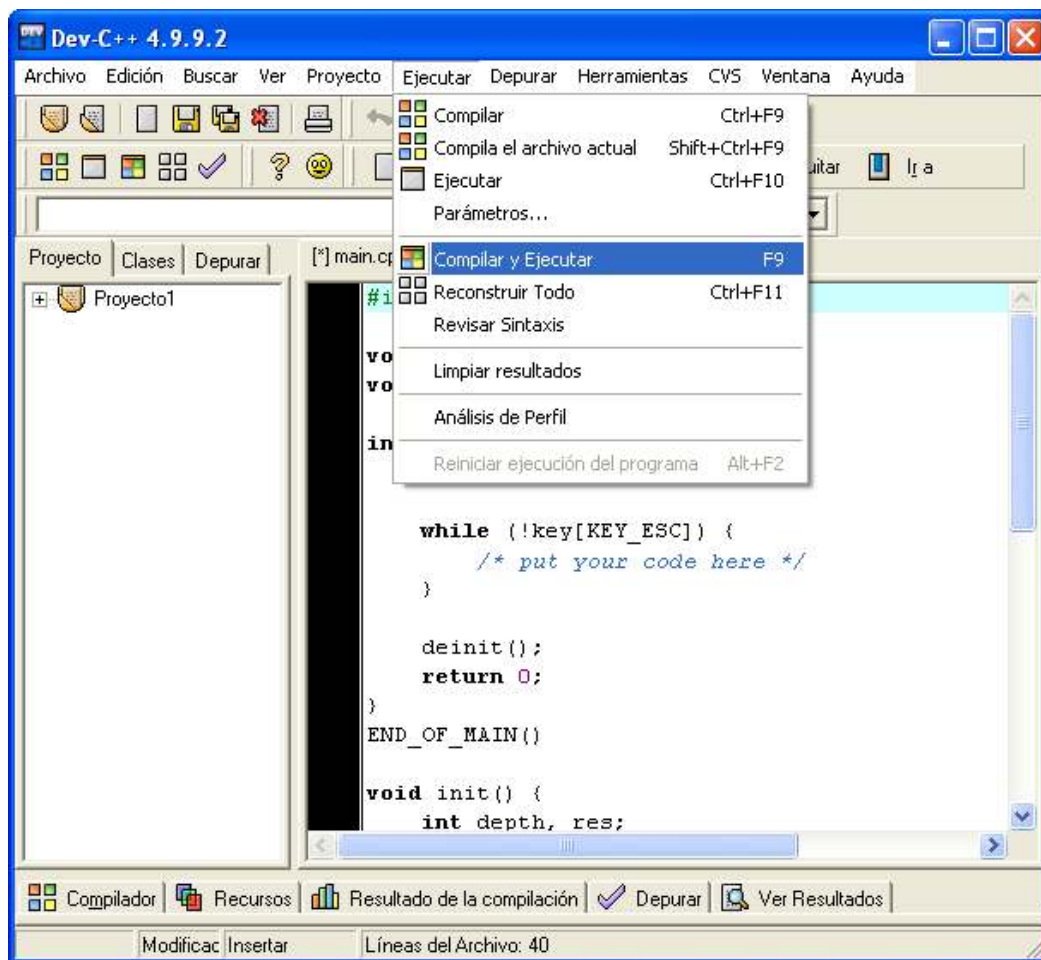
A continuación se nos pedirá que guardemos el proyecto (lo ideal es que aprovechemos este momento para crear una carpeta en la que vayan a estar todos nuestros proyectos, no sólo éste):



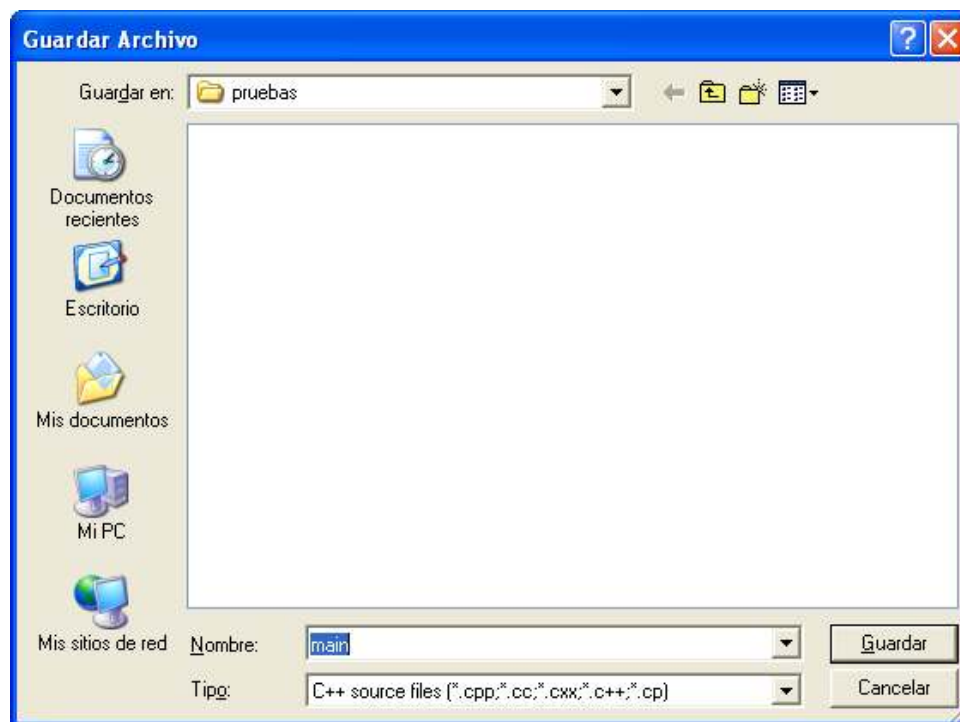
Entonces aparecerá un esqueleto de programa creado con Allegro:



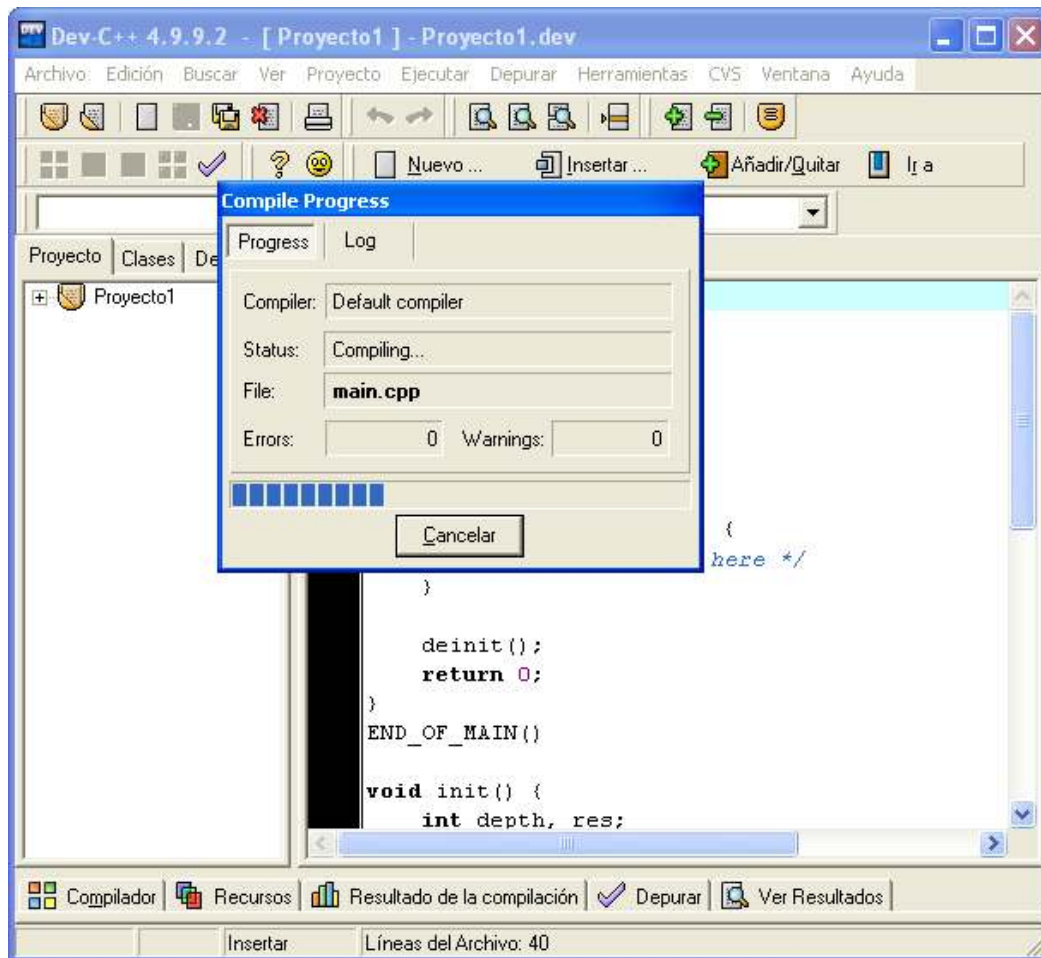
Para probar este ejemplo, entraremos al menú "Ejecutar" y escogeremos la opción "Compilar y Ejecutar":



Si sólo habíamos guardado el proyecto, pero no nuestro fuente (deberíamos acostumbrarnos a guardar los cambios cada poco tiempo), se nos pedirá ahora que lo guardemos:



Entonces comenzará realmente la compilación del programa:



Si no hay errores, debería aparecer el resultado de nuestro programa en pantalla. Para este programa de ejemplo, el resultado es una pantalla negra hasta que pulsemos ESC.

Si todo ha ido bien, tenemos Dev-C++ y Allegro correctamente instalados. En ese caso, ya podemos "copiar y pegar" cualquiera de nuestros fuentes sobre el fuente de ejemplo para poder probarlo. Hay que recordar dejar también dentro de la carpeta del proyecto las imágenes o ficheros de datos adicionales que podamos necesitar.

1.2. Instalando Allegro en Linux.

Casi cualquier distribución de Linux incluye el compilador GCC preinstalado. Por tanto, en nuestro caso basta con "añadir" Allegro.

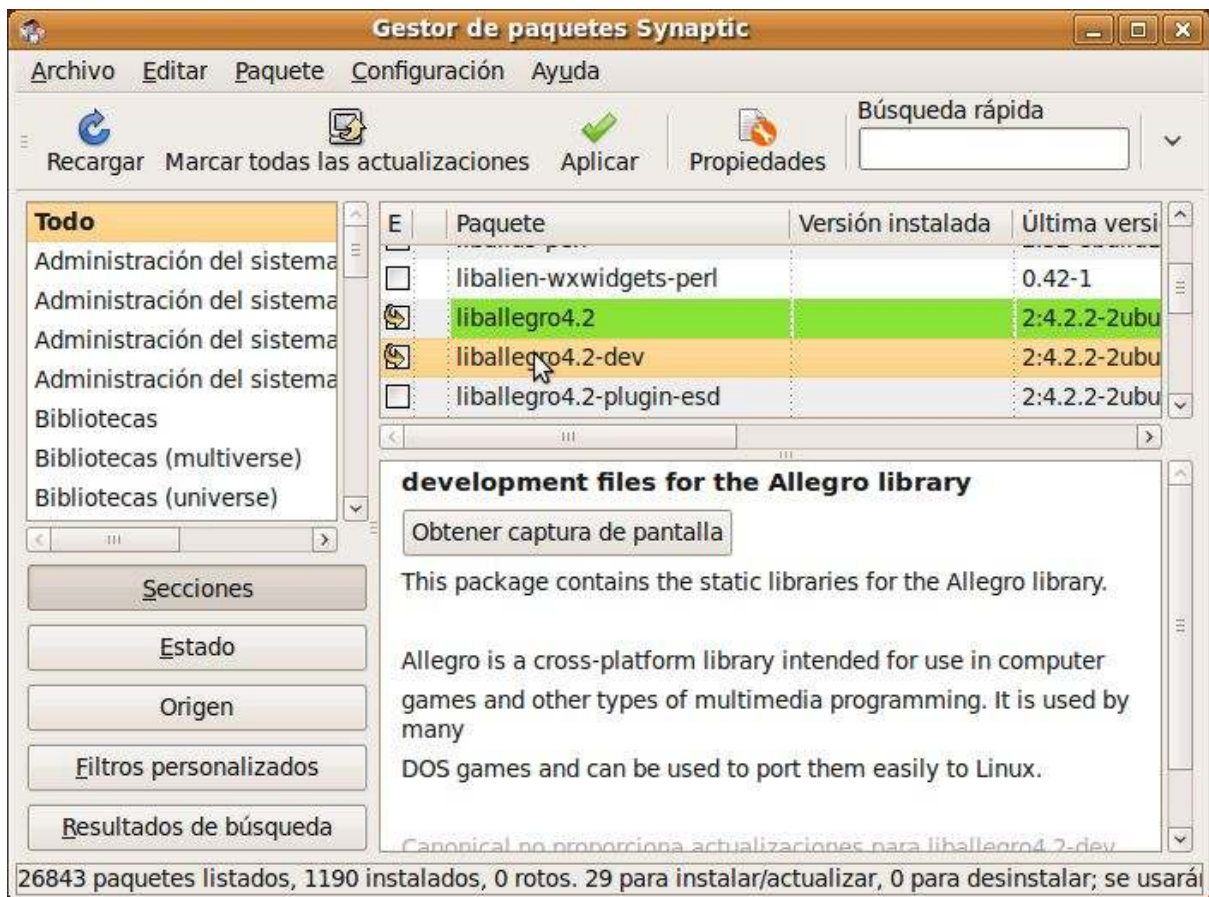
Incluso para instalar Allegro tenemos ayuda: normalmente no hará falta hacerlo "a mano", sino que podremos usar los "gestores de paquetes" que incorporan las distribuciones de Linux.

Si nuestra distribución usa el escritorio Gnome, podremos emplear Synaptic para instalar fácilmente cualquier paquete de software; si nuestro escritorio es KDE, podremos emplear Adept.

Por ejemplo, en Ubuntu Linux 9.04, con el escritorio Gnome, el gestor Synaptic estará dentro del menú Sistema, en el apartado Administración:



Dentro de Synaptic, entre la lista de paquetes a instalar, deberíamos buscar la "librería Allegro" (liballegro), en su versión de desarrollo (liballegro-dev). Si seleccionamos este paquete, automáticamente se seleccionaría la parte básica de Allegro, en caso de que no la tengamos instalada:



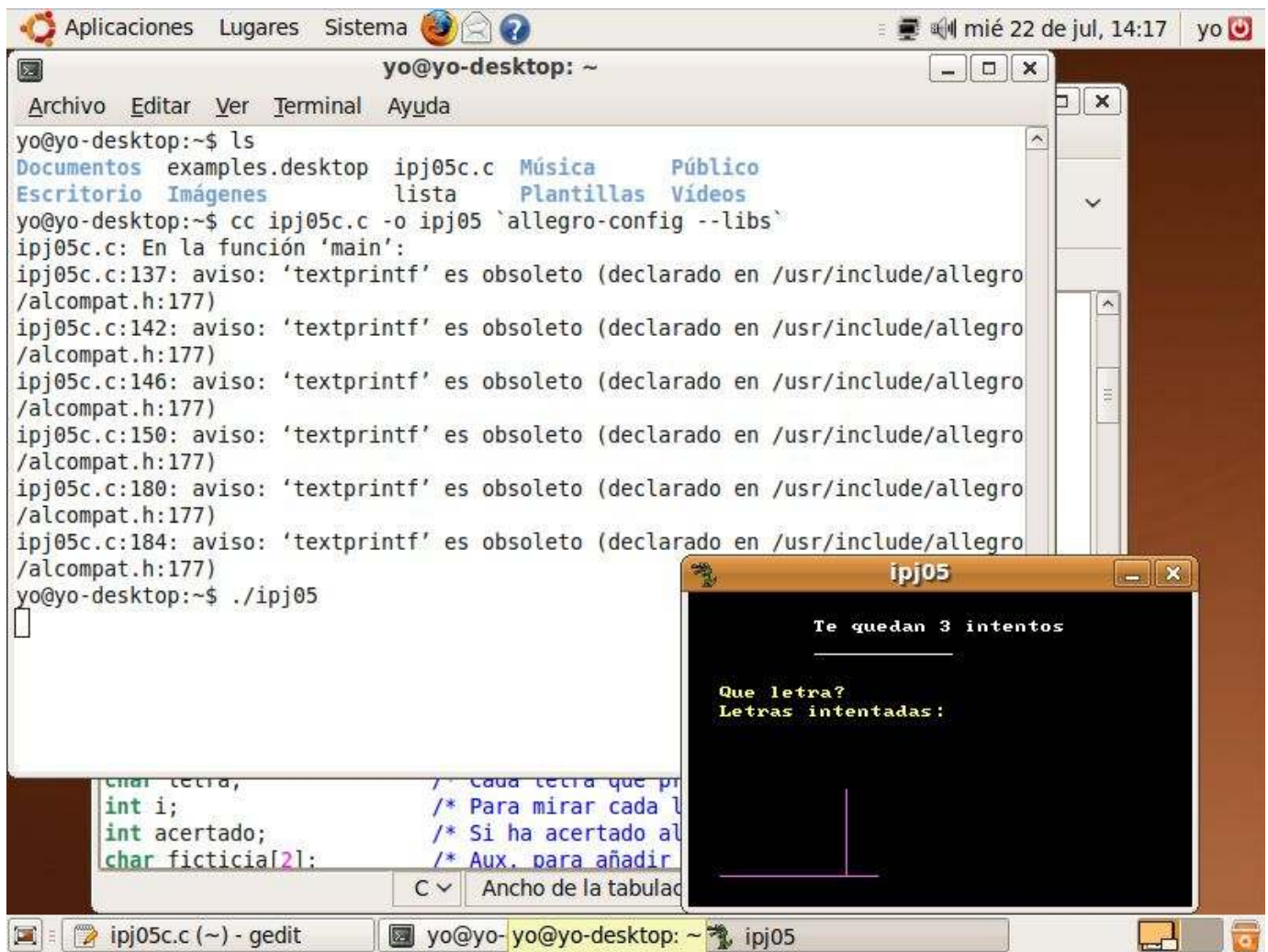
Cuando digamos "Aplicar", se descargará e instalará lo que hemos solicitado. Entonces podremos teclear nuestro fuente con GEdit o cualquier otro editor, abrir un Terminal y compilar cualquier fuente con algo como

```
cc ipj05c.c -o ipj05 `allegro-config --libs`
```

Es decir: el nombre del compilador ("cc"), el de nuestro fuente (por ejemplo, "ipj05c.c"), la opción para detallar el nombre del ejecutable ("-o", de "output"), el nombre que éste tendrá (por ejemplo, "ipj05"), y que use la configuración de Allegro: `allegro-config --libs` (entre comillas simples invertidas, el acento grave, exactamente como aparece en el ejemplo).

Para lanzar el ejecutable, usaríamos su nombre precedido por "./", así:

```
./ipj05
```



Puede que al compilar obtengamos algún mensaje de aviso, como éste que nos recuerda que "textprintf" es una función obsoleta (anticuada), pero aun así debería compilar correctamente y funcionar sin problemas.

1.3. Instalando DJGPP y Allegro en DOS.

(Pronto disponible)

1.4. Instalando Free Pascal para Windows.

Contenido de este apartado:

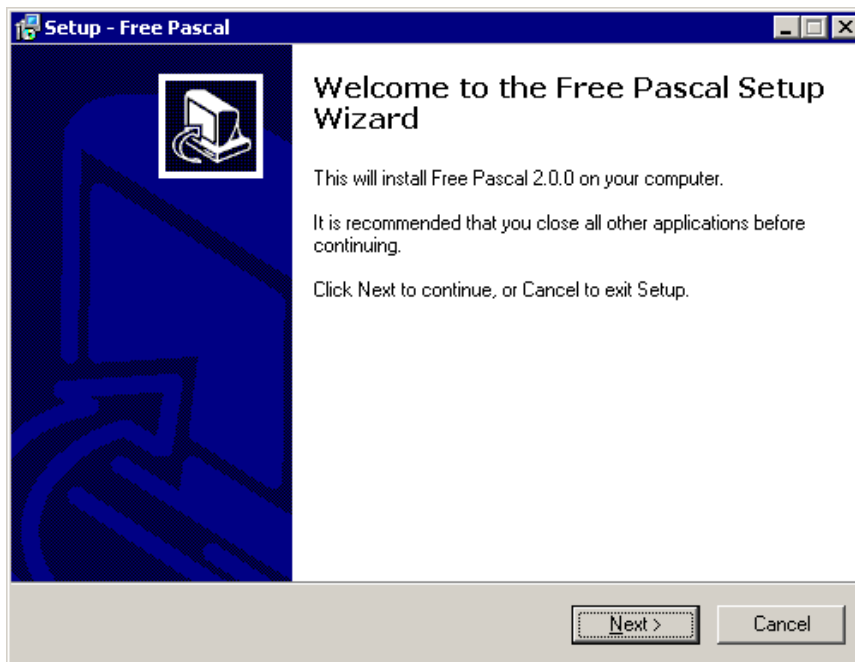
- [¿Dónde encontrar el compilador?](#)
- [¿Cómo instalarlo?](#)
- [Probando el compilador con un ejemplo básico.](#)
- [Probando el modo gráfico.](#)

1.4.1. ¿Dónde encontrar el compilador?

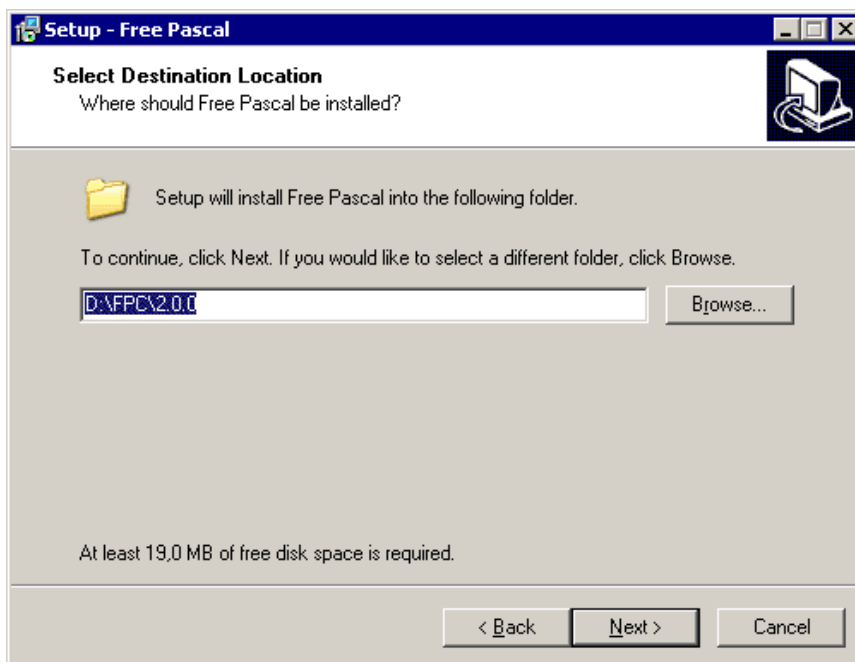
La web oficial de Free Pascal es www.freepascal.org. En el apartado "Downloads" tienes las descargas para los distintos sistemas operativos en los que se puede instalar.

1.4.2. ¿Cómo instalarlo?

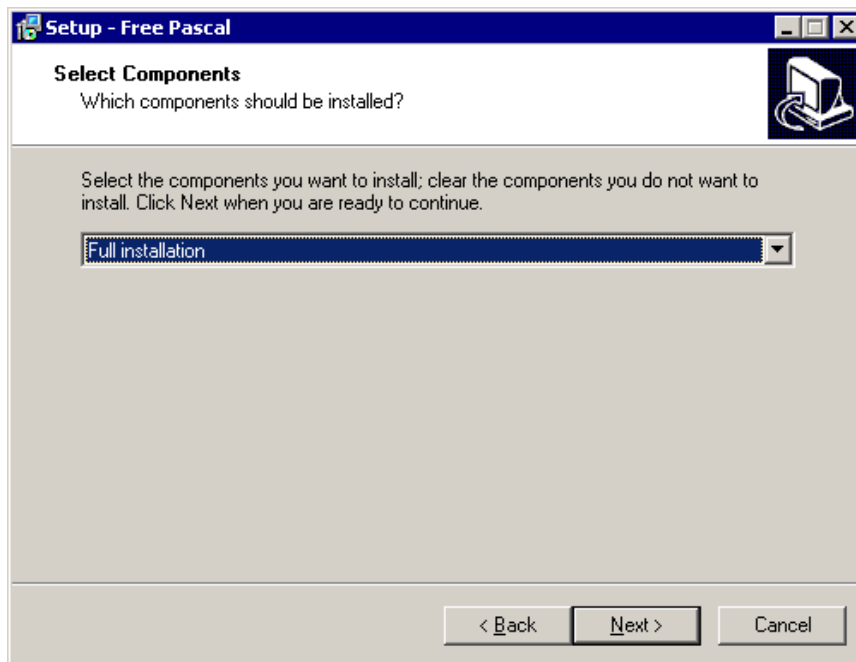
Vamos a ver el caso de Free Pascal 2.0. En sencillo, poco más que hacer doble clic en el fichero que hemos descargado e ir pulsando "Siguiente". La primera pantalla es la de bienvenida:



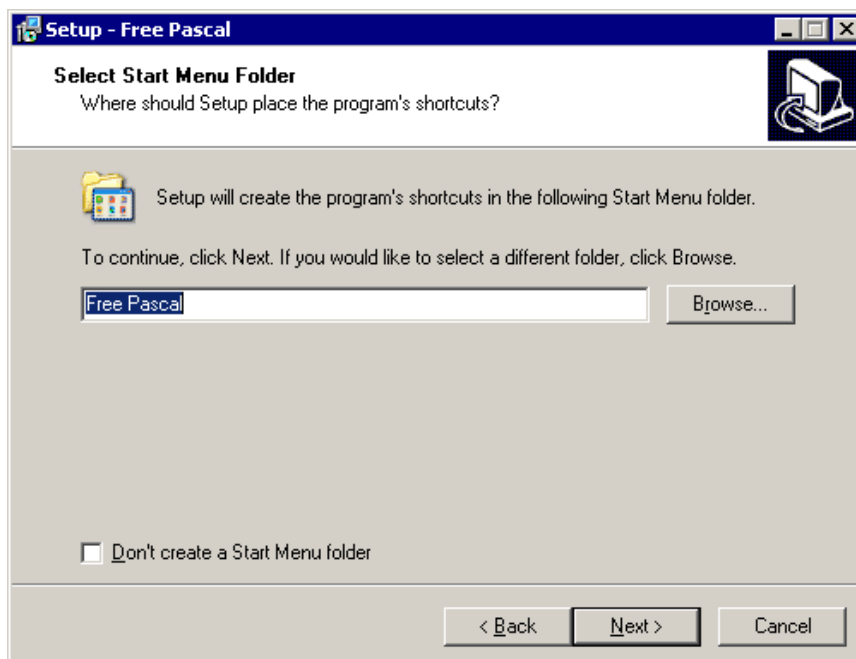
Después se nos preguntará en qué carpeta queremos instalarlo (podemos aceptar la que nos propone):



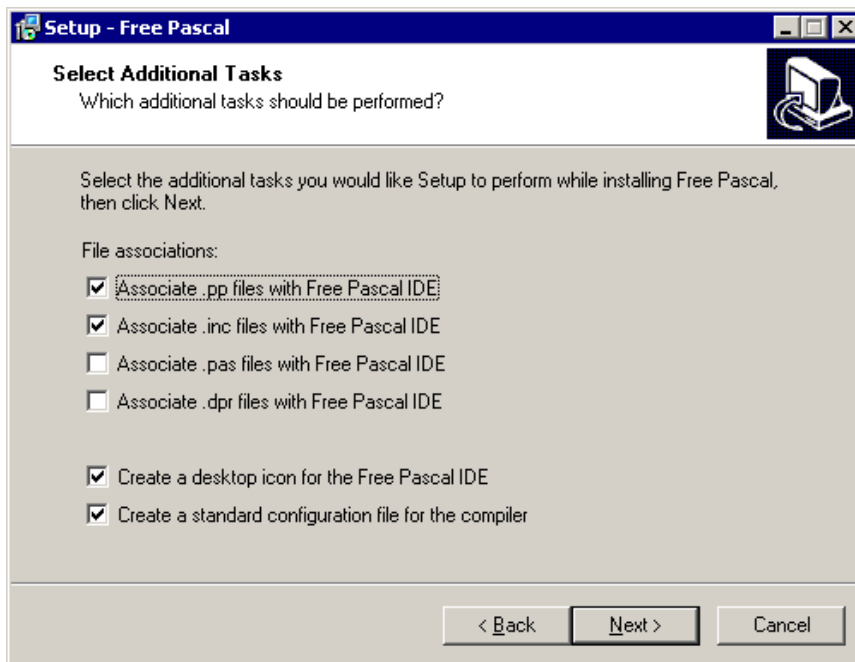
A continuación, nos pedirá que escojamos un tipo de instalación. En caso de duda, suele ser preferible escoger una Instalación Completa ("Full installation"):



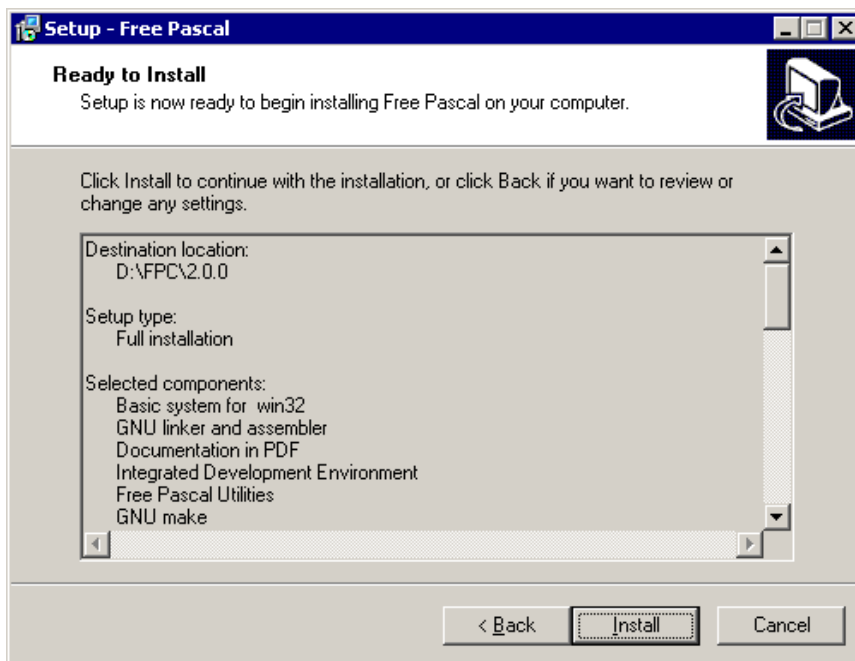
Nos dirá que escojamos un nombre para la carpeta en la que quedará instalado nuestro compilador, dentro del menú de Inicio:



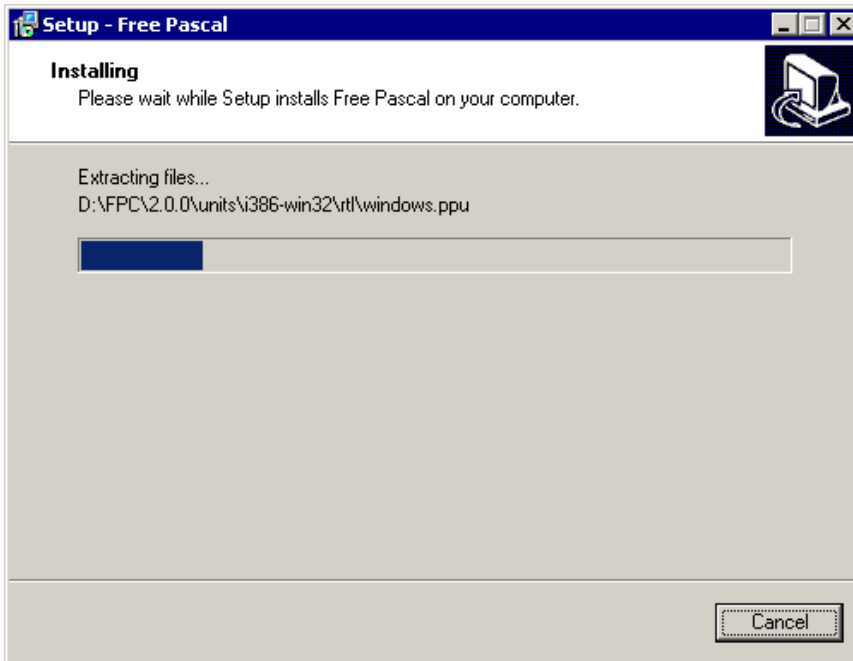
También podemos decir si queremos que se asocien ciertos ficheros con Free Pascal, de modo que se abran automáticamente desde Free Pascal al hacer doble clic sobre ellos. Si tenemos instalado Delphi, quizá sea interesante no asociar los ficheros ".pas" ni los ".dpr". Si no tenemos Delphi, nos puede hacer más cómodo el trabajo el asociar los ".pas" con Free Pascal:



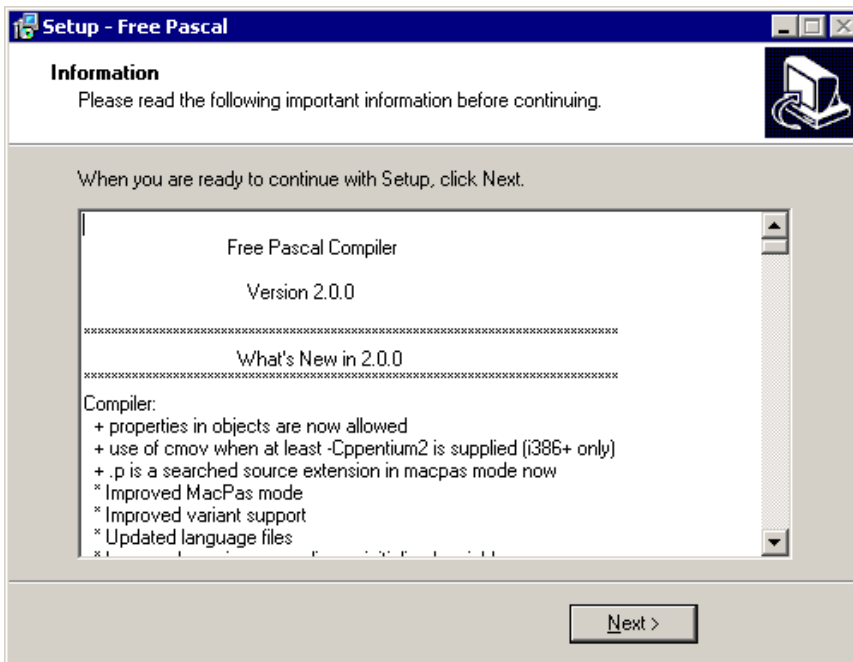
Se nos mostrará un resumen de todo lo que hemos escogido:



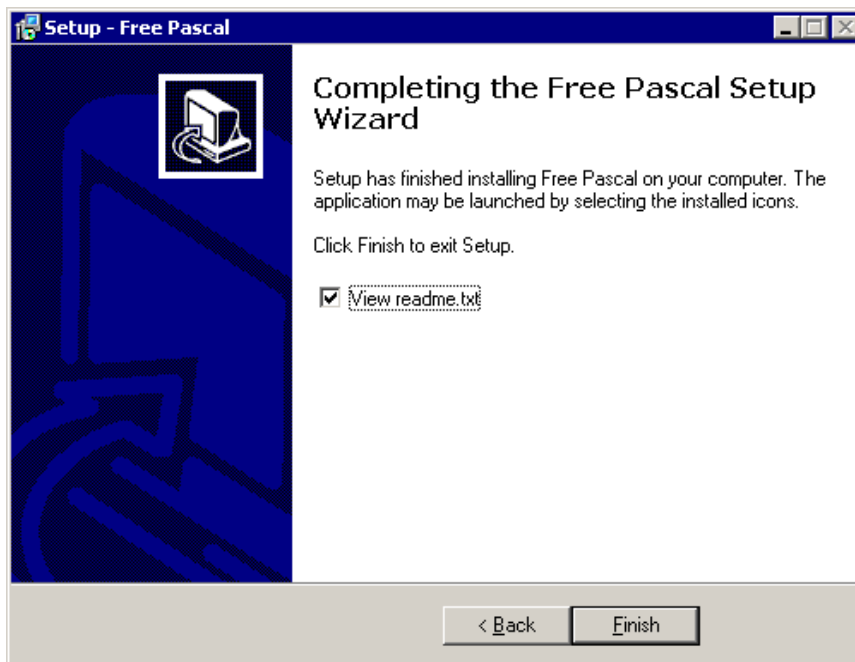
Comenzará la instalación propiamente dicha, y se nos irá informando de cómo avanza:



Después aparece un último aviso con información de última hora que pueda ser importante:



Y con eso ya ha terminado.

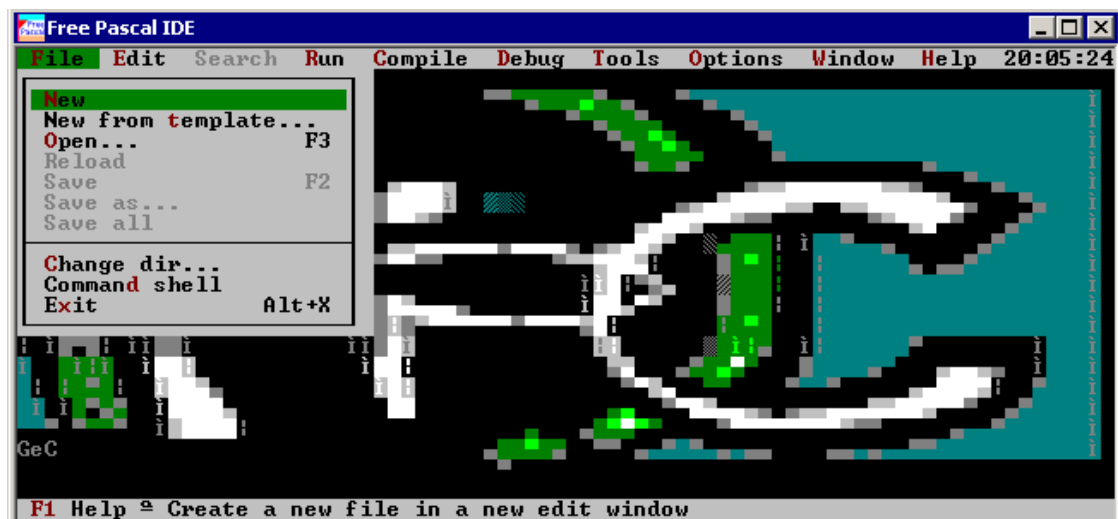


1.4.3. Probando el compilador con un ejemplo básico.

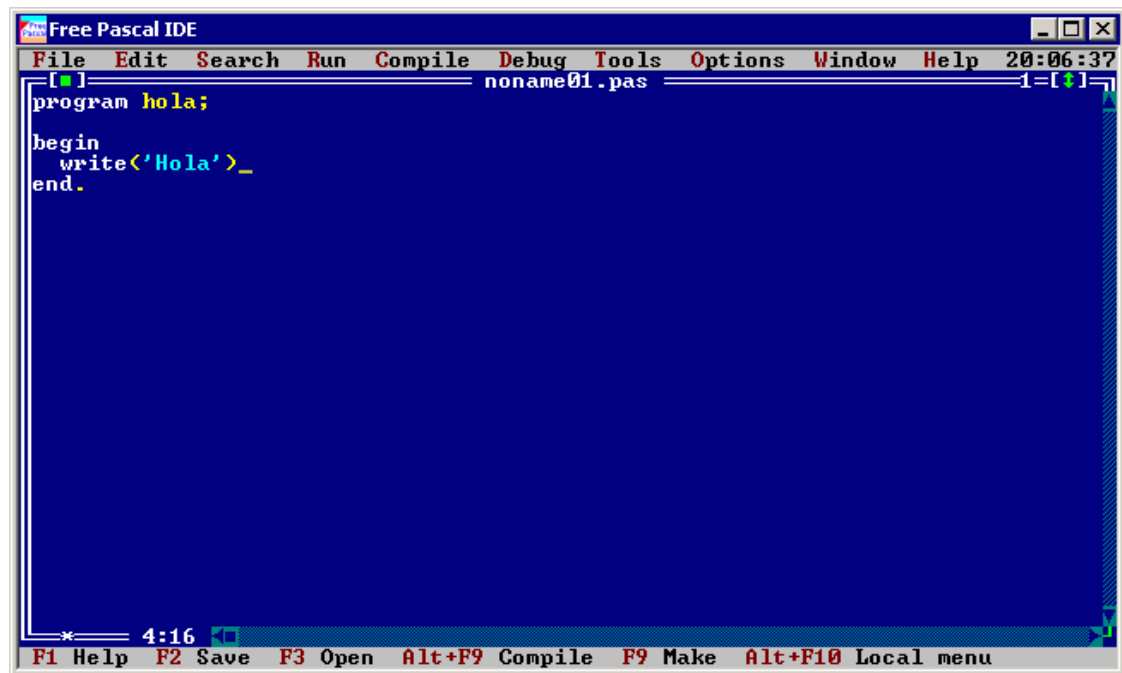
El icono del compilador debería aparecer en nuestra pantalla:



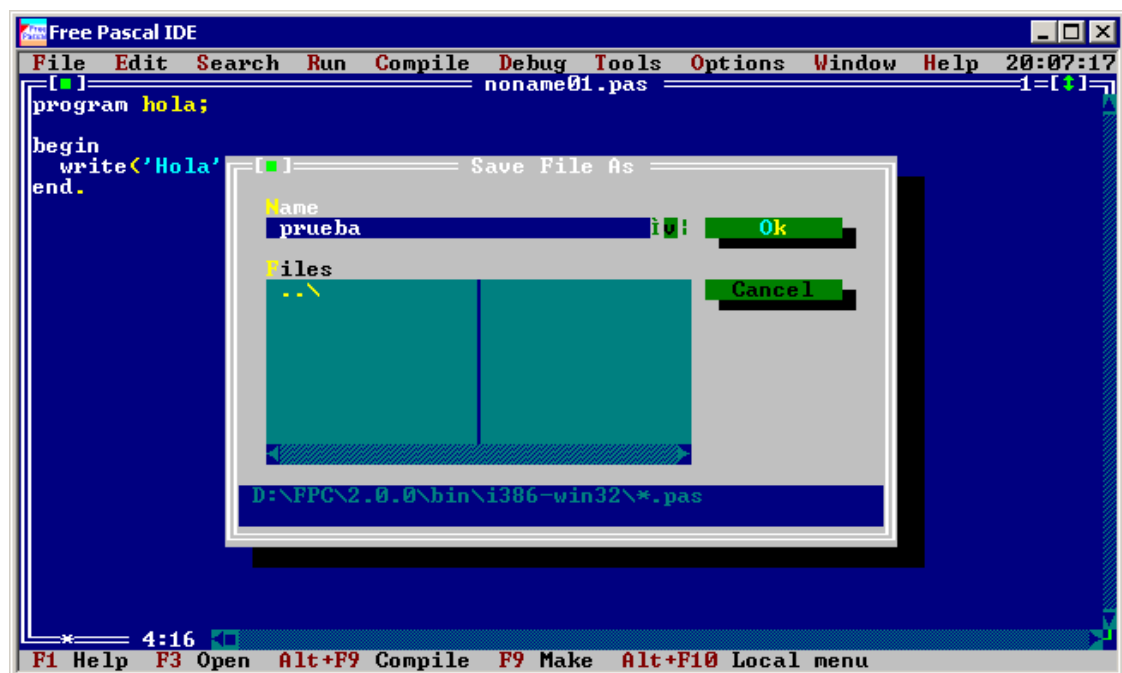
Al hacer doble clic, se abrirá el entorno de desarrollo. Desde el menú "File" (Archivo), con la opción "New" (Nuevo)



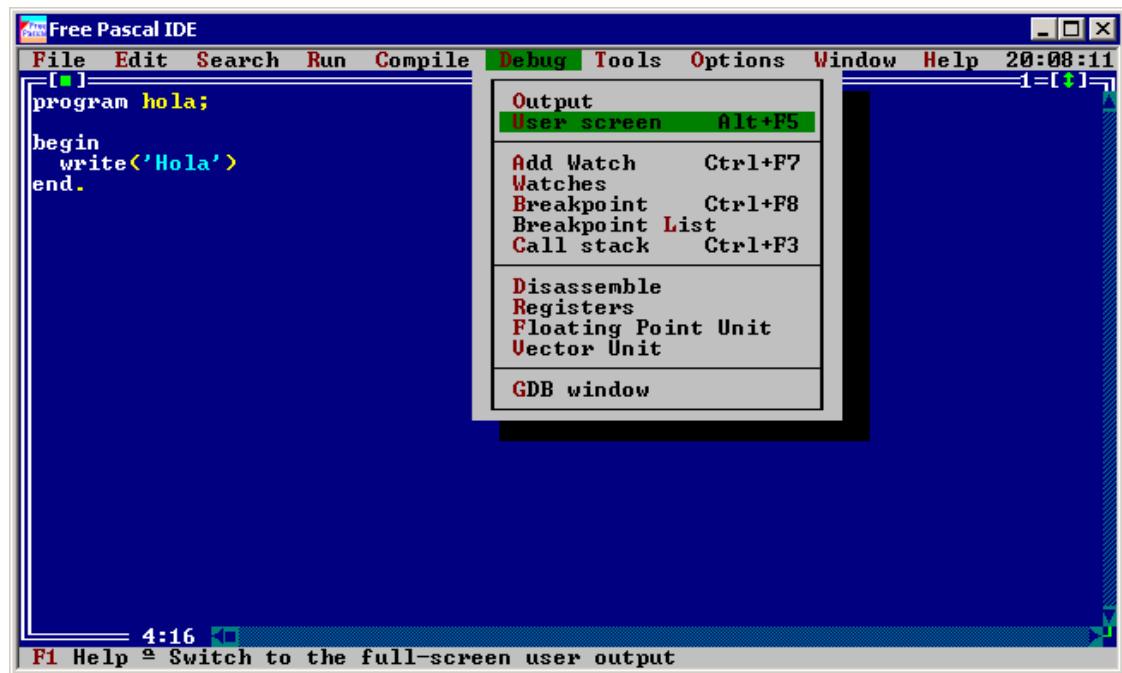
Según vamos escribiendo se realzará la sintaxis en colores, para ayudarnos a descubrir errores:



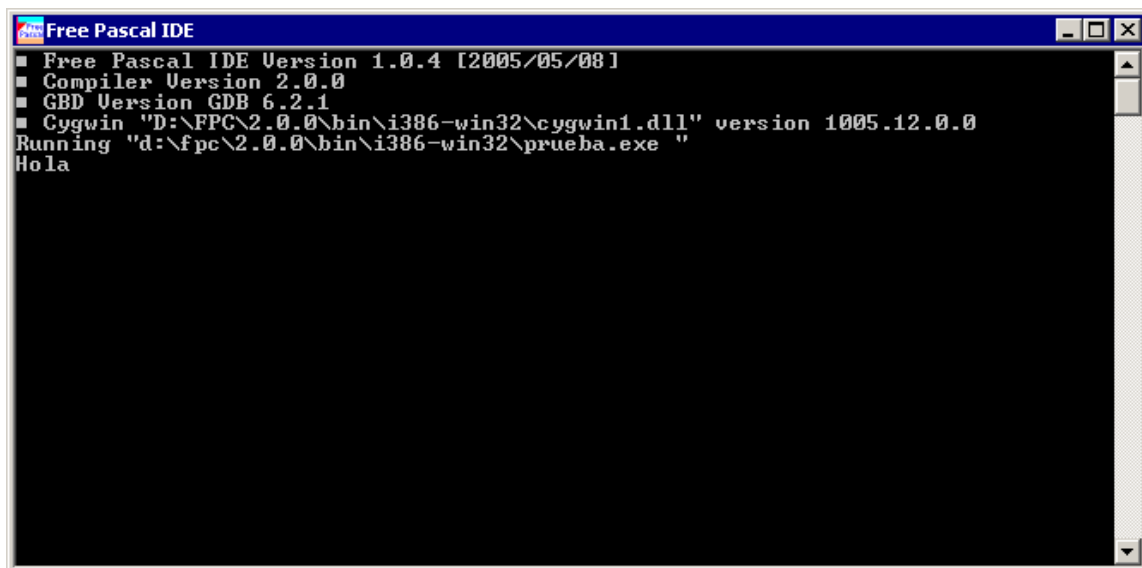
Podemos poner nuestro programa en marcha desde la opción "Run" (Ejecutar), del menú "Run". Eso sí, nos pedirá un nombre con el que guardar nuestro fichero si todavía no lo habíamos hecho.



El único problema es que quizá el programa de prueba se ponga en marcha y termine tan rápido que no tengamos tiempo de ver nada en pantalla. Podemos solucionarlo comprobando qué ha aparecido en la "Pantalla de usuario", en la opción "User Screen" del menú "Debug":

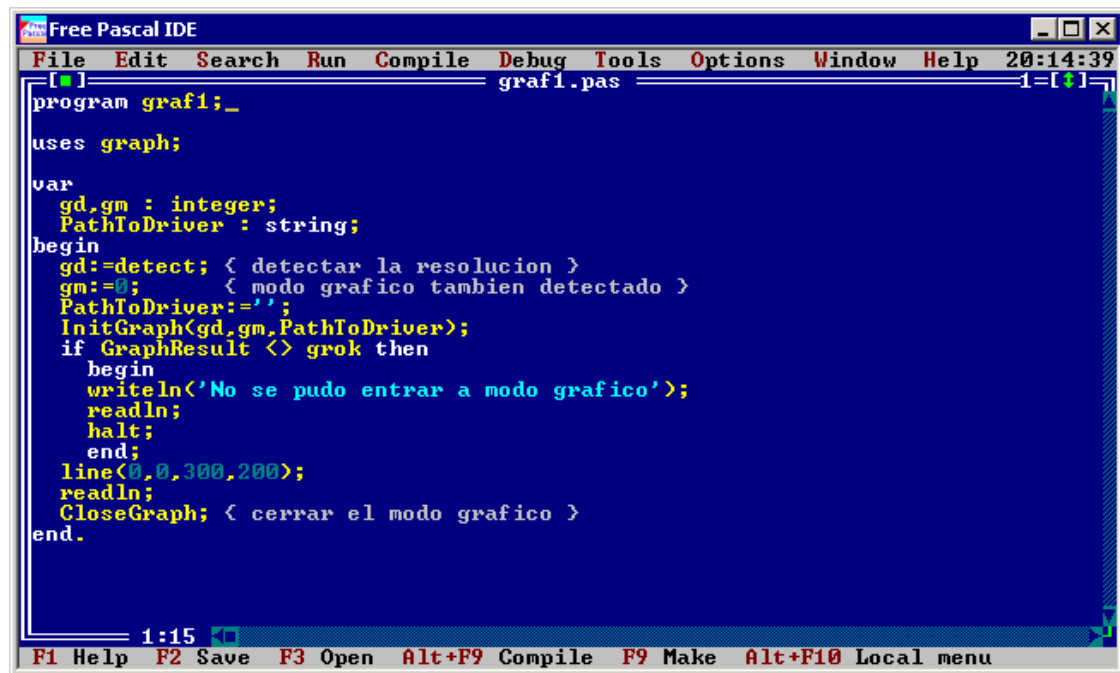


y veríamos algo parecido a esto:



1.4.4. Probando el modo gráfico.

Como todo lo necesario para trabajar en modo gráfico es parte de las bibliotecas incluidas con Free Pascal, no necesitamos nada más para crear aplicaciones gráficas, basta con escribir un fuente sencillo como este (que explicaremos en el siguiente apartado):

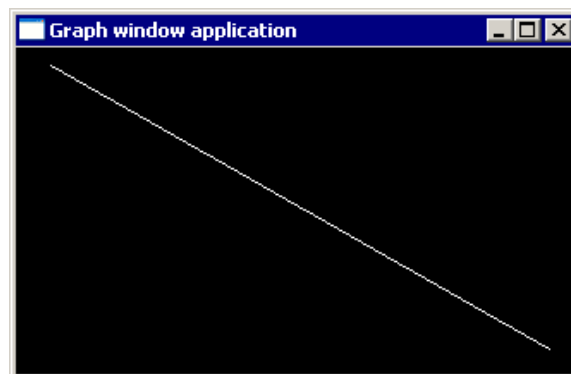


```

Free Pascal IDE
File Edit Search Run Compile Debug Tools Options Window Help 20:14:39
graf1.pas
program graf1;_
uses graph;
var
  gd:gm : integer;
  PathToDriver : string;
begin
  gd:=detect; < detectar la resolucion >
  gm:=0; < modo grafico tambien detectado >
  PathToDriver:='';
  InitGraph(gd,gm,PathToDriver);
  if GraphResult <> grok then
  begin
    writeln('No se pudo entrar a modo grafico');
    readln;
    halt;
  end;
  line(0,0,300,200);
  readln;
  CloseGraph; < cerrar el modo grafico >
end.
1:15
F1 Help F2 Save F3 Open Alt+F9 Compile F9 Make Alt+F10 Local menu

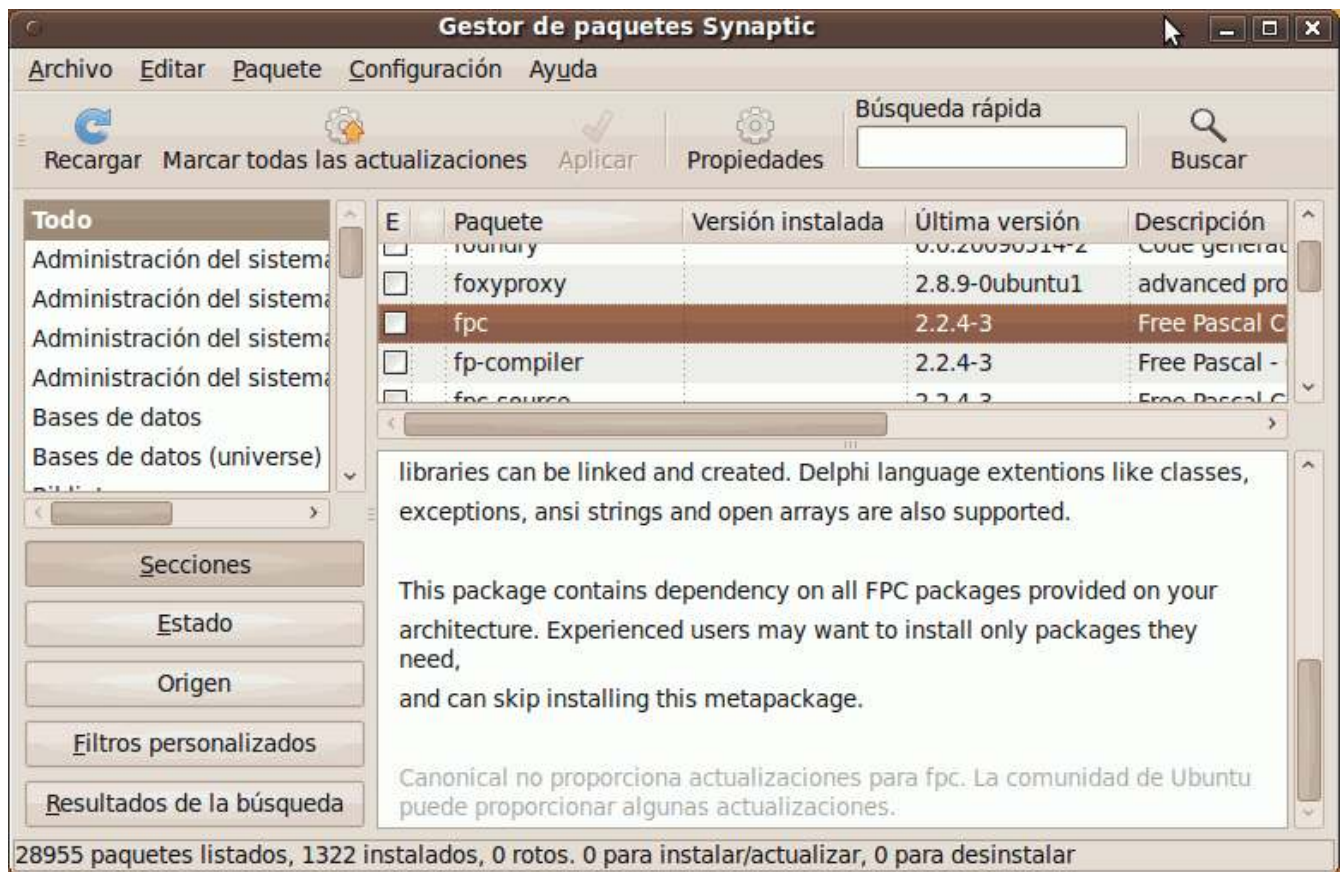
```

Simplemente lo ponemos en marcha (menú "Run", opción "Run") y aparecerá algo como esto:



1.5. Instalando Free Pascal para Linux.

La instalación debería ser sencillo: basta con buscar el paquete "fpc" (o "fp-compiler") en Synaptic o el que sea nuestro gestor de paquetes. El sistema debería instalar automáticamente los paquetes adicionales que fueran necesarios, y un breve instante después de pulsar el botón "Aplicar", debería quedar listo para usar:



Crearíamos nuestro fuente con cualquier editor, como GEdit (si usamos el escritorio Gnome) o Kate (bajo KDE):



Compilaríamos el fuente tecleando "fpc" seguido de su nombre, por ejemplo "fpc hola.pas". Si todo va bien, no tendremos mensajes de error y podremos lanzar el ejecutable con "." seguido de su nombre, por ejemplo "./hola".

```

yo@miPcUbuntu: ~
Archivo  Editar  Ver  Terminal  Ayuda
yo@miPcUbuntu:~$ gedit hola.pas &
[1] 5331
yo@miPcUbuntu:~$ fpc hola.pas
Free Pascal Compiler version 2.2.4-3 [2009/06/04] for i386
Copyright (c) 1993-2008 by Florian Klaempfl
Target OS: Linux for i386
Compiling hola.pas
Linking hola
/usr/bin/ld: warning: link.res contains output sections; did you forget -T?
6 lines compiled, 0.4 sec
yo@miPcUbuntu:~$ ./hola
Hola
yo@miPcUbuntu:~$ █

```

1.6. Instalando Free Pascal para DOS.

(Pronto disponible)

1.7. Instalando Java (JDK) para Windows.

Contenido de este apartado:

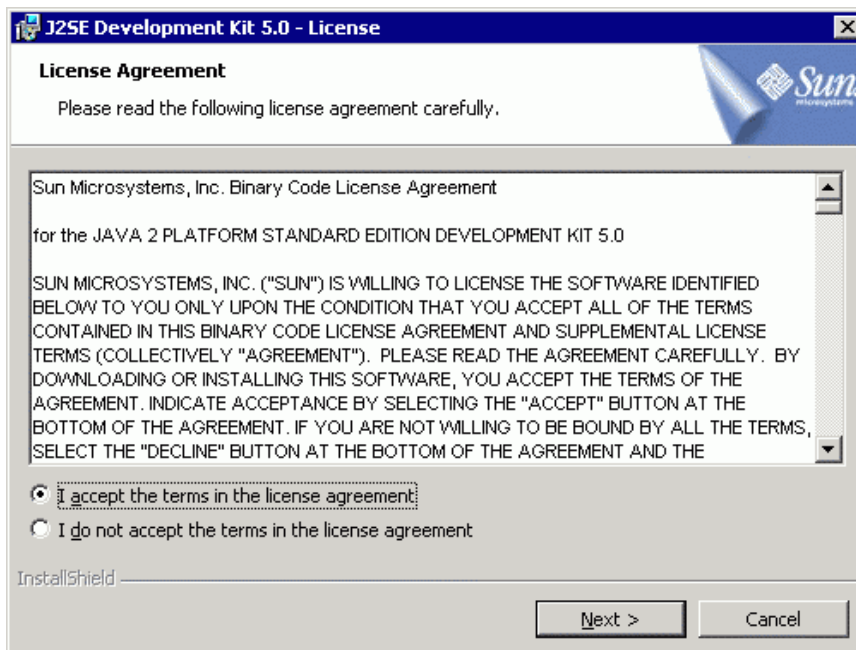
- [¿Dónde encontrar el compilador?](#)
- [¿Cómo instalarlo?](#)
- [Instalando un editor y probando un ejemplo básico.](#)

1.7.1. ¿Dónde encontrar el compilador?

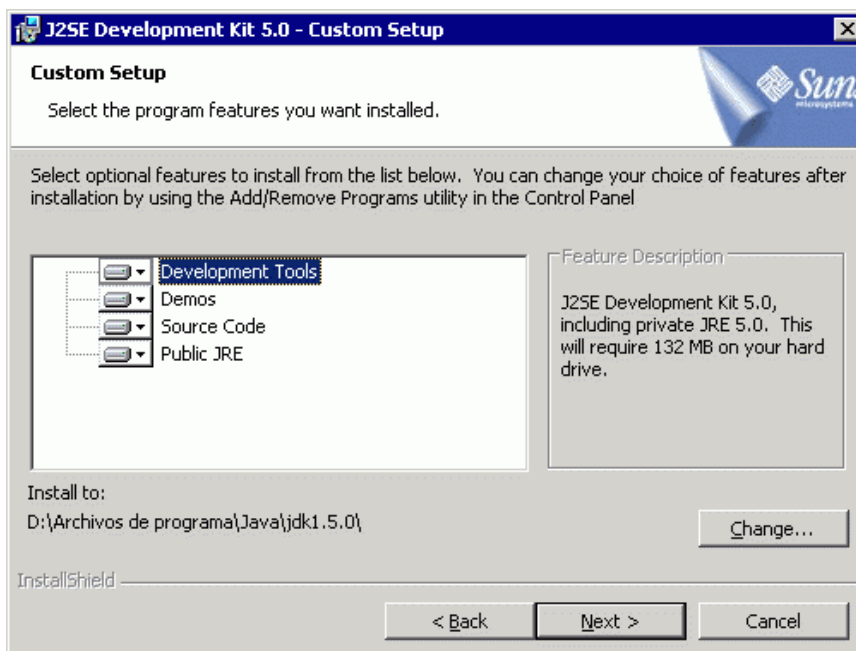
La web oficial de Java es java.sun.com. Hay distintas versiones para descargar, según el sistema operativo que deseemos y según si queremos también el entorno de desarrollo NetBeans o no. La versión sin entorno de desarrollo debería ser una descarga de unos 50 Mb de tamaño, y pasa de los 100 Mb con entorno. Yo voy a utilizar la versión 1.5.0 sin entorno de desarrollo, porque no vamos a crear entornos de usuario, así que podemos permitirnos usar editores sencillos para crear nuestros programas.

1.7.2. ¿Cómo instalarlo?

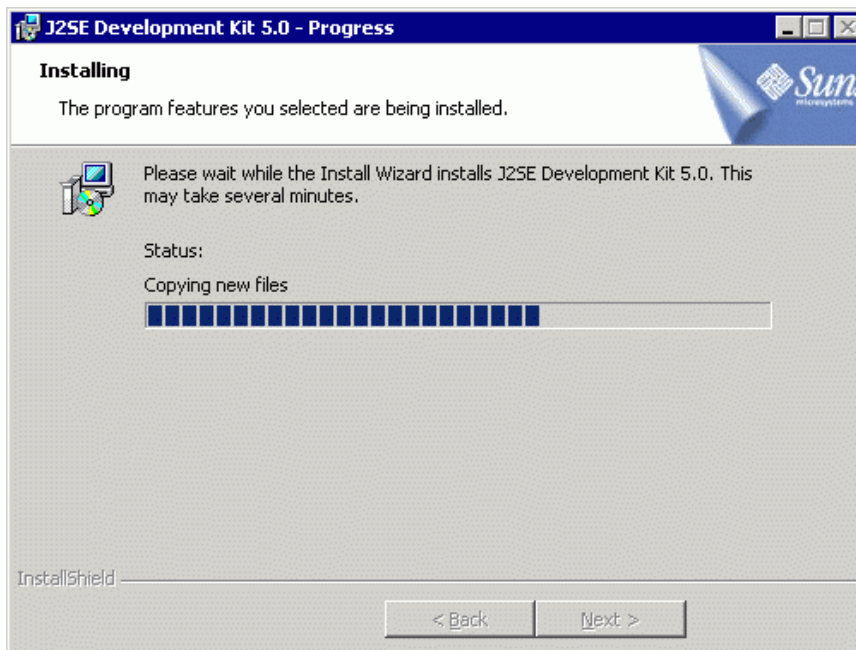
La instalación es sencilla. Hacemos doble clic en el fichero que hemos descargado. Tras la pantalla de bienvenida, se nos pide que aceptemos el contrato de licencia:



El siguiente paso es escoger qué componentes queremos instalar (en principio, todos):



Comenzará el proceso de copia de archivos...



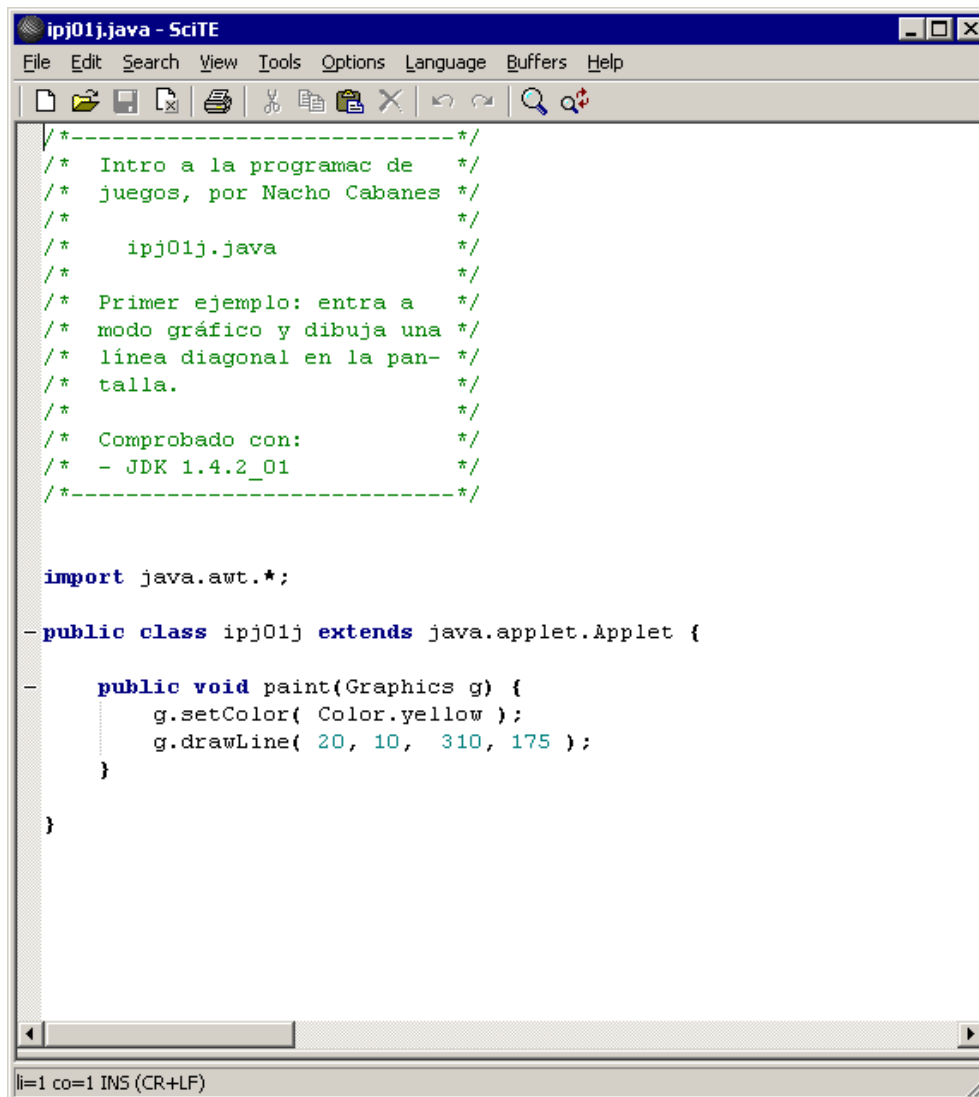
Y terminó:



1.7.3. Instalando un editor y probando un ejemplo básico.

Nos hará falta un editor, ya que hemos descargado el entorno sin ninguno. Yo usaré **SciTe**, gratuito, que se puede descargar desde www.scintilla.org. Este editor se puede usar para compilar con distintos lenguajes, entre ellos Java.

La apariencia del editor (ya con un fuente sencillo de prueba teclado) es ésta:



```
ipj01j.java - SciTE
File Edit Search View Tools Options Language Buffers Help

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* ipj01j.java */
/* Primer ejemplo: entra a */
/* modo gráfico y dibuja una */
/* línea diagonal en la pan- */
/* talla. */
/* Comprobado con: */
/* - JDK 1.4.2_01 */
/*-----*/

import java.awt.*;

public class ipj01j extends java.applet.Applet {

    public void paint(Graphics g) {
        g.setColor( Color.yellow );
        g.drawLine( 20, 10, 310, 175 );
    }
}

li=1 co=1 INS (CR+LF)
```

Para compilar nuestro programa, bastaría con usar la opción "Compile" del menú "Tools":

```

ipj01j.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 ipj01j.java 2 cpp.proper
/*-----
/*  Intro a la
/*  juegos, por
/*  ipj01j.ja
/*
/*  Primer ejemplo: entra a
/*  modo gráfico y dibuja una
/*  línea diagonal en la pan-
/*  talla.
/*
/*  Comprobado con:
/*  - JDK 1.4.2_01
/*-----

import java.awt.*;

public class ipj01j extends java.applet.Applet {

    public void paint(Graphics g) {
        g.setColor( Color.yellow );
    }
}

>d:\j2sdk\bin\javac ipj01j.java
>El sistema no puede hallar la ruta especificada.
>"d:\archivos de programa\java\jdk1.5.0\bin\javac" ipj01j.java
>Exit code: 0
|l=1 co=1 INS (CR+LF)

```

Si no consigue compilar nuestro programa, puede ser porque está buscando el compilador de java ("javac") en la carpeta correcta. Le indicamos donde está, usando la opción "Open cpp properties", del menú "Options". Desde allí se pueden cambiar las opciones que tienen que ver con los lenguajes C, C++ y Java. Al final del fichero de configuración que aparece, se encuentra la orden necesaria para compilar fuentes en Java. Tendríamos que corregirla, para indicar la carpeta en la que lo hemos instalado:

```

command.help.$(file.patterns.cpp)=$(CurrentWord) !G:\Program Files\Microsoft
command.help.subsystem.$(file.patterns.cpp)=4
command.go.*.js=cscript /nologo $(FileNameExt)
# When maintaining old Win16 programs...
# command.help.$(file.patterns.cpp)=$(CurrentWord) !I:\msvc\help\win31wh.hlp
# command.help.subsystem.$(file.patterns.cpp)=5

if PLAT_GTK
    command.help.$(file.patterns.cpp)=man $(CurrentWord) | col -b

# C# is only available on Windows
if PLAT_WIN
    command.build.*.cs=csc /t:winexe $(FileNameExt) /r:system.dll,system.drawing
    command.go.*.cs=$(FileName)
    command.go.subsystem.*.cs=1

command.compile.*.java="d:\archivos de programa\java\jdk1.5.0\bin\javac" $(FileNameE
command.build.*.java=javac *.java
command.go.*.java=java $(FileName)

>d:\j2sdk\bin\javac ipj01j.java
>El sistema no puede hallar la ruta especificada.

```

Después de guardar los cambios, nuestro programa en Java debería compilar correctamente, pero aun no se puede usar.

Vamos a crear **Applets**, pequeños programas en Java que se podrán usar desde cualquier ordenador en el que exista un navegador capaz de "entender" Java. Por eso, deberemos crear también una pequeña **página Web** encargada de mostrar ese Applet.

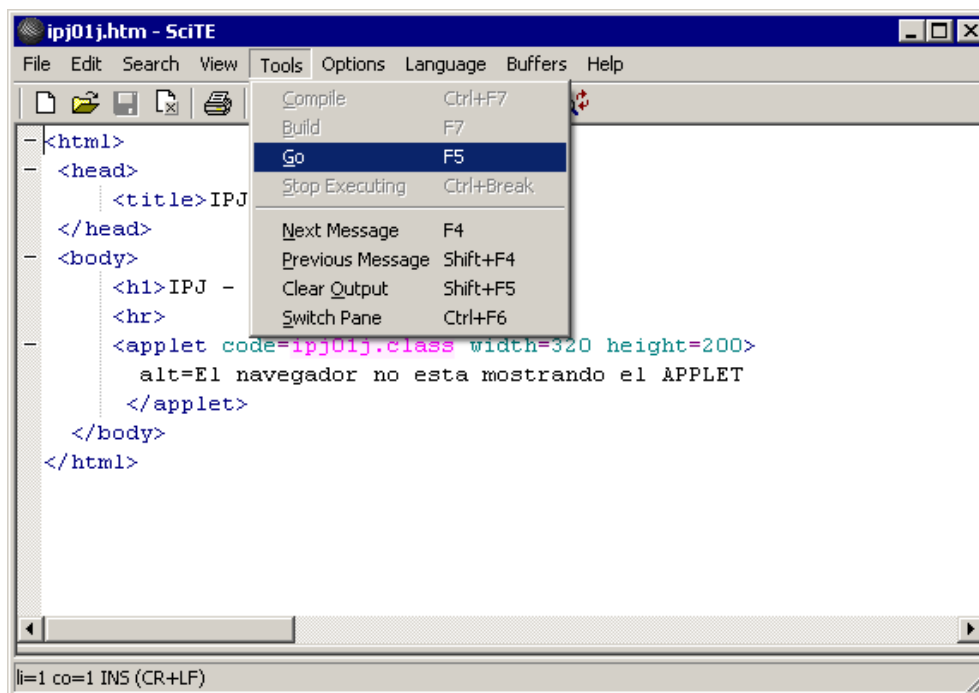
Desde el mismo editor SciTe, tecleamos lo siguiente:

```

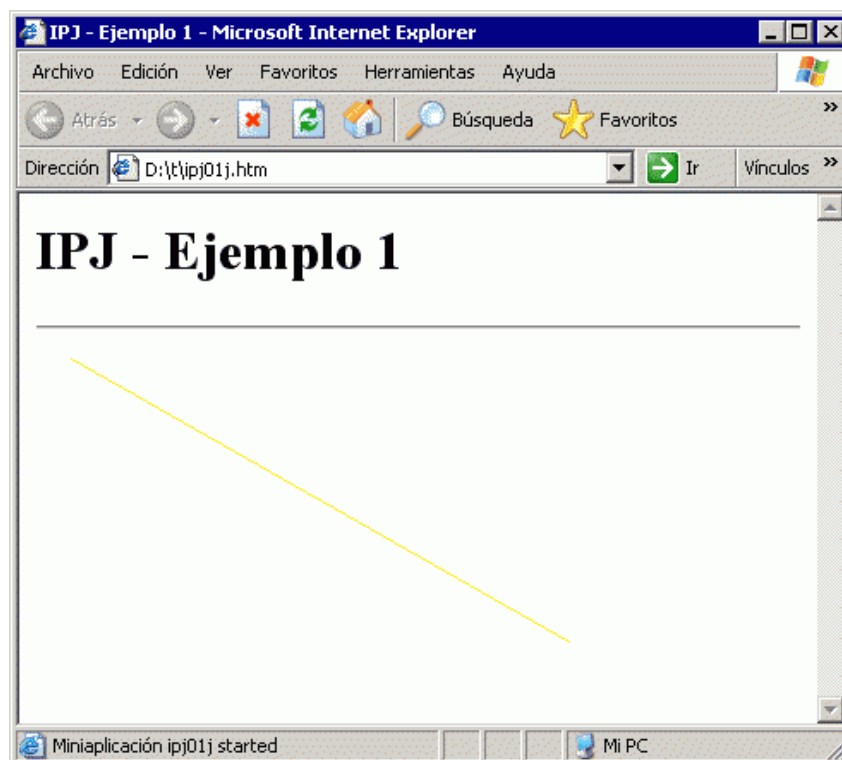
<html>
<head>
    <title>IPJ - Ejemplo 1</title>
</head>
<body>
    <h1>IPJ - Ejemplo 1</h1>
    <hr>
    <applet code=ipj01j.class width=320 height=200>
        alt=El navegador no esta mostrando el APPLET
    </applet>
</body>
</html>

```

Lo guardamos con el nombre que decidamos (terminado en .htm o .html, claro) y lo probamos con la opción "Go" del menú "Tools" (ahora no está disponible "Compile", como era de esperar):



Aparecerá nuestro navegador mostrando el Applet que hemos creado:



1.8. Instalando Java (JDK) para Linux.

(Pronto disponible)

1.9. Instalando Dev-C++ y Allegro para Windows.

Contenido de este apartado:

- [¿Dónde encontrar el compilador?](#)
- [¿Cómo instalarlo?](#)
- [Instalando la biblioteca Allegro.](#)
- [Probando el modo gráfico desde el entorno.](#)

1.9.1. ¿Dónde encontrar el compilador?

Dev-C++ es un entorno que podemos descargar con el compilador GCC ya incluido, todo en un único paquete, de forma que sólo tengamos que instalarlo y ya podamos empezar a trabajar en C o C++ sin preocuparnos más. Necesitaríamos:

- El entorno y el compilador de Dev-C++ están en www.bloodshed.net
- La librería Allegro en alleg.sourceforge.net
- Las rutinas necesarias para que Allegro saque partido de los DirectX de Windows también serán necesarias. Las de DirectX 8 están en el fichero dx80_mgw.zip que puede descargar de alleg.sourceforge.net

1.9.2. ¿Cómo instalarlo?

Haciendo doble clic en el fichero que hemos descargado. Primero se nos preguntará qué componentes queremos instalar (lo habitual será instalar todos):

Luego la carpeta en que vamos a instalarlo y después el idioma del entorno (incluido español):

Cuando termine de instalarse, podemos crear un primer programa estándar de prueba (el clásico "Hola.c", por ejemplo), que debería funcionar ya correctamente.

1.9.3. Instalando la biblioteca Allegro.

Primero descomprimos Allegro (el fichero allegro403.zip o similar) en la carpeta que deseemos (yo lo suelo hacer dentro de DevCpp, en una subcarpeta llamada "contrib" (por eso de que estrictamente no es parte de Dev-C++ sino una contribución).

Después descomprimiremos el fichero "dx80_mgw.zip" dentro de la carpeta de Allegro, conservando las subcarpetas, y sobrescribiendo cualquier fichero ya existente que tenga el mismo nombre que los nuevos y fecha más antigua (si la fecha del existente es más reciente, normalmente será preferible conservarlo).

Entramos a la carpeta en que hemos descomprimido Allegro y tecleamos:

```
SET MINGDIR=C:\DevCpp
fix.bat mingw32
make
```

La primera línea indica dónde hemos instalado el compilador DevC++ (la sintaxis es la misma que para MinGW, ya que es éste el compilador en el que se apoya), la segunda hace unos pequeños cambios a la instalación para adaptarla a MinGW, y la tercera comienza el proceso de instalación propiamente dicho.

Si todo va bien (en mi caso no hubo ningún problema), al cabo de un rato aparecerá:

```
The optimised MinGW32 library has been compiled.
Run make install to complete the installation.
```

Pues vamos con ello. Tecleamos

```
make install
```

que se encarga de copiar cada fichero donde corresponda, y en un instante se nos confirmará que todo está listo:

```
The optimised MinGW32 library has been installed.
```

1.9.4. Probando el modo gráfico desde el entorno integrado.

5) Ya sólo falta comprobar que Allegro también está correctamente instalado. Abrimos cualquiera de los ejemplos desde el editor. Para compilarlo correctamente, nos hace falta añadir la línea "-lalleg" a las opciones del compilador. Esto lo podemos hacer desde el menú "Herramientas", en "Opciones del compilador":

Sólo con ese cambio, los fuentes en C deberían compilar correctamente. Al igual que con MinGW, los ejecutables resultantes son de pequeño tamaño (de hecho, algo menores incluso que con la instalación típica de MinGW) pero junto con ellos hay que distribuir un fichero DLL llamado "alleg40.dll", de unos 900 Kb de tamaño, que inicialmente se encuentra en la carpeta "system" de nuestro Windows).

Si queremos compilar un ejemplo más complejo que esté **formado por varios fuentes**, deberemos crear un proyecto (desde el menú "Proyecto", claro), añadir los fuentes que nos interesen, etc. Eso sí, a la hora de compilar posiblemente tendremos un problema, porque cuando es un proyecto, primero se compila y luego se enlaza. La forma de solucionarlo será añadir "-lalleg" a las opciones del "linker" (el segundo recuadro de la imagen anterior).

1.10. Instalando MinGW y Allegro para Windows.

Contenido de este apartado:

- [¿Dónde encontrar el compilador?](#)
- [¿Cómo instalarlo?](#)
- [Probando el compilador con un ejemplo básico.](#)
- [Instalando la biblioteca Allegro.](#)
- [Probando el modo gráfico desde la línea de comandos.](#)
- [Probando el modo gráfico desde el entorno.](#)

1.10.1. ¿Dónde encontrar el compilador?

Instalar MinGW a veces es un poco engorroso por la cantidad de paquetes que hay que descargar e instalar. Por eso yo me he decantado por una actualización que incluye el compilador listo para usar y un editor. Se llama **MinGW Developer Studio** y se puede descargar desde www.parinyasoft.com. No necesitaremos la versión grande que incluye "wxWindows", porque no vamos a crear entornos de usuario para Windows, así que nos quedamos con la versión "intermedia" (editor y compilador).

1.10.2. ¿Cómo instalarlo?

La instalación es sencilla, poco más que seguir los pasos que nos indique el asistente. La primera pantalla es la de bienvenida:

Y después aceptar la licencia de usuario:

Ahora elegimos los componentes que queremos instalar. Lo razonable sería instalar todo, a no ser que nos falte espacio. En cualquier caso, deberíamos instalar al menos el compilador (MinGW), y sería deseable incluir también el entorno (Developer Studio):

Después se nos preguntará en qué carpeta queremos instalarlo (podemos aceptar la que nos propone):

Veremos cómo va avanzando la copia de ficheros

Y cuando termine, se nos dará la opción de poner en marcha el entorno:

1.10.3. Probando el compilador con un ejemplo básico.

Podemos comenzar a usar el entorno desde la última pantalla de la instalación o bien haciendo doble clic en el icono que habrá aparecido en nuestro escritorio:

Aparecerá el entorno de desarrollo, vacío:

Para comenzar a teclear, deberemos usar la opción "New" (Nuevo) del menú "File" (Archivo), que nos preguntará qué tipo de proyecto queremos crear:

Nos bastará con una "aplicación de consola" (modo texto: "Win 32 Console Application"). Escogemos la carpeta en la que queremos que se guarde nuestro proyecto y daremos un nombre al proyecto:

A su vez, el proyecto deberá contener al menos un fichero fuente, así que volvemos a "File", "New":

Tecleamos el clásico programa capaz de escribir "Hola" en pantalla:

Y lo ponemos en marcha con la opción "Execute..." (Ejecutar), del menú "Build" (Construir):

El resultado será algo así:

1.10.4. Instalando la biblioteca Allegro.

Necesitaremos dos cosas:

- La propia biblioteca Allegro (para la versión 4.03 será el fichero all403.zip, de algo menos de 3 Mb de tamaño).
- El fichero DX70_MGW.ZIP (de menos de 250 Kb de tamaño) , que permite a Allegro usar las bibliotecas DirectX, ya que vamos a crear programas para Windows.

En teoría, deberíamos descomprimir ambos ficheros dentro de la carpeta en la que está instalado el compilador MinGW (que a su vez será una subcarpeta de la de instalación del Developer Studio):

En la práctica, podríamos usar casi cualquier carpeta, porque luego le indicaremos en qué carpeta está instalado.

Una vez descomprimidos, deberemos teclear ciertas órdenes para compilar la biblioteca Allegro:

En primer lugar, entraremos al intérprete de comandos. Por ejemplo, desde el menú "Inicio" de Windows, y la opción "Ejecutar" teclearíamos CMD, si estamos en Windows XP o Windows 2000 (o superior), o bien COMMAND si estamos en una versión más antigua, como Windows 98:

Ya desde la pantalla negra del intérprete de comandos, teclearemos las siguiente órdenes:

D:	O la letra de la unidad de disco en que hemos instalado MinGW Developer Studio, para entrar a esa unidad de disco.
CD MINGWSTUDIO\ALLEGRO	Para entrar a la carpeta en que hemos descomprimido Allegro.
PATH=D:\MINGWSTUDIO\MINGW\BIN;%PATH%	Para que el sistema operativo encuentre las utilidades de MinGW en la carpeta correcta.
SET MINGDIR=D:\MINGWSTUDIO\MINGW	Para que la instalación de Allegro sepa dónde debe colocar los ficheros resultantes.
FIX.BAT MINGW32	Para configurar Allegro para MinGW.

El último paso, en un caso general, sería teclear

MAKE

para que se instale Allegro realmente. En nuestro caso, la herramienta MAKE que incorpora esta versión de MinGW tiene su nombre cambiado, de modo que deberemos teclear

MINGW32-MAKE

Y comenzará el proceso

Es posible que aparezca algún aviso, diciendo que alguna opción está anticuada ("deprecated"), pero no supondrá ningún problema mientras veamos que el proceso no se para.

Si todo va bien, en vez de ningún mensaje de error "grave", obtendremos al final un aviso que nos diga que la biblioteca ha sido compilada y que tecleemos "**make install**" para completar la instalación:

Cuando tecleemos esta última orden, la instalación quedará terminada:

1.10.5. Probando el modo gráfico desde la línea de comandos.

Para comprobar que todo funciona correctamente, bastaría con entrar a la carpeta de ejemplos de ALLEGRO, compilar uno de ellos y ponerlo en funcionamiento.

Las órdenes que usaríamos serían:

CD EXAMPLES	Para entrar a la carpeta de ejemplos de allegro
GCC ex3d.c -lalleg -o ejemplo3d.exe	Para compilar el ejemplo llamado "ex3d.c", incluyendo la biblioteca "alleg" (Allegro) y creando un ejecutable llamado "ejemplo3d.exe"
EJEMPLO3D	Para probar ese ejecutable "ejemplo3d"

Y todo debería funcionar correctamente:

1.10.6. Probando el modo gráfico desde el entorno integrado.

Los pasos serán muy parecidos a cuando probamos el ejemplo básico. En primer lugar, creamos un nuevo proyecto desde "File" y "New" (por ejemplo, llamado pruebaGrafica):

Después, hacemos lo mismo para crear el fuente. Podemos usar un nombre como "pruebaG1.c" (si indicamos la extensión .c sabrá que queremos que sea un fuente en C; si no la indicáramos, se añadiría .cpp para que fuera en C++).

Podemos teclear (o copiar y pegar) cualquiera de los ejemplos básicos para probarlo:

Si intentamos compilar, no lo conseguiremos, porque no sabe dónde encontrar la biblioteca Allegro, de modo que el ejecutable no se creará correctamente. Se lo podemos indicar desde el menú "Project" (Proyecto), en la opción "Settings" (configuración);

El cambio que tenemos que hacer es decirle que debe enlazar también la biblioteca Allegro. Lo hacemos desde la pestaña "Link" (enlazar), tecleando "alleg" en la casilla de "Libraries" (bibliotecas).

Y ahora ya debería funcionar el ejemplo que hayamos escogido:

1.11. Creando programas para Windows con MinGW.

Si queremos crear juegos para Windows con Allegro, pero sin ningún entorno de desarrollo, podemos utilizar el compilador MinGW.

Los pasos que tendríamos que dar son:

1) Descargar los ficheros:

- El compilador MinGW (www.mingw.org)
- La librería Allegro se puede descargar desde la página de DJGPP (ya vista antes) o desde alleg.sourceforge.net
- Ciertas rutinas necesarias para que Allegro saque partido de los DirectX de Windows. Es el fichero dx70_mgw.zip que puede descargar de alleg.sourceforge.net

2) Instalar MinGW haciendo doble clic en el fichero que hemos descargado (es recomendable que lo hagamos en una carpeta que no contenga espacios en el nombre)..

3) Añadir esta carpeta en que hemos instalado el compilador al "PATH" (la lista de carpetas en la que MsDos -y Windows- buscará todo aquello que nosotros tecleemos y que él considere que es una orden, pero que no sabe dónde encontrar).

Lo conseguimos tecleando

```
PATH = C:\MinGW32;%PATH%
```

(en vez de "C:\MinGW32" debería aparecer la carpeta en la que hemos instalado el compilador).

Como suena razonable, en vez de hacer esto cada vez que queramos comenzar a usar nuestro compilador, será más cómodo crear un fichero BAT que nos ahorre trabajo, o incluso añadir esta línea al final de AUTOEXEC.BAT

4) Probar que el compilador funciona: tecleamos el clásico fuente "hola.c" y lo compilamos con

```
gcc hola.c -o hola.exe
```

Si no hay problemas, el compilador ya está listo.

5) Descomprimir las rutinas adicionales para acceder a las DirectX:

Descomprimiremos el fichero "dx70_mgw.zip" dentro de la carpeta de Allegro, conservando las subcarpetas, y sobrescribiendo cualquier fichero ya existente que tenga el mismo nombre que los nuevos.

6) Comenzar la instalación de Allegro:

Entramos a la carpeta en que hemos descomprimido Allegro y tecleamos:

```
SET MINGDIR=C:\MinGW32
cd c:\allegro
fix.bat mingw32
make
```

La primera línea indica dónde hemos instalado el compilador MinGW, la segunda entra a la carpeta de Allegro, la tercera hace unos pequeños cambios a la instalación para adaptarla a MinGW, y la cuarta comienza el proceso de instalación propiamente dicho.

(Como siempre, la primera y la segunda línea serán distintas según en qué carpeta hayamos instalado MingW y Allegro).

Nota: en mi instalación aparecía un error a medio crear esta biblioteca, diciendo que había un carácter incorrecto en la línea 129 del fichero "plugins.h"

Este fichero se encuentra dentro de la carpeta de allegro, en la subcarpeta

```
obj\mingw32\plugins.h
```

Basta con borrar un símbolo extraño que aparece al final, y volver a teclear "make"

Si todo va bien (sea al primer intento, o sea tras hacer alguna corrección como la mencionada), al cabo de un rato aparecerá:

```
The optimised MinGW32 library has been compiled.
Run make install to complete the installation.
```

Pues vamos con ello. Tecleamos

```
make install
```

que se encarga de copiar cada fichero donde corresponda, y en un instante se nos confirmará que todo está listo:

```
The optimised MinGW32 library has been installed.
```

7) Ya sólo falta comprobar que Allegro también está correctamente instalado. Probamos a compilar el ejemplo del ahorcado con

```
gcc ipj04.c -lalleg -o ipj04.exe
```

En mi caso, compiló perfectamente al primer intento, y además generó un fichero ejecutable de sólo 31.162 bytes de tamaño, comparado con los 556.185 bytes del mismo fuente compilado con DJGPP (eso sí, junto con los juegos hay que distribuir un fichero DLL llamado "alleg40.dll", de unos 900 Kb de tamaño, que inicialmente se encuentra en la carpeta "system" de nuestro Windows).

2. Entrando a modo gráfico y dibujando.

Contenido de este apartado:

- [Pautas generales.](#)
- [Cómo hacerlo en el caso del lenguaje C y la biblioteca Allegro.](#)
- [Cómo hacerlo en el caso de Free Pascal.](#)
- [Cómo hacerlo en el caso de Java.](#)

2.1. Pautas generales.

Los pasos básicos serán prácticamente los mismos, usemos el lenguaje que usemos:

- Debemos entrar a modo gráfico, y además generalmente deberemos indicar cuantos puntos queremos en pantalla y cuantos colores. Esta elección no siempre es trivial: cuantos más puntos y más colores queramos en pantalla, más costará "mover" toda esa información, así que se necesitará un ordenador más rápido (no sólo el nuestro, también el de las demás personas que usen nuestro juego) y más habilidad por nuestra parte a la hora de programar.
- Además tenemos el problema añadido de que no todos los ordenadores permiten todos los modos gráficos, por lo que deberíamos descubrir qué permite el ordenador "cliente" de nuestro juego y adaptarnos a sus posibilidades.
- Una vez en modo gráfico, tendremos órdenes preparadas para la mayoría de las necesidades habituales... pero frecuentemente no para todo lo que se nos ocurra, así que alguna vez tendremos que crear nuestras propias rutinas en casos concretos.
- Algunas órdenes a las que estamos acostumbrados no las podremos usar. Por ejemplo, es frecuente que no podamos usar desde modo gráfico la orden que se encarga de leer todo un texto que teclee un usuario (scanf en C, readln en Pascal, input en Basic). En caso de que queramos "imitar" el funcionamiento de esas órdenes, supondrá un trabajo adicional para nosotros (por ejemplo, en el caso de esa orden, se debería poder escribir texto, borrarlo en caso de error sin "estropear" el fondo, controlando también lo que ocurre al llegar al extremo de la pantalla o bien si se pulsan teclas

"especiales" como las flechas y Esc).

Por otra parte, las órdenes más habituales que usaremos serán las siguientes:

- Algunas propias de las bibliotecas gráficas, y que nos permitirán desde lo más básico (que es con lo que empezaremos), como dibujar líneas, rectángulos o escribir textos, hasta opciones avanzadas para representación y manipulación de imágenes planas y de figuras en tres dimensiones (que veremos más adelante).
- Otras generales del lenguaje escogido, que van desde las órdenes "habituales" para controlar el flujo de un programa (si..entonces, repetir..hasta) hasta órdenes más específicas (como las que generan números al azar).

En este apartado comentaremos cómo se entra a modo gráfico y como se dibujan los elementos habituales (líneas, rectángulos, círculos, puntos, etc) con las herramientas que hemos escogido.

En primer lugar, utilizaremos el modo gráfico más estándar, que encontraremos en cualquier tarjeta gráfica VGA o superior (cualquier PC posterior al 1992 debería tenerla): **320x200 puntos, en 256 colores**. Más adelante veremos cómo cambiar a otros modos que nos permitan obtener imágenes de mayor calidad.

Puedes leer todo el apartado o centrarte en un lenguaje:

- [Lenguaje C y la biblioteca Allegro](#).
- [Free Pascal](#).
- [Java](#).

2.2 Cómo hacerlo en el caso del lenguaje C y la biblioteca Allegro.

Los pasos básicos con C y Allegro son:

- Inicialización: Entre los "includes" deberemos añadir `<allegro.h>`, y al principio de nuestro "main" la orden `allegro_init();`
- Si vamos a usar el teclado (aunque en nuestro caso todavía será sólo para esperar a que se pulse una tecla antes de terminar), deberemos añadir `install_keyboard();`
- Entrar a modo gráfico: `set_gfx_mode(GFX_SAFE, ancho, alto, 0, 0)`. De momento sólo nos importan los dos primeros números: anchura y altura (en puntos) de la pantalla (320x200, por ahora). Esta función nos devolverá 0 si todo ha ido bien, o un valor distinto de cero si no se ha podido entrar al modo gráfico que hemos elegido.
- Dibujar una línea entre dos puntos: `line(screen, x1, y1, x2, y2, color);`, donde x1, y1 son las coordenadas horizontal y vertical del primer punto, x2 e y2 son las del segundo punto. "screen" indica dónde queremos dibujar (en la pantalla, ya veremos otros usos). "color" es el color en el que queremos que se dibuje la línea; si queremos que sea un color de la paleta estándar, podemos usar construcciones como `palette_color[15]` (el color 15 de la paleta estándar de un PC es el color blanco).
- Para hacer la pausa final antes de terminar el programa, usamos `readkey();`
- Salir del modo gráfico: añadiremos la línea `END_OF_MAIN();` después de "main".

Vamos a verlo un ejemplo "que funcione", que dibuje una diagonal en pantalla:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* ipj01c.c */
/* */
/* Primer ejemplo: entra a */
/* modo grafico y dibuja una */
/* linea diagonal en la pan- */
/* talla. */
/* */
/* Comprobado con: */
/* - Dev-C++ 4.9.9.2 */

```

```

/*      y Allegro 4.2.1          */
/*-----*/

#include <allegro.h>

int main() {
    allegro_init();
    install_keyboard();

    if (set_gfx_mode(GFX_SAFE, 320, 200, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    line(screen, 20, 10, 310, 175, palette_color[15]);
    readkey();

    return 0;
}

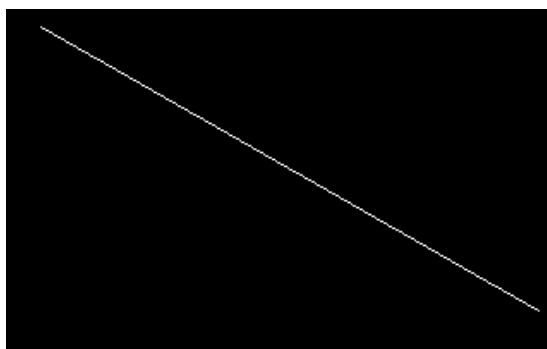
END_OF_MAIN();

```

Sólo hay dos cosas "nuevas" frente a lo que habíamos preparado:

- En caso de error, salimos a modo texto (GFX_TEXT), mostramos un mensaje de error (una aviso nuestro más el mensaje de error preparado por Allegro, y usamos para ello la función auxiliar "allegro_message"), y salimos con el código de error 1.
- Antes de terminar el programa, esperamos a que el usuario pulse una tecla, con la función "readkey()", que también es parte de Allegro.

El resultado será simplemente este (recuerda que [en el apartado 1](#) tienes la forma de teclear y compilar este fuente):



Demos un repaso rápido a las **posibilidades más básicas** de esta biblioteca, en lo que se refiere a modo gráfico.

- Dibujar un punto en un cierto color: void putpixel(BITMAP *bmp, int x, int y, int color);
- Dibujar una línea: void line(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);
- Línea horizontal: void hline(BITMAP *bmp, int x1, int y, int x2, int color);
- Línea vertical: void vline(BITMAP *bmp, int x, int y1, int y2, int color);
- Recuadro: void rect(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);
- Recuadro relleno: void rectfill(BITMAP *bmp, int x1, int y1, int x2, int y2, int color);
- Círculo: void circle(BITMAP *bmp, int x, int y, int radius, int color);
- Círculo relleno: void circlefill(BITMAP *bmp, int x, int y, int radius, int color);
- Elipse: void ellipse(BITMAP *bmp, int x, int y, int rx, int ry, int color);
- Elipse rellena: void ellipsefill(BITMAP *bmp, int x, int y, int rx, int ry, int color);
- Arco circular: void arc(BITMAP *bmp, int x, y, fixed angl, ang2, int r, int color);

(Como hemos comentado antes, el primer parámetro "BITMAP *bmp" especifica dónde se dibujará cada una de esas figuras; para nosotros lo habitual por ahora será indicar "screen" -la pantalla-).

También podemos rellenar una cierta zona de la pantalla con

- `void floodfill(BITMAP *bmp, int x, int y, int color);`

O leer el color de un punto con

- `int getpixel(BITMAP *bmp, int x, int y);`

Para elegir **modo de pantalla** se usa la rutina que ya hemos visto:

```
int set_gfx_mode(int card, int w, int h, int v_w, int v_h);
```

El parámetro "card" (tarjeta) normalmente debería ser `GFX_AUTODETECT` (autodetección); w y h son la anchura (width) y altura (height) en puntos, y v_w y v_h son la anchura y altura de la pantalla "virtual", más grande que la visible, que podríamos utilizar para hacer algún tipo de scroll, y que nosotros no usaremos por ahora.

Pero... ¿y el número de colores? Se indica con

- `void set_color_depth(int depth);`

Donde "depth" es la "profundidad" de color en bits. El valor por defecto es 8 (8 bits = 256 colores), y otros valores posibles son 15, 16 (65.536 colores), 24 y 32 bits ("color verdadero").

Ahora vamos a ver un ejemplo algo más completo:

```

/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*  ipj02c.c                     */
/*                               */
/*  Segundo ejemplo:            */
/*  Figuras basicas en modo    */
/*  640x480 puntos, 64k color  */
/*                               */
/*  Comprobado con:            */
/*  - Dev-C++ 4.9.9.2          */
/*  y Allegro 4.2.1           */
/*-----*/

#include <allegro.h>

int main()
{
    allegro_init();
    install_keyboard();

    set_color_depth(16);
    if (set_gfx_mode(GFX_SAFE, 640, 480, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    line(screen, 20, 10, 310, 175, palette_color[15]);
    line(screen, 639, 0, 0, 479, palette_color[14]);

    rectfill(screen, 30, 30, 300, 200, palette_color[3]);

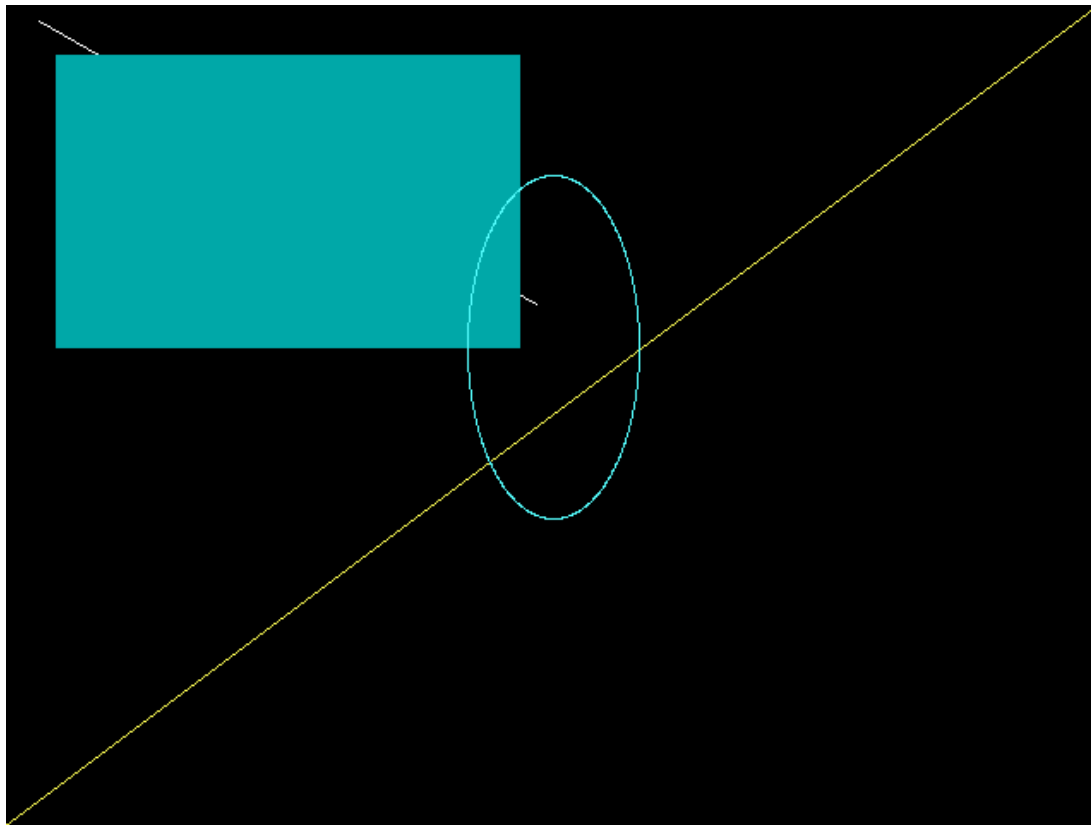
    ellipse (screen, 320, 200, 50, 100, palette_color[11]);

    readkey();
}

END_OF_MAIN();

```

El resultado de este programa sería algo como



Hay muchas más posibilidades, pero las iremos viendo según las vayamos necesitando...

2.3. Cómo hacerlo en el caso de Free Pascal.

En Free Pascal sería:

- Inicialización: Debemos incluir **uses graph;** al principio de nuestro fuente; si compilamos para Windows con Free Pascal y queremos acceder al teclado, deberíamos añadir **winCRT** a la línea de "uses" (si empleamos Turbo Pascal, en vez de "winCRT" deberíamos incluir "CRT")
- Entrar a modo gráfico: **initGraph(gd, gm, '');** En esta orden "gd" es el "driver gráfico" a usar. Para 256 colores, indicaríamos antes **gd := D8bit;** (un poco más adelante veremos los principales drivers disponibles); "gm" es el modo de pantalla (referido a la cantidad de puntos), que en este caso sería **gm := m320x200;** (o bien **m640x480** si no nos permite usar el modo de 320x200, algo frecuente bajo Windows en ordenadores modernos).
- Dibujar una línea entre dos puntos: **line(x1,y1,x2,y2)**, donde x1, y1 son las coordenadas horizontal y vertical del primer punto, x2 e y2 son las del segundo punto. Para indicar el color se la línea, usamos antes **setColor(color)**, donde "color" es un color de la paleta estándar (15 es el blanco).
- Salir del modo gráfico: **closeGraph;** justo antes del fin del programa. Si queremos esperar antes a que se pulse una tecla (que es lo habitual), usaremos **readkey;** (y en ese caso, sí deberá estar "winCRT" entre los "uses" de nuestro programa)

Vamos a verlo en la práctica:

```
(*-----*)
(*  Intro a la programac de  *)
(*  juegos, por Nacho Cabanes *)
(*                               *)
(*    IPJ01P.PAS                *)
(*                               *)
(*  Primer ejemplo: entra a    *)
(*  modo grafico y dibuja una  *)
(*  linea diagonal en la pan-  *)
(*  talla.                      *)
(*                               *)
(*  Comprobado con:           *)
(* - FreePascal 2.0.4 (WinXp) *)
(*-----*)
```

```

uses graph, wincrt;

var
  gd, gm, error : integer;

begin
  gd := D8bit;
  gm := m640x480;
  initgraph(gd, gm, '');

  error := graphResult;
  if error <> grOk then
    begin
      writeln('No se pudo entrar a modo grafico');
      writeln('Error encontrado: '+ graphErrorMsg(error) );
      halt(1);
    end;

  setColor(15);
  line (20,10, 310,175);

  readkey;
  closeGraph;
end.

```

(Recuerda que en el [apartado 1](#) tienes cómo teclearlo y probarlo).

Es posible que obtengamos el error "**Invalid graphics mode**". Sería porque nuestro ordenador no permite el modo gráfico que hemos elegido. Ante la duda, podemos hacer que lo detecte, usando `gd:=0; gm:=0;`

Igual que en el ejemplo en C, hay cosas que aparecen en este fuente y que no habíamos comentado aún:

- `graphResult` nos dice si el paso a modo gráfico ha sido correcto. Si devuelve un valor distinto de `grOk`, querrá decir que no se ha podido, y podremos ayudarnos de "`graphErrorMsg`" para ver qué ha fallado.
- `readkey`, de la unidad `crt` (o `wincrt`), espera a que pulsemos una tecla (en este caso lo usaremos antes de salir, para que tengamos tiempo de ver lo que aparece en pantalla).

Las **posibilidades más básicas** de la biblioteca gráfica de Free Pascal son:

- Dibujar un punto en un color:


```
putpixel(x, y, color);
```
- Cambiar el color de dibujo: `setColor(c);`
- Dibujar una línea en ese color: `line(x1, y1, x2, y2);`
- Recuadro: `rectangle(x1, y1, x2, y2);`
- Recuadro relleno (barra): `bar(x1, y1, x2, y2);`
- Elegir tipo de relleno y color de relleno: `setFillStyle(patrón, color);`
- Círculo: `circle(x, y, radio);`
- Elipse: `ellipse(x, y, anguloIni, anguloFin, radioX, radioY);`
- Elipse rellena: `fillEllipse(x, y, radioX, radioY);`
- Rellenar una cierta zona de la pantalla: `floodfill(x, y, colorBorde);`
- Leer el color de un punto: `getpixel(x, y);` (devuelve un valor de tipo `Word`)

Para elegir **modo de pantalla** se usa la rutina que ya hemos visto:

```
int initgraph( driver, modo, situacDrivers);
```

En su uso más sencillo, tenemos:

- "Driver" indica la cantidad de colores: D1bit para blanco y negro, D2bit para 4 colores, D4bit para 16 colores, D8bit para 256 colores, D16bit para 65536 colores. Existen otros modos que aún no están disponibles (24 bits, 32 bits), y alguno específico de ciertos ordenadores (como 4096 colores para Commodore Amiga).
- "Modo" indica la cantidad de puntos en pantalla. Los que más usaremos son m320x200, m640x480, m800x600, pero existen otros muchos, algunos de los cuales son específicos de ciertos ordenadores (Amiga, Atari, Mac).
- "SituacDeDrivers" se puede dejar en blanco (una cadena vacía, '') salvo si quisiéramos usar tipos de letra de los que Borland creó para Turbo Pascal, pero eso es algo que no haremos, al menos no por ahora.

Un ejemplo un poco más completo es:

```
(*-----*)
(* Intro a la programac de *)
(* juegos, por Nacho Cabanes *)
(* *)
(* IPJ02P.PAS *)
(* *)
(* Segundo ejemplo: *)
(* Figuras basicas en modo *)
(* 640x480 puntos,256 colores *)
(* *)
(* Comprobado con: *)
(* - FreePascal 2.0.4 (WinXp) *)
(*-----*)
uses wincrt, graph;

var
gd,gm, error : integer;

begin
  gd := D8bit;
  gm := m640x480;
  initgraph(gd, gm, '');

  error := graphResult;
  if error <> grOk then
    begin
      writeln('No se pudo entrar a modo grafico');
      writeln('Error encontrado: '+ graphErrorMsg(error));
      halt(1);
    end;

  setColor(15);
  line (20,10,310,175);

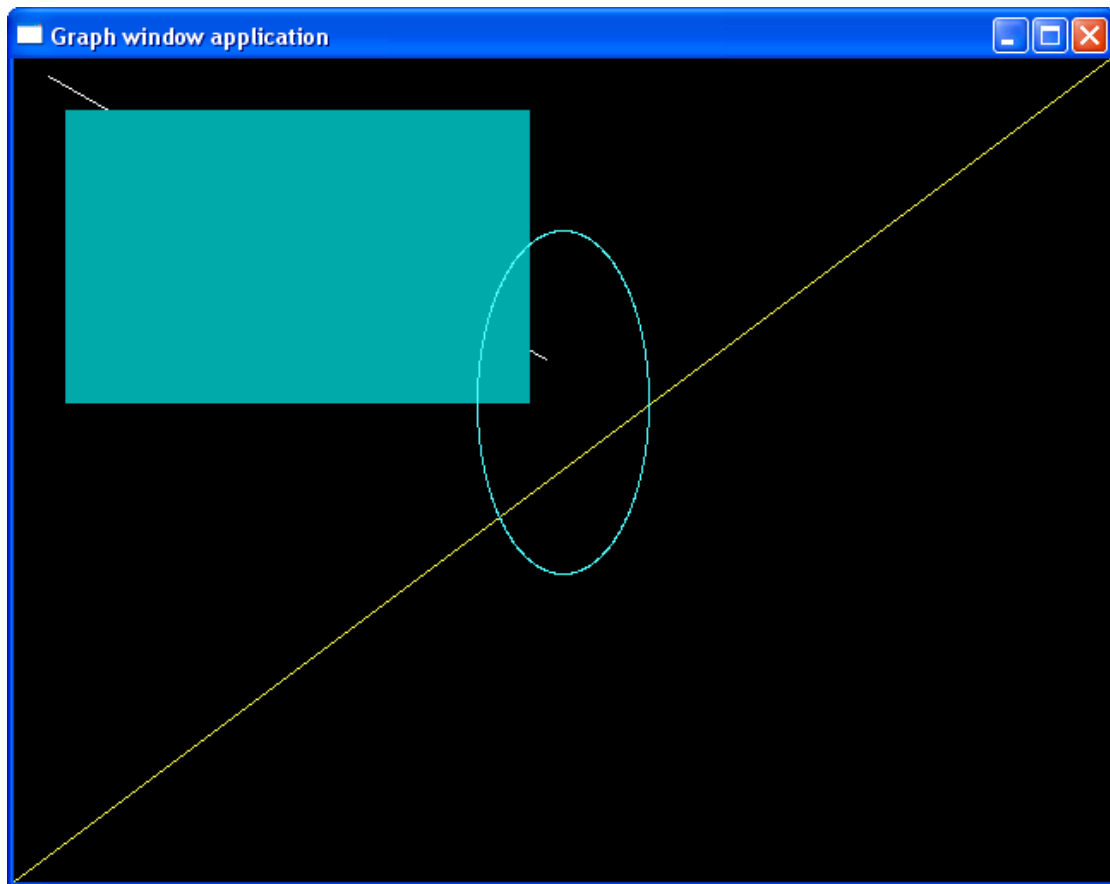
  setColor(14);
  line (639,0,0, 479);

  setFillStyle( SolidFill,3);
  bar(30,30,300,200);

  setColor(11);
  ellipse (320,200,0,360,50,100);

  readkey;
  closeGraph;
end.
```

El resultado de este programa, si compilamos para Windows, es el siguiente:



2.4. Cómo hacerlo en el caso de Java.

En el lenguaje Java las cosas cambian ligeramente. La sintaxis recuerda mucho a la de C, pero muchas de las ideas base son distintas:

- Nuestros programas creados en Java podrán funcionar en cualquier equipo para el que exista una "máquina virtual Java", lo que supone que funcionarán en más de un equipo/sistema operativo distinto sin problemas y sin cambios.
- Por el otro lado, la existencia de esta "capa intermedia" hace que nuestros programas puedan funcionar algo más lento si están creados en Java que si es en C++ o en Pascal. No es grave, con la velocidad que tienen los ordenadores actuales, y menos si nuestros juegos no son excesivamente complejos.
- Además, en Java existe la posibilidad de crear aplicaciones capaces de funcionar "por sí mismas", o bien otras pequeñas aplicaciones que funcionan dentro de una página Web, que son los llamados "Applets". También existen versiones "limitadas" del lenguaje Java para pequeños dispositivos como ciertos teléfonos móviles o PDAs. Nosotros empezaremos por crear "Applets" y más adelante veremos qué cambia si queremos crear aplicaciones completas y si queremos hacer algo para nuestro teléfono móvil.

El ejemplo de cómo cambiar a modo gráfico y dibujar una diagonal en un Applet sería:

```

/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*  ipj01j.java                 */
/*                               */
/*  Primer ejemplo: entra a     */
/*  modo grafico y dibuja una  */
/*  linea diagonal en la pan-  */
/*  talla.                      */
/*                               */
/*  Comprobado con:            */
/*  - JDK 1.5.0                */
/*-----*/

```

```
import java.awt.*;
```

```
public class ipj01j extends java.applet.Applet {

    public void paint(Graphics g) {
        g.setColor( Color.yellow );
        g.drawLine( 20, 10, 310, 175 );
    }

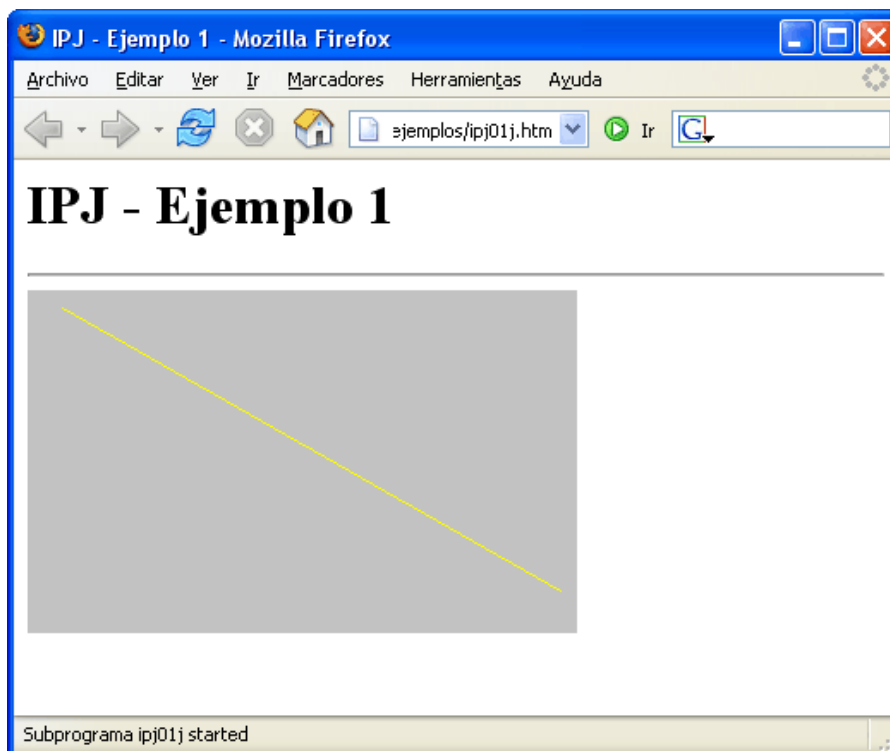
}
```

(Recuerda que en el [apartado 1](#) tienes cómo teclearlo y probarlo).

Como es un Applet, está diseñado para ser usado desde una página Web. Por tanto, tendremos que crear esa página Web. Bastaría con teclear lo siguiente desde cualquier editor de texto:

```
<html>
<head>
  <title>IPJ - Ejemplo 1</title>
</head>
<body>
  <h1>IPJ - Ejemplo 1</h1>
  <hr>
  <applet code=ipj01j.class width=320 height=200>
    alt=El navegador no esta mostrando el APPLET
  </applet>
</body>
</html>
```

Guardamos este fichero y hacemos doble clic para probarlo desde nuestro navegador. Si nuestro navegador no reconoce el lenguaje Java, veríamos el aviso "El navegador no está mostrando el APPLET", pero si todo ha ido bien, debería aparecer algo como



Las **posibilidades más básicas** de las rutinas gráficas de Java son muy similares a las que hemos visto con Allegro y con

Free Pascal, con alguna diferencia. Por ejemplo, en los rectángulos no se indican las coordenadas de las dos esquinas, sino una esquina, la anchura y la altura. De igual modo, en las elipses no se indican el centro y los dos radios, sino como el rectángulo que las rodea. Detallando un poco más:

```
// Escribir texto: mensaje, coordenada x (horizontal) e y (vertical)
g.drawString( "Hola Mundo!", 100, 50 );

// Color: existen nombres predefinidos
g.setColor( Color.black );

// Linea: Coordenadas x e y de los extremos
g.drawLine( x1, y1, x2, y2 );

// Rectangulo: Origen, anchura y altura
g.drawRect( x1, y1, ancho, alto );

// Rectangulo relleno: identico
g.fillRect( x1, y1, ancho, alto );

// Rectangulo redondeado: similar + dos redondeos
g.drawRoundRect( x1, y1, x2, y2, rnd1, rnd2 );

// Rectangulo redondeado relleno: igual
g.fillRoundRect( x1, y1, x2, y2, rnd1, rnd2 );

// Ovalo (elipse): Como rectangulo que lo rodea
g.drawOval( x1, y1, ancho, alto );

// Ovalo relleno: identico
g.fillOval( x1, y1, ancho, alto );
```

```
/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* ipj02j.java */
/* */
/* Segundo ejemplo: */
/* Figuras basicas en modo */
/* 640x480 puntos */
/* */
/* Comprobado con: */
/* - JDK 1.4.2_01 */
/*-----*/

import java.awt.*;

public class ipj02j extends java.applet.Applet {

    public void paint(Graphics g) {
        // Primero borro el fondo en negro
        g.setColor( Color.black );
        g.fillRect( 0, 0, 639, 479 );
        // Y ahora dibujo las figuras del ejemplo
        g.setColor( Color.white );
        g.drawLine( 20, 10, 310, 175 );
        g.setColor( Color.yellow );
        g.drawLine( 639, 0, 0, 479 );
        g.setColor( Color.blue );
        g.fillRect( 30, 30, 270, 170 );
        g.setColor( Color.cyan );
        g.drawOval( 270, 100, 100, 200 );
    }
}
```

La página Web encargada de mostrar este Applet no tendría grandes cambios si comparamos con la anterior. Poco más que el nombre de la clase, los "cartelitos" y el tamaño:

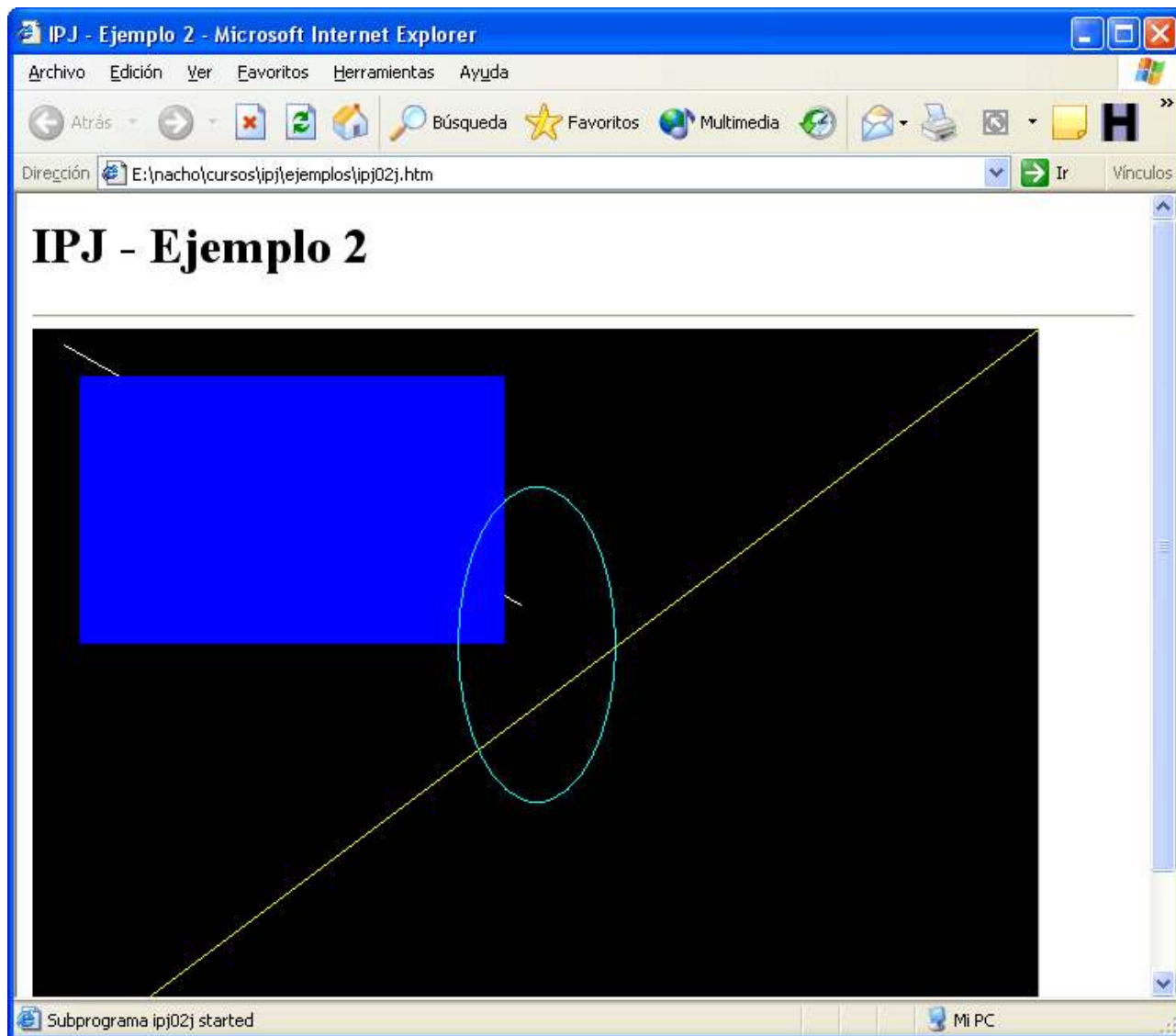
```
<html>
<head>
```

```

<title>IPJ - Ejemplo 2</title>
</head>
<body>
  <h1>IPJ - Ejemplo 2</h1>
  <hr>
  <applet code=ipj02j.class width=640 height=480>
    alt=El navegador no esta mostrando el APPLET
  </applet>
</body>
</html>

```

Y el resultado sería este:



Hay muchas más posibilidades, pero las iremos viendo según las vayamos necesitando...

3. Leyendo del teclado y escribiendo texto.

Contenido de este apartado:

- [Pautas generales.](#)
- [Cómo hacerlo en el caso del lenguaje C y la biblioteca Allegro.](#)
- [Cómo hacerlo en el caso de Free Pascal.](#)
- [Cómo hacerlo en el caso de Java.](#)

3.1. Pautas generales.

Escribir una frase suele ser sencillo. Tendremos una orden capaz de escribir un cierto texto en unas ciertas coordenadas. A veces no será inmediato escribir algo que no sea un texto puro, como es el caso del resultado de una operación matemática, pero este problema es fácil de solucionar. Lo que suele ser algo más difícil es escribir con tipos de letras espectaculares: a veces (pocas) estaremos limitados al tipo de letra de la BIOS de la tarjeta gráfica, y otras muchas estaremos limitados (menos) a los tipos de letra de nuestro sistema operativo o que incorpora nuestro compilador. Más adelante veremos una forma de crear nuestros propios tipos de letra y usarlos en nuestros juegos, a cambio de perder la comodidad de usar funciones "prefabricadas" como las que veremos en este apartado y usaremos en nuestros primeros juegos.

Lo de leer el teclado no suele ser difícil: es habitual tener una función que comprueba si se ha pulsado una tecla o no (para no tener que esperar) y otra que nos dice qué tecla se ha pulsado. Incluso es habitual que tengamos definidas ciertas constantes, para no tener que memorizar cual es el número asociado a una cierta tecla, o el código ASCII que devuelve. En el caso de Java, es algo más incómodo que en C o Pascal, pero tampoco es especialmente complicado.

Vamos a verlo...

3.2 Texto y teclado Con C y Allegro.

A la hora de escribir texto, casi cualquier biblioteca gráfica tendrá disponible una función que nos permita escribir un cierto mensaje en unas ciertas coordenadas. En el caso de Allegro, dicha función es:

- `void textout(BITMAP *bmp, const FONT *f, const char *s, int x, y, int color);`

Debería ser fácil de entender: el primer parámetro es dónde escribiremos el texto (en principio, usaremos "screen", la pantalla), con qué fuente (usaremos una llamada "font", que es la fuente hardware del 8x8 puntos habitual de la pantalla de 320x200 puntos). Finalmente indicamos el texto a escribir, las coordenadas x e y, y terminamos con el color.

Además, tenemos dos variantes, por si queremos que el texto quede centrado en torno a esa posición x (la orden anterior indica la posición de comienzo), o que termine en esa posición:

- `void textout_centre(BITMAP *bmp, const FONT *f, const char *s, int x, y, color);`
- `void textout_right(BITMAP *bmp, const FONT *f, const char *s, int x, y, color);`

Si se trata de un texto con parámetros como los que mostramos con "printf", en la mayoría de bibliotecas gráficas no podríamos escribirlo directamente, deberíamos convertirlo primero a texto, con alguna orden como "sprintf". En Allegro sí tenemos la posibilidad de escribirlo directamente con:

- `void textprintf(BITMAP *bmp, const FONT *f, int x, y, color, const char *fmt, ...);`

Y también tenemos dos órdenes equivalentes pero que ajustan el texto al centro o al lado derecho:

- `void textprintf_centre(BITMAP *bmp, const FONT *f, int x, y, color, const char *fmt, ...);`
- `void textprintf_right(BITMAP *bmp, const FONT *f, int x, y, color, const char *fmt, ...);`

Un ejemplo del uso de estas órdenes sería:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* ipj03c.c */
/* */
/* Tercer ejemplo: escribir */
/* texto en modo grafico */
/* */
/* Comprobado con: */
/* - Dev-C++ 4.9.9.2 */
/* y Allegro 4.2.1 */
/*-----*/

```



```

#include <allegro.h>

int main()
{
    allegro_init();
    install_keyboard();

    if (set_gfx_mode(GFX_SAFE, 320, 200, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    textout(screen, font, "Ejemplo de texto", 160, 80,
        palette_color[15]);
    textout_centre(screen, font, "Ejemplo de texto", 160, 100,
        palette_color[14]);
    textout_right(screen, font, "Ejemplo de texto", 160, 120,
        palette_color[13]);

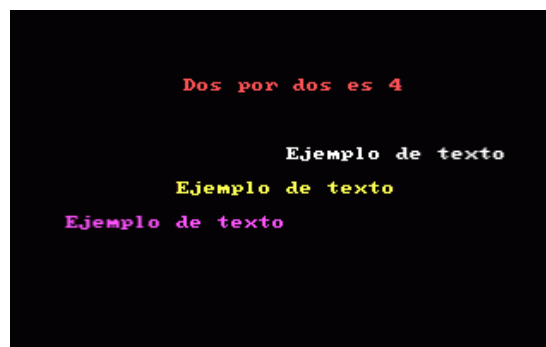
    textprintf(screen, font, 100, 40, palette_color[12],
        "Dos por dos es %d", 2*2);

    readkey();
}

END_OF_MAIN();

```

que tendría como resultado:



Otra orden relacionada con el texto y que puede resultar interesante es:

- `int text_mode(int mode);`

Si el parámetro "mode" es 0 o positivo, el texto será opaco (se borra el fondo), y el color de fondo del texto es el que indique "mode". Si es negativo, el texto será transparente (en sus huecos se verá el fondo, que es lo habitual en otras bibliotecas gráficas).

Si no se indica nada, se asume que el modo de escritura del texto es 0 (el texto es opaco, con fondo negro).

En cuanto a la **introducción de texto**, en la mayoría de bibliotecas gráficas, no tendremos nada parecido a "scanf" ni a "gets" que nos permita teclear un texto entero, y menos todavía corregirlo con las flechas del cursor, retroceso, etc.

Normalmente deberemos leer letra a letra (con funciones como "getch" o alguna equivalente), y filtrarlo nosotros mismos. Allegro sí tiene un pequeño GUI (Interfaz gráfico de usuario), que nos permitirá abreviar ciertas tareas como la entrada de texto o la pulsación de botones. Pero estas posibilidades las veremos más adelante.

Con Allegro tenemos una función para esperar a que se pulse una tecla y comprobar qué tecla se ha pulsado:

- `int readkey();`

Se puede utilizar comprobando el código ASCII (ver qué "letra" corresponde):

```
if((readkey() & 0xff) == 'n') printf("Has pulsado n");
```

o bien comprobando el código de la tecla (el "scancode")

```
if((readkey() >> 8) == KEY_SPACE) printf("Has pulsado Espacio");
```

Esta forma de comprobarlo se debe a que nos devuelve un valor de 2 bytes. El byte bajo contiene el código ASCII (la letra), mientras que el byte alto contiene el código de la tecla.

Si se pulsa Ctrl o Alt a la vez que una cierta tecla, el código (scancode) de la tecla sigue siendo el mismo, pero no es código ASCII. Por ejemplo, mayúsculas + 'a' devolvería 'A', ctrl + 'a' devolvería 1 (y ctrl + 'b' = 2, ctrl + 'c' = 3 y así sucesivamente) y alt + cualquier tecla devuelve 0 (también devuelven 0 las teclas de función F1 a F12 y alguna otra).

Por eso, en los juegos normalmente usaremos el "scancode" de las teclas, que no varía. Eso sí, estos números no son fáciles de recordar, por lo que tenemos unas constantes preparadas, como la KEY_SPACE del ejemplo anterior:

```
KEY_A ... KEY_Z,
KEY_0 ... KEY_9,
KEY_0_PAD ... KEY_9_PAD,
KEY_F1 ... KEY_F12,

KEY_ESC, KEY_TILDE, KEY_MINUS, KEY_EQUALS,
KEY_BACKSPACE, KEY_TAB, KEY_OPENBRACE, KEY_CLOSEBRACE,
KEY_ENTER, KEY_COLON, KEY_QUOTE, KEY_BACKSLASH,
KEY_BACKSLASH2, KEY_COMMA, KEY_STOP, KEY_SLASH,
KEY_SPACE,

KEY_INSERT, KEY_DEL, KEY_HOME, KEY_END, KEY_PGUP,
KEY_PGDN, KEY_LEFT, KEY_RIGHT, KEY_UP, KEY_DOWN,

KEY_SLASH_PAD, KEY_ASTERISK, KEY_MINUS_PAD,
KEY_PLUS_PAD, KEY_DEL_PAD, KEY_ENTER_PAD,

KEY_PRTSCR, KEY_PAUSE,

KEY_LSHIFT, KEY_RSHIFT,
KEY_LCONTROL, KEY_RCONTROL,
KEY_ALT, KEY_ALTGR,
KEY_LWIN, KEY_RWIN, KEY_MENU,
KEY_SCRLOCK, KEY_NUMLOCK, KEY_CAPSLOCK
```

Finalmente, podemos comprobar **si se ha pulsado una tecla**, pero sin quedarnos esperando a que esto ocurra. No lo usaremos en nuestro primer juego, pero aun así vamos a comentarlo. Se haría con:

- `int keypressed();`

que devuelve TRUE si se ha pulsado alguna tecla (después comprobaríamos con "readkey" de qué tecla se trata, y en ese momento, "keypressed" volvería a valer FALSE).

Existe otra forma de comprobar el teclado, que nos permitiría saber si se han pulsado **varias teclas a la vez**, pero eso es algo que no necesitamos aún, y que dejamos para un poco más adelante...

3.3. Teclado y texto con Pascal.

La orden básica para **escribir** texto en la pantalla gráfica con Free Pascal es:

- OutTextXY(x, y, texto);

También existe una variante a la que no se le indica en qué posición queremos escribir, y entonces lo hará a continuación del último texto escrito. Es "OutText(texto)"

Si queremos que el texto esté **alineado** a la derecha o al centro, lo haríamos con "SetTextJustify(horizontal, vertical)", donde "horizontal" puede ser LeftText (izquierda), CenterText (centro) o RightText (derecha) y la alineación vertical puede ser BottomText (texto bajo el puntero), CenterText (centrado) o TopText (sobre el puntero).

Existen otras posibilidades, que no usaremos, al menos por ahora, porque no existen como tal en Allegro y lo haremos de la misma forma que con dicha librería (artesanalmente). Es el caso de emplear distintos tipos de letra, en distintos tamaños o escribir con otras orientaciones (por ejemplo, de arriba a abajo).

Lo que no tenemos en FreePascal es ninguna orden equivalente a "printf" para pantalla gráfica. Sólo podemos escribir cadenas de texto, de modo que si queremos escribir números o expresiones complejas, deberemos convertirlas primero a cadenas de texto.

En cuanto a la **introducción de texto**, en la unidad crt tenemos las funciones equivalentes a las dos principales de Allegro: "keypressed" devuelve "false" mientras no se haya pulsado ninguna tecla, y "true" cuando se ha pulsado. Entonces se puede leer la tecla con "readkey". No tenemos garantías de que funciones como ReadLn vayan a trabajar correctamente en modo gráfico, así que en general tendremos que ser nosotros quienes creamos rutinas fiables si queremos que se pueda teclear un texto entero, y más aún si queremos corregirlo con las flechas del cursor, retroceso, etc.

Eso sí, no tenemos constantes predefinidas con las que comprobar si se ha pulsado una cierta tecla. Para las teclas alfanuméricas no es problema, podemos hacer cosas como

```
tecla := readkey;
if tecla = 'E' then ...
```

Pero para las teclas de función, readkey devuelve 0, y debemos volver a leer su valor para obtener el número de tecla pulsada. Es fácil crear un programa que nos diga el número de cada tecla pulsada, y crearnos nuestras propias constantes simbólicas, así que no profundizamos más por ahora. Si en alguno de nuestros juegos usamos varias teclas, indicaremos entonces los códigos de cada una de ellas.

Un ejemplo similar al anterior en C sería este:

```
(*-----*)
(*  Intro a la programac de  *)
(*  juegos, por Nacho Cabanes *)
(*                          *)
(*    IPJ03P.PAS           *)
(*                          *)
(*  Tercer ejemplo: escribir *)
(*  texto en modo grafico   *)
(*                          *)
(*  Comprobado con:        *)
(* - FreePascal 2.0.4 (WinXp) *)
(*-----*)

uses graph, wincrt, sysutils;

var
  gd,gm, error : integer;

BEGIN
  gd := D8bit;
  gm := m800x600;
  initgraph(gd, gm, '');

  error := graphResult;
  if error <> grOk then
    begin
      writeln('No se pudo entrar a modo grafico');
      writeln('Error encontrado: '+ graphErrorMsg(error) );
      halt(1);
    end;
```

```

setColor(15);
outTextXY(160,80, 'Ejemplo de texto');

setTextJustify(CenterText,CenterText);
setColor(14);
outTextXY(160,100, 'Ejemplo de texto');

setTextJustify(RightText,CenterText);
setColor(13);
outTextXY(160,120, 'Ejemplo de texto');

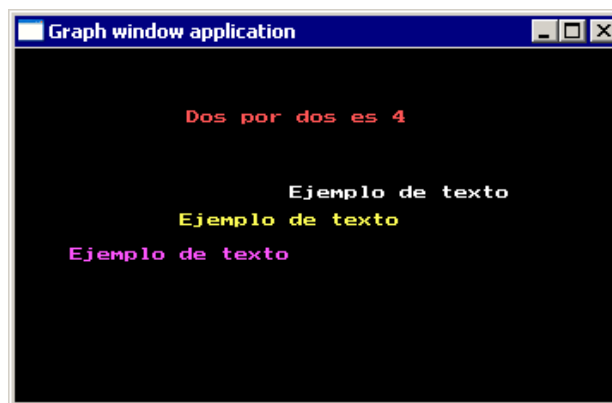
setTextJustify(LeftText,CenterText);
setColor(12);
outTextXY(100,40, 'Dos por dos es '+intToStr(2*2));

readkey;
closeGraph;
END.

```

(En este ejemplo también he incluido la unidad "SysUtils", para poder convertir de una forma sencilla un número en cadena usando "IntToStr", ya que "outTextXY" necesita una cadena como parámetro).

Y su resultado sería



3.4. Teclado y texto con Java.

Escribir en un Applet de Java no es difícil. La orden básica es "**drawString**":

- drawString(String texto, int x, y);

Si usamos esta orden, no es tan sencillo como en Allegro el ajustar el texto a un lado de esas coordenadas, al otro o centrarlo. Existen otras órdenes que nos permitirán hacerlo con facilidad, pero las veremos más adelante.

Lo que sí es algo más complicado en Java es eso de **leer del teclado**. Además es distinto en modo texto (consola) o en modo gráfico (como nuestro Applet). La forma más sencilla (en mi opinión) de hacerlo dentro de un Applet es implementando un "**KeyListener**", que nos da la estructura básica, a cambio de obligarnos a detallar las funciones **keyPressed** (que se pondrá en marcha en el momento de apretar una tecla), **keyReleased** (al dejar de apretar la tecla) y **keyTyped** (que se activa cuando el carácter correspondiente aparecería en pantalla). Las tres funciones admiten un parámetro de tipo KeyEvent, que tiene métodos como "**getChar**" que dice la letra que corresponde a esa tecla pulsada o "**getKeyCode**" que nos indica el código de tecla. La letra pulsada es fácil de comprobar porque tenemos las letras impresas en el teclado, sabemos qué letra corresponde a cada tecla. Pero el código es menos intuitivo, así que tenemos una serie de constantes que nos ayuden a recordarlos:

- VK_A a VK_Z para las teclas alfabéticas.
- VK_0 a VK_9 para las numéricas y VK_NUMPAD0 a VK_NUMPAD9 para el teclado numérico adicional.
- VK_F1 en adelante para las teclas de función.
- VK_LEFT, VK_RIGHT, VK_DOWN, VK_UP para las flechas del teclado.
- VK_SPACE, VK_ENTER, VK_ESCAPE, VK_HOME, VK_END y similares para las demás teclas especiales.

De modo que en la práctica lo usaríamos haciendo cosas como

```
public void keyPressed(KeyEvent e) {
    if (e.getKeyChar() == 'a' ) ...
```

o como

```
public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == VK_ESCAPE ) ...
```

Solo tres comentarios más:

- En el método "init" (que se encarga de inicializar nuestro Applet antes de empezar a dibujar) deberíamos añadir "addKeyListener(this);"
- Sólo se leerán pulsaciones del "componente" actual. Esto quiere decir que si nuestro Applet tuviera varios componentes (cosas que no hemos visto, como cuadros de texto, listas desplegadas, etc), nos podría interesar indicar cual queremos decir que sea el activo con "requestFocus()", que en el ejemplo aparece dentro de un comentario por no ser necesario.
- Por eso mismo, debemos hacer clic con el ratón en nuestro Applet antes de empezar a usar el teclado, o no nos hará caso, si no está "activo".

Todo listo. Vamos a ver un ejemplo sencillo del uso de estas órdenes:

```
/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*  ipj03j.java                 */
/*                               */
/*  Tercer ejemplo: escribir  */
/*  en colores, leer del      */
/*  teclado                    */
/*                               */
/*  Comprobado con:          */
/*  - JDK 1.4.2_01          */
/*-----*/

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ipj03j extends Applet
    implements KeyListener
{
    char letra;

    public void paint(Graphics g) {

        g.setColor(Color.green);
        g.drawString("Ejemplo de texto 1",
            140, 80);

        g.setColor(Color.blue);
        g.drawString("Ejemplo de texto 2",
            160, 100);

        g.setColor(new Color(0, 128, 128));
        g.drawString("Ejemplo de texto 3",
            180, 120);

        if (letra == 'a')
            g.drawString("Has pulsado A",
                60, 60);
    }
}
```

```

public void init() {
    //requestFocus();
    addKeyListener(this);
}

public void keyTyped(KeyEvent e) {
}

public void keyReleased(KeyEvent e) {
}

public void keyPressed(KeyEvent e) {
    letra = e.getKeyChar();
    repaint();
}
}
}

```

que tendría como resultado:



4. Cómo generar números al azar. Un primer juego: adivinar números.

Contenido de este apartado:

- [Pautas generales.](#)
- [Cómo hacerlo en el caso del lenguaje C y la biblioteca Allegro.](#)
- [Cómo hacerlo con Free Pascal.](#)
- [Cómo hacerlo en el caso de Java.](#)

4.1. Pautas generales.

Un requisito fundamental en la mayoría de los juegos es que no sea siempre igual, para que no sea predecible. Si hay cosas al

azar, no bastará con que el jugador memorice, sino que tendrá que enfrentarse a retos que no serán siempre los mismos.

Por eso, vamos a ver cómo generar números al azar. A partir de esos números, haremos el que posiblemente es el juego más sencillo posible: adivinar un número oculto, teniendo una cantidad limitada de intentos.

En cuanto a obtener números aleatorios al azar, suele haber ciertas cosas comunes en casi cualquier lenguaje de programación: las funciones que nos dan ese número se llamarán "rand" o algo muy parecido, porque "aleatorio" en inglés es "random". Por otra parte, en muchos lenguajes es necesario decirle una "semilla" a partir de la que empezar a generar los números. Si la semilla fuera siempre la misma, los números obtenidos, por lo que se suele usar como semilla el reloj interno del ordenador, porque sería casi imposible que dos partidas comenzaran exactamente a la misma hora (con una precisión de centésimas de segundo o más). Para complicar la situación, en algunos lenguajes la función "rand" (o como se llame) nos da un número entre 0 y 1, que nosotros deberíamos multiplicar si queremos números más grandes (eso pasa en Pascal y Java); en otros lenguajes obtenemos un número muy grande, entre 0 y otro valor (a veces cerca de 32.000, otras cerca de 2.000 millones, según la biblioteca que usemos), como es el caso de C, así que para obtener un número más pequeño lo que haremos es dividir y quedarnos con el resto de esa división.

Así, en nuestro caso, en la versión de C del juego haremos (semilla a partir del reloj, y un número enorme, del que tomamos el resto de la división):

```
srand(time(0));
numeroAdivinar = rand() % MAXIMONUMERO;
```

y en Pascal algo como (primero la semilla y luego un número entre 0 y 1, que multiplicamos)

```
randomize;
numeroAdivinar := round(random * MAXIMONUMERO);
```

y en Java (similar a Pascal, pero sin necesidad de pedir que se cree la semilla):

```
numeroAdivinar = (int) Math.round(Math.random() * MAXIMONUMERO);
```

Por lo que respecta a nuestro juego, la idea de lo que tiene que hacer (lo que se suele llamar "pseudocódigo") podría ser algo así:

```
Generar un número al azar entre 0 y 99
acertado = FALSO
repetir
  pedir número al usuario
  si el número tecleado el número al azar, terminado = VERDADERO
    en caso contrario, si el número tecleado es más pequeño, avisar
    en caso contrario, si el número tecleado es mayor, avisar
  incrementar Numero de intentos
hasta que (acertado) o (Numero de intentos = maximo)
si acertado, felicitar
  en caso contrario, decir qué número era el correcto
```

Pasar de aquí a la práctica no debería ser difícil, salvo quizá por el hecho de "pedir número al usuario". Como ya habíamos comentado, en general no tendremos en modo gráfico rutinas que permitan leer entradas complejas por teclado. Pero como esa es la única complicación de este juego, lo podemos evitar de una forma sencilla: obligaremos a que el usuario tenga que teclear un número de 2 cifras entre 00 y 99, de modo que nos bastará con dos órdenes de comprobar una pulsación de teclado (readkey, getch o similar).

Pasar de las dos letras "1" y "4" al número 14 tampoco es difícil. Algunos lenguajes nos permitirán "juntar" (concatenar, hablando más correctamente) las dos letras para obtener "14" y luego decir que calcule el valor numérico de este texto. Otra opción es restar la letra "0" a cada cifra, con lo cual ya tenemos el valor numérico de cada una, es decir 1 y 4; para obtener el 14 basta con multiplicar el primer número por 10 y sumarle el segundo. Este es el método que usaremos en C y en Pascal en este ejemplo, para no necesitar recurrir a bibliotecas externas de funciones para convertir los valores.

(Un consejo: **intenta hacer cada juego**
tú mismo antes de ver cómo lo he resuelto yo.

El enfrentarte con los problemas y comparar con
otras soluciones hará que aprendas mucho
más que si te limitas a observar)

4.2 adivinar Números Con C y Allegro.

Va a ser muy poco más que seguir el pseudocódigo de antes, pero adaptado a la sintaxis del lenguaje C y dando alguna vuelta puntual en casos como el de leer lo que teclaa el usuario, que ya hemos comentado:

```

/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*   ipj04c.c                   */
/*                               */
/*  Cuarto ejemplo: adivinar  */
/*  un numero - modo grafico  */
/*                               */
/*  Comprobado con:          */
/*  - Dev-C++ 4.9.9.2        */
/*  y Allegro 4.2.1         */
/*-----*/

#include <allegro.h>

int numeroAdivinar, numeroTecleado;
char tecla1, tecla2;
int acertado;
int intentos;
int lineaEscritura;

#define MAXIMONUMERO  99
#define NUMEROINTENTOS  6

int main()
{
    allegro_init();
    install_keyboard();

    if (set_gfx_mode(GFX_SAFE, 320, 200, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    intentos = 0;
    lineaEscritura = 50;

    srand(time(0));
    numeroAdivinar = rand() % MAXIMONUMERO;
    acertado = 0;

    textout(screen, font, "Adivinar numeros", 10, 10,
    palette_color[14]);

    do {

        textout(screen, font, "Teclea dos cifras (00 a 99)", 15, lineaEscritura,
        palette_color[13]);

        tecla1 = readkey();
        textprintf(screen, font, 235, lineaEscritura,
        palette_color[13], "%c", tecla1);

        tecla2 = readkey();
        textprintf(screen, font, 243, lineaEscritura,
        palette_color[13], "%c", tecla2);
    }
}

```



```

numeroTecleado = (int) (tecla1 - '0') * 10
                + tecla2 - '0';

if (numeroTecleado == numeroAdivinar) acertado = 1;
else if (numeroTecleado < numeroAdivinar)
textout(screen, font, "Corto", 260, lineaEscritura,
palette_color[12]);
else if (numeroTecleado > numeroAdivinar)
textout(screen, font, "Grande", 260, lineaEscritura,
palette_color[12]);

intentos++;
lineaEscritura += 10;

} while( (!acertado) && (intentos < NUMEROINTENTOS));

if (acertado)
textout(screen, font, "Acertaste!!!", 160, 180,
palette_color[15]);
else
textprintf(screen, font, 160, 180,
palette_color[15], "Era: %d", numeroAdivinar);

readkey();
}

END_OF_MAIN();

```

que se vería así:

```

Adivinar numeros

Teclea dos cifras (00 a 99) 45 Corto
Teclea dos cifras (00 a 99) 78 Grande
Teclea dos cifras (00 a 99) 61 Grande
Teclea dos cifras (00 a 99) 54 Corto
Teclea dos cifras (00 a 99) 57 Corto
Teclea dos cifras (00 a 99) 58 Corto

Era: 60

```

4.3. Adivinar números en Pascal.

El juego en Pascal es casi idéntico al de C (salvando la diferencia de sintaxis, claro):

```

(*-----*)
(* Intro a la programac de *)
(* juegos, por Nacho Cabanes *)
(* *)
(* IPJ04P.PAS *)
(* *)
(* Cuarto ejemplo: adivinar *)
(* un numero - modo grafico *)
(* *)
(* Comprobado con: *)
(* - FreePascal 2.0.4 (WinXp) *)
(*-----*)

uses wincrt, graph, sysUtils;
(* Cambiar por "uses crt, ..." bajo Dos *)

var
gd, gm, error : integer;
numeroAdivinar, numeroTecleado: integer;

```

```

tecla1, tecla2: char;
acertado: boolean;
intentos: integer;
lineaEscritura: integer;

const
  MAXIMONUMERO = 99;
  NUMEROINTENTOS = 5;

begin
  gd := D8BIT;
  gm := m640x480;
  (* Bajo DOS bastaria con m320x200 *)
  initgraph(gd, gm, '');

  error := graphResult;
  if error <> grOk then
    begin
      writeln('No se pudo entrar a modo grafico');
      writeln('Error encontrado: '+ graphErrorMsg(error) );
      halt(1);
    end;

  intentos := 0;
  lineaEscritura := 50;
  randomize;
  numeroAdivinar := round(random * MAXIMONUMERO);
  acertado := false;

  setColor(14);
  outTextXY(10,10, 'Adivinar numeros');

  repeat

    outTextXY(15,lineaEscritura,'Teclea dos cifras (00 a 99)');
    tecla1 := readkey;
    outTextXY(235,lineaEscritura,tecla1);
    tecla2 := readkey;
    outTextXY(243,lineaEscritura,tecla2);
    numeroTecleado := (ord(tecla1)-ord('0')) * 10
      +ord(tecla2)-ord('0');

    if numeroTecleado = numeroAdivinar then acertado := TRUE
    else if numeroTecleado < numeroAdivinar then
      outTextXY(260, lineaEscritura, 'Corto')
    else if numeroTecleado > numeroAdivinar then
      outTextXY(260, lineaEscritura, 'Grande');

    intentos := intentos + 1;
    lineaEscritura := lineaEscritura + 10;

  until acertado or (intentos = NUMEROINTENTOS);

  setColor(15);
  if acertado then
    outTextXY(160,180, 'Acertaste!!!')
  else
    outTextXY(160,180, 'Era: '+intToStr(numeroAdivinar));

  readkey;
  closeGraph;
end.

```

El modo de 320x200 puntos no está disponible en muchos casos si programamos para Windows, por lo que hemos usado un modo de 640x480 puntos, aunque la información en pantalla quede un poco más descentrada.

Si se compila para **MsDos**, sí se podría usar el modo 320x200. Para este sistema operativo, la línea "uses crt, wingraph;" se debería cambiar por "uses crt, graph;"

4.4. Adivinar números en Java.

Hay dos diferencias importantes con la versión en C

- La forma de acceder al teclado es algo más incómoda (por ahora, en ciertos casos nos resultará más práctica esta forma de trabajar), como ya hemos visto
- Las "cosas" dentro de programa están ordenadas de forma un poco especial: por la forma de funcionar los Applets, las rutinas de inicialización aparecen en el método "init". De igual modo, las rutinas de escritura en pantalla aparecen en el método "paint". Finalmente, las operaciones básicas que hay que realizar cuando se pulsa una tecla las he incluido en una de las rutinas de comprobación de teclado (en "keyPressed").

El resto debería ser fácil de seguir:

```

/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*  ipj04j.java                 */
/*                               */
/*  Cuarto ejemplo: adivinar  */
/*  un numero - modo grafico */
/*                               */
/*  Comprobado con:           */
/*  - JDK 1.5.0               */
/*-----*/

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ipj04j extends Applet
    implements KeyListener
{

    int numeroAdivinar, numeroTeclado;
    char tecla1='a', tecla2='a';
    boolean terminado=false;
    int intentos=0;
    int lineaEscritura=50;
    int i;

    final int MAXIMONUMERO = 99;
    final int NUMEROINTENTOS = 6;
    int respuestas[] = {-1,-1,-1,-1,-1,-1};

    char letra;

    public void paint(Graphics g) {

        // Cartel "principal"
        g.setColor(Color.red);
        g.drawString("Adivinar numeros",10,10);

        // Escribo "n" veces lo que ya ha teclado
        for (i=0; i<=intentos; i++) {
            g.setColor(Color.green);
            g.drawString("Teclea dos cifras (00 a 99)",
                15,lineaEscritura+i*10);
            g.setColor(Color.blue);
            // Solo escribo la respuesta si realmente la hay
            if (respuestas[i] != -1){
                g.drawString(""+respuestas[i],
                    235, lineaEscritura+i*10);
                // Y, de paso, compruebo si ha acertado
                if (respuestas[i] == numeroAdivinar) {
                    terminado= true;
                    g.drawString("Acertaste!!!", 160, 180);
                }
                // O si todavia hay que ayudarle
                else if (respuestas[i] < numeroAdivinar)
                    g.drawString("Corto", 260, lineaEscritura+i*10);
                else
                    g.drawString("Grande", 260, lineaEscritura+i*10);
            }
        }

        // Si no quedan intentos, digo cual era

```

```

        if (terminado)
            g.drawString("Era: "+numeroAdivinar, 160, 190);
    }

    public void init() {
        numeroAdivinar =
            (int) Math.round(Math.random() * MAXIMONUMERO);
        addKeyListener(this);
    }

    public void keyTyped(KeyEvent e) {
    }

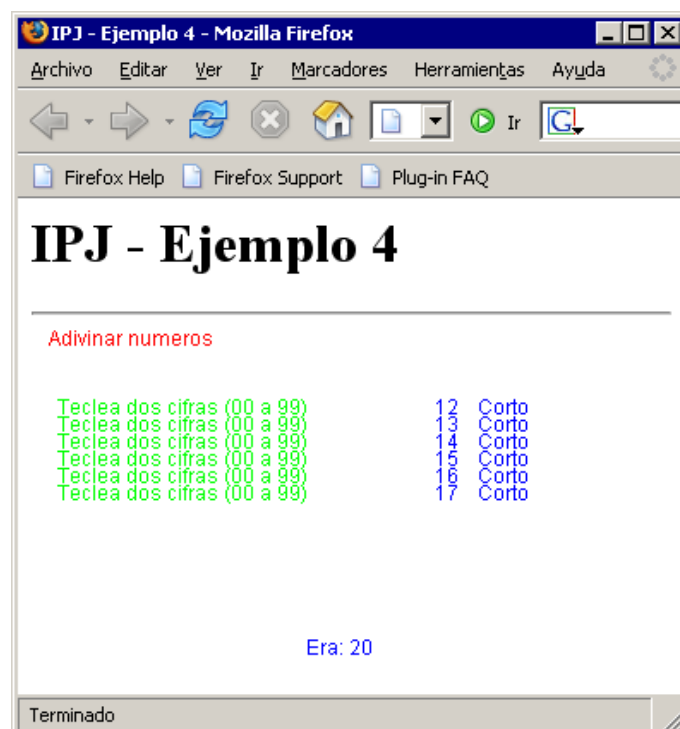
    public void keyReleased(KeyEvent e) {
    }

    public void keyPressed(KeyEvent e) {
        if (!terminado) {
            // Leo la primera tecla si corresponde
            if (tecla1 == 'a') {
                tecla1 = e.getKeyChar();
                tecla2 = 'a';
            }
            else
                // Y si no, leo la segunda
                {
                    tecla2 = e.getKeyChar();
                    // Si era la segunda, calculo el numero tecleado
                    numeroTecleado = new Integer (""+tecla1+tecla2);
                    respuestas[intentos] = numeroTecleado;
                    tecla1='a';
                    intentos++;
                    if (intentos==NUMEROINTENTOS) {
                        terminado = true;
                        intentos --; // Para no salirme en "respuestas[i]"
                    }
                }
        }
    }

    repaint();
}

```

que tendría como resultado:



5. El juego del Ahorcado.

Contenido de este apartado:

- [Pautas generales.](#)
- [Ahorcado en C.](#)
- [La versión en Pascal.](#)
- [Cómo hacerlo en el caso de Java.](#)

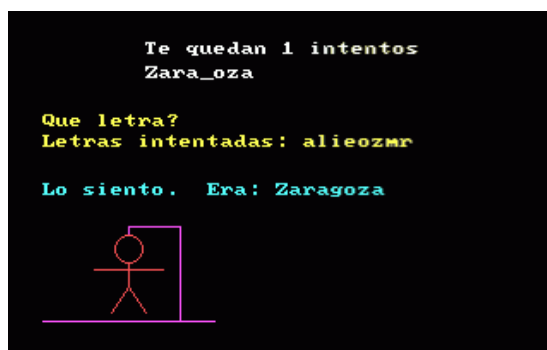
5.1. Pautas generales.

La idea básica de lo que tiene que hacer el juego es la siguiente:

- Generar un número al azar, y con ese número escoger una palabra de entre las predefinidas. Esa es la palabra que deberá adivinar el usuario.
- Repetimos:
 - El usuario elige una letra.
 - Si esa letra está en la palabra buscada, mostramos en que posiciones se encuentra.
 - Si no está en la palabra, le queda un intento menos.
- Hasta que se quede sin intentos o acierte la palabra.

El resto de características ya son "refinamientos". Por ejemplo, si termina sin acertar, le decimos cual era la palabra buscada. Además, por tratarse de un juego gráfico, en cada "pasada" comprobaremos cuantos fallos lleva, para dibujar la correspondiente parte del "patíbulo" de nuestro ahorcado.

Y todavía no buscamos una gran presentación, nos basta con que la **apariencia** sea sencilla, algo así:



Por tanto, la única complicación es: escribir texto en pantalla, dibujar líneas y círculos, y generar números al azar. Todo eso ya lo sabemos hacer, así que vamos con ello...

5.2 Ahorcado en C.

Debería ser fácil de seguir...

```

/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  ipj05c.c                     */
/*                               */
/*  Cuarto ejemplo: juego del    */
/*  ahorcado (versión básica)     */
/*                               */
/*  Comprobado con:              */
/*  - Dlgpp 2.03 (gcc 3.2)       */
/*    y Allegro 4.02 - MsDos     */
/*  - MinGW 2.0.0-3 (gcc 3.2)   */
/*    y Allegro 4.02 - Win 98    */
/*  - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*    y Allegro 4.03 - Win XP    */
/*-----*/

```

```

#include <stdio.h>          /* Rutinas estándar, como "printf" */
#include <string.h>        /* Manejo de cadenas */
#include <stdlib.h>        /* Para "rand" */
#include <time.h>          /* Para "time" */
#include <ctype.h>         /* Para "tolower" */
#include <allegro.h>

#define NUMPALABRAS 10
#define MAXINTENTOS 5
/* No deberíamos modificar el número máximo de intentos,
   porque vamos a dibujar 5 "cosas" cuando se equivoque */

char palabra[80], intento[80], letras[80];
/* La palabra a adivinar, la que */
/* el jugador 2 va consiguiendo y */
/* las letras que se han probado */

int oportunidades;        /* El número de intentos permitido */
char letra;              /* Cada letra que prueba el jug. dos */
int i;                   /* Para mirar cada letra, con "for" */
int acertado;           /* Si ha acertado alguna letra */
char ficticia[2];        /* Aux, para añadir letra a cadena */

char mensaje[80];        /* Los letreros que mostrar en pantalla */

char datosPalabras [NUMPALABRAS][80]=
{
    "Alicante","Barcelona","Guadalajara","Madrid",
    "Toledo","Malaga","Zaragoza","Sevilla",
    "Valencia","Valladolid"
};

void PrimerFallo() /* Primer fallo: */
{
    /* Dibujamos la "plataforma" */
    line(screen,20,180,120,180, palette_color[13]);
}

void SegundoFallo() /* Segundo fallo: */
{
    /* Dibujamos el "palo vertical" */
    line(screen,100,180,100,125, palette_color[13]);
}

void TercerFallo() /* Tercer fallo: */
{
    /* Dibujamos el "palo superior" */
    line(screen,100,125,70,125, palette_color[13]);
}

void CuartoFallo() /* Cuarto fallo: */
{
    /* Dibujamos la "cuerda" */
    line(screen,70,125,70,130, palette_color[13]);
}

void QuintoFallo() /* Quinto fallo: */
{
    int j;          /* Dibujamos la "persona" */

    /* Cabeza */
    circle(screen,70,138,8, palette_color[12]);
    /* Tronco */
    line(screen,70,146,70,160, palette_color[12]);
    /* Brazos */
    line(screen,50,150,90,150, palette_color[12]);
    /* Piernas */
    line(screen,70,160,60,175, palette_color[12]);
    line(screen,70,160,80,175, palette_color[12]);
}

int main()
{
    allegro_init();          /* Inicializamos Allegro */
    install_keyboard();
}

```

```

/* Intentamos entrar a modo grafico */
if (set_gfx_mode(GFX_SAFE,320,200,0,0)!=0){
    set_gfx_mode(GFX_TEXT,0,0,0,0);
    allegro_message(
        "Incapaz de entrar a modo grafico\n%s\n",
        allegro_error);
    return 1;
}

/* Si todo ha ido bien: empezamos */
srand(time(0)); /* Valores iniciales */
strcpy(palabra, datosPalabras[ rand()%(Numpalabras+1)]);
oportunidades = MAXINTENTOS;

strcpy(letras, "");

/* Relleno con _ y " " lo que ve Jug. 2 */
for (i=1; i<=strlen(palabra); i++)
    if (palabra[i-1]==' ')
        intento[i-1]=' ';
    else
        intento[i-1]='_';
intento[i]='\0'; /* Y aseguro que termine correctamente */

/* Parte repetitiva: */
do {
    clear_bitmap(screen);

    /* Dibujo lo que corresponde del "patibulo" */
    if (oportunidades <=4) PrimerFallo();
    if (oportunidades <=3) SegundoFallo();
    if (oportunidades <=2) TercerFallo();
    if (oportunidades <=1) CuartoFallo();

    /* Digo cuantos intentos le quedan */
    textprintf(screen, font, 80,18, palette_color[15],
        "Te quedan %d intentos", oportunidades);

    /* Le muestro cómo va */
    textprintf(screen, font, 80,32, palette_color[15],
        intento, oportunidades);

    /* Las letras intentadas */
    textprintf(screen, font, 20,72, palette_color[14],
        "Letras intentadas: %s", letras);

    /* Y le pido otra letra */
    textprintf(screen, font, 20,60, palette_color[14],
        "Que letra?");

    letra = readkey()&0xff;

    /* Añado esa letra a las tecleadas*/
    strcpy (ficticia,"a"); /* Usando una cadena de texto aux */
    ficticia[0]= letra;
    strcat (letras, ficticia);

    acertado = 0; /* Miro a ver si ha acertado */
    for (i=1; i<=strlen(palabra); i++)
        if(tolower(letra)== tolower(palabra[i-1]))
        {
            intento[i-1]= palabra[i-1];
            acertado = 1;
        }

    if (! acertado ) /* Si falló, le queda un intento menos */
        oportunidades --;
}
while ( strcmp (intento,palabra) /* Hasta que acierte */
    && (oportunidades>0)); /* o gaste sus oportunidades */

/* Le felicito o le digo cual era */

```

```

if ( strcmp (intento,palabra)==0)
    textprintf(screen, font, 20,100, palette_color[11],
        "Acertaste!");
else
    {
    textprintf(screen, font, 20,100, palette_color[11],
        "Lo siento. Era: %s", palabra);
    QuintoFallo();
    }

readkey();
return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

5.3. Ahorcado en Pascal.

Esta versión debería ser aun más fácil de seguir que la de C:

```

(*-----*)
(* Intro a la programac de *)
(* juegos, por Nacho Cabanes *)
(* *)
(* IPJ05P.PAS *)
(* *)
(* Cuarto ejemplo: juego del *)
(* ahorcado (version basica) *)
(* *)
(* Comprobado con: *)
(* - FreePascal 2.04 - WinXP *)
(* - FreePascal 1.10 - Dos *)
(*-----*)

uses graph, wincrt;
(* Cambiar por "uses graph, crt;" bajo Dos *)

const NUMPALABRAS = 10;
const MAXINTENTOS = 5;
(* No deberiamos modificar el numero maximo de intentos,
   porque vamos a dibujar 5 'cosas' cuando se equivoque *)

var
    palabra, intento, letras: string[80];
    (* La palabra a adivinar, la que *)
    (* el jugador 2 va consiguiendo y *)
    (* las letras que se han probado *)

    oportunidades: integer; (* El numero de intentos permitido *)
    oportStr: string; (* Y para convertirlo a cadena *)
    letra: char; (* Cada letra que prueba el jug. dos *)
    i: integer; (* Para mirar cada letra, con 'for' *)
    acertado: boolean; (* Si ha acertado alguna letra *)

    mensaje: string[80]; (* Los letreros que mostar en pantalla *)

const datosPalabras: array [1..NUMPALABRAS] of string= (
    'Alicante','Barcelona','Guadalajara','Madrid',
    'Toledo','Malaga','Zaragoza','Sevilla',
    'Valencia','Valladolid');

procedure PrimerFallo; (* Primer fallo: *)
begin (* Dibujamos la 'plataforma' *)
    setcolor(13);
    line(20, 180, 120, 180);
end;

```



```

procedure SegundoFallo;          (* Segundo fallo: *)
begin                            (* Dibujamos el 'palo vertical' *)
    setcolor(13);
    line(100, 180, 100, 125);
end;

procedure TercerFallo;          (* Tercer fallo: *)
begin                            (* Dibujamos el 'palo superior' *)
    setcolor(13);
    line(100, 125, 70, 125);
end;

procedure CuartoFallo;         (* Cuarto fallo: *)
begin                            (* Dibujamos la 'cuerda' *)
    setcolor(13);
    line(70, 125, 70, 130);
end;

procedure QuintoFallo;        (* Quinto fallo: *)
begin                            (* Dibujamos la 'persona' *)

    setcolor(12);
    (* Cabeza *)
    circle(70, 138, 8);
    (* Tronco *)
    line(70, 146, 70, 160);
    (* Brazos *)
    line(50, 150, 90, 150);
    (* Piernas *)
    line(70, 160, 60, 175);
    line(70, 160, 80, 175);
end;

var
    gd, gm, error : integer;

BEGIN
    gd := D8bit;
    gm := m640x480;
    (* Si falla bajo Dos, probar gm:=m320x200; *)
    initgraph(gd, gm, '');

                                (* Intentamos entrar a modo grafico *)
    error := graphResult;
    if error <> grOk then
        begin
            writeln('No se pudo entrar a modo grafico');
            writeln('Error encontrado: '+ graphErrorMsg(error) );
            halt(1);
        end;

                                (* Si todo ha ido bien: empezamos *)
    randomize;                    (* Valores iniciales *)
    palabra := datosPalabras[ round(random * NUMPALABRAS) + 1 ];
    oportunidades := MAXINTENTOS;

    letras := '';

                                (* Relleno con _ y ' ' lo que ve Jug. 2 *)
    intento := palabra;
    for i:=1 to length(palabra) do
        if palabra[i] = ' ' then
            intento[i] := ' '
        else
            intento[i] := '_';

                                (* Parte repetitiva: *)
    repeat
        clearDevice;

        (* Dibujo lo que corresponde del 'patibulo' *)
        if (oportunidades <= 4) then PrimerFallo;

```

```

if (oportunidades <= 3) then SegundoFallo;
if (oportunidades <= 2) then TercerFallo;
if (oportunidades <= 1) then CuartoFallo;

                                (* Digo cuantos intentos le quedan *)
setColor(15);
str(oportunidades, oportStr);
outTextXY(80, 18, 'Te quedan '+oportStr+' intentos');

                                (* Las letras intentadas *)
outTextXY(80, 32, intento);

setColor(14);
outTextXY(20, 72, 'Letras intentadas: '+letras);

                                (* Y le pido otra letra *)
outTextXY(20, 60, 'Que letra?');

letra := readkey;

                                (* Añado esa letra a las tecleadas*)
letras := letras + letra;

acertado := false;                                (* Miro a ver si ha acertado *)
for i := 1 to length(palabra) do
  if lowercase(letra) = lowercase(palabra[i]) then
    begin
      intento[i] := palabra[i];
      acertado := true;
    end;

if not acertado then                                (* Si falló, le queda un intento menos *)
  oportunidades := oportunidades - 1;

until (intento = palabra)                                (* Hasta que acierte *)
  or (oportunidades=0);                                (* o gaste sus oportunidades *)

                                (* Le felicito o le digo cual era *)

setColor(15);
if intento = palabra then
  outTextXY(20, 100, 'Acertaste!')
else
  begin
    outTextXY(20, 100, 'Lo siento. Era: '+ palabra);
    QuintoFallo;
  end;

readkey;
closeGraph;

end.

```

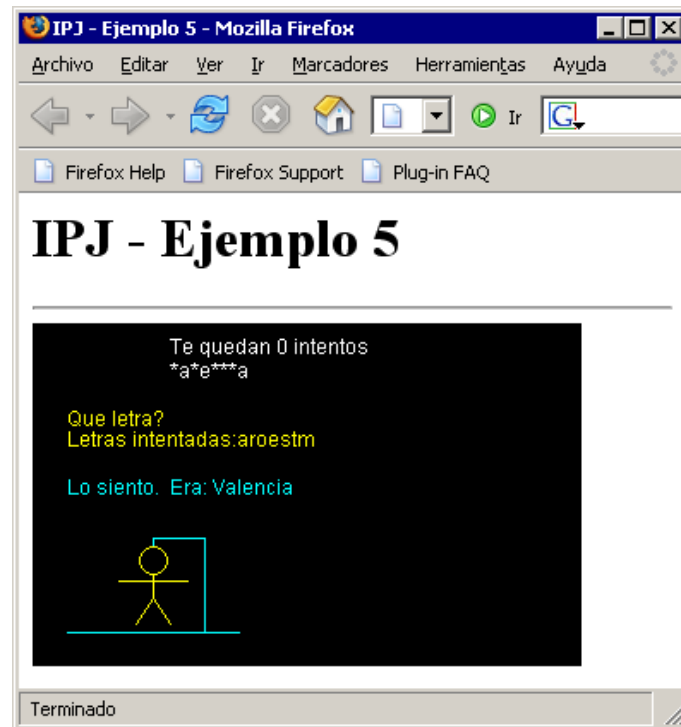
5.4. Ahorcado con Java.

La diferencia grande con la versión en C es la forma en que se estructura, como ya vimos en el ejemplo anterior: en "init" pondremos todo lo que sea inicialización, en "paint" todo lo que dibuje en pantalla (aunque hemos desglosado ligeramente lo que hacer en el caso de cada fallo, igual que en la versión en C), y en "keyPressed" agruparemos las operaciones básicas que hay que realizar cuando se pulsa una tecla.

- Las demás diferencias son básicamente de sintaxis. Por ejemplo:
 - Para escribir un texto y una cifra usamos cosas como expresiones más parecidas a Pascal que a C, cosas como `g.drawString("Te quedan " + oportunidades + " intentos", 80, 18);`
 - Para leer el carácter que hay en una posición de una cadena usaremos `palabra.charAt(i-1)` y para cambiar el carácter que hay en una posición se haría con `palabra.setCharAt(i-1, 'a')`.

- Para comparar dos cadenas de texto usamos la construcción `palabra.equals(otraPalabra)`
- La cadena de texto que debemos modificar con frecuencia (intento) no la definimos como `String` sino como `StringBuffer`, que es el tipo de datos que permite operaciones avanzadas como `SetCharAt`.

La **apariciencia** será algo como:



Y una forma de desarrollarlo sería:

```

/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*    ipj05j.java                */
/*                               */
/*  Cuarto ejemplo: juego del  */
/*  ahorcado (versión básica)  */
/*                               */
/*  Comprobado con:              */
/*  - JDK 1.4.2_01               */
/*  - JDK 1.5.0                  */
/*-----*/

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ipj04j extends Applet
    implements KeyListener
{

    final int NUMPALABRAS = 10;
    final int MAXINTENTOS = 5;
        // No deberíamos modificar el número máximo de intentos,
        // porque vamos a dibujar 5 "cosas" cuando se equivoque

    String palabra;           // La palabra a adivinar
    StringBuffer intento;    // Lo que el jugador 2 va consiguiendo
    String letras="";        // Las letras que se han probado

    int oportunidades;      // El número de intentos permitido
    char letra;             // Cada letra que prueba el jug. dos
    int i;                  // Para mirar cada letra, con "for"
    boolean acertado;       // Si ha acertado alguna letra
    boolean terminado;     // Si la partida ha terminado

```

```

String datosPalabras []=
{
    "Alicante","Barcelona","Guadalajara","Madrid",
    "Toledo","Malaga","Zaragoza","Sevilla",
    "Valencia","Valladolid"
};

void PrimerFallo(Graphics g)
{ // Primer fallo: Dibujamos la "plataforma"
  g.setColor(Color.cyan);
  g.drawLine(20, 180, 120, 180);
}

void SegundoFallo(Graphics g)
{ // Segundo fallo: Dibujamos el "palo vertical"
  g.drawLine(100, 180, 100, 125);
}

void TercerFallo(Graphics g)
{ // Tercer fallo: Dibujamos el "palo superior"
  g.drawLine(100, 125, 70, 125);
}

void CuartoFallo(Graphics g)
{ // Cuarto fallo: Dibujamos la "cuerda"
  g.drawLine(70, 125, 70, 130);
}

void QuintoFallo(Graphics g)
{ // Quinto fallo: Dibujamos la "persona"
int j;

    // Cabeza
    g.setColor(Color.yellow);
    g.drawOval(62, 130, 16, 16);
    // Tronco
    g.drawLine(70, 146, 70, 160);
    // Brazos
    g.drawLine(50, 150, 90, 150);
    // Piernas
    g.drawLine(70, 160, 60, 175);
    g.drawLine(70, 160, 80, 175);
}

public void init() {

    // Valores iniciales
    i = (int) Math.round(Math.random() * NUMPALABRAS);
    palabra = datosPalabras[ i ];
    oportunidades = MAXINTENTOS;

    // Relleno con * y " " lo que ve Jug. 2
    intento = new StringBuffer(palabra);

    for (i=1; i<=palabra.length(); i++)
    if (palabra.charAt(i-1) == ' ' )
        intento.setCharAt(i-1, ' ');
    else
        intento.setCharAt(i-1, '*');

    terminado = false;
    requestFocus();
    addKeyListener(this);
}

public void paint(Graphics g) {

    // Primero borro el fondo en negro
    g.setColor( Color.black );
    g.fillRect( 0, 0, 639, 479 );

    // Digo cuantos intentos le quedan

```

```

g.setColor(Color.white);
g.drawString("Te quedan " + oportunidades + " intentos",
            80, 18);

// Le muestro como va
g.drawString(intento.toString(),
            80, 32);

// Muestro las letras probadas
g.setColor(Color.yellow);
g.drawString("Letras intentadas:" + letras,
            20, 72);

// Y le pido otra letra
g.drawString("Que letra?", 20, 60);

// Dibujo lo que corresponde del "patibulo"
if (oportunidades <= 4) PrimerFallo(g);
if (oportunidades <= 3) SegundoFallo(g);
if (oportunidades <= 2) TercerFallo(g);
if (oportunidades <= 1) CuartoFallo(g);
// Si se acabo: Le felicito o le digo cual era
if ((oportunidades <= 0) || (palabra.equals(intento.toString()))) {
    terminado = true;
    if ( palabra.equals(intento.toString() ) )
        g.drawString("Acertaste!",
                    20, 100);
    else
    {
        g.drawString("Lo siento. Era: " + palabra,
                    20, 100);
        QuintoFallo(g);
    }
}

}

public void keyTyped(KeyEvent e) {
    letra=e.getKeyChar();
    if (! terminado) {
        letras=letras+letra;

        acertado = false;        // Miro a ver si ha acertado
        for (i=1; i<=palabra.length(); i++)
            if (Character.toLowerCase(letra) == Character.toLowerCase(palabra.charAt(i-1)))
                {
                    intento.setCharAt(i-1, palabra.charAt(i-1) );
                    acertado = true;
                }

        if ( ! acertado )        // Si falló, le queda un intento menos
            oportunidades --;
    }
    repaint();
    e.consume();
}

public void keyReleased(KeyEvent e) {
}

public void keyPressed(KeyEvent e) {
}

}

```

6. Evitemos esperar al teclado. Tercer juego: motos de luz.

Contenido de este apartado:

- [Ideas generales.](#)
- [Ahorcado en C.](#)
- [La versión en Pascal.](#)

- [Problemas en el caso de Java.](#)

6.1. Ideas generales.

En la mayoría de los juegos no ocurre lo que en el anterior: no podemos permitirnos que el ordenador que "parado" esperando a que se pulse una tecla, sino que la acción debe proseguir aunque nosotros no toquemos el teclado.

Esto es muy fácil de conseguir. Entre la rutinas estándar de casi cualquier compilador de **C** tenemos la función **kbhit()**, que devuelve "verdadero" (distinto de cero) si se ha pulsado alguna tecla y "falso" (cero) si no se ha pulsado ninguna. Posteriormente, con **getch()** sabríamos exactamente qué tecla es la que se ha pulsado y vaciaríamos el "buffer" (memoria intermedia) del teclado para que se pudieran seguir comprobando nuevas teclas.

En el caso de **Allegro**, la rutina que comprueba si hemos pulsado alguna tecla se llama **keypressed()**, y en la práctica podríamos hacer cosas como esta para que el programa no se pare pero a la vez esté pendiente de qué teclas pulsemos:

```
do
{
  /*Actualizar reloj y pantalla, mover nave, etc */
}
while ( ! keypressed ( ) );
teclaPulsada = readkey();
```

Al principio del programa deberemos añadir la línea "install_keyboard()" para tener acceso a estas rutinas de manejo de teclado.

En el caso de **Pascal** (Free Pascal) es muy similar, incluso con los mismos nombres de funciones auxiliares (que están en la unidad CRT):

```
repeat
  (* Actualizar reloj y pantalla, mover nave, etc *)
until keypressed;

teclaPulsada := readkey;
```

Para **Java**... existe una **dificultad** en este juego con una de las cosas que no hemos comentado aún, y que surgirá enseguida...

Vamos a aplicarlo a un primer juego de habilidad, el clásico de las "**motos de luz**" (si alguien recuerda la película Tron, quizá sepa a qué me refiero; si no se conoce esa película, la imagen inferior dará un pista de lo que perseguimos). La idea es la siguiente:

- Participan dos jugadores.
- Cada uno de los jugadores maneja una "moto", que va dejando un rastro cuando se desplaza.
- Si una de las motos "choca" con el rastro dejado por la otra, o por ella misma, o con el borde de la pantalla, estalla y pierde la partida.
- Por tanto, el objetivo es "arrinconar" al contrario sin ser arrinconado antes por él.

Con un poco más de detalle, lo que tendría que hacer nuestro programa será lo siguiente:

- Borrar la pantalla y dibujar un recuadro alrededor de ella.
- Preparar las variables iniciales: dirección de cada jugador (incremento en X e Y de la posición de cada uno). En el instante inicial, las motos se desplazarán en direcciones contrarias. En un juego "más real", este incremento podría ser variable (o disminuir el tiempo de pausa entre un movimiento y otro), de modo que las motos circularan cada vez más deprisa, para mayor dificultad.
- Parte repetitiva:
 - Si la siguiente posición de alguna de las motos está ocupada, la moto estalla y ese jugador ha perdido: se acabó el juego.
 - Desplazar las motos a su siguiente posición.
 - Si se ha pulsado alguna tecla, analizar esta tecla y cambiar la dirección de la moto según corresponda.
 - Pausa fija (o variable según la dificultad) antes de la siguiente repetición, fijada por nuestro programa, para que la velocidad no dependa de la rapidez del ordenador que se utilice.

- Repetir indefinidamente (la condición de salida la comprobamos "dentro").

Eso de que una moto choque con otra... suena difícil, pero en este primer caso nos bastará con mirar el punto de la pantalla al que nos vamos a mover. Si ese punto tiene un color distinto del fondo, habremos chocado (quizá sea con el borde de la pantalla, quizá con nuestra estela, quizá con la del otro jugador... pero eso nos da igual).

Ese es el problema en el caso de **Java**: no se puede leer un punto de la pantalla con esa facilidad, lo que nos obligaría a "memorizar" de otra forma las posiciones que no se pueden tocar. No es difícil, pero es algo que veremos dentro de poco, así que aplazamos un poco la versión en Java de este juego.

Lo único que aún no sabemos hacer es la **pausa**. Con C y **Allegro** usaremos la función

```
rest(n);
```

(donde "n" es el tiempo en milisegundos), que requiere que antes hayamos instalado el manejador de temporizadores (también será necesario para otras cosas, por ejemplo cuando accedamos al ratón, y al reproducir ciertos tipos de animaciones y de ficheros musicales) con:

```
install_timer();
```

En **Pascal**, la pausa de "n" milisegundos se haría con

```
delay(n);
```

Y en **Java** (que insisto que no usaremos aún en este ejemplo), la pausa sería

```
try {
    Thread.sleep( n );
}
catch ( InterruptedException e ) {
}
```

pero nos obligará también a hacer algunos cambios más en la estructura del fuente.

La apariencia del juego será también sencilla, así:



6.2 Las motos en C.

Debería ser fácil de seguir...

```
/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*    ipj06c.c                   */
/*                               */
/*  Quinto ejemplo: juego de    */
/*  "motos de luz"              */
/*                               */
/*  Comprobado con:             */
/*                               */
```

```

/* - Djgpp 2.03 (gcc 3.2) */
/* y Allegro 4.02 - MsDos */
/* - MinGW 2.0.0-3 (gcc 3.2) */
/* y Allegro 4.02 - Win 98 */
/*-----*/
#include <allegro.h>

/* Posiciones X e Y iniciales de ambos jugadores */
#define POS_X_INI_1 150
#define POS_X_INI_2 170
#define POS_Y_INI_1 100
#define POS_Y_INI_2 100

#define INC_X_INI_1 -1
#define INC_X_INI_2 1
#define INC_Y_INI_1 0
#define INC_Y_INI_2 0

/* Pausa en milisegundos entre un "fotograma" y otro */
#define PAUSA 150

/* Teclas predefinidas para cada jugador */
#define TEC_ARRIBA_1 KEY_E
#define TEC_ABAJO_1 KEY_X
#define TEC_IZQDA_1 KEY_S
#define TEC_DCHA_1 KEY_D

#define TEC_ARRIBA_2 KEY_8_PAD
#define TEC_ABAJO_2 KEY_2_PAD
#define TEC_IZQDA_2 KEY_4_PAD
#define TEC_DCHA_2 KEY_6_PAD

int posX1, posY1, posX2, posY2; /* Posiciones actuales */
int incX1, incY1, incX2, incY2; /* Incremento de la posicion */
int futX1, futY1, futX2, futY2; /* Posiciones futuras */

/* Si ha chocado alguna moto */
int chocado;

/* La tecla pulsada */
int tecla;

int main()
{
    allegro_init(); /* Inicializamos Allegro */
    install_keyboard();
    install_timer();

    /* Intentamos entrar a modo grafico */
    if (set_gfx_mode(GFX_SAFE, 320, 200, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    /* Si todo ha ido bien: empezamos */

    /* Rectangulo amarillo alrededor */
    rect(screen, 0, 0, 319, 199, palette_color[14]);

    /* Valores iniciales */
    posX1 = POS_X_INI_1;
    posX2 = POS_X_INI_2;
    posY1 = POS_Y_INI_1;
    posY2 = POS_Y_INI_2;

    incX1 = INC_X_INI_1;
    incX2 = INC_X_INI_2;
    incY1 = INC_Y_INI_1;
    incY2 = INC_Y_INI_2;

    /* Parte repetitiva: */

```



```

do {
    chocado = FALSE;

    /* Compruebo si alguna va a colisionar */
    futX1 = posX1 + incX1;
    futX2 = posX2 + incX2;
    futY1 = posY1 + incY1;
    futY2 = posY2 + incY2;

    if (getpixel(screen, futX1, futY1)!=0){
        textout(screen, font,
            "La moto 1 ha chocado!", 100,90, palette_color[13]);
        chocado =TRUE;
    }

    if (getpixel(screen, futX2, futY2)!=0){
        textout(screen, font,
            "La moto 2 ha chocado!", 100,110, palette_color[12]);
        chocado = TRUE;
    }

    if (chocado)break;

    /* Si ninguna ha colisionado, avanzan */
    line (screen, posX1, posY1, futX1, futY1, palette_color[13]);
    posX1 = futX1; posY1 = futY1;

    line (screen, posX2, posY2, futX2, futY2, palette_color[12]);
    posX2 = futX2; posY2 = futY2;

    /* Compruebo si se ha pulsado alguna tecla */
    if ( keypressed() ){
        tecla = readkey() >>8;

        switch(tecla){
            case TEC_ARRIBA_1:
                incX1 = 0; incY1 = -1;break;
            case TEC_ABAJO_1:
                incX1 = 0; incY1 = 1;break;
            case TEC_IZQDA_1:
                incX1 = -1; incY1 = 0;break;
            case TEC_DCHA_1:
                incX1 = 1; incY1 = 0;break;

            case TEC_ARRIBA_2:
                incX2 = 0; incY2 = -1;break;
            case TEC_ABAJO_2:
                incX2 = 0; incY2 = 1;break;
            case TEC_IZQDA_2:
                incX2 = -1; incY2 = 0;break;
            case TEC_DCHA_2:
                incX2 = 1; incY2 = 0;break;
        }
    }

    /* Pequeña pausa antes de seguir */
    rest ( PAUSA );
}
while (TRUE);/* Repetimos indefinidamente */
/* (la condición de salida la comprobamos "dentro") */

readkey();
return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

6.3. Las motos desde Pascal.

Muuuuy parecido a la versión en C:

```
(*-----*)
(*  Intro a la programac de  *)
(*  juegos, por Nacho Cabanes *)
(*          *)
(*  IPJ06P.PAS          *)
(*          *)
(*  Sexto ejemplo: juego de  *)
(*  "motos de luz"        *)
(*          *)
(*  Comprobado con:      *)
(*  - FreePascal 1.10 (Dos) *)
(*  - FreePascal 2.0 -Windows *)
(*-----*)

uses graph, crt;
  (* Cambiar por "uses wincrt, ..." bajo Windows *)

const

  (* Posiciones X e Y iniciales de ambos jugadores *)
  POS_X_INI_1 = 150;
  POS_X_INI_2 = 170;
  POS_Y_INI_1 = 100;
  POS_Y_INI_2 = 100;

  INC_X_INI_1 = -1;
  INC_X_INI_2 = 1;
  INC_Y_INI_1 = 0;
  INC_Y_INI_2 = 0;

  (* Pausa en milisegundos entre un "fotograma" y otro *)
  PAUSA = 150;

  (* Teclas predefinidas para cada jugador *)
  TEC_ARRIBA_1 = 'E';
  TEC_ABAJO_1  = 'X';
  TEC_IZQDA_1  = 'S';
  TEC_DCHA_1   = 'D';

  TEC_ARRIBA_2 = '8';
  TEC_ABAJO_2  = '2';
  TEC_IZQDA_2  = '4';
  TEC_DCHA_2   = '6';

var
  posX1, posY1, posX2, posY2: integer;  (* Posiciones actuales *)
  incX1, incY1, incX2, incY2: integer;  (* Incremento de la posicion *)
  futX1, futY1, futX2, futY2: integer;  (* Posiciones futuras *)

  (* Si ha chocado alguna moto *)
  chocado: boolean;

  (* La tecla pulsada *)
  tecla: char;

var
  gd, gm, error : integer;

begin
  gd := D8bit;
  gm := m320x200;
  (* Si falla bajo Windows, probar gd:=0; gm:=0; *)
  initgraph(gd, gm, '');

  (* Intentamos entrar a modo grafico *)
  error := graphResult;
  if error <> grOk then
    begin
      writeln('No se pudo entrar a modo grafico');
      writeln('Error encontrado: '+ graphErrorMsg(error) );
      halt(1);
    end;
```

```

(* Si todo ha ido bien: empezamos *)

(* Rectangulo amarillo alrededor *)
setcolor(14);
rectangle(0,0, 319, 199);

(* Valores iniciales *)
posX1 := POS_X_INI_1;
posX2 := POS_X_INI_2;
posY1 := POS_Y_INI_1;
posY2 := POS_Y_INI_2;

incX1 := INC_X_INI_1;
incX2 := INC_X_INI_2;
incY1 := INC_Y_INI_1;
incY2 := INC_Y_INI_2;

(* Parte repetitiva: *)
repeat
  chocado := FALSE;

  (* Compruebo si alguna va a colisionar *)
  futX1 := posX1 + incX1;
  futX2 := posX2 + incX2;
  futY1 := posY1 + incY1;
  futY2 := posY2 + incY2;

  if (getpixel(futX1, futY1) <> 0) then
    begin
      SetColor(13);
      OutTextXY( 100, 90,
        'La moto 1 ha chocado!');
      chocado := TRUE;
    end;

  if (getpixel(futX2, futY2) <> 0) then
    begin
      SetColor(12);
      OutTextXY( 100, 110,
        'La moto 2 ha chocado!');
      chocado := TRUE;
    end;

  if chocado then break;

  (* Si ninguna ha colisionado, avanzan *)
  setcolor(13);
  line (posX1, posY1, futX1, futY1);
  posX1 := futX1; posY1 := futY1;

  setcolor(12);
  line (posX2, posY2, futX2, futY2);
  posX2 := futX2; posY2 := futY2;

  (* Compruebo si se ha pulsado alguna tecla *)
  if keypressed then
    begin
      tecla := upcase(readkey);

      case tecla of
        TEC_ARRIBA_1:
          begin incX1 := 0; incY1 := -1; end;
        TEC_ABAJO_1:
          begin incX1 := 0; incY1 := 1; end;
        TEC_IZQDA_1:
          begin incX1 := -1; incY1 := 0; end;
        TEC_DCHA_1:
          begin incX1 := 1; incY1 := 0; end;

        TEC_ARRIBA_2:
          begin incX2 := 0; incY2 := -1; end;
        TEC_ABAJO_2:
          begin incX2 := 0; incY2 := 1; end;
      end case;
    end;
end repeat;

```

```

TEC_IZQDA_2:
  begin incX2 := -1; incY2 := 0; end;
TEC_DCHA_2:
  begin incX2 := 1; incY2 := 0; end;
end;
end;

(* Pequea pausa antes de seguir *)
delay ( PAUSA );

until FALSE; (* Repetimos indefinidamente *)
(* (la condicion de salida la comprobamos "dentro") *)

readkey();

end.

```

6.4. Problemas en el caso de Java.

Como hemos dicho en la introducción, en el caso de Java, la situación se complica ligeramente, porque no tenemos ninguna función que nos diga cual es el color de un punto de la pantalla. Así que para imitar el funcionamiento de las versiones anteriores, tendríamos que "memorizar" el contenido de cada punto de la pantalla. Supone guardar información sobre $320 \times 200 = 64.000$ puntos. Hay al menos un par de formas de hacerlo, y las dos son sencillas, pero para llevar un poco de orden, lo aplazamos... el próximo tema nos ayudará a entender lo que nos falta.

También será en el próximo apartado donde veamos cómo hacer eso de que nuestro juego "avance" cada cierto tiempo, incluso aunque no se pulse una tecla.

7. Mapas. Cuarto juego (aproximación "a"): MiniSerpiente 1.

Contenido de este apartado:

- [Ideas generales.](#)
- [Miniserpiente 1 en C.](#)
- [La versión en Pascal.](#)
- [La versión en Java.](#)

7.1. Ideas generales.

El juego de la **serpiente** es casi tan sencillo de crear como el de las motos de luz: una figura va avanzando por la pantalla; si choca con la pared exterior, con su propia "cola" o con algún otro obstáculo, muere. La única complicación adicional es que en la mayoría de versiones de este juego también va apareciendo comida, que debemos atrapar; esto nos dará puntos, pero también hará que la serpiente sea más grande, y por eso, más fácil chocar con nuestra propia cola.

Esta novedad hace que sea algo más difícil de programar. En primer lugar porque la serpiente va creciendo, y en segundo lugar porque puede que en una posición de la pantalla exista un objeto contra el que podemos chocar pero no debemos morir, sino aumentar nuestra puntuación.

Podríamos volver a usar el "truco" de mirar en los puntos de la pantalla, y distinguir la comida usando un color distinto al de los obstáculos. Pero esto no es lo que se suele hacer. No serviría si nuestro fondo fuera un poco más vistoso, en vez de ser una pantalla negra. En lugar de eso, es más cómodo memorizar un "**mapa**" con las posibles posiciones de la pantalla, e indicando cuales están vacías, cuales ocupadas por obstáculos y cuales ocupadas por comida.

Podría ser algo así:

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
X                                     X   X
X      X                             X   X

```

```

X   XXX   F   X   X   X
X           X   X
X           X   F   X
X           X   X   X
X   F   X           XXX   X
X           X           X
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

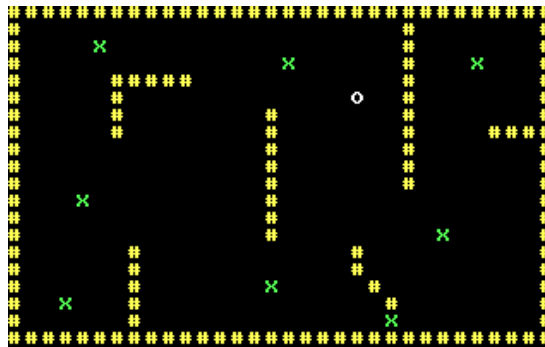
```

En este ejemplo, las X serían las casillas "peligrosas", con las que no debemos chocar si no queremos morir; las F serían las frutas que podemos recoger para obtener puntos extra.

El hecho de consultar este mapa en vez del contenido de la pantalla nos permite dibujar esos obstáculos, esas frutas, el fondo y nuestra propia serpiente con otras imágenes más vistosas.

Vamos a aplicarlo. Tomaremos la base del juego de las motos de luz, porque la idea básica coincide: el objeto se debe seguir moviendo aunque no toquemos ninguna tecla, y en cada paso se debe comprobar si hemos chocado con algún obstáculo. A esta base le añadiremos el uso de un mapa, aunque todavía será poco vistoso...

La apariencia del juego, todavía muy pobre, será:



7.2 Miniserpiente 1 en C.

Sencillo. Muy parecido al anterior, pero no miramos en pantalla sino en nuestro "mapa":

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* ipj07c.c */
/* */
/* Septimo ejemplo: juego de */
/* "miniSerpiente" */
/* */
/* Comprobado con: */
/* - Dgpp 2.03 (gcc 3.2) */
/* y Allegro 4.02 - MsDos */
/* - MinGW 2.0.0-3 (gcc 3.2) */
/* y Allegro 4.02 - Win */
/*-----*/

#include <allegro.h>

/* Posiciones X e Y iniciales */
#define POS_X_INI 16
#define POS_Y_INI 10

#define INC_X_INI 1
#define INC_Y_INI 0

/* Pausa en milisegundos entre un "fotograma" y otro */
#define PAUSA 350

/* Teclas predefinidas */
#define TEC_ARRIBA KEY_E
#define TEC_ABAJO KEY_X
#define TEC_IZQDA KEY_S
#define TEC_DCHA KEY_D

```

```

int posX, posY; /* Posicion actual */
int incX, incY; /* Incremento de la posicion */

/* Terminado: Si ha chocado o comido todas las frutas */
int terminado;

/* La tecla pulsada */
int tecla;

/* Escala: relacion entre tamaño de mapa y de pantalla */
#define ESCALA 10

/* Ancho y alto de los sprites */
#define ANCHOSPRITE 10
#define ALTOSPRITE 10

/* Y el mapa que representa a la pantalla */
/* Como usaremos modo grafico de 320x200 puntos */
/* y una escala de 10, el tablero medira 32x20 */
#define MAXFILAS 20
#define MAXCOLS 32

char mapa[MAXFILAS][MAXCOLS]={
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
    "X                X  X",
    "X   F                X  X",
    "X                F   X  F  X",
    "X   XXXXX                X  X",
    "X   X                  X  X",
    "X   X                X  X",
    "X                X  X",
    "X                X  X",
    "X                X  X",
    "X                X  X",
    "X   F                X  X",
    "X                X  X",
    "X                X  F  X",
    "X   X                X  X",
    "X   X                X  X",
    "X   X                F   X  X",
    "X  F  X                X  X",
    "X   X                F   X",
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
};

int numFrutas = 7;

/* Nuestros sprites */
BITMAP *ladrilloFondo, *comida, *jugador;

typedef
    char tipoSprite[ANCHOSPRITE][ALTOSPRITE];
    /* El sprite en si: matriz de 30x30 bytes */

tipoSprite spriteLadrillo =
    {{0,2,2,2,2,2,2,2,0},
     {2,1,1,1,1,1,1,1,2},
     {2,1,1,1,1,1,1,1,2},
     {2,1,1,1,1,1,1,1,2},
     {2,1,1,1,1,1,1,1,2},
     {2,1,1,1,1,1,1,1,2},
     {2,1,1,1,1,1,1,3,2},
     {2,1,1,1,1,1,1,3,3,2},
     {2,1,1,1,1,1,3,3,2,2},
     {2,2,2,2,2,2,2,2,2,0}
    };

tipoSprite spriteComida =
    {{0,0,0,2,0,0,0,0,0,0},
     {0,0,2,2,0,0,2,2,0,0},
     {0,4,4,4,2,2,4,4,0,0},
     {4,4,4,4,4,2,4,4,4,0},
     {4,4,4,4,4,4,4,4,4,0},
     {4,4,4,4,4,4,4,4,4,0},
     {4,4,4,4,4,4,4,4,4,0},
     {4,4,4,4,4,4,4,4,4,0},
     {4,4,4,4,4,4,4,4,4,0},
     {0,4,4,4,4,4,4,4,4,0}
    };

tipoSprite spriteJugador =

```

```

    {{0,0,3,3,3,3,3,0,0,0},
     {0,3,1,1,1,1,1,3,0,0},
     {3,1,1,1,1,1,1,1,3,0},
     {3,1,1,1,1,1,1,1,3,0},
     {3,1,1,1,1,1,1,1,3,0},
     {3,1,1,1,1,1,1,1,3,0},
     {0,3,1,1,1,1,1,3,0,0},
     {0,0,3,3,3,3,3,0,0,0}
    };

/* ----- Rutina de crear los sprites ----- */

void creaSprites()
{
    int i, j;

    ladrilloFondo = create_bitmap(10, 10);
    clear_bitmap(ladrilloFondo);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(ladrilloFondo, i, j,
                palette_color[ spriteLadrillo[j][i] ]);

    comida = create_bitmap(10, 10);
    clear_bitmap(comida);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(comida, i, j,
                palette_color[ spriteComida[j][i] ]);

    jugador = create_bitmap(10, 10);
    clear_bitmap(jugador);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(jugador, i, j,
                palette_color[ spriteJugador[j][i] ]);
}

/* ----- Rutina de dibujar el fondo ----- */

void dibujaFondo()
{
    int i, j;

    clear_bitmap(screen);

    for(i=0; i<MAXCOLS; i++)
        for (j=0; j<MAXFILAS; j++) {
            if (mapa[j][i] == 'X')
                draw_sprite(screen, ladrilloFondo, i*ESCALA, j*ESCALA);
            if (mapa[j][i] == 'F')
                draw_sprite(screen, comida, i*ESCALA, j*ESCALA);
        }
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    allegro_init();                /* Inicializamos Allegro */
    install_keyboard();
    install_timer();

    /* Intentamos entrar a modo grafico */
    if (set_gfx_mode(GFX_SAFE, 320, 200, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }
}

```

```

/* ----- Si todo ha ido bien: empezamos */

creaSprites();
dibujaFondo();

/* Valores iniciales */
posX = POS_X_INI;
posY = POS_Y_INI;

incX = INC_X_INI;
incY = INC_Y_INI;

/* Parte repetitiva: */
do {
    dibujaFondo();
    draw_sprite (screen, jugador, posX*ESCALA, posY*ESCALA);

    terminado = FALSE;

    /* Si paso por una fruta: la borro y falta una menos */
    if (mapa[posY][posX] == 'F') {
        mapa[posY][posX] = ' ';
        numFrutas --;
        if (numFrutas == 0) {
            textout(screen, font,
                "Ganaste!", 100, 90, palette_color[14]);
            terminado = TRUE;
        }
    }

    /* Si choco con la pared, se acabo */
    if (mapa[posY][posX] == 'X') {
        textout(screen, font,
            "Chocaste!", 100, 90, palette_color[13]);
        terminado = TRUE;
    }

    if (terminado) break;

    /* Compruebo si se ha pulsado alguna tecla */
    if ( keypressed() ) {
        tecla = readkey() >> 8;

        switch (tecla) {
            case TEC_ARRIBA:
                incX = 0; incY = -1; break;
            case TEC_ABAJO:
                incX = 0; incY = 1; break;
            case TEC_IZQDA:
                incX = -1; incY = 0; break;
            case TEC_DCHA:
                incX = 1; incY = 0; break;
        }

    }

    posX += incX;
    posY += incY;

    /* Pequeña pausa antes de seguir */
    rest ( PAUSA );

}
while (TRUE); /* Repetimos indefinidamente */
/* (la condición de salida la comprobamos "dentro") */

readkey();
return 0;

}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```


7.3. Miniserpiente 1 en Pascal.

El único comentario es que en Free Pascal no existen rutinas incorporadas para manejo de Sprites. Se podrían imitar usando las órdenes "getImage" y "putImage", de la librería Graph, pero nosotros lo haremos directamente imitando la idea básica del funcionamiento de un sprite: para cada punto de la figura, se dibuja dicho punto en caso de que no sea transparente (color 0). Además lo haremos en Pascal puro, sin usar ni siquiera lenguaje ensamblador, que haría nuestro programa más rápido pero también menos legible. El resultado es que nuestro programa será mucho más lento que si tuviéramos rutinas preparadas para manejar Sprites o si empleáramos ensamblador, a cambio de que sea muy fácil de entender.

```
(*-----*)
(*  Intro a la programac de  *)
(*  juegos, por Nacho Cabanes *)
(*  *)
(*  ipj07p.pas                *)
(*  *)
(*  Septimo ejemplo: juego de *)
(*  'miniSerpiente'          *)
(*  *)
(*  Comprobado con:           *)
(*  - FreePascal 1.06 (Dos)   *)
(*  - FreePascal 2.0 -Windows *)
(*-----*)

uses graph, crt;
  (* Cambiar por "uses wincrt, ..." bajo Windows *)

  (* Posiciones X e Y iniciales *)
const
  POS_X_INI = 17;
  POS_Y_INI = 11;

  INC_X_INI = 1;
  INC_Y_INI = 0;

  (* Pausa en milisegundos entre un 'fotograma' y otro *)
  PAUSA = 350;

  (* Teclas predefinidas *)
  TEC_ARRIBA = 'E';
  TEC_ABAJO  = 'X';
  TEC_IZQDA  = 'S';
  TEC_DCHA   = 'D';

var
  posX, posY: word;      (* Posicion actual *)
  incX, incY: integer;   (* Incremento de la posicion *)

  (* Terminado: Si ha chocado o comido todas las frutas *)
  terminado: boolean;

  (* La tecla pulsada *)
  tecla: char;

  (* Escala: relacion entre tamaño de mapa y de pantalla *)
const
  ESCALA = 10;

  (* Ancho y alto de los sprites *)
  ANCHOSPRITE = 10;
  ALTOSPRITE  = 10;

  (* Y el mapa que representa a la pantalla *)
  (* Como usaremos modo grafico de 320x200 puntos *)
  (* y una escala de 10, el tablero medira 32x20 *)
  MAXFILAS = 20;
  MAXCOLS  = 32;

  mapa: array[1..MAXFILAS, 1..MAXCOLS] of char = (
    'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX',
    'X          X      X',
    'X    F          X      X',
    'X          F      X  F  X',
```

```

'X      XXXXX          X    X',
'X      X              X    X',
'X      X          X    X    X',
'X      X              X    X',
'X              X      X    X',
'X              X      X    X',
'X              X      X    X',
'X      F          X      X',
'X              X      X    X',
'X              X          F  X',
'X      X          X      X',
'X      X          X      X',
'X      X          F      X    X',
'X      F  X          X      X',
'X      X          F      X',
'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
);

const numFrutas:word = 8;

(* Nuestros sprites *)
(*BITMAP *ladrilloFondo, *comida, *jugador;###*)

type
  tipoSprite = array[1..ANCHOSPRITE,1..ALTOSPRITE] of byte;
              (* El sprite en si: matriz de 30x30 bytes *)

const spriteLadrillo: tipoSprite =
  ((0,2,2,2,2,2,2,2,0),
  (2,1,1,1,1,1,1,1,2),
  (2,1,1,1,1,1,1,1,2),
  (2,1,1,1,1,1,1,1,2),
  (2,1,1,1,1,1,1,1,2),
  (2,1,1,1,1,1,1,1,2),
  (2,1,1,1,1,1,1,1,2),
  (2,1,1,1,1,1,1,1,3,2),
  (2,1,1,1,1,1,1,1,3,3,2),
  (2,1,1,1,1,1,3,3,2,2),
  (2,2,2,2,2,2,2,2,2,0)
  );

const spriteComida: tipoSprite =
  ((0,0,0,2,0,0,0,0,0,0),

  (0,0,2,2,0,0,2,2,0,0),
  (0,4,4,4,2,2,4,4,0,0),
  (4,4,4,4,4,2,4,4,4,0),
  (4,4,4,4,4,4,4,4,4,0),
  (4,4,4,4,4,4,4,4,4,0),
  (4,4,4,4,4,4,4,4,4,0),
  (4,4,4,4,4,4,4,4,4,0),
  (4,4,4,4,4,4,4,4,4,0),
  (4,4,4,4,4,4,4,4,4,0),
  (0,4,4,4,4,4,4,4,0,0)
  );

const spriteJugador: tipoSprite =
  ((0,0,3,3,3,3,3,0,0,0),
  (0,3,1,1,1,1,1,3,0,0),
  (3,1,1,1,1,1,1,1,3,0),
  (3,1,1,1,1,1,1,1,1,3,0),
  (3,1,1,1,1,1,1,1,1,3,0),
  (3,1,1,1,1,1,1,1,1,3,0),
  (3,1,1,1,1,1,1,1,1,3,0),
  (3,1,1,1,1,1,1,1,1,3,0),
  (0,3,1,1,1,1,1,3,0,0),
  (0,0,3,3,3,3,3,0,0,0)
  );

(* ----- Rutina de dibujar sprites ----- *)
(* Simplemente dibuja un sprite definido anteriormente. *)
(* Copia la sintaxis de "draw_sprite" de Allegro, pero es *)
(* una rutina lenta, que solo usa Pascal puro, no seria *)
(* adecuada en programas con muchos sprites *)
procedure draw_sprite(imagen: tipoSprite; x, y: word);
var
  i,j: word;
begin
  for i := 1 to ANCHOSPRITE do

```

```

    for j := 1 to ALTOSPRITE do
    begin
    if imagen[j,i] <> 0 then
        putpixel(x+i-1, y+j-1, imagen[j,i]);
    end;

end;

(* ----- Rutina de dibujar el fondo ----- *)

procedure dibujaFondo;
var
    i, j: word;
begin
    clearDevice;

    for i:= 1 to MAXCOLS do
    for j := 1 to MAXFILAS do
        begin
        if mapa[j,i] = 'X' then
            draw_sprite(spriteLadrillo, (i-1)*ESCALA, (j-1)*ESCALA);
        if mapa[j,i] = 'F' then
            draw_sprite(spriteComida, (i-1)*ESCALA, (j-1)*ESCALA);
        end;

end;

(* ----- *)
(* ----- *)
(* ----- Cuerpo del programa ----- *)
var
    gd,gm, error : integer;

BEGIN
    gd := D8bit;
    gm := m320x200;
    initgraph(gd, gm, '');

                                (* Intentamos entrar a modo grafico *)
    error := graphResult;
    if error <> grOk then
        begin
        writeln('No se pudo entrar a modo grafico');
        writeln('Error encontrado: '+ graphErrorMsg(error) );
        halt(1);
        end;

(* ----- Si todo ha ido bien: empezamos *)

dibujaFondo;

        (* Valores iniciales *)
    posX := POS_X_INI;
    posY := POS_Y_INI;

    incX := INC_X_INI;
    incY := INC_Y_INI;

        (* Parte repetitiva: *)
    repeat
        dibujaFondo;
        draw_sprite (spriteJugador, (posX-1)*ESCALA, (posY-1)*ESCALA);

        terminado := FALSE;

        (* Si paso por una fruta: la borro y falta una menos *)
        if (mapa[posY,posX] = 'F') then
            begin
            mapa[posY,posX] := ' ';
            numFrutas := numFrutas - 1;
            if (numFrutas = 0) then
                begin

```

```

        setColor(14);
        outTextXY( 100, 90, 'Ganaste!' );
        terminado := TRUE;
    end;
end;

(* Si choco con la pared, se acabo *)
if (mapa[posY,posX] = 'X') then
    begin
        setColor(13);
        outTextXY( 100, 90, 'Chocaste!' );
        terminado := TRUE;
    end;

if terminado then break;

(* Compruebo si se ha pulsado alguna tecla *)
if keypressed then
    begin
        tecla := upcase(readkey);

        case tecla of
            TEC_ARRIBA:
                begin incX := 0; incY := -1; end;
            TEC_ABAJO:
                begin incX := 0; incY := 1; end;
            TEC_IZQDA:
                begin incX := -1; incY := 0; end;
            TEC_DCHA:
                begin incX := 1; incY := 0; end;
        end;

    end;

end;

posX := posX + incX;
posY := posY + incY;

(* Pequeña pausa antes de seguir *)
delay ( PAUSA );

until FALSE; (* Repetimos indefinidamente *)
              (* (la condición de salida la comprobamos 'dentro') *)

readkey;
closegraph;
end.

```

7.4. Miniserpiente 1 en Java.

Eso de que el juego no se pare... en Java es algo más complicado: deberemos usar un "Thread", un hilo, que es la parte que sí podremos parar cuando queramos, durante un cierto tiempo o por completo. Vámos a resumir los cambios más importantes:

- En la declaración de la clase deberemos añadir "implements Runnable"
- Tendremos nuevos métodos (funciones): "start" pondrá en marcha la ejecución del hilo, "stop" lo parará, y "run" indicará lo que se debe hacer durante la ejecución del hilo.
- Dentro de este "run" haremos la pausa ("sleep") que necesitamos en cada "fotograma" del juego, y redibujaremos después, así:

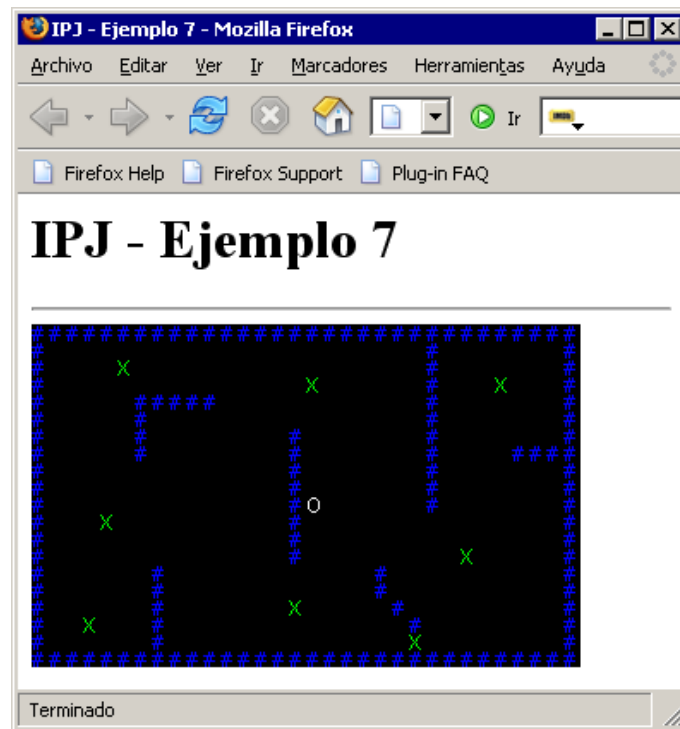
```

try {
    Thread.sleep(PAUSA);
} catch (InterruptedException e){
}

```

Los demás cambios, que no son muchos, son los debidos a la forma de trabajar de los Applets, que ya conocemos (funciones init, paint y las de manejo de teclado), y las propias del lenguaje Java (por ejemplo, va a ser más cómodo definir el mapa como un array de Strings que como un array bidimensional de caracteres).

La apariencia será casi idéntica a las anteriores:



```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* ipj07j.java */
/* */
/* Septimo ejemplo: juego de */
/* "miniSerpiente" (aprox A) */
/* */
/* Comprobado con: */
/* - JDK 1.5.0 */
/*-----*/

```

```

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

```

```

public class ipj07j extends Applet
    implements Runnable, KeyListener
{
    // Posiciones X e Y iniciales
    final int POS_X_INI = 16;
    final int POS_Y_INI = 10;

    final int INC_X_INI = 1;
    final int INC_Y_INI = 0;

    // Pausa en milisegundos entre un "fotograma" y otro
    final int PAUSA = 350;

    // Ahora las imagenes de cada elemento
    final String LADRILLO = "#";
    final String COMIDA = "X";
    final String JUGADOR = "O";

    // Escala: relacion entre tamao de mapa y de pantalla
    final int ESCALA = 10;

    // Y el mapa que representa a la pantalla
    // Como usaremos modo grafico de 320x200 puntos
    // y una escala de 10, el tablero medira 32x20
    final int MAXFILAS = 20;
    final int MAXCOLS = 32;

    String mapa[]={
        "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
        "X          X      X",
        "X    F          X      X",

```

```

"X          F      X  F  X",
"X   XXXXX      X    X",
"X    X         X    X",
"X    X          X    X",
"X    X          X  XXXX",
"X          X    X    X",
"X          X    X    X",
"X          X    X    X",
"X  F        X      X",
"X          X      X",
"X          X      F  X",
"X    X      X      X",
"X    X      X      X",
"X    X      F    X  X",
"X  F  X      X      X",
"X    X      F      X",
"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
};

Thread hilo = null;    // El "hilo" de la animacion

int posX, posY;    // Posicion actual
int incX, incY;    // Incremento de la posicion

// Terminado: Si ha chocado o comido todas las frutas
boolean terminado;

int tecla;        // La tecla pulsada

// Y las teclas por defecto
final char TEC_ARRIBA = 'e';
final char TEC_ABAJO = 'x';
final char TEC_IZQDA = 's';
final char TEC_DCHA = 'd';

int numFrutas = 8;

// Inicializacion
public void init() {

    // Valores iniciales
    posX = POS_X_INI;
    posY = POS_Y_INI;

    incX = INC_X_INI;
    incY = INC_Y_INI;

    terminado = false;

    requestFocus();
    addKeyListener(this);
}

// Escritura en pantalla
public void paint(Graphics g) {

    int i, j;

    // Primero borro el fondo en negro
    g.setColor( Color.black );
    g.fillRect( 0, 0, 639, 479 );

    // Ahora dibujo paredes y comida
    for(i=0; i<MAXCOLS; i++)
        for (j=0; j<MAXFILAS; j++) {
            g.setColor( Color.blue );
            if (mapa[j].charAt(i) == 'X')
                g.drawString(LADRILLO, i*ESCALA, j*ESCALA+10);
            g.setColor( Color.green );
            if (mapa[j].charAt(i) == 'F')
                g.drawString(COMIDA, i*ESCALA, j*ESCALA+10);
        }

    // Finalmente, el jugador
    g.setColor( Color.white );
    g.drawString(JUGADOR, posX*ESCALA, posY*ESCALA+10);

    // Si no quedan frutas, se acabo

```

```

        g.setColor( Color.yellow );
        if (numFrutas == 0) {
            g.drawString("Ganaste!", 100, 90);
            terminado = true;
        }

        // Si choco con la pared, se acabo
        g.setColor( Color.magenta );
        if (mapa[posY].charAt(posX) == 'X') {
            g.drawString("Chocaste!", 100, 90);
            terminado = true;
        }

        if (terminado) hilo=null;
    }

    // La rutina que comienza el "Thread"
    public void start() {
        hilo = new Thread(this);
        hilo.start();
    }

    // La rutina que para el "Thread"
    public synchronized void stop() {
        hilo = null;
    }

    // Y lo que hay que hacer cada cierto tiempo
    public void run() {
        Thread yo = Thread.currentThread();
        while (hilo == yo) {
            try {
                Thread.sleep(PAUSA);
            } catch (InterruptedException e){
            }
            posX += incX;
            posY += incY;

            // Si paso por una fruta: la borro y falta una menos
            if (mapa[posY].charAt(posX) == 'F') {
                // La borro en el mapa
                StringBuffer temp = new StringBuffer(mapa[posY]);
                temp.setCharAt(posX, ' ');
                mapa[posY] = temp.toString();
                // y en el contador
                numFrutas --;
            }

            // En cualquier caso, redibujo
            repaint();
        }
    }

    // Comprobacion de teclado
    public void keyTyped(KeyEvent e) {
        tecla=e.getKeyChar();
        switch (tecla) {
            case TEC_ARRIBA:
                incX = 0; incY = -1; break;
            case TEC_ABAJO:
                incX = 0; incY = 1; break;
            case TEC_IZQDA:
                incX = -1; incY = 0; break;
            case TEC_DCHA:
                incX = 1; incY = 0; break;
        }
        repaint();
        e.consume();
    }

    public void keyReleased(KeyEvent e) {
    }

    public void keyPressed(KeyEvent e) {
    }
}

```

8. Cómo crear figuras multicolor que se muevan. Cuarto juego (aproximación "b"): Miniserpiente 2.

Contenido de este apartado:

- [Ideas generales.](#)
- [Miniserpiente 2 en C.](#)
- [La versión en Pascal.](#)
- [La versión en Java.](#)

8.1. Ideas generales.

Ya sabemos como usar "Mapas" para memorizar los obstáculos y "premios" que el personaje de nuestro juego debe esquivar o alcanzar. Pero nuestros personajes son "pobres", con una presentación muy poco lucida. Va llegando el momento de hacer personajes un poco más vistosos.

Aprenderemos a crear figuras multicolor que se muevan. Además estas figuras serán "transparentes": podrán tener "huecos" alrededor o en su interior y a través de estos huecos deberá verse el fondo. Estas figuras reciben el nombre de "**sprites**".

En Allegro tenemos rutinas para el manejo de Sprites:

```
draw_sprite(screen, figura, posicionX, posicionY);
```

En Pascal también tendríamos algo parecido con la orden "putimage", pero seremos más atrevidos: imitaremos lo que haría la orden "draw_sprite" con una creada por nosotros, para que se entienda mejor el funcionamiento, a cambio de que el resultado sea algo más lento (no tendremos muchas figuras en pantalla, no será grave).

El pseudocódigo de lo que hará esta función (muy parecido a lo que haremos en Pascal) sería:

```
procedimiento dibujar_sprite(imagen, posicX, posicY);
  para i = 1 hasta ANCHOSPRITE
    para j = 1 hasta ALTOSPRITE
      si imagen[j,i] <> 0 entonces
        dibujar_punto(x+i-1, y+j-1, imagen[j,i]);
```

Simplemente dibujamos todos los puntos menos los que sean del color que hayamos considerado transparente (en nuestro caso, el 0).

Y los sprites los vamos a crear desde dentro de nuestro programa (más adelante veremos cómo leer imágenes desde ficheros). Simplemente será un array de dos dimensiones, que indicará el color de cada uno de los puntos que forman la figura:

```
spriteLadrillo =
  {{0,2,2,2,2,2,2,2,2,0},
   {2,1,1,1,1,1,1,1,1,2},
   {2,1,1,1,1,1,1,1,1,2},
   {2,1,1,1,1,1,1,1,1,2},
   {2,1,1,1,1,1,1,1,1,2},
   {2,1,1,1,1,1,1,1,1,2},
   {2,1,1,1,1,1,1,1,3,2},
   {2,1,1,1,1,1,1,3,3,2},
   {2,1,1,1,1,1,3,3,2,2},
   {2,2,2,2,2,2,2,2,2,0}
  };
```

El resto del fuente es idéntico al que hicimos en el apartado anterior. La apariencia que buscamos (algo mejor que la anterior, pero todavía nada espectacular) es

8.2 Miniserpiente 2 en C.

La única diferencia importante con la versión 1 es el manejo de los sprites:

```

/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*  ipj08c.c                      */
/*                               */
/*  Octavo ejemplo: juego de  */
/*  "miniSerpiente" (aprox B) */
/*                               */
/*  Comprobado con:              */
/*  - MinGW DevStudio 2.05      */
/*    (gcc 3.4.2) y Allegro    */
/*    4.03, Windows XP        */
/*-----*/

#include <allegro.h>

/* Posiciones X e Y iniciales */
#define POS_X_INI 16
#define POS_Y_INI 10

#define INC_X_INI 1
#define INC_Y_INI 0

/* Pausa en milisegundos entre un "fotograma" y otro */
#define PAUSA 350

/* Teclas predefinidas */
#define TEC_ARRIBA KEY_E
#define TEC_ABAJO KEY_X
#define TEC_IZQDA KEY_S
#define TEC_DCHA KEY_D

int posX, posY; /* Posicion actual */
int incX, incY; /* Incremento de la posicion */

/* Terminado: Si ha chocado o comido todas las frutas */
int terminado;

/* La tecla pulsada */
int tecla;

/* Escala: relacion entre tamaño de mapa y de pantalla */
#define ESCALA 10

/* Ancho y alto de los sprites */
#define ANCHOSPRITE 10
#define ALTOSPRITE 10

/* Y el mapa que representa a la pantalla */
/* Como usaremos modo grafico de 320x200 puntos */
/* y una escala de 10, el tablero medira 32x20 */
#define MAXFILAS 20
#define MAXCOLS 33

char mapa[MAXFILAS][MAXCOLS]={
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
    "X                               X       X",
    "X   F                           X       X",
    "X           F                   X   F   X",
    "X   XXXXX                       X       X",
    "X   X                             X       X",
    "X   X   X                       X       X",
    "X   X   X   X                   X   XXXX",
    "X           X                   X       X",
    "X           X                   X       X",
    "X           X                   X       X",
    "X   F   X                       X       X",
    "X           X                   X       X",
    "X           X                   X   F   X",
    "X           X                   X       X",
    "X   X           X               X       X",
    "X   X           X               X       X",
    "X   X   F   X                   X       X",
    "X   F   X           X           X       X",
    "X   X           X   F           X       X",
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
}

```

```

};

int numFrutas = 8;

/* Nuestros sprites */
BITMAP *ladrilloFondo, *comida, *jugador;

typedef
char tipoSprite[ANCHOSPRITE][ALTOSPRITE];
/* El sprite en si: matriz de 10x10 bytes */

tipoSprite spriteLadrillo =
{{0,2,2,2,2,2,2,2,2,0},
 {2,1,1,1,1,1,1,1,1,2},
 {2,1,1,1,1,1,1,1,1,2},
 {2,1,1,1,1,1,1,1,1,2},
 {2,1,1,1,1,1,1,1,1,2},
 {2,1,1,1,1,1,1,1,3,2},
 {2,1,1,1,1,1,1,3,3,2},
 {2,1,1,1,1,1,3,3,2,2},
 {2,2,2,2,2,2,2,2,2,0}
};

tipoSprite spriteComida =
{{0,0,0,2,0,0,0,0,0,0},
 {0,0,2,2,0,0,2,2,0,0},
 {0,4,4,4,2,2,4,4,0,0},
 {4,4,4,4,4,2,4,4,4,0},
 {4,4,4,4,4,4,4,4,4,0},
 {4,4,4,4,4,4,4,4,4,0},
 {4,4,4,4,4,4,4,4,4,0},
 {4,4,4,4,4,4,4,4,4,0},
 {0,4,4,4,4,4,4,4,0,0}
};

tipoSprite spriteJugador =
{{0,0,3,3,3,3,3,0,0,0},
 {0,3,1,1,1,1,1,3,0,0},
 {3,1,1,1,1,1,1,1,3,0},
 {3,1,1,1,1,1,1,1,1,3,0},
 {3,1,1,1,1,1,1,1,1,3,0},
 {3,1,1,1,1,1,1,1,1,3,0},
 {0,3,1,1,1,1,1,3,0,0},
 {0,0,3,3,3,3,3,0,0,0}
};

/* ----- Rutina de crear los sprites ----- */

void creaSprites()
{
    int i, j;

    ladrilloFondo = create_bitmap(10, 10);
    clear_bitmap(ladrilloFondo);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(ladrilloFondo, i, j,
                palette_color[ spriteLadrillo[j][i] ]);

    comida = create_bitmap(10, 10);
    clear_bitmap(comida);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(comida, i, j,
                palette_color[ spriteComida[j][i] ]);

    jugador = create_bitmap(10, 10);
    clear_bitmap(jugador);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(jugador, i, j,
                palette_color[ spriteJugador[j][i] ]);
}

/* ----- Rutina de dibujar el fondo ----- */

void dibujaFondo()

```

```

{
    int i, j;

    clear_bitmap(screen);

    for(i=0; i<MAXCOLS; i++)
    for (j=0; j<MAXFILAS; j++) {
        if (mapa[j][i] == 'X')
            draw_sprite(screen, ladrilloFondo, i*ESCALA, j*ESCALA);
        if (mapa[j][i] == 'F')
            draw_sprite(screen, comida, i*ESCALA, j*ESCALA);
    }
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    allegro_init();           /* Inicializamos Allegro */
    install_keyboard();
    install_timer();

                                /* Intentamos entrar a modo grafico */
    if (set_gfx_mode(GFX_SAFE, 320, 200, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    /* ----- Si todo ha ido bien: empezamos */

    creaSprites();
    dibujaFondo();

        /* Valores iniciales */
    posX = POS_X_INI;
    posY = POS_Y_INI;

    incX = INC_X_INI;
    incY = INC_Y_INI;

        /* Parte repetitiva: */
    do {
        dibujaFondo();
        draw_sprite (screen, jugador, posX*ESCALA, posY*ESCALA);

        terminado = FALSE;

        /* Si paso por una fruta: la borro y falta una menos */
        if (mapa[posY][posX] == 'F') {
            mapa[posY][posX] = ' ';
            numFrutas --;
            if (numFrutas == 0) {
                textout(screen, font,
                    "Ganaste!", 100, 90, palette_color[14]);
                terminado = TRUE;
            }
        }

        /* Si choco con la pared, se acabo */
        if (mapa[posY][posX] == 'X') {
            textout(screen, font,
                "Chocaste!", 100, 90, palette_color[13]);
            terminado = TRUE;
        }

        if (terminado) break;

        /* Compruebo si se ha pulsado alguna tecla */
        if ( keypressed() ) {
            tecla = readkey() >> 8;

```

```

switch (tecla) {
    case TEC_ARRIBA:
        incX = 0; incY = -1; break;
    case TEC_ABAJO:
        incX = 0; incY = 1; break;
    case TEC_IZQDA:
        incX = -1; incY = 0; break;
    case TEC_DCHA:
        incX = 1; incY = 0; break;
}

}

posX += incX;
posY += incY;

/* Pequeña pausa antes de seguir */
rest ( PAUSA );

}
while (TRUE); /* Repetimos indefinidamente */
/* (la condición de salida la comprobamos "dentro") */

readkey();
return 0;

}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

8.3. Miniserpiente 2 en Pascal.

Similar, pero con la rutina de dibujar sprites desarrollada, y con la misma apariencia:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* ipj08c.c */
/* Octavo ejemplo: juego de */
/* "miniSerpiente" (aprox B) */
/* Comprobado con: */
/* - MinGW DevStudio 2.05 */
/* (gcc 3.4.2) y Allegro */
/* 4.03, Windows XP */
/*-----*/

#include <allegro.h>

/* Posiciones X e Y iniciales */
#define POS_X_INI 16
#define POS_Y_INI 10

#define INC_X_INI 1
#define INC_Y_INI 0

/* Pausa en milisegundos entre un "fotograma" y otro */
#define PAUSA 350

/* Teclas predefinidas */
#define TEC_ARRIBA KEY_E
#define TEC_ABAJO KEY_X
#define TEC_IZQDA KEY_S
#define TEC_DCHA KEY_D

```

```

int posX, posY; /* Posicion actual */
int incX, incY; /* Incremento de la posicion */

/* Terminado: Si ha chocado o comido todas las frutas */
int terminado;

/* La tecla pulsada */
int tecla;

/* Escala: relacion entre tamaño de mapa y de pantalla */
#define ESCALA 10

/* Ancho y alto de los sprites */
#define ANCHOSPRITE 10
#define ALTOSPRITE 10

/* Y el mapa que representa a la pantalla */
/* Como usaremos modo grafico de 320x200 puntos */
/* y una escala de 10, el tablero medira 32x20 */
#define MAXFILAS 20
#define MAXCOLS 33

char mapa[MAXFILAS][MAXCOLS]={
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
    "X                X      X",
    "X   F                X      X",
    "X           F      X   F   X",
    "X   XXXXX          X      X",
    "X   X              X      X",
    "X   X      X      X      X",
    "X   X      X      X   XXXX",
    "X           X      X      X",
    "X           X      X      X",
    "X           X      X      X",
    "X   F      X      X      X",
    "X           X      X      X",
    "X           X      X      F   X",
    "X           X      X      X",
    "X           X      X      X",
    "X   X      F      X      X",
    "X   F   X      X      X",
    "X   X      F      X      X",
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
};

int numFrutas = 8;

/* Nuestros sprites */
BITMAP *ladrilloFondo, *comida, *jugador;

typedef
    char tipoSprite[ANCHOSPRITE][ALTOSPRITE];
    /* El sprite en si: matriz de 10x10 bytes */

tipoSprite spriteLadrillo =
    {{0,2,2,2,2,2,2,2,2,0},
    {2,1,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,3,2},
    {2,1,1,1,1,1,1,1,3,3,2},
    {2,1,1,1,1,1,1,3,3,2,2},
    {2,2,2,2,2,2,2,2,2,0}
    };

tipoSprite spriteComida =
    {{0,0,0,2,0,0,0,0,0,0},
    {0,0,2,2,0,0,2,2,0,0},
    {0,4,4,4,2,2,4,4,0,0},
    {4,4,4,4,4,2,4,4,4,0},
    {4,4,4,4,4,4,4,4,4,0},
    {4,4,4,4,4,4,4,4,4,0},
    {4,4,4,4,4,4,4,4,4,0},
    {4,4,4,4,4,4,4,4,4,0},
    {0,4,4,4,4,4,4,4,0,0}
    };

```

```

tipoSprite spriteJugador =
    {{0,0,3,3,3,3,3,0,0,0},
     {0,3,1,1,1,1,1,3,0,0},
     {3,1,1,1,1,1,1,1,3,0},
     {3,1,1,1,1,1,1,1,3,0},
     {3,1,1,1,1,1,1,1,3,0},
     {3,1,1,1,1,1,1,1,3,0},
     {3,1,1,1,1,1,1,1,3,0},
     {0,3,1,1,1,1,1,3,0,0},
     {0,0,3,3,3,3,3,0,0,0}
    };

/* ----- Rutina de crear los sprites ----- */

void creaSprites()
{
    int i, j;

    ladrilloFondo = create_bitmap(10, 10);
    clear_bitmap(ladrilloFondo);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(ladrilloFondo, i, j,
                    palette_color[ spriteLadrillo[j][i] ]);

    comida = create_bitmap(10, 10);
    clear_bitmap(comida);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(comida, i, j,
                    palette_color[ spriteComida[j][i] ]);

    jugador = create_bitmap(10, 10);
    clear_bitmap(jugador);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(jugador, i, j,
                    palette_color[ spriteJugador[j][i] ]);
}

/* ----- Rutina de dibujar el fondo ----- */

void dibujaFondo()
{
    int i, j;

    clear_bitmap(screen);

    for(i=0; i<MAXCOLS; i++)
        for (j=0; j<MAXFILAS; j++) {
            if (mapa[j][i] == 'X')
                draw_sprite(screen, ladrilloFondo, i*ESCALA, j*ESCALA);
            if (mapa[j][i] == 'F')
                draw_sprite(screen, comida, i*ESCALA, j*ESCALA);
        }
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    allegro_init();                /* Inicializamos Allegro */
    install_keyboard();
    install_timer();

                                /* Intentamos entrar a modo grafico */
    if (set_gfx_mode(GFX_SAFE, 320, 200, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
    }
}

```

```

    return 1;
}

/* ----- Si todo ha ido bien: empezamos */

creaSprites();
dibujaFondo();

    /* Valores iniciales */
posX = POS_X_INI;
posY = POS_Y_INI;

incX = INC_X_INI;
incY = INC_Y_INI;

    /* Parte repetitiva: */
do {
    dibujaFondo();
    draw_sprite (screen, jugador, posX*ESCALA, posY*ESCALA);

    terminado = FALSE;

    /* Si paso por una fruta: la borro y falta una menos */
    if (mapa[posY][posX] == 'F') {
        mapa[posY][posX] = ' ';
        numFrutas --;
        if (numFrutas == 0) {
            textout(screen, font,
                "Ganaste!", 100, 90, palette_color[14]);
            terminado = TRUE;
        }
    }

    /* Si choco con la pared, se acabo */
    if (mapa[posY][posX] == 'X') {
        textout(screen, font,
            "Chocaste!", 100, 90, palette_color[13]);
        terminado = TRUE;
    }

    if (terminado) break;

    /* Compruebo si se ha pulsado alguna tecla */
    if (keypressed() ) {
        tecla = readkey() >> 8;

        switch (tecla) {
            case TEC_ARRIBA:
                incX = 0; incY = -1; break;
            case TEC_ABAJO:
                incX = 0; incY = 1; break;
            case TEC_IZQDA:
                incX = -1; incY = 0; break;
            case TEC_DCHA:
                incX = 1; incY = 0; break;
        }

    }

    posX += incX;
    posY += incY;

    /* Pequeña pausa antes de seguir */
    rest ( PAUSA );

}
while (TRUE); /* Repetimos indefinidamente */
/* (la condición de salida la comprobamos "dentro") */

readkey();
return 0;

}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

8.4. Miniserpiente 2 en Java.

Al igual que en las versiones en C y Pascal, los únicos cambios en Java se refieren a que en pantalla dibujaremos un Sprite en vez de una letra y a que debemos crear esos Sprites.

De hecho, en Java tenemos incluso un componente llamado **Image** (imagen), que es el equivalente directo de nuestro Sprite. La única complicación es que resulta muy fácil crear un Image a partir de un fichero de imagen, pero es bastante más difícil crearlo a partir de un "array", como hemos hecho en el caso de C y Pascal. Aun así, veamos los pasos:

Crearemos un "array" que contenga la información de los colores de cada punto:

```
int spriteLadrillo[] =
{0,2,2,2,2,2,2,2,2,0,
 2,1,1,1,1,1,1,1,1,2,
 2,1,1,1,1,1,1,1,1,2,
 2,1,1,1,1,1,1,1,1,2,
 2,1,1,1,1,1,1,1,1,2,
 2,1,1,1,1,1,1,1,1,2,
 2,1,1,1,1,1,1,1,3,2,
 2,1,1,1,1,1,1,3,3,2,
 2,1,1,1,1,1,3,3,2,2,
 2,2,2,2,2,2,2,2,2,0
};
```

Pero en Java no existe "paleta de colores Pc estándar" ;-), así que tendremos que convertir esos colores antes de pasarlos al objeto Image. Por ejemplo, el color 1 queremos que sea azul, así que tendremos que darle el nuevo valor $(255 \ll 24) | 255$ (son 4 bytes, el primero es el canal Alpha, que deberá ser 255 para que nuestra figura sea totalmente opaca; los tres siguientes son RGB (rojo, verde y azul, en ese orden), en nuestro caso 0, 0,255. De igual modo, el color 2, que será el verde, debería ser $(255 \ll 24) | (255 \ll 8)$, el color 4, rojo, sería $(255 \ll 24) | (255 \ll 16)$, y el color 3, un azul claro (cyan) podría ser algo como $(255 \ll 24) | (127 \ll 16) | (127 \ll 8) | 255$.

Por tanto, para "corregir" los colores de cada punto haríamos algo como:

```
for(i=0; i<ANCHOSPRITE; i++)
  for (j=0; j<ALTOSPRITE; j++)
  {
    // El color 1 sera azul
    if (spriteLadrillo[i+j*ANCHOSPRITE]==1)
      spriteLadrillo[i+j*ANCHOSPRITE] = (255 << 24) | 255;
    // El color 2 sera verde
    if (spriteLadrillo[i+j*ANCHOSPRITE]==2)
      spriteLadrillo[i+j*ANCHOSPRITE] = (255 << 24) | (255 << 8);
    // El color 3 sera cyan
    if (spriteLadrillo[i+j*ANCHOSPRITE]==3)
      spriteLadrillo[i+j*ANCHOSPRITE] = (255 << 24) | (127 << 16) | (127 << 8) | 255;
    // El color 4 sera rojo
    if (spriteLadrillo[i+j*ANCHOSPRITE]==4)
      spriteLadrillo[i+j*ANCHOSPRITE] = (255 << 24) | (255 << 16);
```

Y ya sólo nos queda pasar toda esa información al componente "Image", que se hace usando un "MemoryImageSource", al que le indicamos el ancho y el alto y desde dónde queremos leer los datos:

```
ladrilloFondo = createImage(new MemoryImageSource(ANCHOSPRITE, ALTOSPRITE,
  spriteLadrillo, 0, ANCHOSPRITE));
```

Para dibujar sí que no hay complicación

```
if (mapa[j].charAt(i) == 'X')
  g.drawImage(ladrilloFondo, i*ESCALA, j*ESCALA, this);
```

La apariencia sería esta

Y el fuente podría ser:


```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* ipj08j.java */
/* */
/* Octavo ejemplo: juego de */
/* "miniSerpiente" (aprox B) */
/* */
/* Comprobado con: */
/* - JDK 1.5.0 */
/*-----*/

import java.applet.Applet;
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;

public class ipj08j extends Applet
    implements Runnable, KeyListener
{
    // Posiciones X e Y iniciales
    final int POS_X_INI = 16;
    final int POS_Y_INI = 10;

    final int INC_X_INI = 1;
    final int INC_Y_INI = 0;

    // Pausa en milisegundos entre un "fotograma" y otro
    final int PAUSA = 350;

    // Ahora las imagenes de cada elemento
    Image ladrilloFondo;
    Image comida;
    Image jugador;

    // Escala: relacion entre tamaño de mapa y de pantalla
    final int ESCALA = 10;

    // Y el mapa que representa a la pantalla
    // Como usaremos modo grafico de 320x200 puntos
    // y una escala de 10, el tablero medira 32x20
    final int MAXFILAS = 20;
    final int MAXCOLS = 32;

    String mapa[]={
        "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
        "X          X          X",
        "X    F          X    X",
        "X      F          X  F  X",
        "X    XXXXX          X    X",
        "X    X            X    X",
        "X    X      X      X    X",
        "X    X      X      X    XXXX",
        "X          X      X    X",
        "X          X      X    X",
        "X          X      X    X",
        "X    F      X      X    X",
        "X          X      X    X",
        "X          X      F    X",
        "X    X      X      X    X",
        "X    X      X      X    X",
        "X    X      F      X    X",
        "X  F  X      X      X    X",
        "X    X      F      X    X",
        "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
    };

    /* Ancho y alto de los sprites */
    final int ANCHOSPRITE = 10;
    final int ALTOSPRITE = 9;

    int spriteLadrillo[] =
    {0,2,2,2,2,2,2,2,2,0,
     2,1,1,1,1,1,1,1,1,2,
     2,1,1,1,1,1,1,1,1,2,
     2,1,1,1,1,1,1,1,1,2,
     2,1,1,1,1,1,1,1,1,2,

```

```

    2,1,1,1,1,1,1,1,1,2,
    2,1,1,1,1,1,1,1,3,2,
    2,1,1,1,1,1,1,3,3,2,
    2,1,1,1,1,1,3,3,2,2,
    2,2,2,2,2,2,2,2,0
};

int spriteComida []=
{0,0,0,2,0,0,0,0,0,0,
0,0,2,2,0,0,2,2,0,0,
0,4,4,4,2,2,4,4,0,0,
4,4,4,4,4,2,4,4,4,0,
4,4,4,4,4,4,4,4,4,0,
4,4,4,4,4,4,4,4,4,0,
4,4,4,4,4,4,4,4,4,0,
4,4,4,4,4,4,4,4,4,0,
4,4,4,4,4,4,4,4,4,0,
0,4,4,4,4,4,4,4,4,0
};

int spriteJugador []=
{0,0,3,3,3,3,0,0,0,
0,3,1,1,1,1,1,3,0,0,
3,1,1,1,1,1,1,1,3,0,
3,1,1,1,1,1,1,1,3,0,
3,1,1,1,1,1,1,1,3,0,
3,1,1,1,1,1,1,1,3,0,
3,1,1,1,1,1,1,1,3,0,
0,3,1,1,1,1,1,3,0,0,
0,0,3,3,3,3,0,0,0
};

Thread hilo = null; // El "hilo" de la animacion

int posX, posY; // Posicion actual
int incX, incY; // Incremento de la posicion

// Terminado: Si ha chocado o comido todas las frutas
boolean terminado;

int tecla; // La tecla pulsada

// Y las teclas por defecto
final char TEC_ARRIBA = 'e';
final char TEC_ABAJO = 'x';
final char TEC_IZQDA = 's';
final char TEC_DCHA = 'd';

int numFrutas = 8;

// Rutina de crear los sprites
void creaSprites()
{
int i, j;

// Doy colores mÃ¡s adecuados a cada figura: ladrillos
for(i=0; i<ANCHOSPRITE; i++)
    for (j=0; j<ALTOSPRITE; j++)
    {
        // El color 1 sera azul
        if (spriteLadrillo[i+j*ANCHOSPRITE]==1)
            spriteLadrillo[i+j*ANCHOSPRITE] = (255 << 24) | 255;
        // El color 2 sera verde
        if (spriteLadrillo[i+j*ANCHOSPRITE]==2)
            spriteLadrillo[i+j*ANCHOSPRITE] = (255 << 24) | (255 << 8);
        // El color 3 sera cyan
        if (spriteLadrillo[i+j*ANCHOSPRITE]==3)
            spriteLadrillo[i+j*ANCHOSPRITE] = (255 << 24) | (127 << 16) | (127 << 8) | 255;
        // El color 4 sera rojo
        if (spriteLadrillo[i+j*ANCHOSPRITE]==4)
            spriteLadrillo[i+j*ANCHOSPRITE] = (255 << 24) | (255 << 16);
    }
// y al final los asigno
ladrilloFondo = createImage(new MemoryImageSource(ANCHOSPRITE, ALTOSPRITE,
    spriteLadrillo, 0, ANCHOSPRITE));

// Lo mismo para la comida
for(i=0; i<ANCHOSPRITE; i++)
    for (j=0; j<ALTOSPRITE; j++)
    {

```

```

    // El color 1 sera azul
    if (spriteComida[i+j*ANCHOSPRITE]==1)
        spriteComida[i+j*ANCHOSPRITE] = (255 << 24) | 255;
    // El color 2 sera verde
    if (spriteComida[i+j*ANCHOSPRITE]==2)
        spriteComida[i+j*ANCHOSPRITE] = (255 << 24) | (255 << 8);
    // El color 3 sera cyan
    if (spriteComida[i+j*ANCHOSPRITE]==3)
        spriteComida[i+j*ANCHOSPRITE] = (255 << 24) | (127 << 16) | (127 << 8) | 255;
    // El color 4 sera rojo
    if (spriteComida[i+j*ANCHOSPRITE]==4)
        spriteComida[i+j*ANCHOSPRITE] = (255 << 24) | (255 << 16);
}
comida = createImage(new MemoryImageSource(ANCHOSPRITE, ALTOSPRITE,
    spriteComida, 0, ANCHOSPRITE));

// Y lo mismo para el jugador
for(i=0; i<ANCHOSPRITE; i++)
    for (j=0; j<ALTOSPRITE; j++)
    {
        // El color 1 sera azul
        if (spriteJugador[i+j*ANCHOSPRITE]==1)
            spriteJugador[i+j*ANCHOSPRITE] = (255 << 24) | 255;
        // El color 2 sera verde
        if (spriteJugador[i+j*ANCHOSPRITE]==2)
            spriteJugador[i+j*ANCHOSPRITE] = (255 << 24) | (255 << 8);
        // El color 3 sera cyan
        if (spriteJugador[i+j*ANCHOSPRITE]==3)
            spriteJugador[i+j*ANCHOSPRITE] = (255 << 24) | (127 << 16) | (127 << 8) | 255;
        // El color 4 sera rojo
        if (spriteJugador[i+j*ANCHOSPRITE]==4)
            spriteJugador[i+j*ANCHOSPRITE] = (255 << 24) | (255 << 16);
    }
jugador = createImage(new MemoryImageSource(ANCHOSPRITE, ALTOSPRITE,
    spriteJugador, 0, ANCHOSPRITE));
}

// Inicializacion
public void init() {

    // Valores iniciales
    posX = POS_X_INI;
    posY = POS_Y_INI;

    incX = INC_X_INI;
    incY = INC_Y_INI;

    terminado = false;

    requestFocus();
    addKeyListener(this);
    creaSprites();
}

// Escritura en pantalla
public void paint(Graphics g) {

    int i, j;

    // Primero borro el fondo en negro
    g.setColor( Color.black );
    g.fillRect( 0, 0, 639, 479 );

    // Ahora dibujo paredes y comida
    for(i=0; i<MAXCOLS; i++)
        for (j=0; j<MAXFILAS; j++) {
            if (mapa[j].charAt(i) == 'X')
                g.drawImage(ladrilloFondo, i*ESCALA, j*ESCALA, this);
            if (mapa[j].charAt(i) == 'F')
                g.drawImage(comida, i*ESCALA, j*ESCALA, this);
        }

    // Finalmente, el jugador
    g.drawImage(jugador, posX*ESCALA, posY*ESCALA, this);

    // Si no quedan frutas, se acabo
    g.setColor( Color.yellow );

```

```

    if (numFrutas == 0) {
        g.drawString("Ganaste!", 100, 90);
        terminado = true;
    }

    // Si choco con la pared, se acabo
    g.setColor( Color.magenta );
    if (mapa[posY].charAt(posX) == 'X') {
        g.drawString("Chocaste!", 100, 90);
        terminado = true;
    }

    if (terminado) hilo=null;
}

// La rutina que comienza el "Thread"
public void start() {
hilo = new Thread(this);
hilo.start();
}

// La rutina que para el "Thread"
public synchronized void stop() {
hilo = null;
}

// Y lo que hay que hacer cada cierto tiempo
public void run() {
    Thread yo = Thread.currentThread();
    while (hilo == yo) {
        try {
            Thread.sleep(PAUSA);
        } catch (InterruptedException e){
        }
        posX += incX;
        posY += incY;

        // Si paso por una fruta: la borro y falta una menos
        if (mapa[posY].charAt(posX) == 'F') {
            // La borro en el mapa
            StringBuffer temp = new StringBuffer(mapa[posY]);
            temp.setCharAt(posX, ' ');
            mapa[posY] = temp.toString();
            // y en el contador
            numFrutas --;
        }

        // En cualquier caso, redibujo
        repaint();
    }
}

// Comprobacion de teclado
public void keyTyped(KeyEvent e) {
    tecla=e.getKeyChar();
    switch (tecla) {
        case TEC_ARRIBA:
            incX = 0; incY = -1; break;
        case TEC_ABAJO:
            incX = 0; incY = 1; break;
        case TEC_IZQDA:
            incX = -1; incY = 0; break;
        case TEC_DCHA:
            incX = 1; incY = 0; break;
    }
    repaint();
    e.consume();
}

public void keyReleased(KeyEvent e) {
}

public void keyPressed(KeyEvent e) {
}
}

```

9. Evitemos los parpadeos. Cuarto juego (aproximación "c"): Miniserpiente 3.

Contenido de este apartado:

- [Idea básica: el doble buffer.](#)
- [Miniserpiente 3 en C.](#)
- [La versión en Pascal.](#)
- [La versión en Java.](#)

9.1. Idea básica: el doble buffer.

Nuestros juegos, a pesar de que aún son sencillos, empiezan a tener problemas de parpadeos. Podríamos sincronizar el refresco de la pantalla con el momento de dibujar nuestras figuras, y el problema se reduciría un poco, pero seguiría existiendo. Y es tanto más grave cuantos más elementos dibujamos en pantalla (especialmente en el caso de Pascal, en que hemos usado una rutina "nuestra" para dibujar los sprites, en vez de rutinas "prefabricadas").

Por eso, en la práctica se suele hacer una aproximación ligeramente distinta a la que hemos usado hasta ahora: no escribiremos directamente en pantalla, sino que prepararemos todo lo que va a aparecer, lo dibujaremos en una "pantalla no visible" y en el último momento volcaremos toda esta "pantalla no visible" a la vez. Esta técnica es lo que se conoce como el empleo de un **"doble buffer"** (en inglés "double buffering").

En informática, un "buffer" es una memoria intermedia en la que se guarda la información antes de llegar a su destino definitivo. Un ejemplo típico es el "buffer" de impresora.

En nuestro caso, esto nos permite preparar toda la información tranquilamente, en un sitio que todavía no será su destino definitivo. Cuando todo está listo, es el momento en el que se vuelca a pantalla, todo a la vez, minimizando los parpadeos.

La forma de conseguir esto con Allegro es algo parecido a:

```
BITMAP *bmp = create_bitmap(320, 200); // Creamos el bitmap auxiliar en memoria
clear_bitmap(bmp); // Lo borramos
putpixel(bmp, x, y, color); // Dibujamos en él
blit(bmp, screen, 0, 0, 0, 0, 320, 200); // Finalmente, volcamos a pantalla
```

Los cambios en nuestro programa no son grandes: creamos un nuevo bitmap llamado "pantallaOculta", dibujamos en él la nave, los marcianos y el disparo, volcamos a pantalla al final de cada secuencia completa de dibujado, y esta vez no nos molestamos en esperar a que realmente haya "información nueva" en pantalla: volvemos a dibujar aunque realmente sea lo mismo que ya había antes. En cualquier caso, ahora no debería haber parpadeos o estos deberían ser mínimos.

9.2 Miniserpiente 3 en C.

Pocos cambios...

```
/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* ipj09c.c */
/* */
/* Noveno ejemplo: juego de */
/* "miniSerpiente" (aprox C) */
/* */
/* Comprobado con: */
/* - MinGW DevStudio 2.05 */
/* (gcc 3.4.2) y Allegro */
/* 4.03, Windows XP */
/*-----*/

#include <allegro.h>
```

```

    /* Posiciones X e Y iniciales */
#define POS_X_INI 16
#define POS_Y_INI 10

#define INC_X_INI 1
#define INC_Y_INI 0

    /* Pausa en milisegundos entre un "fotograma" y otro */
#define PAUSA 350

    /* Teclas predefinidas */
#define TEC_ARRIBA KEY_E
#define TEC_ABAJO KEY_X
#define TEC_IZQDA KEY_S
#define TEC_DCHA KEY_D

int posX, posY; /* Posicion actual */
int incX, incY; /* Incremento de la posicion */

    /* Terminado: Si ha chocado o comido todas las frutas */
int terminado;

    /* La tecla pulsada */
int tecla;

    /* Escala: relacion entre tamaño de mapa y de pantalla */
#define ESCALA 10

    /* Ancho y alto de los sprites */
#define ANCHOSPRITE 10
#define ALTOSPRITE 10

    /* Y el mapa que representa a la pantalla */
    /* Como usaremos modo grafico de 320x200 puntos */
    /* y una escala de 10, el tablero medira 32x20 */
#define MAXFILAS 20
#define MAXCOLS 33

char mapa[MAXFILAS][MAXCOLS]={
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
    "X X X",
    "X F X",
    "X F X",
    "X XXXXX X",
    "X X X",
    "X X X",
    "X X X XXXX",
    "X X X",
    "X X X",
    "X X X",
    "X F X",
    "X X X",
    "X X F X",
    "X X X",
    "X X X",
    "X F X X",
    "X X F X",
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
};

int numFrutas = 8;

    /* Nuestros sprites */
BITMAP *ladrilloFondo, *comida, *jugador;
BITMAP *pantallaOculta; /* Y la pantalla oculta */

typedef
    char tipoSprite[ANCHOSPRITE][ALTOSPRITE];
    /* El sprite en si: matriz de 10x10 bytes */

tipoSprite spriteLadrillo =
    {{0,2,2,2,2,2,2,2,2,0},
    {2,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,3,2},
    }}

```

```

    {2,1,1,1,1,1,1,3,3,2},
    {2,1,1,1,1,1,3,3,2,2},
    {2,2,2,2,2,2,2,2,2,0}
};

tipoSprite spriteComida =
    {{0,0,0,2,0,0,0,0,0,0},
    {0,0,2,2,0,0,2,2,0,0},
    {0,4,4,4,2,2,4,4,0,0},
    {4,4,4,4,4,2,4,4,4,0},
    {4,4,4,4,4,4,4,4,4,0},
    {4,4,4,4,4,4,4,4,4,0},
    {4,4,4,4,4,4,4,4,4,0},
    {4,4,4,4,4,4,4,4,4,0},
    {4,4,4,4,4,4,4,4,4,0},
    {0,4,4,4,4,4,4,4,0,0}
};

tipoSprite spriteJugador =
    {{0,0,3,3,3,3,3,0,0,0},
    {0,3,1,1,1,1,1,3,0,0},
    {3,1,1,1,1,1,1,1,3,0},
    {3,1,1,1,1,1,1,1,1,3,0},
    {3,1,1,1,1,1,1,1,1,3,0},
    {3,1,1,1,1,1,1,1,1,3,0},
    {3,1,1,1,1,1,1,1,1,3,0},
    {0,3,1,1,1,1,1,3,0,0},
    {0,0,3,3,3,3,3,0,0,0}
};

/* ----- Rutina de crear los sprites ----- */

void creaSprites()
{
    int i, j;

    ladrilloFondo = create_bitmap(10, 10);
    clear_bitmap(ladrilloFondo);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(ladrilloFondo, i, j,
                palette_color[ spriteLadrillo[j][i] ]);

    comida = create_bitmap(10, 10);
    clear_bitmap(comida);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(comida, i, j,
                palette_color[ spriteComida[j][i] ]);

    jugador = create_bitmap(10, 10);
    clear_bitmap(jugador);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(jugador, i, j,
                palette_color[ spriteJugador[j][i] ]);

    pantallaOculto = create_bitmap(320, 200);
}

/* ----- Rutina de dibujar el fondo ----- */

void dibujaFondo()
{
    int i, j;

    clear_bitmap(pantallaOculto);

    for(i=0; i<MAXCOLS; i++)
        for (j=0; j<MAXFILAS; j++) {
            if (mapa[j][i] == 'X')
                draw_sprite(pantallaOculto, ladrilloFondo, i*ESCALA, j*ESCALA);
            if (mapa[j][i] == 'F')
                draw_sprite(pantallaOculto, comida, i*ESCALA, j*ESCALA);
        }
}

/* ----- */
/* ----- */

```

```

/* ----- Cuerpo del programa ----- */

int main()
{
    allegro_init();           /* Inicializamos Allegro */
    install_keyboard();
    install_timer();

                                /* Intentamos entrar a modo grafico */
    if (set_gfx_mode(GFX_SAFE, 320, 200, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    /* ----- Si todo ha ido bien: empezamos */

    creaSprites();
    dibujaFondo();

                                /* Valores iniciales */
    posX = POS_X_INI;
    posY = POS_Y_INI;

    incX = INC_X_INI;
    incY = INC_Y_INI;

                                /* Parte repetitiva: */
    do {
        dibujaFondo();
        draw_sprite (pantallaOculto, jugador, posX*ESCALA, posY*ESCALA);

        terminado = FALSE;

        /* Si paso por una fruta: la borro y falta una menos */
        if (mapa[posY][posX] == 'F') {
            mapa[posY][posX] = ' ';
            numFrutas--;
            if (numFrutas == 0) {
                textout(pantallaOculto, font,
                    "Ganaste!", 100, 90, palette_color[14]);
                terminado = TRUE;
            }
        }

        /* Si choco con la pared, se acabo */
        if (mapa[posY][posX] == 'X') {
            textout(pantallaOculto, font,
                "Chocaste!", 100, 90, palette_color[13]);
            terminado = TRUE;
        }

        /* En cualquier caso, vuelco la pantalla oculta */
        blit(pantallaOculto, screen, 0, 0, 0, 0, 320, 200);

        if (terminado) break;

        /* Compruebo si se ha pulsado alguna tecla */
        if (keypressed()) {
            tecla = readkey() >> 8;

            switch (tecla) {
                case TEC_ARRIBA:
                    incX = 0; incY = -1; break;
                case TEC_ABAJO:
                    incX = 0; incY = 1; break;
                case TEC_IZQDA:
                    incX = -1; incY = 0; break;
                case TEC_DCHA:
                    incX = 1; incY = 0; break;
            }
        }

        posX += incX;

```



```

    posY += incY;

    /* Pequeña pausa antes de seguir */
    rest ( PAUSA );

}
while (TRUE); /* Repetimos indefinidamente */
/* (la condición de salida la comprobamos "dentro") */

readkey();
destroy_bitmap(pantallaOculto);
destroy_bitmap(jugador);
destroy_bitmap(comida);
destroy_bitmap(ladrilloFondo);
return 0;
}

/* Termina con la "macro" que me pide Allegro */
END_OF_MAIN();

```

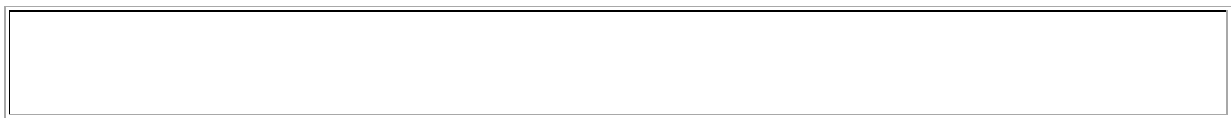
9.3. Miniserpiente 3 en Pascal.

(Todavía no disponible: la biblioteca gráfica estándar de Free Pascal no permite usar un doble buffer)



9.4. Miniserpiente 3 en Java.

(Todavía no disponible; pronto lo estará)



10. Más sobre la paleta de colores.

Hasta ahora nos hemos limitado a utilizar el color 1, el color 2, etc., tomando dichos colores de la paleta estándar de un ordenador tipo PC.

Esto es muy limitado, comparado con lo que podríamos hacer: Un ordenador actual es capaz de utilizar millones de colores simultáneamente en la pantalla, pero es que incluso un ordenador de 1990 es capaz de mostrar 256 colores simultáneos, escogidos de una paleta de más de 250.000, de modo que incluso con un ordenador bastante antiguo, podríamos estar haciendo cosas bastante más vistosas que las que estamos haciendo.

Así que ya va siendo hora de ver cómo saber a qué número corresponde cada color, de aprender a utilizar más colores de estos 256 "estándar" y de trabajar con una gama casi ilimitada de colores en un ordenador actual.

En primer lugar, veamos a qué corresponden los **números de color** habitual (la "paleta estándar" de un ordenador que siga el patrón marcado por los primeros IBM PC) :

Números de color	Equivale a
0	Negro
1	Azul
2	Verde
3	Cyan

4	Rojo
5	Violeta
6	Marrón
7	Gris claro
8	Gris oscuro
9	Azul claro
10	Verde claro
11	Cyan claro
12	Rojo claro
13	Violeta claro
14	Amarillo
15	Blanco

Estos son los 16 colores estándar en cualquier ordenador "tipo PC" que tenga pantalla a color (tarjeta gráfica CGA o superiores). Estos colores son los que se emplean en una pantalla de texto en el modo normal de 16 colores (como curiosidad, se puede observar que se trata de 8 colores básicos, con dos intensidades de brillo). Para no tener que recordar sus nombres, la mayoría de los compiladores de lenguajes como C y Pascal tienen definidas unas constantes (en inglés):

```
Black = 0;
Blue = 1;
Green = 2;
Cyan = 3;
Red = 4;
Magenta = 5;
Brown = 6;
LightGray = 7;
DarkGray = 8;
LightBlue = 9;
LightGreen = 10;
LightCyan = 11;
LightRed = 12;
LightMagenta = 13;
Yellow = 14;
White = 15;
```

Si queremos ver la apariencia de estos 16 colores y los otros 240 que forman la " **paleta estándar VGA** ", podemos crear un pequeño programita que muestre todos ellos. Por ejemplo, en 16 filas y 16 columnas, dibujando "recuadritos" de 8 puntos de ancho y 8 puntos de alto, así:

Esto lo conseguimos simplemente con una orden como esta:

```
for (i=0; i<=15; i++)
  for (j=0; j<=15; j++)

    rectfill(
      screen,
      j*10+80, i*10+20, j*10+88, i*10+28,
      palette_color[i * 16 + j]
    );
```

Además, con la mayoría de bibliotecas gráficas podemos **modificar el color** al que corresponde cada una de estas posiciones, indicando qué cantidad de rojo, verde y azul queremos que lo compongan. Tenemos 64 tonalidades posibles de cada uno de estos colores básicos, lo que nos permite utilizar nada menos que 262.144 tonos de color distintos.

Por ejemplo, podríamos cambiar el color 15, que inicialmente es el blanco, para convertirlo en un rojo puro: basta asignarle 63 de rojo, 0 de verde, 0 de azul. La orden que lo permite en los lenguajes de Borland (como Turbo C++ y Turbo Pascal) es

```
setRGBpalette ( color, r, g, b);
```

donde "color" es el número del color de la paleta que queremos cambiar (15 en nuestro caso), r es la cantidad de rojo (red), g es la cantidad de verde (green) y b es la cantidad de azul (blue). Por tanto, para convertir el color 15 en rojo puro haríamos:

```
setRGBpalette ( 15, 63, 0, 0);
```

En el caso de Allegro, la construcción es algo más complicada, aunque no mucho:

```
void set_color(int index, const RGB *p);
```

donde el tipo RGB es un tipo predefinido en Allegro:

```
typedef struct RGB
{
    unsigned char r, g, b;
} RGB;
```

de modo que podemos crear nuestros colores RGB con

```
RGB negro = { 0, 0, 0 };
RGB blanco = { 63, 63, 63 };
RGB rojo = { 63, 0, 0 };
RGB violeta = { 63, 0, 63 };
```

y entonces podríamos asignar cualquiera de estos colores a una de las posiciones de la paleta:

```
set_color(15, &violeta);
```

Una recomendación adicional es utilizar la orden "vsync()" antes de modificar un color, para sincronizar con el momento en el que se actualiza la información en pantalla, y así evitar que aparezca "nieve". Existe una variante de la orden "set_color", llamada "_set_color", que no necesita sincronizar con el barrido de la pantalla, pero a cambio sólo funciona en los modos estándar de una tarjeta gráfica VGA (320x200 puntos y algunos modos extendidos adicionales llamados "modos X") y no en los modos de alta resolución, por lo que no la usaremos.

También podemos **jugar con toda la paleta** de colores: memorizar la paleta de colores actual (para poder modificarla y recuperarla después), oscurecer la paleta, convertirla gradualmente en otra, etc.

Por ejemplo,

```
void fade_out(int velocidad);
```

Pasa gradualmente de la paleta de colores actual a una pantalla negra. La velocidad va desde 1 (la más lenta) hasta 64 (instantáneo).

```
void fade_in(const PALETTE p, int velocidad);
```

Pasa gradualmente de una pantalla negra a la paleta de colores "p". La velocidad nuevamente va desde 1 (la más lenta) hasta 64 (instantáneo).

```
void fade_from(const PALETTE p1, const PALETTE p2, int velocidad);
```

Pasa de la paleta "p1" a la paleta "p2".

En todos los casos, el tipo PALETTE está definido en Allegro, y se trata de un "array" de 256 estructuras de tipo "RGB".

Podemos usar **más de 256 colores** (si nuestra tarjeta gráfica lo permite), cambiando al modo de pantalla correspondiente y empleando la orden "putpixel" habitual. El único cambio es indicar el color a partir de sus componentes r,g,b, con:

```
int makecol(int r, int g, int b);
```

Donde r, g, b van desde 0 a 255 (y el propio Allegro se encargará de convertirlo al color "físico" correspondiente según el modo de pantalla que usemos), por ejemplo:

```
putpixel(screen, 320, 200, makecol(255, 70, 50) );
```

Por otra parte, tenemos unas variantes del putpixel normal que son específicas para cada modo de pantalla, de 8 bits (256 colores), 15 bits (32.768 colores), 16 bits (65.536 colores), 24 bits (262.144 colores) o 32 bits (lo que se suele llamar "true color" - color auténtico):

```
void _putpixel(BITMAP *bmp, int x, int y, int color);
void _putpixel15(BITMAP *bmp, int x, int y, int color);
void _putpixel16(BITMAP *bmp, int x, int y, int color);
void _putpixel24(BITMAP *bmp, int x, int y, int color);
void _putpixel32(BITMAP *bmp, int x, int y, int color);
```

Estas variantes son más rápidas que el putpixel() "normal", pero tienen varios inconvenientes:

- No comprueban si nos salimos de los límites de la pantalla (operación que se suele llamar "clipping", y que sí hace putpixel), de lo que nos deberemos encargar nosotros si no queremos que nuestro programa falle.
- No funcionan en los modos extendidos (modos X) de una tarjeta VGA estándar.
- Tampoco hacen caso al "modo de escritura" (que nosotros no hemos usado por ahora, pero que puede ser útil para conseguir ciertos efectos).

11. Cuarto juego (completo): Serpiente.

Contenido de este apartado:

- [Idea básica: cambio de paleta, serpiente creciente.](#)
- [Serpiente en C.](#)
- [La versión en Pascal.](#)
- [La versión en Java.](#)

11.1. Idea básica: cambio de paleta, serpiente creciente.

Con muy pocas modificaciones, podemos mejorar el juego de la "Mini-serpiente" que acabamos de hacer:

- Por una parte, ya sabemos cómo cambiar la paleta de colores estándar, para conseguir un apariencia algo más vistosa.
- Por otra parte, podemos hacer que la serpiente sea más larga, en vez de ocupar una única posición. Para ser más concretos, haremos que su tamaño vaya aumentando cada vez que coma una fruta.

La primera parte (cambiar colores) no debería suponer ningún problema. Por ejemplo, si queremos que los ladrillos de la pared sean de tres tonos distintos de gris, podríamos redefinir los colores 1, 2 y 3 haciendo:

```

RGB gris30 = { 30, 30, 30 };
RGB gris40 = { 40, 40, 40 };
RGB gris40verde45 = { 40, 45, 40 };

set_color( 1, &gris40 );
set_color( 2, &gris30 );
set_color( 3, &gris40verde45 );

```

Lo de cambiar el tamaño de la serpiente puede sonar más complicado, pero no lo es: podemos hacerlo de una forma bastante sencilla: definimos la serpiente como un "array", para el que reservaremos tanto espacio como el máximo que vayamos a permitir. En la posición 0 de este array memorizaremos el punto de la pantalla en el que se encuentra el primer segmento de la serpiente. Cuando coma una fruta, la longitud aumenta a 2: en la posición 0 sigue estando la "cabeza" de la serpiente, y en la posición 1 está el segundo segmento. Cuando coma otra fruta, usaremos las posiciones 0, 1 y 2 del array, y así sucesivamente.

La única dificultad es plantear cómo podemos hacer que "se mueva" toda la serpiente de manera conjuntada. Lo que haremos es que la "cabeza" de la serpiente (la posición 0) se mueva tal y como lo hacía la mini-serpiente: hacia la dirección que le indiquemos al pulsar una tecla, o continuará moviéndose en la misma dirección si no pulsamos ninguna tecla. Por lo que respecta al resto de la serpiente, irá siguiendo a la cabeza: la posición 1 pasará a ocupar el punto de pantalla que antes ocupaba la posición 0, la 2 pasará a ocupar el de la 1 y así sucesivamente. Se podría conseguir así:

```

for (i=longSerpiente-1; i>=1; i--) {
    trozoSerpiente[i].x = trozoSerpiente[i-1].x;
    trozoSerpiente[i].y = trozoSerpiente[i-1].y;
}

trozoSerpiente[0].x = posX;

```

```
trozoSerpiente[0].y = posY;
```

La apariencia que obtendremos será algo así:

Por supuesto, para llegar a un juego "de verdad" todavía queda mucho: distintos niveles de dificultad, varias "vidas", puntuación y tabla de records... pero todo irá llegando.

11.2 Serpiente en C.

Los cambios con la versión anterior no son grandes...

```
/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  ipj11c.c                     */
/*                               */
/*  Undecimo ejemplo: juego de   */
/*  "la Serpiente"              */
/*                               */
/*  Comprobado con:             */
/*  - Djgpp 2.03 (gcc 3.2)      */
/*  y Allegro 4.02 - MsDos      */
/*  - MinGW 2.0.0-3 (gcc 3.2)   */
/*  y Allegro 4.02 - Win        */
/*  - Gcc 3.2.2 + All. 4.03     */
/*  en Mandrake Linux 9.1      */
/*  - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*  y Allegro 4.03 - Win XP     */
/*-----*/

#include <allegro.h>

/* Posiciones X e Y iniciales */
#define POS_X_INI 16
#define POS_Y_INI 10

#define INC_X_INI 1
#define INC_Y_INI 0

/* Pausa en milisegundos entre un "fotograma" y otro */
#define PAUSA 350

/* Teclas predefinidas */
#define TEC_ARRIBA KEY_E
#define TEC_ABAJO KEY_X
#define TEC_IZQDA KEY_S
#define TEC_DCHA KEY_D

int posX, posY; /* Posicion actual */
int incX, incY; /* Incremento de la posicion */

/* Terminado: Si ha chocado o comida todas las frutas */
int terminado;

/* La tecla pulsada */
int tecla;

/* Escala: relacion entre tamaño de mapa y de pantalla */
#define ESCALA 10

/* Ancho y alto de los sprites */
#define ANCHOSPRITE 10
#define ALTOSPRITE 10

/* Y el mapa que representa a la pantalla */
/* Como usaremos modo grafico de 320x200 puntos */
/* y una escala de 10, el tablero medira 32x20 */
#define MAXFILAS 20
#define MAXCOLS 32
```

```

char mapa[MAXFILAS][MAXCOLS]={
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXX",
    "X          X          X",
    "X    F          X          X",
    "X          F          X    F    X",
    "X    XXXXX          X          X",
    "X    X            X          X",
    "X    X            X          X",
    "X          X          X          X",
    "X          X          X          X",
    "X          X          X          X",
    "X    F          X          X",
    "X          X          X          X",
    "X          X          X          X",
    "X    X            X          X",
    "X    X            X          X",
    "X    X            F          X",
    "X    F    X            X          X",
    "X    X            F          X",
    "XXXXXXXXXXXXXXXXXXXXXXXXXXXX"
};

int numFrutas = 8;

/* Nuestros sprites */
BITMAP *ladrilloFondo, *comida, *jugador;

typedef
char tipoSprite[ANCHOSPRITE][ALTOSPRITE];
/* El sprite en si: matriz de 30x30 bytes */

tipoSprite spriteLadrillo =
{
    {0,2,2,2,2,2,2,2,2,0},
    {2,1,1,1,1,2,2,1,1,2},
    {2,2,1,1,1,1,1,1,1,2},
    {2,2,1,1,1,1,1,1,1,2},
    {2,1,3,1,1,1,1,1,1,2},
    {2,1,1,1,1,1,1,1,3,2},
    {2,1,1,1,1,1,1,1,3,3,2},
    {2,2,1,1,1,1,3,3,2,2},
    {0,2,2,2,2,2,2,2,2,0}
};

tipoSprite spriteComida =
{
    {00,00,00,12,00,00,00,00,00,00},
    {00,00,11,12,11,00,12,11,00,00},
    {00,14,14,14,12,12,14,14,00,00},
    {14,13,14,14,14,11,14,14,14,00},
    {14,14,14,14,14,14,14,13,14,00},
    {14,14,14,14,14,14,13,13,14,00},
    {14,14,14,14,14,14,13,14,14,00},
    {14,14,14,14,14,14,14,14,14,00},
    {00,14,14,14,14,14,14,14,14,00}
};

tipoSprite spriteJugador =
{
    {0,0,4,4,4,4,4,0,0,0},
    {0,4,5,5,5,5,5,4,0,0},
    {4,5,6,6,6,6,6,5,4,0},
    {4,5,6,5,6,6,6,5,4,0},
    {4,5,6,6,6,6,6,5,4,0},
    {4,5,6,6,6,6,6,5,4,0},
    {4,5,6,6,6,6,6,5,4,0},
    {0,4,5,5,5,5,5,4,0,0},
    {0,0,4,4,4,4,4,0,0,0}
};

/* ----- Dato de la serpiente ----- */

int longSerpiente=1;

struct {
    int x;
    int y;
}
trozoSerpiente[10];

```

```
/* ----- Rutina de crear los sprites ----- */
```

```
void creaSprites()
{
    int i, j;

    /* Los ladrillos serán grises */

    RGB gris30 =      { 30, 30, 30 };
    RGB gris40      = { 40, 40, 40 };
    RGB gris40verde45 = { 40, 45, 40 };

    set_color( 1, &gris40 );
    set_color( 2, &gris30 );
    set_color( 3, &gris40verde45 );

    ladrilloFondo = create_bitmap(10, 10);
    clear_bitmap(ladrilloFondo);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(ladrilloFondo, i, j,
                palette_color[ spriteLadrillo[j][i] ]);

    /* Los comida será roja (y verde) */

    RGB rojo60 =      { 60, 0, 0 };
    RGB rojo50      = { 50, 0, 0 };
    RGB verde40     =  { 0, 40, 0 };
    RGB verde60     =  { 0, 60, 0 };

    set_color( 11, &verde40 );
    set_color( 12, &verde60 );
    set_color( 13, &rojo60 );
    set_color( 14, &rojo50 );

    comida = create_bitmap(10, 10);
    clear_bitmap(comida);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(comida, i, j,
                palette_color[ spriteComida[j][i] ]);

    /* La serpiente será azul */

    RGB azul60 =      { 0, 0, 60 };
    RGB azul60gris10 = { 10, 10, 60 };
    RGB azul60gris20 = { 20, 20, 60 };

    set_color( 4, &azul60gris20 );
    set_color( 5, &azul60gris10);
    set_color( 6, &azul60 );

    jugador = create_bitmap(10, 10);
    clear_bitmap(jugador);
    for(i=0; i<ANCHOSPRITE; i++)
        for (j=0; j<ALTOSPRITE; j++)
            putpixel(jugador, i, j,
                palette_color[ spriteJugador[j][i] ]);
}

/* ----- Rutina de dibujar el fondo ----- */

void dibujaFondo()
{
    int i, j;

    clear_bitmap(screen);

    for(i=0; i<MAXCOLS; i++)
        for (j=0; j<MAXFILAS; j++) {
            if (mapa[j][i] == 'X')
                draw_sprite(screen, ladrilloFondo, i*ESCALA, j*ESCALA);
            if (mapa[j][i] == 'F')
                draw_sprite(screen, comida, i*ESCALA, j*ESCALA);
        }
}
```

```

}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int i;

    allegro_init();           /* Inicializamos Allegro */
    install_keyboard();
    install_timer();

                                /* Intentamos entrar a modo grafico */
    if (set_gfx_mode(GFX_SAFE, 320, 200, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    /* ----- Si todo ha ido bien: empezamos */

    creaSprites();
    dibujaFondo();

                                /* Valores iniciales */
    posX = POS_X_INI;
    posY = POS_Y_INI;

    incX = INC_X_INI;
    incY = INC_Y_INI;

                                /* Parte repetitiva: */
    do {
        dibujaFondo();

        /* Añadido en la versión completa: dibuja todos los segmentos de la serpiente */
        for (i=1; i<=longSerpiente; i++)
            draw_sprite (screen, jugador, trozoSerpiente[i-1].x*ESCALA,
                trozoSerpiente[i-1].y*ESCALA);

        terminado = FALSE;

        /* Si paso por una fruta: la borro y falta una menos */
        if (mapa[posY][posX] == 'F') {
            mapa[posY][posX] = ' ';
            numFrutas --;
            if (numFrutas == 0) {
                textout(screen, font,
                    "Ganaste!", 100, 90, palette_color[14]);
                terminado = TRUE;
            }
            /* Esta es la novedad en la version "completa": aumenta la serpiente */
            longSerpiente ++;
        }

        /* Si choco con la pared, se acabo */
        if (mapa[posY][posX] == 'X') {
            textout(screen, font,
                "Chocaste!", 100, 90, palette_color[13]);
            terminado = TRUE;
        }

        if (terminado) break;

        /* Compruebo si se ha pulsado alguna tecla */
        if (keypressed() ) {
            tecla = readkey() >> 8;

            switch (tecla) {
                case TEC_ARRIBA:
                    incX = 0; incY = -1; break;

```



```

    case TEC_ABAJO:
        incX = 0; incY = 1; break;
    case TEC_IZQDA:
        incX = -1; incY = 0; break;
    case TEC_DCHA:
        incX = 1; incY = 0; break;
}

}

posX += incX;
posY += incY;

/* Esta es la novedad en la version "completa": aumenta la serpiente */
for (i=longSerpiente-1; i>=1; i--) {
    trozoSerpiente[i].x = trozoSerpiente[i-1].x;
    trozoSerpiente[i].y = trozoSerpiente[i-1].y;
}
trozoSerpiente[0].x = posX;
trozoSerpiente[0].y = posY;

/* Pequeña pausa antes de seguir */
rest ( PAUSA );

}
while (TRUE); /* Repetimos indefinidamente */
/* (la condición de salida la comprobamos "dentro") */

readkey();
return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

11.3. Serpiente en Pascal.

(Todavía no disponible)

11.4. Serpiente en Java.

(Todavía no disponible; pronto lo estará)

12. Utilizando el ratón

La mayoría de los lenguajes "de propósito general", como C, C++ o Pascal, **no suelen incluir** "por sí mismos" órdenes para manejar el ratón. Aun así, en el mundo de la informática se tiende a crear entornos que sean fáciles de manejar para el usuario, y el ratón es una gran ayuda para manejar los entornos gráficos actuales, como Windows, MacOS o las X-Window de Linux y los demás sistemas Unix. Por eso, es muy fácil encontrar **bibliotecas** de funciones ya creadas, que nos permitan utilizar el ratón desde estos lenguajes.

En estas bibliotecas tendremos **posibilidades** como: mostrar el ratón (habitualmente se llamará "showMouse"), ocultar el ratón (hideMouse), leer en qué posición se encuentra (getMouseX y getMouseY), mover el ratón a cierto punto (setMouseX y setMouseY) o ver si se ha pulsado un cierto botón del ratón (getMouseButton). Otras posibilidades más avanzadas, como comprobar si se ha hecho un doble clic, o cambiar la forma del puntero, o leer la posición de la rueda (si nuestro ratón tiene una y el driver lo reconoce), o limitar el movimiento a ciertas zonas de la pantalla, pueden no estar disponibles en algunas bibliotecas, y por supuesto los nombres exactos cambiarán de una biblioteca a otra, ya que no hay un estándar claro

En el caso de Allegro, algunas de las posibilidades que tenemos son:

- `int mouse_x` : Nos devuelve la coordenada x (horizontal) del punto en el que se encuentra el ratón. Va desde 0 (parte izquierda de la pantalla) hasta el valor máximo que permita nuestro modo de pantalla (por ejemplo hasta 639 si estamos en un modo de 640x480 puntos).
- `int mouse_y` : Nos devuelve la coordenada y (vertical) del punto en el que se encuentra el ratón. Va desde 0 (parte superior de la pantalla) hasta el valor máximo que permita nuestro modo de pantalla (por ejemplo hasta 479 si estamos en un modo de 640x480 puntos).
- `int mouse_z` : Nos da la posición de la rueda del ratón (insisto: sólo si nuestro ratón tiene una y el driver lo reconoce).
- `int mouse_b` : Indica si se ha pulsado algún botón del ratón. Cada botón corresponde a un bit, de modo que usaríamos "if (mouse_b & 1)" para comprobar si se ha pulsado el primer botón y "if (mouse_b & 2)" para ver el segundo.
- `void position_mouse` (int x, int y): Desplaza el ratón a una cierta posición.
- `void set_mouse_range` (int x1, int y1, int x2, int y2): Limita el movimiento del ratón a una zona de la pantalla, entre las coordenadas x1 y x2 (horizontal), y1 y y2 (vertical).
- `void show_mouse` (BITMAP *bmp): Muestra el ratón en una cierta zona (que normalmente será "screen"). Para ocultarlo, se usaría "show_mouse(NULL)".
- `void set_mouse_sprite` (BITMAP *sprite): Cambia la forma del puntero del ratón. Para volver a la flecha normal, se usaría "set_mouse_sprite(NULL)".
- `void scare_mouse()` . Oculta el ratón. Normalmente será necesario antes de dibujar algo en pantalla, en la zona en que se encuentra el ratón. De lo contrario , cuando movamos el ratón se borraría lo que acabamos de dibujar (se vería el fondo anterior).
- `void unscare_mouse()` . Vuelve a mostrar el ratón después de ocultarlo con "scare_mouse()".

En cualquier caso, deberá aparecer "install_mouse()" e "install_timer()" en nuestro programa antes de usar estas funciones. En el próximo apartado veremos un ejemplo sencillo de cómo usar el ratón en nuestros juegos.

12b. Quinto Juego: Puntería.

El juego será poco espectacular porque todavía no sabemos algo que sería casi fundamental para un juego de este tipo (y casi de cualquiera): cómo medir el tiempo. Lo usaremos simplemente para ver cómo utilizar el ratón en la práctica.

La idea básica del juego será:

```
Dibujar un recuadro en pantalla, con posición y tamaño al azar
Comprobar si se pulsa el ratón
Si se ha pulsado dentro del recuadro -> un punto más para el jugador
En cualquier caso, dibujar otro rectángulo distinto.
Todo ello, repetido hasta que se pulse una tecla.
```

Así de sencillo. Sería más razonable que el juego acabase después de un cierto número de intentos, o de un cierto tiempo. De igual modo, sería más entretenido si los puntos obtenidos dependiesen del tiempo que tardáramos en acertar, incluso también del tamaño de la diana. Es más, sería preferible que la diana no fuese cuadrada, pero eso haría que fuese más difícil comprobar si se ha pulsado el ratón dentro de la zona de la diana, y esa es una complicación que no nos interesa todavía.

Por eso, en la próxima entrega veremos algo sobre temporizadores, unas nociones mínimas de matemáticas que podamos aplicar a nuestros juegos, y eso nos permitirá basarnos en la idea de este juego, pero con blancos móviles y que tengan formas "más reales".

De momento allá va nuestra toma de contacto:

```
/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes  */
/*                               */
/*    ipj12c.c                   */
/*                               */
/*  Duodécimo ejemplo: juego de  */
/*  "punteria"                   */
/*  (Version inicial, sin        */
```

```

/* temporizadores)          */
/*                          */
/* Comprobado con:         */
/* - Dlgpp 2.03 (gcc 3.2)  */
/*   y Allegro 4.02 - MsDos */
/* - MinGW 2.0.0-3 (gcc 3.2) */
/*   y Allegro 4.02 - Win   */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*   y Allegro 4.03 - Win XP */
/*-----*/

#include <stdlib.h>          /* Para "rand" */
#include <allegro.h>

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{

#define ANCHOPANTALLA 320
#define ALTOPANTALLA 200
#define MAXLADODIANA 50

int
    posXraton = 160,
    posYraton = 100,
    posXdiana,
    posYdiana,
    ladoDiana,
    puntos = 0,
    dibujarDiana = 1;

allegro_init();           /* Inicializamos Allegro */
install_keyboard();
install_timer();
install_mouse();

/* Intentamos entrar a modo grafico */
if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message(
        "Incapaz de entrar a modo grafico\n%s\n",
        allegro_error);
    return 1;
}

/* ----- Si todo ha ido bien: empezamos */

srand(time(0));
show_mouse(screen);

/* Parte repetitiva: */
do {
    rest(50); /*Pausa de 50 ms */

    if (dibujarDiana) {
        /*Calculo nueva posicion de la diana */
        posXdiana = rand() % (ANCHOPANTALLA - ladoDiana);
        posYdiana = rand() % (ALTOPANTALLA - ladoDiana);
        ladoDiana = (rand() % MAXLADODIANA) + 2;

        /* Oculto raton y redibujo */
        scare_mouse();
        clear_bitmap(screen);
        rectfill(screen,
            posXdiana, posYdiana, posXdiana+ladoDiana, posYdiana+ladoDiana,
            palette_color[14]);
        textprintf(screen, font, 4,4, palette_color[13],
            "Puntos: %d", puntos);

        /*Vuelvo a mostrar ratony marco como dibujado */
        unscare_mouse();
        dibujarDiana = 0;
    }

    // Si se pulsa el botón, compruebo si es dentro del recuadro.

```

```

// Si es así, aumento puntos. En cualquier caso, dibujo nueva diana
if (mouse_b & 1) {
    if ((mouse_x >= posXdiana) && (mouse_x <= posXdiana+ladoDiana) &&
        (mouse_y >= posYdiana) && (mouse_y <= posYdiana+ladoDiana)) {
        puntos ++;
    }
    dibujarDiana = 1;
}
while ( !keypressed() ); /* Repetimos hasta pulsar tecla */

return 0;
}

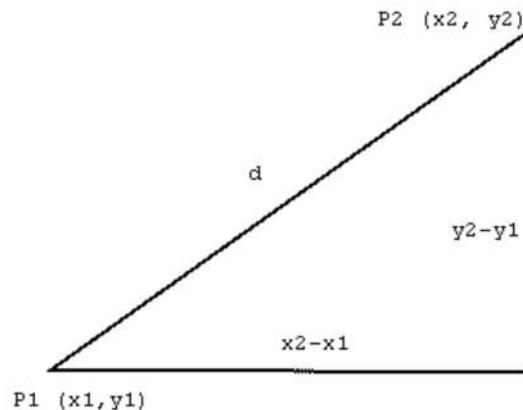
/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

13. Un poco de matemáticas para juegos. Sexto Juego: TiroAlPlato.

Siempre que queramos imitar fenómenos físicos, necesitaremos tener unos ciertos conocimientos de matemáticas, saber qué fórmulas nos permitirán representar esos fenómenos de una forma creíble. De momento veremos unas primeras operaciones básicas (distancias, círculos, parábolas y su utilización); otras operaciones algo más avanzadas las dejamos para más adelante (por ejemplo, la forma de representar figuras 3D y de rotarlas).

Distancias . La ecuación que nos da la distancia entre dos puntos procede directamente del teorema de Pitágoras, aquello de "el cuadrado de la hipotenusa es igual a la suma de los cuadrados de los catetos". Si unimos dos puntos con una línea recta, podríamos formar un triángulo rectángulo: su hipotenusa sería la recta que une los dos puntos, el tamaño de un cateto sería la diferencia entre las coordenadas horizontales (x) de los puntos, y el tamaño del otro cateto sería la diferencia entre las coordenadas verticales (y), así:

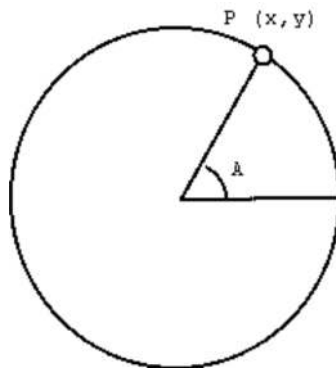


Por tanto, la forma de hallar la distancia entre dos puntos sería

$$d = \text{raíz} ((x_2 - x_1)^2 + (y_2 - y_1)^2)$$

Así, en nuestro último juego (puntería) podríamos haber usado un blanco que fuera redondo, en vez de rectangular. Comprobaríamos si se ha acertado simplemente viendo si la distancia desde el centro del círculo hasta el punto en el que se ha pulsado el ratón es menor (o igual) que el radio del círculo.

Círculo . La ecuación que nos da las coordenadas de los puntos de una circunferencia a partir del radio de la circunferencia y del ángulo en que se encuentra ese punto son



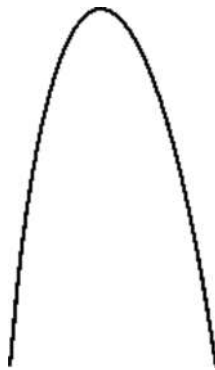
$$x = \text{radio} * \cos(\text{ángulo})$$

$$y = \text{radio} * \sin(\text{ángulo})$$

- Nota: eso es en el caso de que el centro sea el punto (0,0); si no lo fuera, basta con sumar a estos valores las coordenadas x e y del centro del círculo -

Normalmente no necesitaremos usar estas expresiones para dibujar un círculo, porque casi cualquier biblioteca de funciones gráficas tendrá incluidas las rutinas necesarias para dibujarlos. Pero sí nos pueden resultar tremendamente útiles si queremos rotar un objeto en el plano. Lo haremos dentro de poco (y más adelante veremos las modificaciones para girar en 3D).

Parábola . La forma general de una parábola es $y = ax^2 + bx + c$. Los valores de a, b, y c dependen de cómo esté desplazada la parábola y de lo "afilada" que sea. La aplicación, que veremos en la práctica en el próximo ejemplo, es que si lanzamos un objeto al aire y vuelve a caer, la curva que describe es una parábola (no sube y baja a la misma velocidad en todo momento):



Eso sí, al menos un par de comentarios:

- En "el mundo real", una parábola como la de la imagen anterior debe tener el coeficiente "a" (el número que acompaña a x^2) negativo; si es un número positivo, la parábola sería "al revés", con el hueco hacia arriba. Eso sí, en la pantalla del ordenador, las coordenadas verticales (y) se suelen medir de arriba a abajo, de modo que esa ecuación será la útil para nosotros, tal y como aparece.
- Nos puede interesar saber valores más concretos de "a", "b" y "c" en la práctica, para conseguir que la parábola pase por un cierto punto. Detallaré sólo un poco más:

Una parábola que tenga su vértice en el punto (x_1, y_1) y que tenga una distancia "p" desde dicho vértice hasta un punto especial llamado "foco" (que está "dentro" de la parábola, en la misma vertical que el vértice, y tiene ciertas propiedades que no veremos) será:

$$(x-x_1)^2 = 2p(y-y_1)$$

Podemos desarrollar esta expresión y obtener cuánto tendría que ser la "a", la "b" y la "c" a partir de las coordenadas del vértice (x_1, y_1) y de la distancia "p" (de la que sólo diré que cuanto menor sea p, más "abierto" será la parábola):

$$a = 1 / 2p$$

$$b = -x_1 / p$$

$$c = (x_1^2 / 2p) + y_1$$

¿Un ejemplo de cómo se usa esto? Claro, en el siguiente juego...

13b. Sexto Juego: TiroAlPlato.

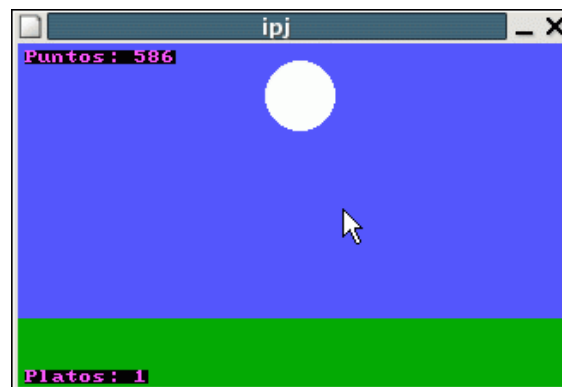
Ahora ya sabemos cómo utilizar el ratón, como medir el tiempo y conocemos algunas herramientas matemáticas sencillas. Con todo esto, podemos mejorar el juego de puntería, hacerlo "más jugable". Ahora los blancos estarán en movimiento, siguiendo una curva que será una parábola, y serán circulares. La puntuación dependerá del tiempo que se tarde en acertar. Habrá un número limitado de "platos", tras el cual se acabará la partida.

Con todo esto, la mecánica del juego será:

```
Inicializar variables
Repetir para cada plato:
  Dibujar plato a la izqda de la pantalla
  Repetir
    Si se pulsa el ratón en plato
      Aumentar puntuación
    Si no, al cabo de un tiempo
      Calcular nueva posición del plato
      Redibujar
  Hasta que el plato salga (dcha) o se acierte
Hasta que se acaben los platos
Mostrar puntuación final
```

No suena difícil, ¿no?

La apariencia (todavía muy sobria) podría ser



Y el fuente podría ser así:

```
/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes   */
/*                               */
/*  ipj13c.c                    */
/*                               */
/*  Decimoteracer ejemplo: juego */
/*  de "Tiro Al Plato"         */
/*                               */
/*  Comprobado con:            */
/*  - Djgpp 2.03 (gcc 3.2)     */
/*  y Allegro 4.02 - MsDos     */
/*  - MinGW 2.0.0-3 (gcc 3.2)  */
/*  y Allegro 4.02 - Win       */
/*  - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*  y Allegro 4.03 - Win XP    */
/*-----*/

#include <stdlib.h>          // Para "rand"
#include <math.h>           // Para "sqrt"
#include <allegro.h>
```

```

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 320
#define ALTOPANTALLA 200
#define MAXRADIODIANA 25
#define MINRADIODIANA 5
#define NUMDIANAS 12
#define MAXINCREMXDIANA 20
#define MININCREMXDIANA 10
#define RETARDO 7

/* ----- Variables globales ----- */
int
    TamanyoDianaActual,
    numDianaActual,
    posXdiana,
    posYdiana,
    radioDiana,
    incremXdiana,
    incremYdiana,
    acertado = 0; // Si se acierta -> plato nuevo

long int
    puntos = 0,
    contadorActual = 0;

float
    a,b,c; // Para la parábola del plato

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    allegro_init(); // Inicializamos Allegro
    install_keyboard();
    install_timer();
    install_mouse();

    // Intentamos entrar a modo grafico
    if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    // Si he podido entrar a modo gráfico,
    // ahora inicializo las variables
    numDianaActual = 1;
    srand(time(0));
    show_mouse(screen);

    // Y termino indicando que no ha habido errores
    return 0;
}

/* ----- Rutina de nuevo plato ----- */
void nuevoPlato()
{
    int xVerticeParabola,
        yVerticeParabola;
    float pParabola;

    // Un radio al azar entre el valor máximo y el mínimo
    radioDiana = (rand() % (MAXRADIODIANA - MINRADIODIANA))
        + MINRADIODIANA;
    // La velocidad (incremento de X), similar
    incremXdiana = (rand() % (MAXINCREMXDIANA - MININCREMXDIANA))
        + MININCREMXDIANA;

    // Vértice de la parábola, cerca del centro en horizontal
    xVerticeParabola = ANCHOPANTALLA/2 + (rand() % 40) - 20;
    // Y mitad superior de la pantalla, en vertical
    yVerticeParabola = (rand() % (ALTOPANTALLA/2));

    // Calculo a, b y c de la parábola
    pParabola = ALTOPANTALLA/2;

```

```

a = 1 / (2*pParabola);
b = -xVerticeParabola / pParabola;
c = ((xVerticeParabola*xVerticeParabola) / (2*pParabola) )
    + yVerticeParabola;

// Posición horizontal: junto margen izquierdo
posXdiana = radioDiana;
// Posición vertical: según la parábola
posYdiana =
    a*posXdiana*posXdiana +
    b*posXdiana +
    c;
}

/* ----- Rutina de redibujar pantalla ---- */
void redibujaPantalla()
{
    // Oculto ratón
    scare_mouse();
    // Borro pantalla
    clear_bitmap(screen);
    // Sincronizo con barrido para menos parpadeos
    vsync();

    // Y dibujo todo lo que corresponda
    rectfill(screen,0,0,ANCHOPANTALLA,ALTOPANTALLA-40,
        makecol(70, 70, 255)); //Cielo
    textprintf(screen, font, 4,4, palette_color[13],
        "Puntos: %d", puntos); // Puntuación
    rectfill(screen,0,ALTOPANTALLA-40,ANCHOPANTALLA,ALTOPANTALLA,
        makecol(0, 150, 0)); //Suelo
    circlefill(screen,
        posXdiana, posYdiana, radioDiana,
        palette_color[15]); // Diana
    if (numDianaActual <= NUMDIANAS) {
        textprintf(screen, font, 4,190, palette_color[13],
            "Platos: %d", NUMDIANAS-numDianaActual);
    } // Restantes, si no acabó

    unscare_mouse();
}

/* ----- Distancia entre dos puntos ----- */
float distancia(int x1, int x2, int y1, int y2) {
    return (sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2)) );
}

/* ----- Rutinas de temporización ---- */
volatile long int contador = 0;

void aumentaContador(void) { contador++; }
END_OF_FUNCTION(aumentaContador);

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    // Intentamos inicializar
    if (inicializa() != 0)
        exit(1);

    // Bloqueamos la variable y la función del temporizador
    LOCK_VARIABLE( contador );
    LOCK_FUNCTION( aumentaContador );

    // Y ponemos el temporizador en marcha: cada 10 milisegundos
    install_int(aumentaContador, 10);

    do { // Parte que se repite para cada plato

        nuevoPlato(); // Calculo su posición inicial
    }
}

```



```

redibujaPantalla(); // Y dibujo la pantalla
acertado = 0;      // Todavía no se ha acertado, claro

do { // Parte que se repite mientras se mueve

    // Compruebo el ratón
    if (mouse_b & 1) {
        if (distancia(mouse_x, posXdiana, mouse_y, posYdiana)
            <= radioDiana) {
            puntos += ANCHOPANTALLA-posXdiana;
            acertado = 1;
        }
    }

    // Si ya ha pasado el retardo, muevo
    if (contador >= contadorActual+RETARDO) {
        contadorActual = contador+RETARDO;
        posXdiana += incremXdiana;
        posYdiana =
            a*posXdiana*posXdiana +
            b*posXdiana +
            c;
        redibujaPantalla();
    }

} while ((posXdiana <= ANCHOPANTALLA - radioDiana)
        && (acertado == 0));

numDianaActual ++; // Siguiete diana

} while (numDianaActual <= NUMDIANAS);

redibujaPantalla();
scare_mouse();
textprintf(screen, font, 40,100, palette_color[15],
           "Partida terminada");
unscare_mouse();
readkey();
return 0;

}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

14. Cómo reproducir sonidos. Séptimo juego: SimeonDice.

La mayoría de las versiones actuales de lenguajes como C y Pascal incluyen posibilidades básicas de creación de sonidos a través del altavoz del ordenador. En ciertos entornos como Windows será fácil reproducir sonidos digitalizados, porque el propio sistema nos da facilidades. Otras melodías complejas pueden requerir que sepamos exactamente de qué forma se almacenan las notas y los instrumentos musicales empleados, o bien que usemos bibliotecas de funciones que nos simplifiquen un poco esa tarea.

Vamos a comentar los principales tipos de sonidos por ordenador que nos pueden interesar para nuestros juegos y a ver cómo reproducirlos:

1) Sonidos simples mediante el altavoz. Muchos lenguajes permiten emitir un sonido de una cierta frecuencia durante un cierto tiempo.

Por ejemplo, con Turbo Pascal empezamos a emitir un sonido con la orden "sound(freq)" donde "freq" es la frecuencia del sonido (más adelante veremos alguna de las frecuencias que podemos usar). Paramos este sonido con "nosound". Para que el sonido dure un cierto tiempo, podemos usar "delay(ms)" para esperar una cantidad de milisegundos, o bien hacer otra cosa mientras se escucha el sonido y comprobar la hora continuamente.

Esta es la misma idea que sigue Free Pascal (aunque esta posibilidad puede no estar disponible en todas las plataformas). También existen estas órdenes para muchos compiladores de C como Turbo C y DJGPP (para DJGPP están declaradas en "pc.h"):

```

sound(freq);
...
nosound();

```

2) Sonidos digitalizados. También existe la posibilidad de capturar un sonido con la ayuda de un micrófono y reproducirlo posteriormente. El estándar en Windows es el formato **WAV**, que podremos reproducir fácilmente desde lenguajes diseñados para este sistema operativo. Otro formato conocido, y que se usaba bastante en los tiempos de MDos, es el **VOC**, de Creative Labs, la casa desarrolladora de las tarjetas de sonido SoundBlaster. Y un tercer formato, también usado actualmente es el **AU**, el estándar para Java. No es difícil encontrar herramientas (incluso gratuitas) para convertir de un formato a otro nuestros sonidos, si nos interesa. Un cuarto formato que es imprescindible mencionar es el formato **MP3**, que es un formato comprimido, lo que supone una cierta pérdida de calidad a cambio de ocupar menos espacio (habitualmente cerca de un 10% de lo que ocuparía el WAV correspondiente).

En nuestro caso, podríamos capturar un sonido con la Grabadora de Windows, guardarlo en formato WAV y reproducirlo con una secuencia de órdenes parecida a ésta:

```
SAMPLE *sonido;
int pan = 128;
int pitch = 1000;

...

allegro_init();

...

install_timer();

...

sonido = load_sample(nombreFichero);

if (!sonido) {
    allegro_message("Error leyendo el fichero WAV '%s'\n", nombreFichero);
    return 1;
}

play_sample(
    sonido, // Sonido a reproducir
    255,    // Volumen: máximo
    0,     // Desplazamiento: ninguno
    1000,  // Frecuencia: original
    FALSE); // Repetir: no

...

destroy_sample(sonido);
```

En la orden "play_sample" los parámetros que se indican son: el **sonido** a reproducir, el **volumen** (0 a 255), el **desplazamiento** (posición a partir de la que reproducir, "pan", 0 a 255), la **frecuencia** (relativa: 1000 es la velocidad original, 2000 es el doble y así sucesivamente) y si se debe **repetir** (TRUE para si y FALSE para no repetir). Si un sonido se está repitiendo, para pararlo usaremos "stop_sample".

3) Melodías MIDI. Son melodías formadas por secuencias de notas. Suelen ser relativamente fáciles de crear con programas que nos muestran partituras en pantalla, o incluso conectando ciertos instrumentos musicales electrónicos (teclados, por ejemplo) a una conexión que muchos ordenadores incluyen (MIDI: Musical Instrument Device Interface). Entornos como Windows (e incluso muchos teléfonos móviles actuales) incluyen reproductores de sonido capaces de manejar ficheros **MID**. Nosotros podríamos escucharlos haciendo algo como

```
MIDI *miMusica;

...

allegro_init();

...

install_timer();

...

if (install_sound(DIGI_AUTODETECT, MIDI_AUTODETECT, argv[0]) != 0) {
```

```

    allegro_message("Error inicializando el sistema de sonido\n%s\n", allegro_error);
    return 1;
}

miMusica = load_midi(nombreFichero);

if (!miMusica) {
    allegro_message("Error leyendo el fichero MID '%s'\n", nombreFichero);
    return 1;
}

play_midi(miMusica, TRUE);
...
destroy_midi(miMusica);

```

En la orden "play_midi" los dos únicos recuerdan a los de "play_sample": la **música** a reproducir y si se debe **repetir** (TRUE para si y FALSE para no repetir). Si una música se está repitiendo, seguirá hasta que se indique una nueva o se use "stop_midi".

4) Melodías MOD y similares (S3M, XM, etc). Son formatos más complejos que el anterior, ya que junto con las notas a reproducir se detalla cómo "suena" cada uno de los instrumentos (normalmente usando una "muestra" digitalizada -en inglés: "sample"-). Allegro no incluye rutinas para reproducir ficheros de este tipo (al menos en la versión existente a fecha de escribir este texto).

14b. Séptimo juego: SimeonDice.

(Aun no disponible)

El "Simon" es un juego clásico de memoria. La versión más habitual es un tablero electrónico con 4 pulsadores, de 4 colores distintos, cada uno de los cuales emite un sonido distinto:

El juego consiste en repetir la secuencia de sonidos (y luces) que nos va proponiendo el ordenador: primero será una sola nota; si la recordamos, se añade un segunda nota; si recordamos estas dos se añade una tercera, después una cuarta y así sucesivamente hasta que cometamos un error.

De modo que la secuencia de nuestra versión del juego, ya en algo más cercano al lenguaje que el ordenador entiende sería:

```

secuenciaDeNotas = vacia
repetir
    generar nuevaNota al azar
    anadir nuevaNota a secuenciaDeNotas
    reproducir secuenciaDeNotas
    fallo = FALSO
    desde i = 1 hasta (i=numeroDeNotas) o hasta (fallo)
        esperar a que el usuario escoja una nota
        si notaEscogida != nota[i] entonces fallo=VERDADERO
    finDesde
hasta fallo
puntuacion = numeroNotasAcertadas * 10

```

Y nuestro tablero podría ser sencillamente así (debajo de cada color recordamos la tecla que se debería pulsar):

Las notas que yo he reproducido en este ejemplo son en formato MIDI. Si quieres hacer lo mismo en tu ordenador necesitarás algún editor de melodías MIDI (no debería ser difícil encontrar alguno gratuito en Internet). Si no tienes acceso a ninguno y/o prefieres usar las mismas notas que he creado yo, aquí las tienes [comprimidas en un fichero ZIP](#).

Yo he plasmado todo el juego así (nota: en mi ordenador funciona correctamente bajo Windows, pero si compilo para DOS no se oye el sonido, se supone que porque no he instalado el driver MIDI en MsDos para mi tarjeta de sonido):

```

/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  ipj14c.c                     */
/*                               */

```

```

/* Decimocuarto ejemplo:      */
/* "Simeon dice"             */
/*                            */
/* Comprobado con:           */
/* - MinGW 2.0.0-3 (gcc 3.2)  */
/*   y Allegro 4.02 - Win XP  */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*   y Allegro 4.03 - Win XP  */
/*-----*/

/* -----

Planteamiento del juego:

secuenciaDeNotas = vacia
repetir
    generar nuevaNota al azar
    anadir nuevaNota a secuenciaDeNotas
    reproducir secuenciaDeNotas
    fallo = FALSO
    desde i = 1 hasta (i=numeroDeNotas) o hasta (fallo)
        esperar a que el usuario escoja una nota
        si notaEscogida != nota[i] entonces fallo=VERDADERO
    finDesde
hasta fallo
puntuacion = numeroNotasAcertadas * 10

----- */

#include <stdlib.h>          // Para "rand"
#include <ctype.h>          // Para "tolower"
#include <allegro.h>

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 320
#define ALTOPANTALLA 200
#define MAXNOTAS 300
#define RETARDONOTA 200
#define RETARDOETAPA 1000

#define TECLA1 'w'
#define TECLA2 'e'
#define TECLA3 's'
#define TECLA4 'd'

#define FICHEROSONIDO1 "simon1.mid"
#define FICHEROSONIDO2 "simon2.mid"
#define FICHEROSONIDO3 "simon3.mid"
#define FICHEROSONIDO4 "simon4.mid"

/* ----- Variables globales ----- */
int
    notaActual = 0, // Número de nota actual
    notas[MAXNOTAS], // Secuencia de notas
    acertado; // Si se ha acertado o no

MIDI *sonido1, *sonido2, *sonido3, *sonido4;

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    allegro_init(); // Inicializamos Allegro
    install_keyboard();
    install_timer();

    // Intentamos entrar a modo grafico
    if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    // e intentamos usar midi
    if (install_sound(DIGI_AUTODETECT, MIDI_AUTODETECT, "") != 0) {
        allegro_message("Error inicializando el sistema de sonido\n%s\n", allegro_error);
        return 2;
    }
}

```

```

}

sonido1 = load_midi(FICHEROSONIDO1);
if (!sonido1) {
    allegro_message("Error leyendo el fichero MID '%s'\n", FICHEROSONIDO1);
    return 3;
}

sonido2 = load_midi(FICHEROSONIDO2);
if (!sonido2) {
    allegro_message("Error leyendo el fichero MID '%s'\n", FICHEROSONIDO2);
    return 3;
}

sonido3 = load_midi(FICHEROSONIDO3);
if (!sonido3) {
    allegro_message("Error leyendo el fichero MID '%s'\n", FICHEROSONIDO3);
    return 3;
}

sonido4 = load_midi(FICHEROSONIDO4);
if (!sonido4) {
    allegro_message("Error leyendo el fichero MID '%s'\n", FICHEROSONIDO4);
    return 3;
}

// Preparo números aleatorios
srand(time(0));

// Volumen al máximo, por si acaso
set_volume(255,255);

// Y termino indicando que no ha habido errores
return 0;
}

/* ----- Rutina de dibujar pantalla ----- */
void dibujaPantalla()
{
    // Borro pantalla
    clear_bitmap(screen);

    // Primer sector: SupIzq -> verde
    rectfill(screen, 10,10,
        ANCHOPANTALLA/2-10,ALTOPANTALLA/2-30,
        makecol(0, 150, 0));
    textprintf(screen, font, ANCHOPANTALLA/4, ALTOPANTALLA/2-28,
        makecol(0, 150, 0), "%c",
        TECLA1);

    // Segundo sector: SupDcha -> rojo
    rectfill(screen, ANCHOPANTALLA/2+10,10,
        ANCHOPANTALLA-10,ALTOPANTALLA/2-30,
        makecol(150, 0, 0));
    textprintf(screen, font, ANCHOPANTALLA/4*3, ALTOPANTALLA/2-28,
        makecol(150, 0, 0), "%c",
        TECLA2);

    // Tercer sector: InfIzq -> amarillo
    rectfill(screen, 10,ALTOPANTALLA/2-10,
        ANCHOPANTALLA/2-10,ALTOPANTALLA-50,
        makecol(200, 200, 0));
    textprintf(screen, font, ANCHOPANTALLA/4, ALTOPANTALLA-48,
        makecol(200, 200, 0), "%c",
        TECLA3);

    // Cuarto sector: InfDcha -> azul
    rectfill(screen, ANCHOPANTALLA/2+10,ALTOPANTALLA/2-10,
        ANCHOPANTALLA-10,ALTOPANTALLA-50,
        makecol(0, 0, 150));
    textprintf(screen, font, ANCHOPANTALLA/4*3, ALTOPANTALLA-48,
        makecol(0, 0, 150), "%c",
        TECLA4);

    textprintf(screen, font, 4,ALTOPANTALLA-20, palette_color[13],
        "Puntos: %d", notaActual*10); // Puntuación
}

```

```

}

/* ----- Rutina de reproducir notas ----- */
void reproduceNotas()
{
    int i;

    for (i=0; i<=notaActual; i++) {

        if (notas[i] == 0) {
            play_midi(sonido1, FALSE);
            rectfill(screen, 10,10,
                ANCHOPANTALLA/2-10,ALTOPANTALLA/2-30,
                makecol(255, 255, 255));
        }

        if (notas[i] == 1) {
            play_midi(sonido2, FALSE);
            rectfill(screen, ANCHOPANTALLA/2+10,10,
                ANCHOPANTALLA-10,ALTOPANTALLA/2-30,
                makecol(255, 255, 255));
        }

        if (notas[i] == 2) {
            play_midi(sonido3, FALSE);
            rectfill(screen, 10,ALTOPANTALLA/2-10,
                ANCHOPANTALLA/2-10,ALTOPANTALLA-50,
                makecol(255, 255, 255));
        }

        if (notas[i] == 3) {
            play_midi(sonido4, FALSE);
            rectfill(screen, ANCHOPANTALLA/2+10,ALTOPANTALLA/2-10,
                ANCHOPANTALLA-10,ALTOPANTALLA-50,
                makecol(255, 255, 255));
        }
        rest(RETARDONOTA);
        dibujaPantalla();
    }
}

/* ----- Rutina de comparar notas ----- */
int comparaNota(char tecla, int notaActual)
{
    int i;

    // Presupongo que no ha acertado y comparare 1 x 1
    int seHaAcertado = 0;

    if ( (tecla == TECLA1) && (notas[notaActual] == 0) ){
        play_midi(sonido1, FALSE);
        rectfill(screen, 10,10,
            ANCHOPANTALLA/2-10,ALTOPANTALLA/2-30,
            makecol(255, 255, 255));
        seHaAcertado = 1;
    }

    if ( (tecla == TECLA2) && (notas[notaActual] == 1) ){
        play_midi(sonido2, FALSE);
        rectfill(screen, ANCHOPANTALLA/2+10,10,
            ANCHOPANTALLA-10,ALTOPANTALLA/2-30,
            makecol(150, 0, 0));
        seHaAcertado = 1;
    }

    if ( (tecla == TECLA3) && (notas[notaActual] == 2) ){
        play_midi(sonido3, FALSE);
        rectfill(screen, 10,ALTOPANTALLA/2-10,
            ANCHOPANTALLA/2-10,ALTOPANTALLA-50,
            makecol(200, 200, 0));
        seHaAcertado = 1;
    }

    if ( (tecla == TECLA4) && (notas[notaActual] == 3) ){
        play_midi(sonido4, FALSE);
        rectfill(screen, ANCHOPANTALLA/2+10,ALTOPANTALLA/2-10,
            ANCHOPANTALLA-10,ALTOPANTALLA-50,
            makecol(255, 255, 255));
    }
}

```

```

        seHaAcertado = 1;
    }

    return seHaAcertado;
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int i;
    char tecla;

    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    dibujaPantalla();
    textprintf(screen, font, ANCHOPANTALLA/4, ALTOPANTALLA/2 - 40,
        makecol(255, 255, 255), " S I M E O N   D I C E ");
    textprintf(screen, font, ANCHOPANTALLA/4, ALTOPANTALLA/2,
        makecol(255, 255, 255), "Pulsa una tecla para jugar");
    readkey();

    do { // Parte que se repite hasta que falle

        dibujaPantalla(); // Dibujo la pantalla de juego

        // Genero nueva nota y reproduzco todas
        notas[notaActual] = rand() % 4;
        reproduceNotas();

        acertado = 1; // Presupongo que acertará

        // Ahora el jugador intenta repetir
        i = 0;
        do {
            tecla = tolower(readkey()); // Leo la tecla
            if (tecla == 27) break; // Salgo si es ESC
            acertado = comparaNota(tecla, i); // Comparo
            i++; // Y paso a la siguiente
        } while ((i<=notaActual) && (acertado == 1));

        // Una nota más
        if (acertado == 1) {
            textprintf(screen, font,
                ANCHOPANTALLA/4, ALTOPANTALLA/2,
                makecol(255, 255, 255), "Correcto!");
            notaActual++;
        }
        else {
            textprintf(screen, font,
                ANCHOPANTALLA/4, ALTOPANTALLA/2,
                makecol(255, 255, 255), "Fallaste!");
        }

        rest (RETARDOETAPA);

    } while ((acertado == 1) && (notaActual < MAXNOTAS));

    textprintf(screen, font,
        ANCHOPANTALLA/4, ALTOPANTALLA/2 + 20, palette_color[15],
        "Partida terminada");
    readkey();
    return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

15. Formatos de ficheros de imágenes más habituales. Cómo leer imágenes desde ficheros.

Cuando se habla de imágenes creadas por ordenador, se suele distinguir entre dos grandes **grupos** :

- **Bitmap** : (o mapa de bits). Un tipo de imágenes para ordenador, en las que se almacena información sobre los puntos que las componen y el color de cada punto. Esto supone que al ampliar la imagen no se gana en definición, sino que se ven "puntos gordos".
- **Vectorial** : Un tipo de imágenes para ordenador, en las que se almacena información sobre las líneas y figuras geométricas que las componen. Esto permite que no pierdan definición si se amplían, al contrario de lo que ocurre con las imágenes "Bitmap".

También podríamos distinguir el caso de imágenes en 3 dimensiones, en las que además se indica (como mínimo) su situación en el espacio (la de cada elemento que la forma o la de uno que se toma como referencia, y también se pueden indicar los ángulos de rotación, las texturas que se le deben aplicar...). Es un caso bastante más complicado que lo que nos interesa por ahora, así que las trataremos más adelante.

En los juegos clásicos, se suelen usar imágenes Bitmap, y es enormemente frecuente que las imágenes no se creen "desde el programa", como hemos hecho hasta ahora, sino que se diseñan con un programa de manipulación de imágenes, se guardan en un cierto formato y desde nuestro programa nos limitamos a cargarlas y manipularlas.

Pues bien, los formatos de imagen Bitmap más habituales son:

- **JPG** o **JPEG**: Es un formato comprimido, pero que **PIERDE CALIDAD** en la compresión. Se puede indicar el grado de compresión (o de calidad) que se desea, para conseguir un tamaño más pequeño o bien una mayor fidelidad al original. Por ejemplo, una compresión del 15% (o una calidad del 85%) es un valor que permite imágenes con bastante calidad pero con un tamaño no muy grande.
- **GIF** : Es un formato comprimido, que no puede mostrar más de 256 colores, pero que no pierde calidad en la compresión. También permite otras características "avanzadas", como incluir varias imágenes en un mismo fichero (muy utilizado para crear animaciones sencillas en Internet) o definir un color transparente (que no se dibujará en pantalla al mostrar la imagen, también muy usado en Internet). Como inconveniente adicional, es un formato que tiene copyright, por lo que algunas utilidades no permiten usarlo.
- **TIFF** : Es un formato con que no pierde calidad y que se usa con frecuencia para intercambiar información entre ordenadores muy distintos (PC y Macintosh, por ejemplo) cuando se debe conservar toda la calidad de la imagen original. Existen distintas variantes, unas sin comprimir y otras comprimidas, pero aun así comprime la información mucho menos que el formato JPG.
- **BMP** : Formato nativo de Windows. Permite desde imágenes en blanco y negro hasta "color auténtico", y existe también una variante sin comprimir y una variante comprimida (que, al igual que los TIFF, ocupa mucho más que el JPG).

Otros menos utilizados son, por ejemplo:

- **PCX** : Formato diseñado por los creadores de **Paintbrush** (un programa de dibujo que se utilizó mucho en los tiempos de MsDos). Es comprimido, pero compacta la información menos que el formato GIF. Hoy en día este formato se usa poco, pero es un formato muy sencillo, por lo que es habitual encontrar bibliotecas de funciones para juegos que lo utilizan.
- **LBM** : Es un caso muy similar. Se trata del formato usado por el antiguo programa de dibujo **Deluxe Paint** .
- **PNG** : Diseñado como alternativa a GIF cuando los creadores de este formato decidieron cobrar royalties por su uso. Además incluye alguna mejora, como el soporte para imágenes de más de 256 colores. Se utiliza en Internet, pero no está tan extendido como GIF.
- **TGA** : Es un formato sencillo y que permite imágenes de una alta calidad (separa los componentes rojo, verde y azul de cada punto de la imagen) pero a cambio de ocupar mucho espacio en disco.
- **PSD** : Formato nativo de Photoshop, uno de los programas de retoque de imágenes más utilizados (los ficheros no podrán ser visualizados desde muchas otras aplicaciones).
- **CPT** : Formato nativo de Corel Photopaint, otro conocido programa de retoque (con el mismo inconveniente que el anterior).

En nuestro caso, tendremos cuenta que:

- Es muy fácil usar ciertos tipos de ficheros cuando la plataforma (sistema operativo) los soporta directamente. Por ejemplo, casi cualquier compilador para Windows nos permitirá mostrar imágenes BMP de una forma muy sencilla.
- Otros formatos muy frecuentes, como JPG o GIF no suelen estar tan soportados por la plataforma (cada día más), pero sí es fácil encontrar bibliotecas que se encarguen de ello por nosotros.
- Finalmente, otros (como el PCX) son tan sencillos que nosotros mismos podríamos crear fácilmente las rutinas para mostrarlos.
- En el caso de **Allegro** , la versión que estamos usando en este momento permite manejar imágenes en formato PCX,

LBM, BMP y TGA.

15. Cómo leer imágenes desde ficheros. (*)

Vamos a empezar por centrarnos en el caso de Allegro. En la versión 4.0.2 podemos leer imágenes en formatos BMP, LBM, PCX y TGA usando

```
BITMAP *load_bmp(const char *nombreFich, RGB *paleta);
(Para imágenes de 256 colores o color auténtico de 24 bits en formato BMP de Windows o de OS/2).
```

```
BITMAP *load_lbm(const char *nombreFich, RGB *paleta);
(Para imágenes de 256 colores en formato LBM).
```

```
BITMAP *load_pcx(const char *nombreFich, RGB *paleta);
(Para imágenes de 256 colores o color auténtico de 24 bits en formato PCX).
```

```
BITMAP *load_tga(const char *nombreFich, RGB *paleta);
(Para imágenes de 256 colores, color de alta densidad de 15 bits, color auténtico de 24 bits o color auténtico +
transparencia de 32 bits en formato TGA).
```

y tenemos también una función genérica, que decide automáticamente qué tipo de fichero es, a partir de su extensión:

```
BITMAP *load_bitmap(const char *nombreFich, RGB *paleta);
```

Nosotros somos los responsables de comprobar que no ha habido ningún error (el puntero devolvería NULL), de destruir el BITMAP después de usarlo y de cambiar a un cierto modo gráfico antes de cargar la imagen o de indicar cómo se debe convertir los colores de la imagen a los de la pantalla con la orden "set_color_conversion(modo)", donde los modos son constantes con nombres:

```
COLORCONV_8_TO_24
COLORCONV_24_TO_8
COLORCONV_32_TO_24
COLORCONV_32A_TO_24
...
```

(en la documentación de Allegro están todos los valores posibles, aunque los nombres son igual de intuitivos que esos)

En [el apartado 17](#) leeremos nuestras primeras imágenes.

Es menos habitual que un juego tenga que **guardar imágenes** (al menos a nuestro nivel actual), pero aun así, conviene comentar que también tenemos las funciones equivalentes (excepto para el LBM, que no podemos guardar):

```
int save_bmp(const char *nombreFich, BITMAP *bmp, const RGB *paleta);
int save_pcx(const char *nombreFich, BITMAP *bmp, const RGB *paleta);
int save_tga(const char *nombreFich, BITMAP *bmp, const RGB *paleta);
```

y la genérica

```
int save_bitmap(const char *nombreFich, BITMAP *bmp, const RGB *paleta);
```

En cualquier caso, devuelve un valor distinto de cero si hay algún error (no guardaremos imágenes todavía, pero sí las leeremos pronto... en [el apartado 17](#)).

(Información para Pascal y Java...Aun no disponible) (*)

16. Octavo juego: Marciano 1. (acercamiento)

Vamos a aproximarnos a nuestro primer "matamarcianos". La idea del juego es la siguiente:

- Manejamos una nave, que será capaz de moverse a izquierda y derecha (por supuesto, no debe salir de la pantalla).
- En la parte superior de la pantalla habrá un "marciano", capaz de moverse de izquierda a derecha y de ir bajando paulatinamente hacia nuestra posición. Para simplificar (por ahora), sólo hay un enemigo y además no será capaz de disparar.

- Nuestra nave sí debe ser capaz de disparar, y el programa ha de detectar cuando nuestro disparo alcanza al enemigo, momento en el que termina la partida. También como simplificación, no puede haber dos disparos nuestros en pantalla a la vez.
- El enemigo debe "cambiar de forma" a medida que se mueve, no ser totalmente "estático" (basta con dos "fotogramas").
- Los tres elementos (nave, enemigo y disparo deben ser capaces de moverse independientemente uno del otro, y a velocidades distintas).

Para quien tenga poca imaginación, la pantalla del juego podría ser algo así: (en la parte superior he dejado indicado cuales podrían ser las dos formas del "marciano" y la de la explosión; esta apariencia está basada en la del clásico juego "Space Invaders"):

invader1

Como hay varias cosas nuevas, lo resolveremos en cuatro pasos para que sea más fácil de entender:

1. **Paso 1:** Todavía no hay movimiento del "marciano" ni de la nave, nos limitaremos a ver cómo cargar las imágenes que manejaremos en los otros 3 pasos.
2. **Paso 2:** Movimiento sólo del "marciano", de lado a lado, bajando en la pantalla y cambiando de forma. Se podrá interrumpir pulsando Esc. No deberá salirse por los lados, y si llega a la parte inferior deberá volver a la superior.
3. **Paso 3:** Añadiremos nuestra nave, capaz de moverse de lado a lado, a velocidad distinta. Tampoco deberá poderse salir por los lados.
4. **Paso 4:** Finalmente, incluiremos también la posibilidad de disparar, el movimiento vertical del disparo y la detección de colisiones entre el disparo y el enemigo. El disparo "desaparecerá" cuando golpee al enemigo o llegue a la parte superior de la pantalla, y sólo entonces se podrá volver a disparar.

17. Cargar y mostrar imágenes. Octavo juego (aproximación "a"): Marciano 1 (*)

Antes de ver cómo mover a la vez varias figuras que hemos leído desde el disco... ¿deberíamos tener claro cómo leer esas figuras!

Pues bien, en [el apartado 15](#) hemos comentado la idea básica de cómo leer imágenes desde un fichero. Ahora vamos a llevarlo a la práctica. Además lo haremos como realmente se suele hacer. Me explico: si en nuestro juego hay 20 tipos de enemigos distintos, podríamos tener 20 ficheros de imágenes, cada uno de los cuales tendría guardada la apariencia de un enemigo. Pero eso no es lo habitual. Es más eficiente y más cómodo tener todas las imágenes en un solo fichero. En nuestro caso, podríamos crear una única imagen con esta apariencia:



(Esta imagen está al triple de su tamaño real, para que se aprecie mejor).

En esta imagen tenemos más incluso de lo que usaremos por ahora:

- Las cifras del 0 al 9, para mostrar la puntuación.
- Los "letreros" que nos interesan en pantalla (SCORE=Puntuación, UP para formar 1UP= Primer jugador y 2UP=Segundo jugador, HI- para HI-SCORE= mejor puntuación).
- Los primeros "personajes": las dos apariencias posibles del "marciano", la de nuestra nave, la de unas "torres" debajo de las que nos podríamos refugiar (no lo haremos por ahora), y la del disparo.

El "truco" para que nos sea cómodo consiste en que las imágenes que queremos mostrar tengan un tamaño fijo. Por razones "históricas" (limitaciones del hardware de antiguos ordenadores personales), es frecuente que ese tamaño sea un múltiplo de 8 píxeles: 8, 16, 32 o incluso 64. Si sabemos que cada pequeño "sprite" tiene 16 puntos de ancho y 16 puntos de alto, es facilísimo leer el sprite que hemos dibujado en la 4ª fila y la 8ª columna de nuestra imagen, por ejemplo.

En mi caso todas las figuras de la imagen anterior tiene 16 píxeles de alto. Las letras, cifras y el disparo tienen 8 píxeles de ancho, mientras que el marciano, la nave y la torre defensiva tienen 16 píxeles de ancho. Por tanto, el sprite de la nave comenzará en la posición (32, 32), porque tiene 2 figuras de 16 píxeles a su izquierda y dos filas de 16 píxeles por encima.

Pues vamos con ello...

Un ejemplo que mostrara esta imagen al completo, la nave en la parte inferior de la pantalla y el marciano cambiando de forma en la parte superior de la pantalla, podría ser:

```

/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*  ipj17c.c                     */
/*                               */
/*  Leer imágenes desde         */
/*  un fichero                   */
/*                               */
/*  Comprobado con:             */
/*  - Djgpp 2.03 (gcc 3.2)      */
/*  y Allegro 4.02 - MsDos     */
/*-----*/

#include <allegro.h>

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 320
#define ALTOPANTALLA 200
#define RETARDO 300

/* ----- Variables globales ----- */
PALETTE pal;
BITMAP *imagen;
BITMAP *nave;
BITMAP *marciano1;
BITMAP *marciano2;

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    allegro_init();          // Inicializamos Allegro
    install_keyboard();
    install_timer();

                                // Intentamos entrar a modo grafico
    if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }
    set_palette(pal);

                                // e intentamos abrir imágenes
    imagen = load_pcx("spr_inv.pcx", pal);
    if (!imagen) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("No se ha podido abrir la imagen\n");
        return 1;
    }

    // Ahora reservo espacio para los otros sprites
    nave = create_bitmap(16, 16);
    marciano1 = create_bitmap(16, 16);
    marciano2 = create_bitmap(16, 16);

    // Y los extraigo de la imagen "grande"
    blit(imagen, nave // bitmaps de origen y destino
        , 32, 32 // coordenadas de origen
        , 0, 0 // posición de destino
        , 16, 16); // anchura y altura

    blit(imagen, marciano1, 0, 32, 0, 0, 16, 16);
    blit(imagen, marciano2, 16, 32, 0, 0, 16, 16);

    // Y termino indicando que no ha habido errores
    return 0;
}

```

```

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int fotograma = 1;

    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    do { // Parte que se repite hasta pulsar tecla

        clear_bitmap(screen);

        // Dibujo la figura y la nave
        draw_sprite(screen, imagen, 120, 80);
        draw_sprite(screen, nave, 70, 150);

        // Dibujo un marciano u otro, alternando
        if (fotograma==1) {
            draw_sprite(screen, marciano1, 250, 40);
            fotograma = 2;
        } else {
            draw_sprite(screen, marciano2, 250, 40);
            fotograma = 1;
        }
        // Y espero un poco
        rest(RETARDO);

    } while (!keypressed());

    destroy_bitmap(imagen);
    readkey();
    return 0;
}

/* Termina con la "macro" que me pide Allegro */
END_OF_MAIN();

```

(Por cierto, si quieres usar exactamente la misma imagen que yo, y ya convertida a formato PCX, [la tienes aquí](#)).

Debería resultar fácil de seguir. Para volcar un trozo de un sprite a otro se usa la orden "blit", a la que se le indica el sprite de origen y el de destino, la posición desde la que queremos comenzar a leer, la posición de destino, la anchura y la altura. Para que un sprite "cambie de forma" basta con leer varias imágenes distintas para él, e ir cambiando de una a otra según la circunstancia (puede ser un simple contador, como en este caso, o puede modificarse en casos especiales, como cuando estalle el marciano si nuestro disparo impacta en él).

Ahora vamos a mover la nave enemiga...

(Pascal y Java: Aun no completo - Pronto estará disponible) (*)

18. Mostrar y mover sólo el marciano. Octavo juego (aproximación "b"): Marciano 2 (*)

La idea de esta segunda aproximación es:

- Movimiento sólo del "marciano", de lado a lado, bajando en la pantalla y cambiando de forma. Se podrá interrumpir pulsando Esc. No deberá salirse por los lados, y si llega a la parte inferior deberá volver a la superior.

Debería resultar bastante fácil. La única novedad por ahora es eso de "cambiar de forma". Según el caso, nuestro "personaje" puede cambiar de apariencia cada vez que lo dibujamos, de forma sucesiva, o mantener una misma forma durante varios fotogramas. En este ejemplo, vamos a coger esta última opción, pero en su variante más sencilla: cada X fotogramas (y siempre los mismos, por ejemplo, cada 3) cambiaremos la apariencia del personaje y volveremos a empezar a contar.

Todo esto, en lenguaje mucho más cercano a los de programación sería:

```
cargar imagen 1 y 2 del marciano
entrar a modo gráfico
dibujar marciano en punto inicial
```

repetir

```
  aumentar posicion horizontal
```

```
  si se acerca a margen lateral de la pantalla
    cambiar sentido horizontal de avance
    bajar una línea en vertical
```

```
  si se acerca a parte inferior de la pantalla
    mover a parte superior
```

```
  aumentar contador de "fotogramas"
  si contador de fotogramas = numFotogramasPorImagen
    cambiar apariencia de marciano
    contador de fotogramas = 0
```

```
  redibujar pantalla
  pausa
```

hasta que se pulse ESC

Si se ha entendido la aproximación anterior, capaz de mostrar las imágenes, esta añade pocas novedades: calcular la nueva posición del marciano (cambiando de línea y de dirección o volviendo a la parte superior de la pantalla si corresponde) y salir sólo si se pulsa la tecla ESC, no con las demás teclas:

```
/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*    ipj18c.c                   */
/*                               */
/*  Decimoquinto ejemplo:       */
/*    Movimiento de un         */
/*    marciano animado         */
/*                               */
/*  Comprobado con:             */
/*    - Djgpp 2.03 (gcc 3.2)    */
/*    y Allegro 4.02 - MsDos    */
/*-----*/
```

```
/* -----
```

Planteamiento de esta aproximación:

```
cargar imagen 1 y 2 del marciano
entrar a modo gráfico
dibujar marciano en punto inicial
```

repetir

```
  aumentar posicion horizontal
  si se acerca a margen lateral de la pantalla
    cambiar sentido horizontal de avance
    bajar una línea en vertical
```

```
  si se acerca a parte inferior de la pantalla
    mover a parte superior
```

```
  aumentar contador de "fotogramas"
  si contador de fotogramas = numFotogramasPorImagen
    cambiar apariencia de marciano
    contador de fotogramas = 0
```

```
  redibujar pantalla
  pausa
```

hasta que se pulse ESC

```
----- */
```

```

#include <allegro.h>

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 320
#define ALTOPANTALLA 200
#define RETARDO 250

#define MARGENSUP 40
#define MARGENDCHO 30
#define MARGENIZQDO 30
#define MARGENINF 50
#define INCREMX 5
#define INCREMY 15

/* ----- Variables globales ----- */
PALETTE pal;
BITMAP *imagen;
BITMAP *nave;
BITMAP *marciano1;
BITMAP *marciano2;

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    allegro_init();          // Inicializamos Allegro
    install_keyboard();
    install_timer();

    // Intentamos entrar a modo grafico
    if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    // e intentamos abrir imágenes
    imagen = load_pcx("spr_inv.pcx", pal);
    if (!imagen) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("No se ha podido abrir la imagen\n");
        return 1;
    }
    set_palette(pal);

    // Ahora reservo espacio para los otros sprites
    nave = create_bitmap(16, 16);
    marciano1 = create_bitmap(16, 16);
    marciano2 = create_bitmap(16, 16);

    // Y los extraigo de la imagen "grande"
    blit(imagen, nave // bitmaps de origen y destino
        , 32, 32 // coordenadas de origen
        , 0, 0 // posición de destino
        , 16, 16); // anchura y altura

    blit(imagen, marciano1, 0, 32, 0, 0, 16, 16);
    blit(imagen, marciano2, 16, 32, 0, 0, 16, 16);

    // Y termino indicando que no ha habido errores
    return 0;
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int fotograma = 1;

    int xMarciano = MARGENIZQDO;

```

```

int yMarciano = MARGENSUP;
int desplMarciano = INCREMX;

int salir = 0;

// Intento inicializar
if (inicializa() != 0)
    exit(1);

do { // Parte que se repite hasta pulsar tecla

    clear_bitmap(screen);

    // Dibujo un marciano u otro, alternando
    if (fotograma==1) {
        draw_sprite(screen, marciano1, xMarciano, yMarciano);
        fotograma = 2;
    } else {
        draw_sprite(screen, marciano2, xMarciano, yMarciano);
        fotograma = 1;
    }
    // Y espero un poco
    rest(RETARDO);

    // Finalmente, calculo nueva posición
    xMarciano += desplMarciano;
    // Compruebo si debe bajar
    if ((xMarciano > ANCHOPANTALLA-16-MARGENDCHO)
        || (xMarciano < MARGENIZQDO))
    {
        desplMarciano = -desplMarciano; // Dirección contraria
        yMarciano += INCREMY;           // Y bajo una línea
    }
    // O si debe volver arriba
    if (yMarciano > ALTOPANTALLA-16-MARGENINF) {
        xMarciano = MARGENIZQDO;
        yMarciano = MARGENSUP;
        desplMarciano = INCREMX;
    }
    if (keypressed())
        if (readkey() >> 8 == KEY_ESC)
            salir = 1;

} while (!salir);

destroy_bitmap(imagen);
readkey();
return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

Si está todo entendido, vamos ahora a mover dos cosas a velocidad distinta (el "marciano" y nuestra nave)...

19. Moviendo una nave y un enemigo a la vez con velocidades distintas. Octavo juego (aproximación "c"): Marciano 3. (*)

La idea de esta tercera aproximación es:

- Añadiremos nuestra nave, capaz de moverse de lado a lado, a velocidad distinta. Tampoco deberá poderse salir por los lados.

El hecho de que nuestra nave se mueva, y que no pueda salir de la pantalla, son cosas que ya sabemos hacer. La dificultad está en que se mueva a la vez que el marciano y a velocidad distinta de la de éste. Ahora no podremos usar "pausa" porque el programa no deberá esperar, lo que deberemos es comprobar continuamente si ha llegado el momento de mover alguno de los dos personajes.

La idea básica sería algo como:

repetir

```

si hayQueRedibujarMarciano
    dibujar marciano
    hayQueRedibujarMarciano = FALSO

si hayQueRedibujarNave
    dibujar nave
    hayQueRedibujarNave = FALSO

si tiempoTranscurridoNave = tiempoHastaMoverNave
    comprobar si se ha pulsado alguna tecla
    calcular nueva posición de nave
    hayQueRedibujarNave = VERDADERO

si tiempoTranscurridoMarciano = tiempoHastaMoverMarciano
    calcular nueva posición de marciano
    hayQueRedibujarMarciano = VERDADERO

```

hasta que se pulse ESC

No debería tener problemas. Está claro que frases como "calcular nueva posición de marciano" equivalen a varias órdenes (de hecho, a casi toda la aproximación anterior) y que expresiones como "tiempoTranscurridoMarciano" serían simplemente una resta entre dos tiempos, algo así como "HoraActual - HoraUltimoMovimientoMarciano" si tenemos un reloj disponible, o algo como "contador > RetardoEsperado" si tenemos que usar un contador, como en Allegro.



Se puede hacer así (como siempre, no es la única forma posible de plantearlo, ni siquiera la mejor, pero espero que al menos sea fácil de seguir):

```

/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*  ipj19c.c                     */
/*                               */
/*  Decimosexto ejemplo:        */
/*  Un marciano y una nave      */
/*  moviéndose de forma         */
/*  independiente               */
/*                               */
/*  Comprobado con:            */
/*  - Djgpp 2.03 (gcc 3.2)     */
/*  y Allegro 4.02 - MsDos     */
/*  - MinGW 2.0.0-3 (gcc 3.2)  */
/*  y Allegro 4.02 - WinXP     */
/*-----*/

```



```

/* -----
Planteamiento de esta aproximación:

repetir

    si hayQueRedibujarMarciano
        dibujar marciano
        hayQueRedibujarMarciano = FALSO

    si hayQueRedibujarNave
        dibujar nave
        hayQueRedibujarNave = FALSO

    si tiempoTranscurridoNave = tiempoHastaMoverNave
        comprobar si se ha pulsado alguna tecla
        calcular nueva posición de nave
        hayQueRedibujarNave = VERDADERO

    si tiempoTranscurridoMarciano = tiempoHastaMoverMarciano
        calcular nueva posición de marciano
        hayQueRedibujarMarciano = VERDADERO

hasta que se pulse ESC

-----

Planteamiento de la anterior:

cargar imagen 1 y 2 del marciano
entrar a modo gráfico
dibujar marciano en punto inicial

repetir
    aumentar posicion horizontal
    si se acerca a margen lateral de la pantalla
        cambiar sentido horizontal de avance
        bajar una línea en vertical
    si se acerca a parte inferior de la pantalla
        mover a parte superior
        aumentar contador de "fotogramas"
    si contador de fotogramas = numFotogramasPorImagen
        cambiar apariencia de marciano
        contador de fotogramas = 0
    redibujar pantalla
    pausa
hasta que se pulse ESC

----- */

#include <allegro.h>

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 320
#define ALTOPANTALLA 200

// Los dos siguientes valores son los retardos
// hasta mover la nave y el marciano, pero ambos
// en centésimas de segundo
#define RETARDOCN 10
#define RETARDOCM 25

#define MARGENSUP 40
#define MARGENDCHO 30
#define MARGENIZQDO 30
#define MARGENINF 50
#define INCREMX 5
#define INCREMY 15

/* ----- Variables globales ----- */
PALETTE pal;
BITMAP *imagen;
BITMAP *nave;
BITMAP *marciano1;
BITMAP *marciano2;

int fotograma = 1;

```

```

int xMarciano = MARGENIZQDO;
int yMarciano = MARGENSUP;
int xNave = ANCHOPANTALLA / 2;
int yNave = ALTOPANTALLA-16-MARGENINF;

int desplMarciano = INCREMX;
int desplNave = INCREMX;

int puedeMoverMarciano = 1;
int puedeMoverNave = 1;
int tecla;

int salir = 0;

/* ----- Rutinas de temporización ----- */
// Contadores para temporización
volatile int contadorN = 0;
volatile int contadorM = 0;

// La rutina que lo aumenta cada cierto tiempo
void aumentaContadores(void)
{
    contadorN++;
    contadorM++;
}

END_OF_FUNCTION(aumentaContadores);

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    allegro_init();          // Inicializamos Allegro
    install_keyboard();
    install_timer();

    // Intentamos entrar a modo grafico
    if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

    // e intentamos abrir imágenes
    imagen = load_pcx("spr_inv.pcx", pal);
    if (!imagen) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("No se ha podido abrir la imagen\n");
        return 1;
    }
    set_palette(pal);

    // Ahora reservo espacio para los otros sprites
    nave = create_bitmap(16, 16);
    marciano1 = create_bitmap(16, 16);
    marciano2 = create_bitmap(16, 16);

    // Y los extraigo de la imagen "grande"
    blit(imagen, nave // bitmaps de origen y destino
        , 32, 32 // coordenadas de origen
        , 0, 0 // posición de destino
        , 16, 16); // anchura y altura

    blit(imagen, marciano1, 0, 32, 0, 0, 16, 16);
    blit(imagen, marciano2, 16, 32, 0, 0, 16, 16);

    // Rutinas de temporización
    // Bloqueamos las variables y la función del temporizador
    LOCK_VARIABLE( contadorN );
    LOCK_VARIABLE( contadorM );
    LOCK_FUNCTION( aumentaContadores );

    // Y ponemos el temporizador en marcha: cada 10 milisegundos
    // (cada centésima de segundo)
    install_int(aumentaContadores, 10);

```

```

    // Y termino indicando que no ha habido errores
    return 0;
}

/* ----- Rutina de inicialización ----- */
int mueveMarciano()
{
    // Finalmente, calculo nueva posición
    xMarciano += desplMarciano;
    // Compruebo si debe bajar
    if ((xMarciano > ANCHOPANTALLA-16-MARGENDCHO)
        || (xMarciano < MARGENIZQDO))
    {
        desplMarciano = -desplMarciano; // Dirección contraria
        yMarciano += INCREMY;           // Y bajo una línea
    }
    // O si debe volver arriba
    if (yMarciano > ALTOPANTALLA-16-MARGENINF) {
        xMarciano = MARGENIZQDO;
        yMarciano = MARGENSUP;
        desplMarciano = INCREMX;
    }
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    do { // Parte que se repite hasta pulsar tecla

        // Sólo se pueden mover tras un cierto tiempo
        if (contadorM >= RETARDOCM) {
            puedeMoverMarciano = 1;
            contadorM = 0;
        }
        if (contadorN >= RETARDOCN) {
            puedeMoverNave = 1;
            contadorN = 0;
        }

        if (puedeMoverMarciano || puedeMoverNave) {

            // Compruebo teclas pulsadas para salir
            // o mover la nave
            if (keypressed()) {
                tecla = readkey() >> 8;
                if (tecla == KEY_ESC)
                    salir = 1;
                if (puedeMoverNave) {
                    if ((tecla == KEY_RIGHT)
                        && (xNave < ANCHOPANTALLA-16-MARGENDCHO)) {
                        xNave += INCREMX;
                        puedeMoverNave = 0;
                    }
                    if ((tecla == KEY_LEFT)
                        && (xNave > MARGENIZQDO+16)) {
                        xNave -= INCREMX;
                        puedeMoverNave = 0;
                    }
                }
                clear_keybuf();
            }

            // Sincronizo con el barrido para evitar
            // parpadeos, borro la pantalla y dibujo la nave
            vsync();
            clear_bitmap(screen);
            draw_sprite(screen, nave, xNave, yNave);

            // Dibujo un marciano u otro, alternando
            if (fotograma==1) {
                draw_sprite(screen, marciano1, xMarciano, yMarciano);
            }
        }
    }
}

```

```

        if (puedeMoverMarciano)
            fotograma = 2;
    } else {
        draw_sprite(screen, marciano2, xMarciano, yMarciano);
        if (puedeMoverMarciano)
            fotograma = 1;
    }

    // Y calculo su nueva posición si corresponde
    if (puedeMoverMarciano) {
        mueveMarciano();
        puedeMoverMarciano = 0;
    }
}

} while (!salir);

destroy_bitmap(imagen);
readkey();
return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

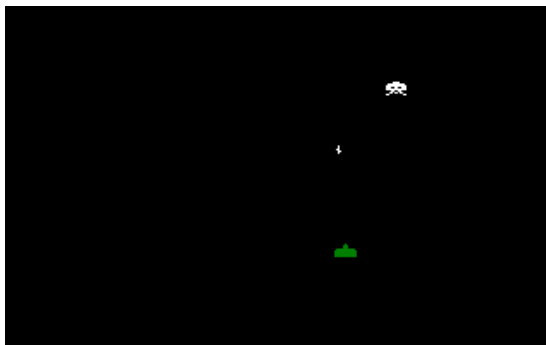
Ahora añadamos el disparo y la detección de colisiones...

20. Incluyendo un disparo y comprobación de colisiones. Octavo juego (aproximación "d"): Marciano 4. (*)

La idea de esta cuarta aproximación es:

- Incluiremos también la posibilidad de disparar, el movimiento vertical del disparo y la detección de colisiones entre el disparo y el enemigo. El disparo "desaparecerá" cuando golpee al enemigo o llegue a la parte superior de la pantalla, y sólo entonces se podrá volver a disparar.

Sin grandes novedades. Un tercer "personaje" a controlar, el disparo, que no añade casi ninguna complicación, salvo por el hecho de que no siempre está "activo" (puede haber momentos en los que no estemos disparando). La novedad es cómo comprobar si el disparo ha impactado en la nave enemiga. Una forma muy sencilla de conseguirlo puede ser simplemente comparar la posición del disparo y la del marciano. Como también sabemos el "tamaño" del marciano (su anchura y su altura), podemos saber si el disparo ha acertado (si su coordenada X está entre XMarciano y XMarciano+AnchoMarciano, y lo mismo para la coordenada Y).



Es muy similar a la aproximación anterior, aunque ligeramente más largo:

```

/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes */
/*                               */
/*  ipj20c.c                    */
/*                               */
/*  Ejemplo:                    */
/*  Un marciano y una nave     */
/*  moviéndose de forma       */
/*  independiente; la nave    */

```

```

/* puede disparar y */
/* "matar" al marciano */
/* */
/* Comprobado con: */
/* - Dlgpp 2.03 (gcc 3.2) */
/* y Allegro 4.02 - MsDos */
/*-----*/

```

```

/* -----

```

Planteamiento de esta aproximación:

repetir

```

    si hayQueRedibujarMarciano
        dibujar marciano
        hayQueRedibujarMarciano = FALSO

    si hayQueRedibujarNave
        dibujar nave
        hayQueRedibujarNave = FALSO

    si (hayQueRedibujarDisparo) y (disparoActivo)
        dibujar disparo
        hayQueRedibujarDisparo = FALSO

    si tiempoTranscurridoNave = tiempoHastaMoverNave
        comprobar si se ha pulsado alguna tecla
        calcular nueva posición de nave
        hayQueRedibujarNave = VERDADERO
        si teclaPulsada = teclaDisparo
            disparoActivo = VERDADERO

    si tiempoTranscurridoMarciano = tiempoHastaMoverMarciano
        calcular nueva posición de marciano
        hayQueRedibujarMarciano = VERDADERO

    si (tiempoTranscurridoDisparo = tiempoHastaMoverDisparo)
        y (disparoActivo)
        calcular nueva posición de disparo
        hayQueRedibujarDisparo = VERDADERO

    si disparoFueraDeLaPantalla
        disparoActivo = FALSO

    si colisionDisparoYMarciano
        finDeLaPartida

```

hasta que se pulse ESC

```

-----

```

Planteamiento de la inicial:

```

cargar imagen 1 y 2 del marciano
entrar a modo gráfico
dibujar marciano en punto inicial

```

repetir

```

    aumentar posicion horizontal
    si se acerca a margen lateral de la pantalla
        cambiar sentido horizontal de avance
        bajar una línea en vertical
    si se acerca a parte inferior de la pantalla
        mover a parte superior
    aumentar contador de "fotogramas"
    si contador de fotogramas = numFotogramasPorImagen
        cambiar apariencia de marciano
        contador de fotogramas = 0
    redibujar pantalla
    pausa

```

hasta que se pulse ESC

```

----- */

```

```

#include <allegro.h>

```

```

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 320

```

```

#define ALTOPANTALLA 200

// Los dos siguientes valores son los retardos
// hasta mover la nave y el marciano, pero ambos
// en centésimas de segundo
#define RETARDOCN 10
#define RETARDOCM 25
// Y también el del disparo
#define RETARDOCD 15

#define MARGENSUP 40
#define MARGENDCHO 30
#define MARGENIZQDO 30
#define MARGENINF 50
#define INCREMX 5
#define INCREMY 15

/* ----- Variables globales ----- */
PALETTE pal;
BITMAP *imagen;
BITMAP *nave;
BITMAP *marciano1;
BITMAP *marciano2;
BITMAP *disparo;

int fotograma = 1;

int xMarciano = MARGENIZQDO;
int yMarciano = MARGENSUP;
int xNave = ANCHOPANTALLA / 2;
int yNave = ALTOPANTALLA-16-MARGENINF;
int xDisparo = 0;
int yDisparo = 0;

int desplMarciano = INCREMX;
int desplNave = INCREMX;

int puedeMoverMarciano = 1;
int puedeMoverNave = 1;
int puedeMoverDisparo = 0;
int disparoActivo = 0;
int tecla;

int salir = 0;

/* ----- Rutinas de temporización ----- */
// Contadores para temporización
volatile int contadorN = 0;
volatile int contadorM = 0;
volatile int contadorD = 0;

// La rutina que lo aumenta cada cierto tiempo
void aumentaContadores(void)
{
    contadorN++;
    contadorM++;
    contadorD++;
}

END_OF_FUNCTION(aumentaContadores);

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    allegro_init(); // Inicializamos Allegro
    install_keyboard();
    install_timer();

    // Intentamos entrar a modo grafico
    if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }
}

```

```

                                // e intentamos abrir imágenes
imagen = load_pcx("spr_inv.pcx", pal);
if (!imagen) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("No se ha podido abrir la imagen\n");
    return 1;
}
set_palette(pal);

// Ahora reservo espacio para los otros sprites
nave = create_bitmap(16, 16);
marciano1 = create_bitmap(16, 16);
marciano2 = create_bitmap(16, 16);
disparo = create_bitmap(8, 8);

// Y los extraigo de la imagen "grande"
blit(imagen, nave // bitmaps de origen y destino
     , 32, 32 // coordenadas de origen
     , 0, 0 // posición de destino
     , 16, 16); // anchura y altura

blit(imagen, marciano1, 0, 32, 0, 0, 16, 16);
blit(imagen, marciano2, 16, 32, 0, 0, 16, 16);
blit(imagen, disparo, 72, 32, 0, 0, 8, 8);

// Rutinas de temporización
// Bloqueamos las variables y la función del temporizador
LOCK_VARIABLE( contadorN );
LOCK_VARIABLE( contadorM );
LOCK_FUNCTION( aumentaContadores );

// Y ponemos el temporizador en marcha: cada 10 milisegundos
// (cada centésima de segundo)
install_int(aumentaContadores, 10);

// Y termino indicando que no ha habido errores
return 0;
}

/* ----- Rutina de mover marciano ----- */
int mueveMarciano()
{
    // Calculo nueva posición
    xMarciano += desplMarciano;
    // Compruebo si debe bajar
    if ((xMarciano > ANCHOPANTALLA-16-MARGENDCHO)
        || (xMarciano < MARGENIZQDO))
    {
        desplMarciano = -desplMarciano; // Dirección contraria
        yMarciano += INCREMY; // Y bajo una línea
    }
    // O si debe volver arriba
    if (yMarciano > ALTOPANTALLA-16-MARGENINF) {
        xMarciano = MARGENIZQDO;
        yMarciano = MARGENSUP;
        desplMarciano = INCREMX;
    }
}

/* ----- Rutina de mover disparo ----- */
int mueveDisparo()
{
    // Calculo nueva posición
    yDisparo -= 4;
    // Compruebo si ha chocado
    if ((xDisparo >= xMarciano)
        && (xDisparo <= xMarciano + 8)
        && (yDisparo >= yMarciano)
        && (yDisparo <= yMarciano + 8))
    {
        textprintf(screen, font, xDisparo, yDisparo,
                  palette_color[11], "Boom!");
        readkey();
        salir = 1; // Y se acabó
    }
    // O si ha llegado arriba
    if (yDisparo < MARGENSUP-8) {
        xDisparo = 0; yDisparo = 0;
        puedeMoverDisparo = 0;
        disparoActivo = 0;
    }
}

```

```

    }
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    do { // Parte que se repite hasta pulsar tecla

        // Sólo se pueden mover tras un cierto tiempo
        if (contadorM >= RETARDOCM) {
            puedeMoverMariano = 1;
            contadorM = 0;
        }
        if (contadorN >= RETARDOCN) {
            puedeMoverNave = 1;
            contadorN = 0;
        }
        if ((contadorD >= RETARDOCD) && (disparoActivo)) {
            puedeMoverDisparo = 1;
            contadorD = 0;
        }

        if (puedeMoverMariano || puedeMoverNave || puedeMoverDisparo)
        {
            // Compruebo teclas pulsadas para salir
            // o mover la nave
            if (keypressed()) {
                tecla = readkey() >> 8;
                if ( tecla == KEY_ESC)
                    salir = 1;
                if (puedeMoverNave) {
                    if (( tecla == KEY_RIGHT)
                        && (xNave < ANCHOPANTALLA-16-MARGENDCHO)) {
                        xNave += INCREMX;
                        puedeMoverNave = 0;
                    }
                    if (( tecla == KEY_LEFT)
                        && (xNave > MARGENIZQDO+16)) {
                        xNave -= INCREMX;
                        puedeMoverNave = 0;
                    }
                    if (( tecla == KEY_SPACE)
                        && (!disparoActivo)) {
                        disparoActivo = 1;
                        puedeMoverDisparo = 1;
                        contadorD = 0;
                        xDisparo = xNave;
                        yDisparo = yNave-2;
                    }
                }
                clear_keybuf();
            }
        }

        // Sincronizo con el barrido para evitar
        // parpadeos, borro la pantalla y dibujo la nave
        vsync();
        clear_bitmap(screen);
        draw_sprite(screen, nave, xNave, yNave);

        // Dibujo un marciano u otro, alternando
        if (fotograma==1) {
            draw_sprite(screen, marciano1, xMarciano, yMarciano);
            if (puedeMoverMariano)
                fotograma = 2;
        } else {
            draw_sprite(screen, marciano2, xMarciano, yMarciano);
            if (puedeMoverMariano)
                fotograma = 1;
        }
    }
}

```



```

// Y calculo su nueva posición si corresponde
if (puedeMoverMarciano) {
    mueveMarciano();
    puedeMoverMarciano = 0;
}

// Y análogo para el disparo
if (puedeMoverDisparo) {
    mueveDisparo();
    puedeMoverDisparo = 0;
}
if (disparoActivo)
    draw_sprite(screen, disparo, xDisparo, yDisparo);
}

} while (!salir);

textprintf(screen, font, 1, 1,
           palette_color[11], "Partida terminada.");
destroy_bitmap(imagen);
readkey();
return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

Ahora incluyamos varios marcianos...

21. Moviendo varios marcianos a la vez. Octavo juego (aproximación "e"): Marciano 5. (*)

Si se ha entendido la aproximación anterior al juego, la versión "completa" no aporta muchas dificultades. Apenas dos importantes:

- Ahora no tendremos un único enemigo, sino varios. Además, no siempre habrá que redibujar todos los enemigos, sino que algunos todavía estarán "vivos", mientras que a otros ya los habremos "matado". El juego terminará cuando matemos a todos los enemigos (en un juego "real" se debería pasar a un nivel de dificultad mayor) o cuando se nos mate a nosotros (en la práctica debería ser cuando agotáramos un cierto número de "vidas", habitualmente 3).
- Además, para que el juego tenga más aliciente, sería deseable que los enemigos también sean capaces de dispararnos a nosotros.

Por eso, haremos nuevamente varias aproximaciones sucesivas hasta tener un juego "razonablemente jugable".

En este primer acercamiento, simplemente mostrará varios enemigos moviéndose a la vez.

Las diferencias entre tener un único enemigo o varios no son muchas, pero hay que tener varias cosas en cuenta:

- Habrá que mover varios enemigos a la vez. Esto se puede hacer guardando la información de todos ellos dentro de un "array", tanto en nuestro caso, en el que todos los enemigos se moverán a la vez, como en el caso de que tuvieran movimientos relativamente independientes. Para cada marciano nos podría interesar (según el caso) guardar información como: posición actual, trayectoria actual, apariencia actual, etc.
- Podemos ir "matando marcianos", de modo que no estarán siempre todos visibles. Podríamos eliminar realmente a los marcianos (pasaríamos de tener "n" a tener "n-1") o simplemente marcarlos como "no activos" (siempre tendríamos "n", pero algunos estarían activos y otros no).
- El espacio que recorre cada marciano no es siempre el mismo: el "bloque" formado por todos los marcianos se mueve todo lo que pueda a la derecha y todo lo que pueda a la izquierda. A medida que desaparezcan los marcianos de los extremos, los centrales podrán avanzar más hacia cada lado.

Vamos poco a poco:

El hecho de dibujar varios marcianos se podría resolver simplemente con algo como

```

for (i=0; i<COLUMNASMARCIANOS; i++)
    for (j=0; j<FILASMARCIANOS; j++) {
        draw_sprite(screen, marciano1,

```

```

        xMarciano + i*DISTANCIAXMARCIANOS,
        yMarciano + j*DISTANCIAYMARCIANOS);
    ...

```

Donde las constantes son las que indican el número de marcianos en vertical y en horizontal, así como la distancia entre ellos:

```

#define FILASMARCIANOS 5
#define COLUMNASMARCIANOS 11
#define DISTANCIAXMARCIANOS 20
#define DISTANCIAYMARCIANOS 12

```

Además, los marcianos deberán avanzar menos hacia la derecha que si estuvieran solos, por ejemplo con:

```

if ((xMarciano > ANCHOPANTALLA-16-MARGENDCHO-COLUMNASMARCIANOS*DISTANCIAXMARCIANOS)
    || (xMarciano < MARGENIZQDO))
{
    desplMarciano = -desplMarciano; // Dirección contraria
    yMarciano += INCREMY;           // Y bajo una línea
}

```

Pero este planteamiento no hace que sea fácil calcular cuando se ha matado a un marciano concreto, ni cuando el bloque de marcianos debe avanzar más hacia la derecha o hacia la izquierda. Una opción más razonable sería crear una "estructura" (nos vamos acercando a los objetos) que representa a cada marciano, con datos como su posición y si está activo:

```

// Datos de cada marciano
typedef struct {
    int posX;
    int posY;
    int activo;
} marciano;

// El array que controla a todos los marcianos
marciano marcianos[COLUMNASMARCIANOS][FILASMARCIANOS];

```

Y apenas cambia un poco la forma de dibujar a los marcianos, que ahora se hará dentro de un doble bucle "for", para recorrer todos, y comprobando si están activos:

```

// Dibujo un marciano u otro, alternando
if (puedeMoverMarciano) {
    for (i=0; i<COLUMNASMARCIANOS; i++)
        for (j=0; j<FILASMARCIANOS; j++)
            if (marcianos[i][j].activo)
                {
                    if (fotograma==1) {
                        draw_sprite(screen, marciano1,
                                    marcianos[i][j].posX, marcianos[i][j].posY);
                        fotograma = 2;
                    } else {
                        draw_sprite(screen, marciano2,
                                    marcianos[i][j].posX, marcianos[i][j].posY);
                        fotograma = 1;
                    }
                }
}
}

```

o la de comprobar si un disparo ha acertado, con los mismos cambios:

```

for (i=0; i<COLUMNASMARCIANOS; i++)
    for (j=0; j<FILASMARCIANOS; j++)
        if (marcianos[i][j].activo) {
            if ((xDisparo >= marcianos[i][j].posX)
                && (xDisparo <= marcianos[i][j].posX + 8)
                && (yDisparo >= marcianos[i][j].posY)
                && (yDisparo <= marcianos[i][j].posY + 8))
            {
                textprintf(screen, font, xDisparo, yDisparo,

```

```

        palette_color[11], "Boom!");
marcianos[i][j].activo = 0;
disparoActivo = 0;
puntos += 10;
numMarcianos --;
if (numMarcianos < 1) {
textprintf(screen, font, 10, 10,
palette_color[11], "Partida ganada!");
salir = 1; // Y se acabó
}
}
}

```

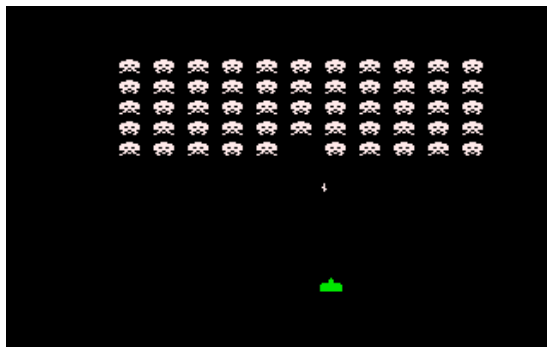
o hasta donde deben moverse los marcianos, porque habrá que calcular la "primera x" (la del marciano más a la izquierda) y la "última x" (la del marciano más a la derecha):

```

// Calculo nueva posición
for (i=0; i<COLUMNASMARCIANOS; i++)
for (j=0; j<FILASMARCIANOS; j++) {
marcianos[i][j].posX += desplMarciano;
// Y veo si es el más a la derecha o a la izqda
if ((marcianos[i][j].activo) && (marcianos[i][j].posX > ultimaX))
ultimaX = marcianos[i][j].posX;
if ((marcianos[i][j].activo) && (marcianos[i][j].posX < primeraX))
primeraX = marcianos[i][j].posX;
}

// Compruebo si debe bajar
if ((ultimaX > ANCHOPANTALLA-16-MARGENDCHO)
|| (primeraX < MARGENIZQDO))
{
desplMarciano = -desplMarciano; // Dirección contraria
for (i=0; i<COLUMNASMARCIANOS; i++)
for (j=0; j<FILASMARCIANOS; j++) {
marcianos[i][j].posY += INCREMY; // Y bajo una línea
}
}
}

```



De paso, he añadido un par de **refinamientos**: una puntuación (10 puntos por cada marciano eliminado) y el final de la partida si los marcianos nos invaden (si alcanzan la parte inferior de la pantalla).

El problema es que dibujamos unas 57 figuras en pantalla cada vez, lo que provoca un serio **problema de parpadeo**, que corregiremos en el [próximo apartado](#). Aun así, para quien quiera ver cómo sería el fuente completo de esta versión preliminar, aquí está:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* ipj21c.c */
/* */
/* Ejemplo: */
/* Varios marcianos y una */
/* nave moviéndose de */
/* forma independiente; la */
/* nave puede disparar y */
/* "matar" a los marcianos */
/* Problema: parpadeos */
/* */

```

```

/* Comprobado con:          */
/* - Dlgpp 2.03 (gcc 3.2)   */
/*   y Allegro 4.02 - MsDos */
/* - MinGW 2.0.0-3 (gcc 3.2) */
/*   y Allegro 4.02 - WinXP  */
/* - DevC++ 4.9.9.2(gcc 3.4.2) */
/*   y Allegro 4.03 - WinXP  */
/*-----*/

#include <allegro.h>

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 320
#define ALTOPANTALLA 200

// Los dos siguientes valores son los retardos
// hasta mover la nave y el marciano, pero ambos
// en centésimas de segundo
#define RETARDOCN 10
#define RETARDOCM 35
// Y también el del disparo
#define RETARDOCD 15

#define MARGENSUP 20
#define MARGENDCHO 25
#define MARGENIZQDO 25
#define MARGENINF 30
#define INCREMX 3
#define INCREMY 7

#define FILASMARCIANOS 5
#define COLUMNASMARCIANOS 11
#define DISTANCIAXMARCIANOS 20
#define DISTANCIAYMARCIANOS 12

// Datos de cada marciano
typedef struct {
    int posX;
    int posY;
    int activo;
} marciano;

/* ----- Variables globales ----- */
PALETTE pal;
BITMAP *imagen;
BITMAP *nave;
BITMAP *marciano1;
BITMAP *marciano2;
BITMAP *disparo;

// El array que controla a todos los marcianos
marciano marcianos[COLUMNASMARCIANOS][FILASMARCIANOS];

int fotograma = 1;

int xNave = ANCHOPANTALLA / 2;
int yNave = ALTOPANTALLA-16-MARGENINF;
int xDisparo = 0;
int yDisparo = 0;

int numMarcianos = FILASMARCIANOS*COLUMNASMARCIANOS;

int desplMarciano = INCREMX;
int desplNave = INCREMX;

int puedeMoverMarciano = 1;
int puedeMoverNave = 1;
int puedeMoverDisparo = 0;
int disparoActivo = 0;
int puntos = 0;
int tecla;

int salir = 0;

/* ----- Rutinas de temporización ----- */
// Contadores para temporización
volatile int contadorN = 0;
volatile int contadorM = 0;

```

```

volatile int contadorD = 0;

// La rutina que lo aumenta cada cierto tiempo
void aumentaContadores(void)
{
    contadorN++;
    contadorM++;
    contadorD++;
}

END_OF_FUNCTION(aumentaContadores);

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    int i,j;

    allegro_init();          // Inicializamos Allegro
    install_keyboard();
    install_timer();

                                // Intentamos entrar a modo grafico
    if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

                                // e intentamos abrir imágenes
    imagen = load_pcx("spr_inv.pcx", pal);
    if (!imagen) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("No se ha podido abrir la imagen\n");
        return 1;
    }
    set_palette(pal);

    // Ahora reservo espacio para los otros sprites
    nave = create_bitmap(16, 16);
    marciano1 = create_bitmap(16, 16);
    marciano2 = create_bitmap(16, 16);
    disparo = create_bitmap(8, 8);

    // Y los extraigo de la imagen "grande"
    blit(imagen, nave          // bitmaps de origen y destino
        , 32, 32              // coordenadas de origen
        , 0, 0                // posición de destino
        , 16, 16);           // anchura y altura

    blit(imagen, marciano1, 0, 32, 0, 0, 16, 16);
    blit(imagen, marciano2, 16, 32, 0, 0, 16, 16);
    blit(imagen, disparo, 72, 32, 0, 0, 8, 8);

    // Valores iniciales de los marcianos
    for (i=0; i<COLUMNASMARCIANOS; i++)
        for (j=0; j<FILASMARCIANOS; j++) {
            marcianos[i][j].posX = MARGENIZQDO + i*DISTANCIAXMARCIANOS;
            marcianos[i][j].posY = MARGENSUP + j*DISTANCIAYMARCIANOS;
            marcianos[i][j].activo = 1;
        }

    // Rutinas de temporización
    // Bloqueamos las variables y la función del temporizador
    LOCK_VARIABLE( contadorN );
    LOCK_VARIABLE( contadorM );
    LOCK_FUNCTION( aumentaContadores );

    // Y ponemos el temporizador en marcha: cada 10 milisegundos
    // (cada centésima de segundo)
    install_int(aumentaContadores, 10);

    // Y termino indicando que no ha habido errores
    return 0;
}

/* ----- Rutina de mover marciano ----- */

```

```

int mueveMarciano()
{
    int primeraX = ANCHOPANTALLA;
    int ultimaX = 0;
    int i,j;

    if (fotograma==1) fotograma = 2;
    else fotograma = 1;

    // Calculo nueva posición
    for (i=0; i<COLUMNASMARCIANOS; i++)
        for (j=0; j<FILASMARCIANOS; j++) {
            marcianos[i][j].posX += desplMarciano;
            // Y veo si es el más a la derecha o a la izqda
            if ((marcianos[i][j].activo) && (marcianos[i][j].posX > ultimaX))
                ultimaX = marcianos[i][j].posX;
            if ((marcianos[i][j].activo) && (marcianos[i][j].posX < primeraX))
                primeraX = marcianos[i][j].posX;
        }

    // Compruebo si debe bajar
    if ((ultimaX > ANCHOPANTALLA-16-MARGENDCHO)
        || (primeraX < MARGENIZQDO))
    {
        desplMarciano = -desplMarciano; // Dirección contraria
        for (i=0; i<COLUMNASMARCIANOS; i++)
            for (j=0; j<FILASMARCIANOS; j++) {
                marcianos[i][j].posY += INCREMY;           // Y bajo una línea
            }
    }

    // O si se ha "invadido", partida acabada)
    for (i=0; i<COLUMNASMARCIANOS; i++)
        for (j=0; j<FILASMARCIANOS; j++) {
            if ((marcianos[i][j].activo)
                && (marcianos[i][j].posY > ALTOPANTALLA-20-MARGENINF)) {
                textprintf(screen, font, 10, 10,
                    palette_color[11], "Invadido!");
                salir = 1;           // Y se acabó
            }
        }
}

/* ----- Rutina de mover disparo ----- */
int mueveDisparo()
{
    int i,j;

    // Calculo nueva posición
    yDisparo -= 4;

    // Compruebo si ha chocado
    for (i=0; i<COLUMNASMARCIANOS; i++)
        for (j=0; j<FILASMARCIANOS; j++)
            if (marcianos[i][j].activo) {
                if ((xDisparo >= marcianos[i][j].posX)
                    && (xDisparo <= marcianos[i][j].posX + 8)
                    && (yDisparo >= marcianos[i][j].posY)
                    && (yDisparo <= marcianos[i][j].posY + 8))
                {
                    textprintf(screen, font, xDisparo, yDisparo,
                        palette_color[11], "Boom!");
                    marcianos[i][j].activo = 0;
                    disparoActivo = 0;
                    puntos += 10;
                    numMarcianos --;
                    if (numMarcianos < 1) {
                        textprintf(screen, font, 10, 10,
                            palette_color[11], "Partida ganada!");
                        salir = 1;           // Y se acabó
                    }
                }
            }
}

// O si ha llegado arriba
if (yDisparo < MARGENSUP-8) {
    xDisparo = 0; yDisparo = 0;
    puedeMoverDisparo = 0;
    disparoActivo = 0;
}

```

```

}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int i,j;

    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    do { // Parte que se repite hasta pulsar tecla

        // Sólo se pueden mover tras un cierto tiempo
        if (contadorM >= RETARDOCM) {
            puedeMoverMarciano = 1;
            contadorM = 0;
        }
        if (contadorN >= RETARDOCN) {
            puedeMoverNave = 1;
            contadorN = 0;
        }
        if ((contadorD >= RETARDOCD) && (disparoActivo)) {
            puedeMoverDisparo = 1;
            contadorD = 0;
        }

        // Compruebo teclas pulsadas para salir
        // o mover la nave
        if (keypressed()) {
            tecla = readkey() >> 8;
            if ( tecla == KEY_ESC)
                salir = 1;
            if (puedeMoverNave) {
                if (( tecla == KEY_RIGHT)
                    && (xNave < ANCHOPANTALLA-16-MARGENDCHO)) {
                    xNave += INCREMX;
                    puedeMoverNave = 0;
                }
                if (( tecla == KEY_LEFT)
                    && (xNave > MARGENIZQDO+16)) {
                    xNave -= INCREMX;
                    puedeMoverNave = 0;
                }
                if (( tecla == KEY_SPACE)
                    && (!disparoActivo)) {
                    disparoActivo = 1;
                    puedeMoverDisparo = 1;
                    contadorD = 0;
                    xDisparo = xNave;
                    yDisparo = yNave-2;
                }
            }
            clear_keybuf();
        }
    }

    if (puedeMoverMarciano || puedeMoverNave || puedeMoverDisparo)
    {
        // Sincronizo con el barrido para evitar
        // parpadeos, borro la pantalla y dibujo la nave
        vsync();
        clear_bitmap(screen);
        draw_sprite(screen, nave, xNave, yNave);

        // Dibujo un marciano u otro, alternando
        if (puedeMoverMarciano) {
            for (i=0; i<COLUMNASMARCIANOS; i++)
                for (j=0; j<FILASMARCIANOS; j++)
                    if (marcianos[i][j].activo)
                    {
                        if (fotograma==1)
                            draw_sprite(screen, marciano1,

```

```

                marcianos[i][j].posX, marcianos[i][j].posY);
            else
                draw_sprite(screen, marciano2,
                    marcianos[i][j].posX, marcianos[i][j].posY);
        }
    }

    // Y calculo su nueva posición si corresponde
    if (puedeMoverMarciano) {
        mueveMarciano();
        puedeMoverMarciano = 0;
    }

    // Y análogo para el disparo
    if (puedeMoverDisparo) {
        mueveDisparo();
        puedeMoverDisparo = 0;
    }
    if (disparoActivo)
        draw_sprite(screen, disparo, xDisparo, yDisparo);

    textprintf(screen, font, 100, 1,
        palette_color[11], "Puntos: %d", puntos);
}

} while (!salir);

textprintf(screen, font, 1, 1,
    palette_color[11], "Partida terminada.");
destroy_bitmap(imagen);
destroy_bitmap(nave);
destroy_bitmap(marciano1);
destroy_bitmap(marciano2);
destroy_bitmap(disparo);
readkey();
rest(2000);
return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

[Eliminemos ese parpadeo...](#)

22. Un doble buffer para evitar parpadeos. Octavo juego (aproximación "e"): Marciano 6. (*)

Habíamos comentado en el [tema 9](#) cómo podíamos hacer un "doble buffer", para dibujar en memoria, en vez de hacer directamente en pantalla, y volcar el resultado en el momento final, evitando parpadeos:

```

BITMAP *bmp = create_bitmap(320, 200); // Creamos el bitmap auxiliar en memoria
clear_bitmap(bmp); // Lo borramos
putpixel(bmp, x, y, color); // Dibujamos en él
// ...
blit(bmp, screen, 0, 0, 0, 0, 320, 200); // Finalmente, volcamos a pantalla

```

Nuestro "bitmap auxiliar" se llamará "pantallaOculta". Los cambios en el fuente son muy pocos:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* ipj22c.c */
/* */
/* Ejemplo: */
/* Varios marcianos y una */
/* nave moviéndose de */
/* forma independiente; la */
/* nave puede disparar y */
/* "matar" a los marcianos */
/* Eliminados parpadeos */
/* */
/* Comprobado con: */
/* - Dlgpp 2.03 (gcc 3.2) */
/* y Allegro 4.02 - MsDos */

```



```

/* - MinGW 2.0.0-3 (gcc 3.2) */
/*   y Allegro 4.02 - WinXP */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*   y Allegro 4.03 - WinXP */
/* - Gcc 3.2.2 + All. 4.03 */
/*   en Mandrake Linux 9.1 */
/*-----*/

#include <allegro.h>

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 320
#define ALTOPANTALLA 200

// Los dos siguientes valores son los retardos
// hasta mover la nave y el marciano, pero ambos
// en centésimas de segundo
#define RETARDOCN 8
#define RETARDOCM 30
// Y también el del disparo
#define RETARDOCD 12

#define MARGENSUP 20
#define MARGENDCHO 25
#define MARGENIZQDO 25
#define MARGENINF 30
#define INCREMX 3
#define INCREMY 7

#define FILASMARCIANOS 5
#define COLUMNASMARCIANOS 11
#define DISTANCIAXMARCIANOS 20
#define DISTANCIAYMARCIANOS 12

// Datos de cada marciano
typedef struct {
    int posX;
    int posY;
    int activo;
} marciano;

/* ----- Variables globales ----- */
PALETTE pal;
BITMAP *imagen;
BITMAP *nave;
BITMAP *marciano1;
BITMAP *marciano2;
BITMAP *disparo;
BITMAP *pantallaOculta;

// El array que controla a todos los marcianos
marciano marcianos[COLUMNASMARCIANOS][FILASMARCIANOS];

int fotograma = 1;

int xNave = ANCHOPANTALLA / 2;
int yNave = ALTOPANTALLA-16-MARGENINF;
int xDisparo = 0;
int yDisparo = 0;

int numMarcianos = FILASMARCIANOS*COLUMNASMARCIANOS;

int desplMarciano = INCREMX;
int desplNave = INCREMX;

int puedeMoverMarciano = 1;
int puedeMoverNave = 1;
int puedeMoverDisparo = 0;
int disparoActivo = 0;
int puntos = 0;
int tecla;

int salir = 0;

/* ----- Rutinas de temporización ----- */
// Contadores para temporización
volatile int contadorN = 0;
volatile int contadorM = 0;
volatile int contadorD = 0;

```

```

// La rutina que los aumenta cada cierto tiempo
void aumentaContadores(void)
{
    contadorN++;
    contadorM++;
    contadorD++;
}

END_OF_FUNCTION(aumentaContadores);

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    int i,j;

    allegro_init();          // Inicializamos Allegro
    install_keyboard();
    install_timer();

                                // Intentamos entrar a modo grafico
    if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

                                // e intentamos abrir imágenes
    imagen = load_pcx("spr_inv.pcx", pal);
    if (!imagen) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("No se ha podido abrir la imagen\n");
        return 1;
    }
    set_palette(pal);

    // Ahora reservo espacio para los otros sprites
    nave = create_bitmap(16, 16);
    marciano1 = create_bitmap(16, 16);
    marciano2 = create_bitmap(16, 16);
    disparo = create_bitmap(8, 8);

    // Y los extraigo de la imagen "grande"
    blit(imagen, nave // bitmaps de origen y destino
        , 32, 32 // coordenadas de origen
        , 0, 0 // posición de destino
        , 16, 16); // anchura y altura

    blit(imagen, marciano1, 0, 32, 0, 0, 16, 16);
    blit(imagen, marciano2, 16, 32, 0, 0, 16, 16);
    blit(imagen, disparo, 72, 32, 0, 0, 8, 8);

    // Valores iniciales de los marcianos
    for (i=0; i<COLUMNASMARCIANOS; i++)
        for (j=0; j<FILASMARCIANOS; j++) {
            marcianos[i][j].posX = MARGENIZQDO + i*DISTANCIAXMARCIANOS;
            marcianos[i][j].posY = MARGENSUP + j*DISTANCIAYMARCIANOS;
            marcianos[i][j].activo = 1;
        }

    // Pantalla oculta para evitar parpadeos
    // (doble buffer)
    pantallaOculata = create_bitmap(320, 200);

    // Rutinas de temporización
    // Bloqueamos las variables y la función del temporizador
    LOCK_VARIABLE( contadorN );
    LOCK_VARIABLE( contadorM );
    LOCK_VARIABLE( contadorD );
    LOCK_FUNCTION( aumentaContadores );

    // Y ponemos el temporizador en marcha: cada 10 milisegundos
    // (cada centésima de segundo)
    install_int(aumentaContadores, 10);

```

```

// Y termino indicando que no ha habido errores
return 0;
}

/* ----- Rutina de mover marciano ----- */
int mueveMarciano()
{
    int primeraX = ANCHOPANTALLA;
    int ultimaX = 0;
    int i,j;

    if (fotograma==1) fotograma = 2;
        else fotograma = 1;

    // Calculo nueva posición
    for (i=0; i<COLUMNASMARCIANOS; i++)
        for (j=0; j<FILASMARCIANOS; j++) {
            marcianos[i][j].posX += desplMarciano;
            // Y veo si es el más a la derecha o a la izqda
            if ((marcianos[i][j].activo) && (marcianos[i][j].posX > ultimaX))
                ultimaX = marcianos[i][j].posX;
            if ((marcianos[i][j].activo) && (marcianos[i][j].posX < primeraX))
                primeraX = marcianos[i][j].posX;
        }

    // Compruebo si debe bajar
    if ((ultimaX > ANCHOPANTALLA-16-MARGENDCHO)
        || (primeraX < MARGENIZQDO))
    {
        desplMarciano = -desplMarciano; // Dirección contraria
        for (i=0; i<COLUMNASMARCIANOS; i++)
            for (j=0; j<FILASMARCIANOS; j++) {
                marcianos[i][j].posY += INCREMY; // Y bajo una línea
            }
    }

    // O si se ha "invadido", partida acabada)
    for (i=0; i<COLUMNASMARCIANOS; i++)
        for (j=0; j<FILASMARCIANOS; j++) {
            if ((marcianos[i][j].activo)
                && (marcianos[i][j].posY > ALTOPANTALLA-20-MARGENINF)) {
                /*textprintf(screen, font, 10, 10,
                    palette_color[11], "Invadido!");*/
                salir = 1; // Y se acabó
            }
        }
}

/* ----- Rutina de mover disparo ----- */
int mueveDisparo()
{
    int i,j;

    // Calculo nueva posición
    yDisparo -= 4;

    // Compruebo si ha chocado
    for (i=0; i<COLUMNASMARCIANOS; i++)
        for (j=0; j<FILASMARCIANOS; j++)
            if (marcianos[i][j].activo) {
                if ((xDisparo >= marcianos[i][j].posX)
                    && (xDisparo <= marcianos[i][j].posX + 8)
                    && (yDisparo >= marcianos[i][j].posY)
                    && (yDisparo <= marcianos[i][j].posY + 8))
                {
                    //textprintf(screen, font, xDisparo, yDisparo,
                    //    palette_color[11], "Boom!");
                    marcianos[i][j].activo = 0;
                    disparoActivo = 0;
                    puntos += 10;
                    numMarcianos --;
                    if (numMarcianos < 1) {
                        /*textprintf(screen, font, 10, 10,
                            palette_color[11], "Partida ganada!");*/
                        salir = 1; // Y se acabó
                    }
                }
            }
}

// O si ha llegado arriba

```

```

    if (yDisparo < MARGENSUP-8) {
        xDisparo = 0; yDisparo = 0;
        puedeMoverDisparo = 0;
        disparoActivo = 0;
    }
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int i,j;

    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    do { // Parte que se repite hasta pulsar tecla

        // Sólo se pueden mover tras un cierto tiempo
        if (contadorM >= RETARDOCM) {
            puedeMoverMarciano = 1;
            contadorM = 0;
        }
        if (contadorN >= RETARDOCN) {
            puedeMoverNave = 1;
            contadorN = 0;
        }
        if ((contadorD >= RETARDOCD) && (disparoActivo)) {
            puedeMoverDisparo = 1;
            contadorD = 0;
        }

        //if (puedeMoverMarciano || puedeMoverNave || puedeMoverDisparo)

        // Sincronizo con el barrido para evitar
        // parpadeos, borro la pantalla y dibujo la nave
        clear_bitmap(pantallaOculto);
        draw_sprite(pantallaOculto, nave, xNave, yNave);

        // Dibujo un marciano u otro, alternando
        for (i=0; i<COLUMNASMARCIANOS; i++)
            for (j=0; j<FILASMARCIANOS; j++)
                if (marcianos[i][j].activo)
                    if (fotograma==1)
                        draw_sprite(pantallaOculto, marciano1,
                                    marcianos[i][j].posX, marcianos[i][j].posY);
                    else
                        draw_sprite(pantallaOculto, marciano2,
                                    marcianos[i][j].posX, marcianos[i][j].posY);

        // Dibujo el disparo
        if (disparoActivo)
            draw_sprite(pantallaOculto, disparo, xDisparo, yDisparo);

        // Escribo la puntuación
        textprintf(pantallaOculto, font, 100, 1,
                  palette_color[11], "Puntos: %d",puntos);

        // Sincronizo con el barrido para evitar parpadeos
        // y vuelco la pantalla oculta
        vsync();
        blit(pantallaOculto, screen, 0, 0, 0, 0,
             ANCHOPANTALLA, ALTOPANTALLA);

        // Compruebo teclas pulsadas para salir
        // o mover la nave
        if (keypressed()) {
            tecla = readkey() >> 8;
            if ( tecla == KEY_ESC)
                salir = 1;
            if (puedeMoverNave) {
                if (( tecla == KEY_RIGHT)

```

```

        && (xNave < ANCHOPANTALLA-16-MARGENDCHO)) {
    xNave += INCREMX;
    puedeMoverNave = 0;
}
if (( tecla == KEY_LEFT)
    && (xNave > MARGENIZQDO+16)) {
    xNave -= INCREMX;
    puedeMoverNave = 0;
}
if (( tecla == KEY_SPACE)
    && (!disparoActivo)) {
    disparoActivo = 1;
    puedeMoverDisparo = 1;
    contadorD = 0;
    xDisparo = xNave;
    yDisparo = yNave-2;
}
clear_keybuf();
}
}

// Y calculo la nueva posición de los marcianos
if (puedeMoverMarciano) {
    mueveMarciano();
    puedeMoverMarciano = 0;
}

// Y análogo para el disparo
if (puedeMoverDisparo) {
    mueveDisparo();
    puedeMoverDisparo = 0;
}

} while (!salir);

textprintf(screen, font, 1, 1,
           palette_color[11], "Partida terminada.      ");
destroy_bitmap(imagen);
destroy_bitmap(nave);
destroy_bitmap(marciano1);
destroy_bitmap(marciano2);
destroy_bitmap(disparo);
destroy_bitmap(pantallaOculto);
readkey();
rest(2000);
return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

Ahora añadamos disparos de los enemigos...

23. Enemigos que disparan: Marciano 7. (*)

El hecho de que los enemigos disparen no reviste una gran dificultad, pero sí supone algunos cambios:

- Al igual que en el disparo del personaje, cada disparo de un enemigo puede estar activo o inactivo.
- Si queremos que puedan disparar varios enemigos a la vez, necesitaremos un array de disparos, en vez de un único disparo.
- Al igual que hacíamos con el disparo del personaje, deberemos comprobar si hay alguna colisión para los disparos de los enemigos. En este caso, no nos importará si el disparo colisiona con un enemigo, sino con el personaje. En ese caso, habría que restar una vida (y terminar la partida al llegar a cero vidas).
- También podemos comprobar si hay colisiones entre el disparo del personaje y los disparos de los enemigos. Si dos disparos chocan, ambos desaparecen.
- Sólo queda un detalle: cómo aparecen los disparos. En el caso del disparo del personaje, es cuando pulsamos una tecla, pero para los enemigos, no hay tecla que pulsar, sino que deben aparecer al cabo de un cierto tiempo. Una forma sencilla de hacerlo es contando "fotogramas" de juego. Por ejemplo, es frecuente añadir una "pausa" al final de cada "pasada" por el bucle de juego, para que la velocidad del juego sea independiente del ordenador, y no vaya exageradamente rápido en ordenadores muy modernos. Así, para que nuestro juego se mueva a 25 fps (fotogramas por segundo), que es una velocidad suficiente para engañar al ojo humano, y dar sensación de movimiento continuo, podríamos añadir una pausa de 40 milisegundos al final de cada "pasada" (1 segundo / 25 fps = 40 milisegundos por

fotograma). Así, si queremos que aparezca un disparo cada 2 segundos, habría que esperar $25 \times 2 = 50$ fotogramas antes de activar cada nuevo disparo.

- En la práctica, un disparo cada 2 segundos es "demasiado matemático", demasiado poco natural. Suele dar mejores resultados si la espera es un poco más aleatoria. Por ejemplo, podemos generar un número al azar entre 25 (un segundo) y 75 (tres segundos), y que esa sea la cantidad de fotogramas que hay que esperar antes de activar el siguiente disparo. Aun así, para afinar este tipo de valores, no hay más remedio que probarlos: quizá dos segundos parezca un número adecuado "sobre el papel", pero a la hora de probar el juego puede resultar demasiado rápido o demasiado lento, y habrá que terminar de afinarlo, cambiándolo por el valor que mejor se comporte, de forma que la dificultad del juego sea la correcta.

En principio, no voy a incluir ningún fuente que lo resuelva. Lo dejo propuesto como ejercicio para lo seguidores del curso. Eso sí, si alguien envía algún fuente que lo solucione, lo revisaré y lo publicaré.

En el siguiente apartado veremos la apariencia que suele tener un "bucle de juego" normal, y lo aplicaremos para imitar un juego clásico...

24. Un "bucle de juego" clásico: aproximación a Columns.(*)

Vamos a imitar un juego clásico, muy entretenido y razonablemente fácil. Lo llamaremos Columns, para evitar problemas de Copyright. La apariencia del original es ésta:



Para conseguirlo, en vez de crear un "main" cada vez más grande, como pasaba con el "matamarcianos", lo haremos más modular desde un principio. Seguiremos la apariencia de un "bucle de juego" clásico, que podría ser algo así:

```
repetir
  comprobar teclas
  calcular siguiente posición de elementos
  comprobar colisiones entre elementos
  dibujar elementos en pantalla
  pausa hasta tiempo final de fotograma
hasta final de partida
```

Si nos ayudamos de funciones auxiliares, esto se convertiría simplemente en algo como:

```
void buclePrincipal() {
  partidaTerminada = FALSE;
  do {
    comprobarTeclas();
    moverElementos();
    comprobarColisiones();
    dibujarElementos();
    pausaFotograma();
  } while (partidaTerminada != TRUE);
}
```

Donde, a su vez "comprobarTeclas" sería algo tan sencillo como:

```
void comprobarTeclas() {
  if (keypressed()) {
    tecla = readkey() >> 8;
    if ( tecla == KEY_ESC )
```

```

    partidaTerminada = TRUE;
    if ( tecla == KEY_RIGHT )
        intentarMoverDerecha();
    if ( tecla == KEY_LEFT )
        intentarMoverIzquierda();
    if ( tecla == KEY_SPACE )
        rotarColores();
    clear_keybuf();
}
}

```

Mientras que "moverElementos" no haría casi nada en un juego como éste, al no existir enemigos que se muevan solos. Se limitaría a encargarse del movimiento "automático" de nuestro personaje, que baja automáticamente poco a poco, pero eso lo haremos en la próxima entrega; por ahora estará vacío:

```

void moverElementos() {
    // Ningun elemento extra que mover por ahora
}

```

De igual modo, "comprobarColisiones" tampoco haría nada en este juego, al no existir "disparos" ni nada parecido. En la siguiente entrega sí deberemos ver si nuestra ficha realmente se puede mover a la derecha o existe algún obstáculo, pero eso quizá sea más razonable hacerlo desde el "intentarMoverDerecha", cuando el jugador indica que se quiere mover en cierta dirección.

```

void comprobarColisiones() {
    // Por ahora, no hay colisiones que comprobar
}

```

"dibujarElementos" se limitará a dibujar el fondo (en la siguiente entrega) y la pieza (ya en esta entrega) en la pantalla oculta, y luego volcar esa pantalla oculta a la visible para evitar parpadeos (la técnica del "doble buffer" que ya habíamos mencionado).

```

void dibujarElementos() {

    // Borro la pantalla y dibujo la nave
    clear_bitmap(pantallaOculta);

    // Dibujo el "fondo", con los trozos de piezas anteriores
    // (aun no)

    // Dibujo la "pieza"
    draw_sprite(pantallaOculta, pieza, x, y);

    // Sincronizo con el barrido para evitar parpadeos
    // y vuelco la pantalla oculta
    vsync();
    blit(pantallaOculta, screen, 0, 0, 0, 0,
        ANCHOPANTALLA, ALTOPANTALLA);
}

```

Finalmente, "pausaFotograma" esperará el tiempo que sea necesario hasta el momento de dibujar el siguiente fotograma en pantalla. Si queremos 25 fotogramas por segundo, como en la televisión, para lograr una sensación de movimiento continuo, deberíamos hacer una pausa de 40 milisegundos entre un fotograma y otro ($1000 / 25 = 40$). Estamos suponiendo que el tiempo de cálculo y dibujado es despreciable, pero esto es verdad en un juego tan sencillo como éste. En un juego "más real", calcularíamos cuanto se tarda en calcular posiciones, colisiones, dibujar elementos, etc., y lo restaríamos. Por ejemplo, si se tarda 10 milisegundos en esas tareas, la espera que realmente haríamos sería $40 - 10 = 30$ milésimas de segundo. Esas mejoras las dejamos para más adelante. De momento, la pausa de 40 ms la haremos así:

```

void pausaFotograma() {
    // Para 25 fps: 1000/25 = 40 milisegundos de pausa
    rest(40);
}

```

De momento usaremos esta imagen: [columnas_pieza0.bmp](#)



Y el fuente completo sería por ahora poco más que juntar esas rutinas, una función de inicializar que entrase a modo gráfico y un "main" que cargase e bucle principal del juego, de modo que podría quedar así

```

/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  ipj24c.c                     */
/*                               */
/*  Ejemplo:                     */
/*  Primer acercamiento a      */
/*  Columnas                    */
/*                               */
/*  Comprobado con:            */
/*  - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*  y Allegro 4.03 - WinXP     */
/*-----*/

#include <allegro.h>

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 640
#define ALTOPANTALLA 480

#define MARGENSUP 20
#define MARGENDCHO 25
#define MARGENIZQDO 25
#define MARGENINF 30

/* ----- Variables globales ----- */
PALETTE pal;
BITMAP *pieza;
BITMAP *pantallaOculta;

int partidaTerminada;
int x = 300;
int y = 200;
int incrX = 4;
int incrY = 4;
int tecla;

// Prototipos de las funciones que usaremos
void comprobarTeclas();
void moverElementos();
void comprobarColisiones();
void dibujarElementos();
void pausaFotograma();
void intentarMoverDerecha();
void intentarMoverIzquierda();
void rotarColores();

// --- Bucle principal del juego -----
void buclePrincipal() {
    partidaTerminada = FALSE;
    do {
        comprobarTeclas();
        moverElementos();
        comprobarColisiones();
        dibujarElementos();
        pausaFotograma();
    } while (partidaTerminada != TRUE);
}

// -- Comprobac de teclas para mover personaje o salir
void comprobarTeclas() {

    if (keypressed()) {
        tecla = readkey() >> 8;
        if ( tecla == KEY_ESC )

```



```

        partidaTerminada = TRUE;
    if ( tecla == KEY_RIGHT )
        intentarMoverDerecha();
    if ( tecla == KEY_LEFT )
        intentarMoverIzquierda();
    if ( tecla == KEY_SPACE )
        rotarColores();
    clear_keybuf();
}

// -- Intenta mover la "pieza" hacia la derecha
void intentarMoverDerecha() {
    x += incrX;
}

// -- Intenta mover la "pieza" hacia la izquierda
void intentarMoverIzquierda() {
    x -= incrX;
}

// -- Rotar los colores de la "pieza"
void rotarColores() {
    // (Todavía no)
}

// -- Mover otros elementos del juego
void moverElementos() {
    // Ningun elemento extra que mover por ahora
}

// -- Comprobar colisiones de nuestro elemento con otros, o disparos con enemigos, etc
void comprobarColisiones() {
    // Por ahora, no hay colisiones que comprobar
}

// -- Dibujar elementos en pantalla
void dibujarElementos() {

    // Borro la pantalla y dibujo la nave
    clear_bitmap(pantallaOculto);

    // Dibujo el "fondo", con los trozos de piezas anteriores
    // (aun no)

    // Dibujo la "pieza"
    draw_sprite(pantallaOculto, pieza, x, y);

    // Sincronizo con el barrido para evitar parpadeos
    // y vuelco la pantalla oculta
    vsync();
    blit(pantallaOculto, screen, 0, 0, 0, 0,
        ANCHOPANTALLA, ALTOPANTALLA);
}

// -- Pausa hasta el siguiente fotograma
void pausaFotograma() {
    // Para 25 fps: 1000/25 = 40 milisegundos de pausa
    rest(40);
}

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    int i,j;

    allegro_init();           // Inicializamos Allegro
    install_keyboard();
    install_timer();

                                // Intentamos entrar a modo grafico
}

```

```

set_color_depth(32);
if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message(
        "Incapaz de entrar a modo grafico\n%s\n",
        allegro_error);
    return 1;
}

// e intentamos abrir imágenes
pieza = load_bmp("columnas_pieza0.bmp", pal);
if (!pieza) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("No se ha podido abrir la imagen\n");
    return 1;
}
set_palette(pal);

// Pantalla oculta para evitar parpadeos
// (doble buffer)
pantallaOcultas = create_bitmap(ANCHOPANTALLA, ALTOPANTALLA);

// Y termino indicando que no ha habido errores
return 0;
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int i,j;

    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    buclePrincipal();

    destroy_bitmap(pieza);
    destroy_bitmap(pantallaOcultas);

    rest(2000);
    return 0;
}

/* Termina con la "macro" que me pide Allegro */
END_OF_MAIN();

```

25. Avanzando Columnas: primera parte de la lógica de juego.(*)

La entrega anterior mostraba la apariencia que suele tener el "bucle de juego", y lo aplicaba para un primer acercamiento al nuestro juego "Columnas". Hasta entonces, una pieza ficticia se podía mover de lado a lado. Ahora llega el momento de ampliar ese "esqueleto" con parte de la lógica que corresponde realmente a este juego.

Tendremos que hacer cosas como:

- La pieza no será siempre igual: cada vez que aparece una nueva pieza, ésta estará formada por 3 componentes elegidos al azar.
- Además, podemos "rotar" la pieza, con lo que la distribución de colores cambiará.
- La pieza no se podrá mover tan libremente como en la primera aproximación, sino que sólo habrá unas pocas columnas verticales en las que podrá estar (por ejemplo, 6).
- La pieza va bajando poco a poco.
- Cuando toca el suelo, deberemos comprobar si hay 3 o más piezas del mismo color en línea (horizontal, vertical o diagonal), y si es así, deberán eliminarse.
- Cuando ya han caído piezas anteriores, nuestra pieza no deberá llegar hasta la parte inferior de la pantalla, sino que sólo caerá hasta tocar esas piezas; de igual modo, si hay una pieza a nuestra derecha o izquierda, no podremos movernos en esa dirección.
- La partida acaba cuando una de las columnas se queda ocupada en toda su altura (que puede ser, por ejemplo, de 14

filas).

- También hay otra pequeña ampliación que tenemos que incluir: que se pueda pulsar la flecha hacia abajo para que la pieza caiga un poco más rápido.

Vamos a ver cómo se podría hacer cada una de esas cosas:

Para que la pieza esté formada por tres fragmentos al azar, podemos leer la imagen que contiene todos los fragmentos posibles y "trocearla": [columnas_pieza0.bmp](#)



(Realmente, esta imagen contiene un séptimo fragmento, que no usaremos... todavía...)

```
#define NUMFRAGMENTOS 6
BITMAP *pieza[NUMFRAGMENTOS];
imagenFragmentos = load_bmp("columnas_piezas.bmp", pal);
// Ahora reservo espacio para los otros sprites
for (i=0; i<NUMFRAGMENTOS; i++)
{
    pieza[i] = create_bitmap(32, 32);

    // Y los extraigo de la imagen "grande"
    blit(imagenFragmentos, pieza[i] // bitmaps de origen y destino
        , 32*i, 0 // coordenadas de origen
        , 0, 0 // posición de destino
        , 32, 32); // anchura y altura
}
```

Y luego podemos guardar en un "array" los 3 tipos de fragmento que contiene la pieza actualmente

```
// Tipo de imagen que corresponde a cada fragmento de pieza
int tipoFragmento[3];
```

Cuando se crea una pieza nueva, esos fragmentos se escogen al azar:

```
for (i=0; i<3; i++)
    tipoFragmento[i] = rand() % NUMFRAGMENTOS;
```

Y se consulta este array para dibujar la pieza:

```
for (i=0; i<3; i++)
    draw_sprite(pantallaOculta,
        pieza[ tipoFragmento[i] ], x, y+32*i);
```

Por otra parte, la forma de rotar colores sería asignar a la segunda casilla el color que tenía la primera, asignar a la tercera el que tenía la segunda y así sucesivamente:

```
int auxiliar = tipoFragmento[0];
tipoFragmento[0] = tipoFragmento[1];
tipoFragmento[1] = tipoFragmento[2];
tipoFragmento[2] = auxiliar;
```

Para que la pieza no se podrá mover tan libremente como en la primera aproximación, sino que sólo habrá unas pocas columnas verticales, los saltos (incrementos) verticales deberán ser algo mayores, y deberemos comprobar que no se salga por ninguno de los dos extremos de la "pantalla de juego":

```
#define TAMANYOPIEZA 32
int incrX = TAMANYOPIEZA;
...
void intentarMoverIzquierda() {
    if (x > MARGENIZQDO)
        x -= incrX;
```

```
}

```

La pieza debe ir bajando poco a poco.... así que nuestro "moverElementos" ya tendrá que hacer algo aunque no pulsemos ninguna tecla. Si hacemos "y += incrY;" en cada pasada, la pieza se moverá demasiado deprisa, así que será ás jugable si llevamos un contador de fotogramas dibujados, de modo que sólo avance una vez cada cierto número de fotogramas (por ejemplo, 10):

```
void moverElementos() {
    contadorFotogramas ++;
    if (contadorFotogramas >= 10) {
        y += incrY;
        contadorFotogramas = 0;
    }
}
```

También queremos que se pueda pulsar la flecha hacia abajo para que la pieza caiga un poco más rápido. Eso es muy fácil:

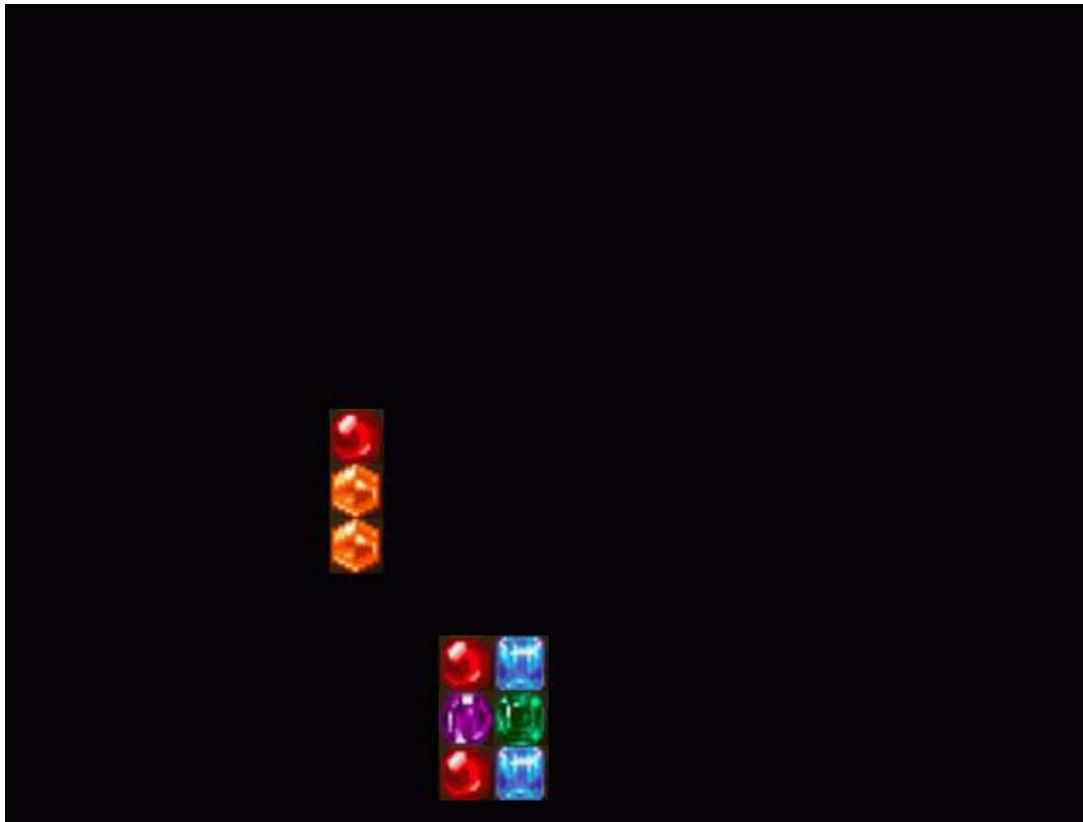
```
if ( tecla == KEY_DOWN )
    intentarBajar();
...
void intentarBajar() {
    y += incrY;
    if (y >= ALTOPANTALLA-MARGENINF-3*TAMANYOPIEZA)
        colocarPiezaEnFondo();
}
```

Cuando toca el suelo, haremos que las figuras que al componen pasen a ser parte del fondo, pero aplazamos para la siguiente entregar el comprobar si hay 3 o más piezas del mismo color en línea para eliminarlas. También aplazaremos por ahora la comprobación de choques en vertical y en horizontal, y el final de la partida (seguiremos teniendo que pulsar ESC). Para comprobar si ha llegado al suelo, podríamos añadir estas líneas al final de "intentarBajar" y de "moverElementos":

```
if (y >= ALTOPANTALLA-MARGENINF-3*TAMANYOPIEZA)
    colocarPiezaEnFondo();
```

Y esa función "colocarPiezaEnFondo" se encargaría de guardar en un array de dos dimensiones cada fragmento de la pieza actual, calculando la posición del "tablero" que le corresponde a cada fargmento (para lo que habrá que restar el margen y dividir entre el tamaño de la pieza):

```
for (i=0; i<3; i++) {
    tablero
        [ (x-MARGENIZQDO)/TAMANYOPIEZA ]
        [ (y-MARGENSUP)/TAMANYOPIEZA + i]
        = tipoFragmento[i];
}
```



Con todo esto, la nueva versión del fuente podría ser así

```

/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  ipj25c.c                     */
/*                               */
/*  Ejemplo:                     */
/*  Segundo acercamiento a      */
/*  Columnas                     */
/*                               */
/*  Comprobado con:             */
/*  - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*  y Allegro 4.03 - WinXP      */
/*-----*/

#include <allegro.h>

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 640
#define ALTOPANTALLA 480

#define ANCHOTABLERO 10
#define ALTOTABLERO 14

#define MARGENSUP 16
#define MARGENDCHO 160
#define MARGENIZQDO 160
#define MARGENINF 16

// Número de fragmentos distintos con los que
// formar las piezas
#define NUMFRAGMENTOS 6

// Ancho y alto de cada pieza
#define TAMANYOPIEZA 32

/* ----- Variables globales ----- */
PALETTE pal;
BITMAP *imagenFragmentos;
BITMAP *pantallaOculta;

BITMAP *pieza[NUMFRAGMENTOS];

```

```

int partidaTerminada;
int x = MARGENIZQDO + TAMANYOPIEZA*5;
int y = MARGENSUP;
int incrX = TAMANYOPIEZA;
int incrY = 4;
int tecla;

// Tipo de imagen que corresponde a cada fragmento de pieza
int tipoFragmento[3];

// El tablero de fondo
int tablero[ANCHOTABLERO][ALTOTABLERO];

// Contador de fotogramas, para regular la velocidad de juego
int contadorFotogramas = 0;

// Prototipos de las funciones que usaremos
void comprobarTeclas();
void moverElementos();
void comprobarColisiones();
void dibujarElementos();
void pausaFotograma();
void intentarMoverDerecha();
void intentarMoverIzquierda();
void intentarBajar();
void rotarColores();
void colocarPiezaEnFondo();

// --- Bucle principal del juego -----
void buclePrincipal() {
    partidaTerminada = FALSE;
    do {
        comprobarTeclas();
        moverElementos();
        comprobarColisiones();
        dibujarElementos();
        pausaFotograma();
    } while (partidaTerminada != TRUE);
}

// -- Comprobac de teclas para mover personaje o salir
void comprobarTeclas() {

    if (keypressed()) {
        tecla = readkey() >> 8;
        if ( tecla == KEY_ESC )
            partidaTerminada = TRUE;
        if ( tecla == KEY_RIGHT )
            intentarMoverDerecha();
        if ( tecla == KEY_LEFT )
            intentarMoverIzquierda();
        if ( tecla == KEY_DOWN )
            intentarBajar();
        if (( tecla == KEY_SPACE ) || ( tecla == KEY_UP ))
            rotarColores();
        clear_keybuf();
    }
}

// -- Intenta mover la "pieza" hacia la derecha
void intentarMoverDerecha() {
    if (x < ANCHOPANTALLA-MARGENDCHO-TAMANYOPIEZA)
        x += incrX;
}

// -- Intenta mover la "pieza" hacia la izquierda
void intentarMoverIzquierda() {
    if (x > MARGENIZQDO)
        x -= incrX;
}

// -- Intenta mover la "pieza" hacia abajo
void intentarBajar() {

```

```

    y += incrY;
    if (y >= ALTOPANTALLA-MARGENINF-3*TAMANYOPIEZA)
        colocarPiezaEnFondo();
}

// -- Rotar los colores de la "pieza"
void rotarColores() {
    int auxiliar = tipoFragmento[0];
    tipoFragmento[0] = tipoFragmento[1];
    tipoFragmento[1] = tipoFragmento[2];
    tipoFragmento[2] = auxiliar;
}

// -- Mover otros elementos del juego
void moverElementos() {
    contadorFotogramas++;
    if (contadorFotogramas >= 10) {
        y += incrY;
        contadorFotogramas = 0;
        if (y >= ALTOPANTALLA-MARGENINF-3*TAMANYOPIEZA)
            colocarPiezaEnFondo();
    }
}

// -- Comprobar colisiones de nuestro elemento con otros, o disparos con enemigos, etc
void comprobarColisiones() {
    // Por ahora, no hay colisiones que comprobar
}

// -- Dibujar elementos en pantalla
void dibujarElementos() {
    int i,j;

    // Borro la pantalla y dibujo la pieza
    clear_bitmap(pantallaOcultas);

    // Dibujo el "fondo", con los trozos de piezas anteriores
    for (i=0; i<ANCHOTABLERO; i++)
        for (j=0; j<ALTOTABLERO; j++)
            if (tablero[i][j] != -1)
                draw_sprite(pantallaOcultas,
                    pieza[ tablero[i][j] ],
                    MARGENIZQDO + TAMANYOPIEZA*i,
                    MARGENSUP + TAMANYOPIEZA*j);

    // Dibujo la "pieza"
    for (i=0; i<3; i++)
        draw_sprite(pantallaOcultas,
            pieza[ tipoFragmento[i] ], x, y+32*i);

    // Sincronizo con el barrido para evitar parpadeos
    // y vuelco la pantalla oculta
    vsync();
    blit(pantallaOcultas, screen, 0, 0, 0, 0,
        ANCHOPANTALLA, ALTOPANTALLA);
}

// -- Pausa hasta el siguiente fotograma
void pausaFotograma() {
    // Para 25 fps: 1000/25 = 40 milisegundos de pausa
    rest(40);
}

// -- Crea una nueva pieza con componentes al azar
void crearNuevaPieza() {
    int i;
    for (i=0; i<3; i++)
        tipoFragmento[i] = rand() % NUMFRAGMENTOS;
    x = MARGENIZQDO + TAMANYOPIEZA*5;
    y = MARGENSUP;
}

```

```

// -- Coloca una pieza como parte del fondo, cuando
// llegamos a la parte inferior de la pantalla
void colocarPiezaEnFondo(){
    int i;
    for (i=0; i<3; i++) {
        tablero
            [ (x-MARGENIZQDO)/TAMANYOPIEZA ]
            [ (y-MARGENSUP)/TAMANYOPIEZA + i ]
            = tipoFragmento[i];
    }
    crearNuevaPieza();
}

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    int i,j;

    allegro_init();          // Inicializamos Allegro
    install_keyboard();
    install_timer();

                                // Intentamos entrar a modo grafico
    set_color_depth(32);
    if (set_gfx_mode(GFX_SAFE,ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

                                // e intentamos abrir imágenes
    imagenFragmentos = load_bmp("columnas_piezas.bmp", pal);
    if (!imagenFragmentos) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("No se ha podido abrir la imagen\n");
        return 1;
    }

    // Ahora reservo espacio para los otros sprites
    for (i=0; i<NUMFRAGMENTOS; i++)
    {
        pieza[i] = create_bitmap(32, 32);

        // Y los extraigo de la imagen "grande"
        blit(imagenFragmentos, pieza[i] // bitmaps de origen y destino
            , 32*i, 0 // coordenadas de origen
            , 0, 0 // posición de destino
            , 32, 32); // anchura y altura
    }

    set_palette(pal);

    srand(time(0));

    // Pantalla oculta para evitar parpadeos
    // (doble buffer)
    pantallaOculto = create_bitmap(ANCHOPANTALLA, ALTOPANTALLA);

    // Vacío el tablero de fondo
    for (i=0; i<ANCHOTABLERO; i++)
        for (j=0; j<ALTOTABLERO; j++)
            tablero[i][j] = -1;

    crearNuevaPieza();

    // Y termino indicando que no ha habido errores
    return 0;
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{

```



```

int i,j;

// Intento inicializar
if (inicializa() != 0)
    exit(1);

// Bucle principal del juego
buclePrincipal();

// Libero memoria antes de salir
for (i=0; i<NUMFRAGMENTOS; i++)
    destroy_bitmap(pieza[i]);
destroy_bitmap(pantallaOculto);

rest(1000);
return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

26. Avanzando Columnas: segunda parte de la lógica de juego.(*)

La entrega anterior ya permitía que cayeran las piezas, y que pudiéramos "rotarlas" mientras caían. Aun así, quedaban varias cosas para que el juego realmente fuera jugable, como por ejemplo:

- Las piezas "caían" siempre hasta el fondo de la pantalla, aunque ya hubiera otras piezas por debajo, en vez de depositarse sobre ellas.
- Las piezas se podían mover a un lado y a otro, aunque existieran otras piezas con las que "chocaran", en vez de quedar bloqueadas por ellas.
- Cuando las piezas habían terminado de caer, no se eliminaban los fragmentos de el mismo color que estuvieran en línea.

Vayamos por partes...

Para que la pieza no caiga siempre hasta el fondo, sino que se compruebe si en algún momento toca alguna otra pieza anterior, deberemos cambiar un poco la comprobación de si debe dejar de moverse. Antes era:

```

if (y >= ALTOPANTALLA-MARGENINF-3*TAMANYOPIEZA)
    colocarPiezaEnFondo();

```

Y ahora en su lugar podríamos crear una función más completa, que además de comprobar si ha llegado a la parte inferior la pantalla, viera si existe ya alguna pieza en el fondo ("tablero"), debida a algún movimiento anterior. Para eso calculamos la posición X y la posición Y de la parte inferior de nuestra pieza, y miramos si en el tablero

```

// -- Devuelve "true" si toca fondo y no puede bajar mas
bool tocadoFondo() {
    bool colision = false;
    // Si llega abajo del todo
    if (y >= ALTOPANTALLA-MARGENINF-3*TAMANYOPIEZA)
        colision = true;
    // o si toca con una pieza interior
    int posX = (x-MARGENIZQDO)/TAMANYOPIEZA;
    int posYfinal = (y-MARGENSUP)/TAMANYOPIEZA + 3;
    if (tablero[posX][posYfinal] != -1)
        colision = true;
    return colision;
}

```

De modo que ahora "intentarBajar" quedaría así, usando esta nueva función:

```

// -- Intenta mover la "pieza" hacia abajo
void intentarBajar() {
    y += incrY;
    if ( tocadoFondo() )

```

```

    colocarPiezaEnFondo();
}

```

De igual modo, al mover hacia los lados, podríamos comprobar si colisiona con:

```

// -- Intenta mover la "pieza" hacia la izquierda
void intentarMoverIzquierda() {
    if (x > MARGENIZQDO) // Si no he llegado al margen
    {
        // Y la casilla no esta ocupada
        int posX = (x-incrX-MARGENIZQDO)/TAMANYOPIEZA;
        int posYfinal = (y-MARGENSUP)/TAMANYOPIEZA + 3;
        if (tablero[posX][posYfinal] == SINPIEZA)
            x -= incrX;
    }
}

```

Y además, despues de colocar la pieza como parte del fondo deberemos eliminar los bloques de piezas iguales que estén conectados:

```

// -- Coloca una pieza como parte del fondo, cuando
// llegamos a la parte inferior de la pantalla
// o tocamos una pieza inferior colocada antes
void colocarPiezaEnFondo(){
    int i;
    for (i=0; i<3; i++) {
        tablero
            [ (x-MARGENIZQDO)/TAMANYOPIEZA ]
            [ (y-MARGENSUP)/TAMANYOPIEZA + i ]
            = tipoFragmento[i];
    }
    eliminarBloquesConectados();
    crearNuevaPieza();
}

```

Una forma sencilla (pero no totalmente fiable) de comprobar si 3 piezas están alineadas en horizontal podría ser

```

for (i=0; i<ANCHOTABLERO-2; i++)
    for (j=0; j<ALTOTABLERO; j++)
        if (tablero[i][j] != SINPIEZA // Si hay pieza
            if ((tablero[i][j] == tablero[i+1][j]) && (tablero[i][j] == tablero[i+2][j])))
            {
                tablero[i][j] = PIEZAEXPLOSION;
                tablero[i+1][j] = PIEZAEXPLOSION;
                tablero[i+2][j] = PIEZAEXPLOSION;
                hayQueBorrar = true;
            }
}

```

Si hay 3 piezas alineadas, las convertimos en "piezas que explotan" (PIEZAEXPLOSION), como paso previo a que desaparezcan. Esta forma de comprobar si están en línea no es la mejor: fallará cuando haya 4 piezas iguales en horizontal, y sólo señalará 3 de ellas. Aun así, como primera aproximación puede servir, y la terminaremos de corregir en la siguiente entrega. La forma de comprobar si hay 3 piezas en vertical sería básicamente igual:

```

for (i=0; i<ANCHOTABLERO; i++)
    for (j=0; j<ALTOTABLERO-2; j++)
        if (tablero[i][j] != SINPIEZA) // Si hay pieza
            if ((tablero[i][j] == tablero[i][j+1]) && (tablero[i][j] == tablero[i][j+2]))
            {
                tablero[i][j] = PIEZAEXPLOSION;
                tablero[i][j+1] = PIEZAEXPLOSION;
                tablero[i][j+2] = PIEZAEXPLOSION;
                hayQueBorrar = true;
            }
}

```

Y tampoco hay grandes cambios si están alineadas en diagonal, caso en el que aumentaremos X e Y a la vez, o disminuirémos una mientras aumentamos la otra:

```

// Diagonal 1
for (i=0; i<ANCHOTABLERO-2; i++)
  for (j=0; j<ALTOTABLERO-2; j++)
    if (tablero[i][j] != SINPIEZA) // Si hay pieza
      if ((tablero[i][j] == tablero[i+1][j+1]) && (tablero[i][j] == tablero[i+2][j+2]))
        {
          tablero[i][j] = PIEZAEXPLOSION;
          tablero[i+1][j+1] = PIEZAEXPLOSION;
          tablero[i+2][j+2] = PIEZAEXPLOSION;
          hayQueBorrar = true;
        }

// Diagonal 2
for (i=2; i<ANCHOTABLERO; i++)
  for (j=0; j<ALTOTABLERO-2; j++)
    if (tablero[i][j] != SINPIEZA) // Si hay pieza
      if ((tablero[i][j] == tablero[i-1][j+1]) && (tablero[i][j] == tablero[i-2][j+2]))
        {
          tablero[i][j] = PIEZAEXPLOSION;
          tablero[i-1][j+1] = PIEZAEXPLOSION;
          tablero[i-2][j+2] = PIEZAEXPLOSION;
          hayQueBorrar = true;
        }

```

Finalmente, cuando ya hemos marcado con la "explosión" todas las piezas que debemos borrar, las mostramos un instante:

```

if (hayQueBorrar) {
  dibujarElementos();
  rest(200);
  ...
}

```

y después las borramos, dejando "caer" las piezas que tuvieran por encima:

```

// Mientras haya "explosiones", recoloco
algoHaCambiado = false;
for (i=0; i<ANCHOTABLERO; i++)
  for (j=ALTOTABLERO-1; j>=0; j--) // Abajo a arriba
    if (tablero[i][j] == PIEZAEXPLOSION)
      {
        // Bajo las superiores
        for (k=j; k>0; k--)
          tablero[i][k] = tablero[i][k-1];
        tablero[i][0] = SINPIEZA;
      }

```

Y finalmente, si realmente hemos borrado alguna pieza, hacemos una pequeña pausa y mostramos cómo queda la pantalla de juego. Además, en ese caso deberemos comprobar si al caer las piezas se ha formado algún nuevo bloque que hay que eliminar:

```

if (algoHaCambiado) {
  dibujarElementos();
  rest(200);
  eliminarBloquesConectados();
}

```

Por supuesto, quedan cosas por hacer: la rutina de marcado de piezas no es buena (falla si hay 4 o más en línea), la presentación es muy pobre, y no se lleva cuenta de la puntuación. Por tanto, prepararemos una cuarta entrega que borre correctamente, que dibuje los límites de la pantalla de juego y que calcule y muestre la puntuación de la partida.

El fuente completo de esta tercera entrega podría ser;

```

/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  ipj26c.c                     */
/*                               */
/*  Ejemplo:                     */
/*  Tercer acercamiento a       */

```

```

/*      Columnas      */
/*      */
/* Comprobado con:      */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*   y Allegro 4.03 - WinXP   */
/*-----*/

#include <allegro.h>

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 640
#define ALTOPANTALLA 480

#define ANCHOTABLERO 10
#define ALTOTABLERO 14

#define MARGENSUP 16
#define MARGENDCHO 160
#define MARGENIZQDO 160
#define MARGENINF 16

// Número de fragmentos distintos con los que
// formar las piezas
#define NUMFRAGMENTOS 7

// Ancho y alto de cada pieza
#define TAMANYOPIEZA 32

// Pieza especial como paso previo cuando explotan
#define PIEZAEXPLOSION 6
#define SINPIEZA -1

/* ----- Variables globales ----- */
PALETTE pal;
BITMAP *imagenFragmentos;
BITMAP *pantallaOcultas;

BITMAP *pieza[NUMFRAGMENTOS];

int partidaTerminada;
int x = MARGENIZQDO + TAMANYOPIEZA*5;
int y = MARGENSUP;
int incrX = TAMANYOPIEZA;
int incrY = 4;
int tecla;

// Tipo de imagen que corresponde a cada fragmento de pieza
int tipoFragmento[3];

// El tablero de fondo
int tablero[ANCHOTABLERO][ALTOTABLERO];

// Contador de fotogramas, para regular la velocidad de juego
int contadorFotogramas = 0;

// Prototipos de las funciones que usaremos
void comprobarTeclas();
void moverElementos();
void comprobarColisiones();
void dibujarElementos();
void pausaFotograma();
void intentarMoverDerecha();
void intentarMoverIzquierda();
void intentarBajar();
void rotarColores();
void colocarPiezaEnFondo();

// --- Bucle principal del juego -----
void buclePrincipal() {
    partidaTerminada = FALSE;
    do {
        comprobarTeclas();
        moverElementos();
        comprobarColisiones();
        dibujarElementos();
        pausaFotograma();
    } while (partidaTerminada != TRUE);
}

```

```

}

// -- Comprobac de teclas para mover personaje o salir
void comprobarTeclas() {

    if (keypressed()) {
        tecla = readkey() >> 8;
        if ( tecla == KEY_ESC )
            partidaTerminada = TRUE;
        if ( tecla == KEY_RIGHT )
            intentarMoverDerecha();
        if ( tecla == KEY_LEFT )
            intentarMoverIzquierda();
        if ( tecla == KEY_DOWN )
            intentarBajar();
        if (( tecla == KEY_SPACE ) || ( tecla == KEY_UP ))
            rotarColores();
        clear_keybuf();
    }
}

// -- Intenta mover la "pieza" hacia la derecha
void intentarMoverDerecha() {
    // Si no he llegado al margen
    if (x < ANCHOPANTALLA-MARGENDCHO-TAMANYOPIEZA)
    {
        // Y la casilla no esta ocupada
        int posX = (x+incrX-MARGENIZQDO)/TAMANYOPIEZA;
        int posYfinal = (y-MARGENSUP)/TAMANYOPIEZA + 3;
        if (tablero[posX][posYfinal] == SINPIEZA)
            x += incrX;
    }
}

// -- Intenta mover la "pieza" hacia la izquierda
void intentarMoverIzquierda() {
    if (x > MARGENIZQDO) // Si no he llegado al margen
    {
        // Y la casilla no esta ocupada
        int posX = (x-incrX-MARGENIZQDO)/TAMANYOPIEZA;
        int posYfinal = (y-MARGENSUP)/TAMANYOPIEZA + 3;
        if (tablero[posX][posYfinal] == SINPIEZA)
            x -= incrX;
    }
}

// -- Devuelve "true" si toca fondo y no puede bajar mas
bool tocadoFondo() {
    bool colision = false;
    // Si llega abajo del todo
    if (y >= ALTOPANTALLA-MARGENINF-3*TAMANYOPIEZA)
        colision = true;
    // o si toca con una pieza interior
    int posX = (x-MARGENIZQDO)/TAMANYOPIEZA;
    int posYfinal = (y-MARGENSUP)/TAMANYOPIEZA + 3;
    if (tablero[posX][posYfinal] != SINPIEZA)
        colision = true;
    return colision;
}

// -- Intenta mover la "pieza" hacia abajo
void intentarBajar() {
    y += incrY;
    if ( tocadoFondo() )
        colocarPiezaEnFondo();
}

// -- Rotar los colores de la "pieza"
void rotarColores() {
    int auxiliar = tipoFragmento[0];
    tipoFragmento[0] = tipoFragmento[1];
    tipoFragmento[1] = tipoFragmento[2];
    tipoFragmento[2] = auxiliar;
}

```

```

// -- Mover otros elementos del juego
void moverElementos() {
    // Las piezas bajan solas, pero unicamente un
    // fotograma de cada 10
    contadorFotogramas++;
    if (contadorFotogramas >= 10) {
        y += incrY;
        contadorFotogramas = 0;
        if ( tocadoFondo() )
            colocarPiezaEnFondo();
    }
}

// -- Comprobar colisiones de nuestro elemento con otros, o disparos con enemigos, etc
void comprobarColisiones() {
    // Por ahora, no hay colisiones que comprobar
}

// -- Dibujar elementos en pantalla
void dibujarElementos() {
    int i,j;

    // Borro la pantalla y dibujo la pieza
    clear_bitmap(pantallaOculta);

    // Dibujo el "fondo", con los trozos de piezas anteriores
    for (i=0; i<ANCHOTABLERO; i++)
        for (j=0; j<ALTOTABLERO; j++)
            if (tablero[i][j] != SINPIEZA)
                draw_sprite(pantallaOculta,
                    pieza[ tablero[i][j] ],
                    MARGENIZQDO + TAMANYOPIEZA*i,
                    MARGENSUP + TAMANYOPIEZA*j);

    // Dibujo la "pieza"
    for (i=0; i<3; i++)
        draw_sprite(pantallaOculta,
            pieza[ tipoFragmento[i] ], x, y+32*i);

    // Sincronizo con el barrido para evitar parpadeos
    // y vuelco la pantalla oculta
    vsync();
    blit(pantallaOculta, screen, 0, 0, 0, 0,
        ANCHOPANTALLA, ALTOPANTALLA);
}

// -- Pausa hasta el siguiente fotograma
void pausaFotograma() {
    // Para 25 fps: 1000/25 = 40 milisegundos de pausa
    rest(40);
}

// -- Crea una nueva pieza con componentes al azar
void crearNuevaPieza() {
    int i;
    for (i=0; i<3; i++)
        tipoFragmento[i] = rand() % (NUMFRAGMENTOS-1);
    x = MARGENIZQDO + TAMANYOPIEZA*5;
    y = MARGENSUP;
}

// -- Revisa el fondo para comprobar si hay varias
// piezas iguales unidas, que se puedan eliminar
void eliminarBloquesConectados() {
    int i,j,k;
    bool hayQueBorrar = false;
    // Busco en horizontal
    for (i=0; i<ANCHOTABLERO-2; i++)
        for (j=0; j<ALTOTABLERO; j++)
            if (tablero[i][j] != SINPIEZA) // Si hay pieza
                if ((tablero[i][j] == tablero[i+1][j]) && (tablero[i][j] == tablero[i+2][j]))
                    {
                        tablero[i][j] = PIEZAEXPLOSION;
                    }
}

```

```

        tablero[i+1][j] = PIEZAEXPLOSION;
        tablero[i+2][j] = PIEZAEXPLOSION;
        hayQueBorrar = true;
    }

// Busco en vertical
for (i=0; i<ANCHOTABLERO; i++)
    for (j=0; j<ALTOTABLERO-2; j++)
        if (tablero[i][j] != SINPIEZA) // Si hay pieza
            if ((tablero[i][j] == tablero[i][j+1]) && (tablero[i][j] == tablero[i][j+2]))
                {
                    tablero[i][j] = PIEZAEXPLOSION;
                    tablero[i][j+1] = PIEZAEXPLOSION;
                    tablero[i][j+2] = PIEZAEXPLOSION;
                    hayQueBorrar = true;
                }

// Diagonal 1
for (i=0; i<ANCHOTABLERO-2; i++)
    for (j=0; j<ALTOTABLERO-2; j++)
        if (tablero[i][j] != SINPIEZA) // Si hay pieza
            if ((tablero[i][j] == tablero[i+1][j+1]) && (tablero[i][j] == tablero[i+2][j+2]))
                {
                    tablero[i][j] = PIEZAEXPLOSION;
                    tablero[i+1][j+1] = PIEZAEXPLOSION;
                    tablero[i+2][j+2] = PIEZAEXPLOSION;
                    hayQueBorrar = true;
                }

// Diagonal 2
for (i=2; i<ANCHOTABLERO; i++)
    for (j=0; j<ALTOTABLERO-2; j++)
        if (tablero[i][j] != SINPIEZA) // Si hay pieza
            if ((tablero[i][j] == tablero[i-1][j+1]) && (tablero[i][j] == tablero[i-2][j+2]))
                {
                    tablero[i][j] = PIEZAEXPLOSION;
                    tablero[i-1][j+1] = PIEZAEXPLOSION;
                    tablero[i-2][j+2] = PIEZAEXPLOSION;
                    hayQueBorrar = true;
                }

// Borro, si es el caso
if (hayQueBorrar) {
    dibujarElementos();
    rest(200);
    bool algoHaCambiado;
    do {
        // Mientras haya "explosiones", recoloco
        algoHaCambiado = false;
        for (i=0; i<ANCHOTABLERO; i++)
            for (j=ALTOTABLERO-1; j>=0; j--) // Abajo a arriba
                if (tablero[i][j] == PIEZAEXPLOSION)
                    {
                        algoHaCambiado = true;
                        // Bajo las superiores
                        for (k=j; k>0; k--)
                            tablero[i][k] = tablero[i][k-1];
                        tablero[i][0] = SINPIEZA;
                    }
    } while (algoHaCambiado);
    // Y vuelvo a comprobar si hay nuevas cosas que borrar
    if (algoHaCambiado) {
        dibujarElementos();
        rest(200);
        eliminarBloquesConectados();
    }
}

// -- Coloca una pieza como parte del fondo, cuando
// llegamos a la parte inferior de la pantalla
// o tocamos una pieza inferior colocada antes
void colocarPiezaEnFondo(){
    int i;
    for (i=0; i<3; i++) {
        tablero
            [ (x-MARGENIZQDO)/TAMANYOPIEZA ]
            [ (y-MARGENSUP)/TAMANYOPIEZA + i ]
            = tipoFragmento[i];
    }
}

```

```

    }
    eliminarBloquesConectados();
    crearNuevaPieza();
}

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    int i,j;

    allegro_init();          // Inicializamos Allegro
    install_keyboard();
    install_timer();

                                // Intentamos entrar a modo grafico
    set_color_depth(32);
    if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        return 1;
    }

                                // e intentamos abrir imágenes
    imagenFragmentos = load_bmp("columnas_piezas.bmp", pal);
    if (!imagenFragmentos) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("No se ha podido abrir la imagen\n");
        return 1;
    }

    // Ahora reservo espacio para los otros sprites
    for (i=0; i<NUMFRAGMENTOS; i++)
    {
        pieza[i] = create_bitmap(32, 32);

        // Y los extraigo de la imagen "grande"
        blit(imagenFragmentos, pieza[i] // bitmaps de origen y destino
            , 32*i, 0 // coordenadas de origen
            , 0, 0 // posición de destino
            , 32, 32); // anchura y altura
    }

    set_palette(pal);

    srand(time(0));

    // Pantalla oculta para evitar parpadeos
    // (doble buffer)
    pantallaOcultas = create_bitmap(ANCHOPANTALLA, ALTOPANTALLA);

    // Vacío el tablero de fondo
    for (i=0; i<ANCHOTABLERO; i++)
        for (j=0; j<ALTOTABLERO; j++)
            tablero[i][j] = SINPIEZA;

    crearNuevaPieza();

    // Y termino indicando que no ha habido errores
    return 0;
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int i,j;

    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    // Bucle principal del juego
    buclePrincipal();
}

```



```

// Libero memoria antes de salir
for (i=0; i<NUMFRAGMENTOS; i++)
    destroy_bitmap(pieza[i]);
destroy_bitmap(pantallaOculto);

rest(1000);
return 0;
}

/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

27. Completando Columnas: borrado correcto, puntuación. (*)

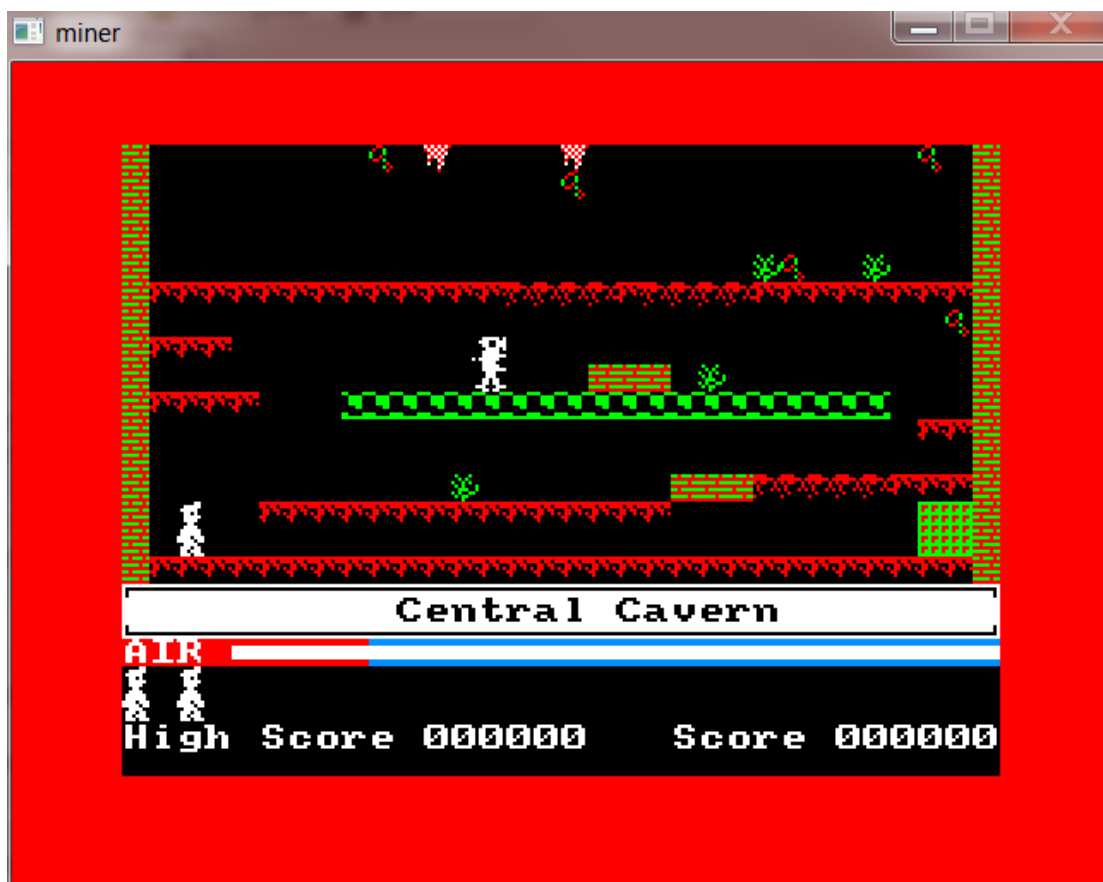
La última entrega de Columnas, aunque empieza a ser jugable, sigue necesiitando mejoras: la rutina de marcado de piezas no es buena (falla si hay 4 o más en línea), la presentación es muy pobre, y no se lleva cuenta de la puntuación.

Por eso, esta cuarta entrega borrará correctamente, que dibujará los límites de la pantalla de juego y calculará (y mostrará) la puntuación de la partida.

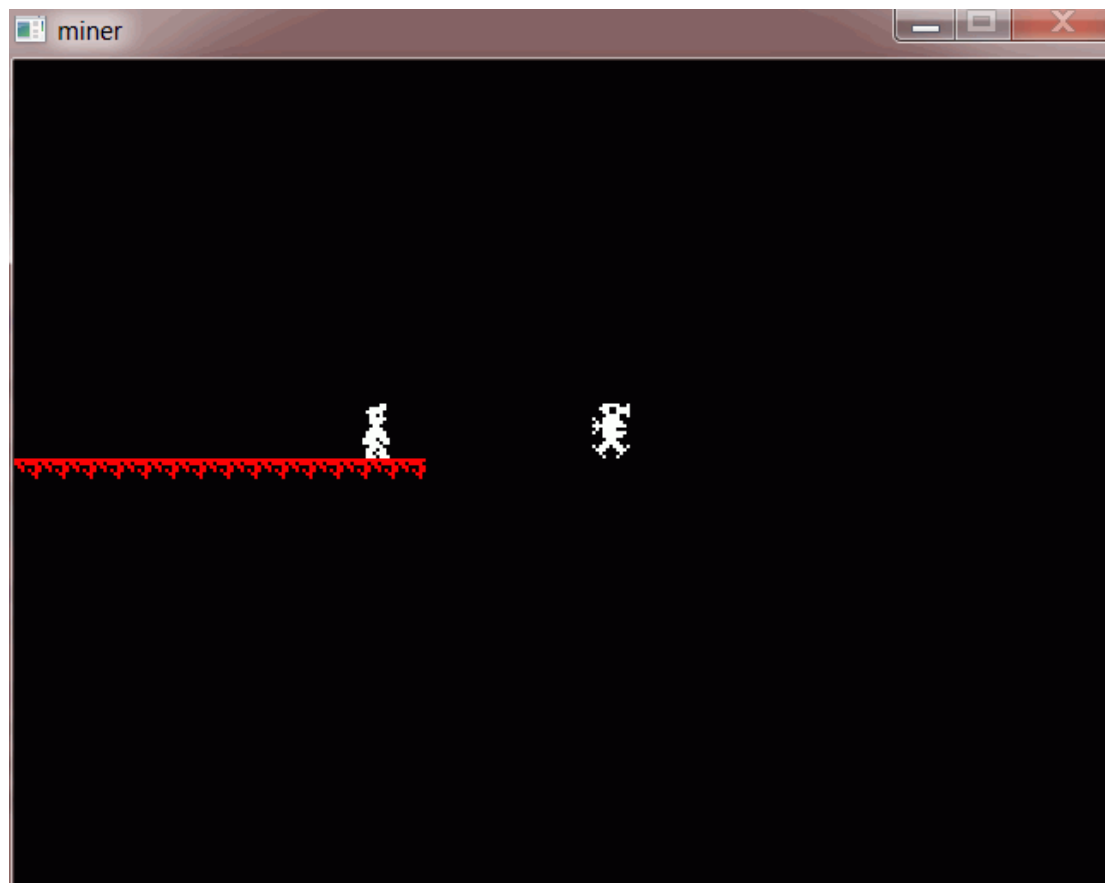
(Pronto disponible)

28. La aproximación orientada a objetos (1). Toma de contacto con un primer "arcade": MiniMiner (1) (*)

Vamos a hacer buena parte de la estructura de un juego "arcade". Imitaremos un clásico de principio de los 80, en el que el minero Willy tenía que ir recogiendo llaves hasta abrir la puerta que le permitía pasar al siguiente nivel:



En este primer acercamiento partiremos del esqueleto de juego que desarrollamos en el [tema 24](#), con un "bucle de juego" que se encarga de repetir las tareas habituales en un juego de este tipo. Ampliaremos un poco el esqueleto, de forma que muestre un personaje que podremos mover a derecha e izquierda, un enemigo que se moverá por sí mismo (también a derecha e izquierda) y un fragmento de suelo. Será algo tan sencillo como esto:



Añadiremos también una "pantalla de presentación" básica, que mostrará la imagen original del juego hasta que pulsemos una tecla. Pero apenas con estos pocos cambios, ya empezaremos a notar cosas "mejorables" en nuestra estructura.

Los cambios van a ser:

- Este juego cargará cuatro imágenes, con lo que la función "inicializa" será mucho más repetitiva que antes.
- La función "dibujarElementos" dibujará también el fondo (que es algo que dentro de poco ya no será tan trivial, así que crearemos una función específica) y el enemigo, lo que no supone grandes cambios.
- Ahora hay un enemigo que mover, así que "moverElementos" llamará ahora a un "moverEnemigo", que cambiará su X de modo que se mueva de un lado a otro de la pantalla.
- El enemigo, como es de esperar, tendrá su propia X y su propia Y, por lo que necesitamos otras dos variables xEnemigo e "yEnemigo" (realmente, también un incremento de X, que equivaldrá a su velocidad). Esto empieza a hacer el fuente menos legible. Una alternativa ligeramente más legible sería que el enemigo fuera un "struct" que contuviera su imagen, su "x" y su "y", etc. Eso se parece más a lo que haremos, pero realmente no usaremos "structs" sino "clases", ya veremos por qué.
- Además, añadiremos una función "lanzarPresentación", que será la que muestre la imagen que por ahora va a ser nuestra presentación rudimentaria.

El fuente podría quedar así:

```

/*-----*/
/*  Intro a la programac de  */
/*  juegos, por Nacho Cabanes  */
/*                               */
/*  miner01.cpp                 */
/*                               */
/*  Ejemplo:                    */
/*  Primer acercamiento a      */
/*  "MiniMiner"                */
/*                               */
/*  Comprobado con:            */
/*  - DevC++ 4.9.9.2(gcc 3.4.2) */
/*  y Allegro 4.03 - WinXP     */
/*-----*/

#include <allegro.h>

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 640

```

```

#define ALTOPANTALLA 480

/* ----- Variables globales ----- */
PALETTE pal;
BITMAP *personaje;
BITMAP *enemigo;
BITMAP *presentacion;
BITMAP *fragmentoSuelo;
BITMAP *pantallaOcultas;

int partidaTerminada;
int x = 200;
int y = 200;
int incrX = 4;
int incrY = 4;
int tecla;
int xEnemigo = 500;
int incrXEnemigo = 2;
int yEnemigo = 200;
int ySuelo = 232;

// Prototipos de las funciones que usaremos
void comprobarTeclas();
void moverElementos();
void comprobarColisiones();
void dibujarElementos();
void pausaFotograma();
void moverDerecha();
void moverIzquierda();
void lanzarPresentacion();
void moverEnemigo();
void dibujarFondo();

// --- Bucle principal del juego -----
void buclePrincipal() {
    partidaTerminada = FALSE;
    do {
        comprobarTeclas();
        moverElementos();
        comprobarColisiones();
        dibujarElementos();
        pausaFotograma();
    } while (partidaTerminada != TRUE);
}

// -- Comprobac de teclas para mover personaje o salir
void comprobarTeclas() {

    if (keypressed()) {
        tecla = readkey() >> 8;
        if ( tecla == KEY_ESC )
            partidaTerminada = TRUE;
        if ( tecla == KEY_RIGHT )
            moverDerecha();
        if ( tecla == KEY_LEFT )
            moverIzquierda();
        clear_keybuf();
    }
}

// -- Intenta mover el personaje hacia la derecha
void moverDerecha() {
    x += incrX;
}

// -- Intenta mover el personaje hacia la izquierda
void moverIzquierda() {
    x -= incrX;
}

// -- Mover otros elementos del juego
void moverElementos() {
    moverEnemigo();
}

```

```

}

// -- Comprobar colisiones de nuestro elemento con otros, o disparos con enemigos, etc
void comprobarColisiones() {
    // Por ahora, no hay colisiones que comprobar
}

// -- Dibujar elementos en pantalla
void dibujarElementos() {

    // Borro la pantalla
    clear_bitmap(pantallaOculto);

    // Dibujo el "fondo", con los trozos de piezas anteriores
    dibujarFondo();

    // Dibujo el personaje y el enemigo
    draw_sprite(pantallaOculto, personaje, x, y);
    draw_sprite(pantallaOculto, enemigo, xEnemigo, yEnemigo);

    // Sincronizo con el barrido para evitar parpadeos
    // y vuelco la pantalla oculta
    vsync();
    blit(pantallaOculto, screen, 0, 0, 0, 0,
        ANCHOPANTALLA, ALTOPANTALLA);
}

// -- Pausa hasta el siguiente fotograma
void pausaFotograma() {
    // Para 25 fps: 1000/25 = 40 milisegundos de pausa
    rest(40);
}

// -- Funciones que no son de la logica juego, sino de
// funcionamiento interno de otros componentes

// -- Pantalla de presentacion
void lanzarPresentacion() {
    draw_sprite(pantallaOculto, presentacion, 0, 0);
    blit(pantallaOculto, screen, 0, 0, 0, 0,
        ANCHOPANTALLA, ALTOPANTALLA);
    readkey();
}

// -- Mover el enemigo a su siguiente posicion
void moverEnemigo() {
    xEnemigo += incrXEnemigo;
    // Da la vuelta si llega a un extremo
    if ((xEnemigo > ANCHOPANTALLA-30) || (xEnemigo < 30))
        incrXEnemigo = -incrXEnemigo;
}

// -- Dibuja el fondo (por ahora, apenas un fragmento de suelo)
void dibujarFondo() {
    int i;
    int anchoImagen = 16;
    for (i=0; i<15; i++)
        draw_sprite(pantallaOculto, fragmentoSuelo,
            i*anchoImagen, ySuelo);
}

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    int i,j;

    allegro_init();           // Inicializamos Allegro
    install_keyboard();
    install_timer();

                                // Intentamos entrar a modo grafico
    set_color_depth(32);
    if (set_gfx_mode(GFX_SAFE, ANCHOPANTALLA, ALTOPANTALLA, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",

```

```

        allegro_error);
    return 1;
}

// e intentamos abrir imágenes
personaje = load_bmp("personaje.bmp", pal);
if (!personaje) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("No se ha podido abrir la imagen\n");
    return 1;
}

enemigo = load_bmp("enemigo.bmp", pal);
if (!enemigo) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("No se ha podido abrir la imagen\n");
    return 1;
}

fragmentoSuelo = load_bmp("suelo.bmp", pal);
if (!fragmentoSuelo) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("No se ha podido abrir la imagen\n");
    return 1;
}

presentacion = load_bmp("miner.bmp", pal);
if (!presentacion) {
    set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
    allegro_message("No se ha podido abrir la imagen\n");
    return 1;
}

// Pantalla oculta para evitar parpadeos
// (doble buffer)
pantallaOcultas = create_bitmap(ANCHOPANTALLA, ALTOPANTALLA);

// Y termino indicando que no ha habido errores
return 0;
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int i,j;

    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    lanzarPresentacion();
    buclePrincipal();

    destroy_bitmap(personaje);
    destroy_bitmap(enemigo);
    destroy_bitmap(fragmentoSuelo);
    destroy_bitmap(presentacion);
    destroy_bitmap(pantallaOcultas);

    rest(1000);
    return 0;
}

/* Termina con la "macro" que me pide Allegro */
END_OF_MAIN();

```

A pesar de que el juego todavía no hace prácticamente nada, ya ocupa 254 líneas, ya empieza a tener fragmentos repetitivos, y además está lleno por todos lados de órdenes que son propias de Allegro, por lo que migrar el programa para que use otra biblioteca gráfica sería costoso... en la próxima entrega empezaremos a solucionar estos problemas...

29. La aproximación orientada a objetos (1). MiniMiner 2: Aislado del hardware. (*)

Queremos que nuestro juego siga los principios de la "Programación orientada a objetos": en vez de tratarse de un único "macroprograma", lo plantearemos como una serie de objetos que se pasan mensajes unos a otros. Esto tendrá una serie de ventajas, entre las que podemos destacar:

- Es un planteamiento "más natural", porque nuestro juego tiene un "personaje" que manejar, varias "pantallas" que recorrer, en las cuales hay "llaves" que recoger, pero existen "enemigos" que tratarán de evitarlo. Todo ello suena a que participan "objetos" de distintas "clases".
- La "herencia" nos ayudará a escribir menos código repetitivo: si creamos una clase de objetos llamada "ElementoGráfico", que sea capaz de mostrarse en una cierta posición de pantalla, y decimos que nuestro "Personaje" es un tipo de "ElementoGráfico", no será necesario indicarle cómo debe dibujarse en pantalla. Como es un "ElementoGráfico", esa será una de las cosas que "sabrán hacer".
- Como ahora el fuente estará desglosado en varios objetos, cada uno de los cuales tiene unas funciones claras, sería más fácil repartir el trabajo entre varias personas, si formáramos parte de un grupo de programadores que trabaja en un proyecto común.
- Además, podemos crear clases auxiliares, que no sean estrictamente necesarias para nuestro juego, pero que nos puedan permitir trabajar con más comodidad. Por ejemplo, podemos definir una clase Hardware que nos oculte los detalles de Allegro o de la biblioteca gráfica que estamos empleando, de modo que el trabajo de adaptar nuestro juego para que funcione usando SDL o cualquier otra biblioteca gráfica alternativa (si llegara a ser necesario) sea mucho menor.

La primera mejora que vamos a hacer va a estar relacionada con la última ventaja que hemos mencionado: aislaremos bastante el juego de la biblioteca Allegro, ayudándonos de dos primeras clases:

- Una clase "Hardware", que nos oculte el acceso a teclado (con funciones para comprobar qué tecla se ha pulsado) y a pantalla (con funciones para borrar la pantalla, dibujar una imagen en la pantalla oculta o hacer visible esa pantalla oculta -lo que habíamos llamado el "doble buffer"-).
- Una clase "Elemento Gráfico" (ElementoGraf), con funciones para crearlo a partir de un fichero, moverlo a ciertas coordenadas, comprobar si ha "chocado" con otro elemento gráfico, etc.
- El resto del juego seguirá estando contenido en el fichero "miner.cpp", que será bastante parecido a como era antes (pero algo más sencillo, si lo hemos hecho bien).

Necesitaremos un fichero de cabecera (.h) y uno de desarrollo (.cpp) para cada clase de objetos que queramos definir. Por tanto, en este momento tendremos 5 ficheros: los dos de la clase Hardware, los dos de la clase ElementoGraf, y el cuerpo del juego.

El fichero de cabecera de Hardware podría ser así:

```

/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  Hardware.h:                 */
/*  Clase "Hardware"            */
/*  para Allegro                */
/*  Fichero de cabecera         */
/*                               */
/*  Parte de "MiniMiner"        */
/*                               */
/*  Ejemplo:                    */
/*  Primer acercamiento a      */
/*  "MiniMiner"                 */
/*                               */
/*  Comprobado con:            */
/*  - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*  y Allegro 4.03 - WinXP      */
/*-----*/

#ifdef Hardware_h
#define Hardware_h

#include <vector>
#include <string>
using namespace std;

#include <allegro.h>

```

```

#include "ElementoGraf.h"

#define TECLA_ESC KEY_ESC
#define TECLA_DCHA KEY_RIGHT
#define TECLA_ARRB KEY_UP
#define TECLA_ABAJ KEY_DOWN
#define TECLA_IZQD KEY_LEFT
#define TECLA_ESPACIO KEY_SPACE
#define TECLA_A KEY_a
#define TECLA_B KEY_b
#define TECLA_C KEY_c
#define TECLA_D KEY_d
#define TECLA_E KEY_e
#define TECLA_F KEY_f
#define TECLA_G KEY_g
#define TECLA_H KEY_h
#define TECLA_I KEY_i
#define TECLA_J KEY_j
#define TECLA_K KEY_k
#define TECLA_L KEY_l
#define TECLA_M KEY_m
#define TECLA_N KEY_n
#define TECLA_O KEY_o
#define TECLA_P KEY_p
#define TECLA_Q KEY_q
#define TECLA_R KEY_r
#define TECLA_S KEY_s
#define TECLA_T KEY_t
#define TECLA_U KEY_u
#define TECLA_V KEY_v
#define TECLA_W KEY_w
#define TECLA_X KEY_x
#define TECLA_Y KEY_y
#define TECLA_Z KEY_z

#define TECLA_0 KEY_0
#define TECLA_1 KEY_1
#define TECLA_2 KEY_2
#define TECLA_3 KEY_3
#define TECLA_4 KEY_4
#define TECLA_5 KEY_5
#define TECLA_6 KEY_6
#define TECLA_7 KEY_7
#define TECLA_8 KEY_8
#define TECLA_9 KEY_9

#define TECLA_F1 KEY_F1

class Hardware {
public:

    void inicializar(int ancho, int alto);
    bool comprobarTecla();

    bool comprobarTecla(int codigoTecla);
    int esperarTecla();
    void vaciarBufferTeclado();
    bool algunaTeclaPulsada();

    void borrarOculto();
    void dibujarOculto(ElementoGraf e);

    void visualizarOculto();

    void pausa(long ms);

private:

    int anchoPantalla;
    int altoPantalla;

    BITMAP *pantallaOculto;
    int maxX;
    int maxY;
    int colores;
    int teclaPulsada;
    int posXRaton;
    int posYRaton;

```

```
private:
    BITMAP pantallaVisible;
    BITMAP fondo;

};
#endif
```

Y el desarrollo podría ser así:

```
/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* Hardware.cpp: */
/* Clase "Hardware" */
/* para Allegro */
/* Fichero de desarrollo */
/* */
/* Parte de "MiniMiner" */
/* */
/* Ejemplo: */
/* Primer acercamiento a */
/* "MiniMiner" */
/* */
/* Comprobado con: */
/* - DevC++ 4.9.9.2(gcc 3.4.2) */
/* y Allegro 4.03 - WinXP */
/*-----*/

#include "Hardware.h"
#include "ElementoGraf.h"
#include "iostream"

#ifdef __cplusplus
    #include <cstdlib>
#else
    #include <stdlib.h>
#endif

#include <math.h>
#include <string>

#include <allegro.h>

void Hardware::inicializar(int ancho, int alto)
{
    cout << "hard-ctor";
    allegro_init(); // Inicializamos Allegro
    install_keyboard();
    install_timer();

    anchoPantalla = ancho;
    altoPantalla = alto;

    // Intentamos entrar a modo grafico
    set_color_depth(32);
    if (set_gfx_mode(GFX_SAFE, ancho, alto, 0, 0) != 0) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message(
            "Incapaz de entrar a modo grafico\n%s\n",
            allegro_error);
        exit( 1 );
    }

    // Pantalla oculta para evitar parpadeos
    // (doble buffer)
    pantallaOculto = create_bitmap(ancho, alto);
}

/** teclaPulsada: devuelve TRUE si alguna tecla
 * ha sido pulsada
```



```

    */
bool Hardware::algunaTeclaPulsada()
{
    return keypressed;
}

/** comprobarTecla: devuelve TRUE si una cierta tecla
 * ha sido pulsada
 */
bool Hardware::comprobarTecla(int codigoTecla)
{
    return key[codigoTecla];
}

/** esperarTecla: pausa hasta que se pulse una tecla,
 * devuelve el codigo de tecla pulsada
 */
int Hardware::esperarTecla()
{
    return readkey() >> 8;
}

void Hardware::borrarOculto()
{
    // Borrar pantalla de fondo
    clear_bitmap(pantallaOculto);
}

void Hardware::dibujarOculto(ElementoGraf e)
{
    draw_sprite( pantallaOculto, e.leerImagen(), e.leerX(), e.leerY() );
}

void Hardware::visualizarOculto()
{
    // Sincronizo con el barrido para evitar parpadeos
    // y vuelco la pantalla oculta
    vsync();
    blit(pantallaOculto, screen, 0, 0, 0, 0,
        anchoPantalla, altoPantalla);
}

void Hardware::pausa(long ms)
{
    rest(ms);
}

```

Del mismo modo, el fichero de cabecera de ElementoGraf podría ser:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* ElementoGraf.h: */
/* Clase "elemento gráfico" */
/* para Allegro */
/* Fichero de cabecera */
/* */
/* Parte de "MiniMiner" */
/* */
/* Ejemplo: */
/* Primer acercamiento a */
/* "MiniMiner" */
/* */
/* Comprobado con: */
/* - DevC++ 4.9.9.2(gcc 3.4.2) */
/* y Allegro 4.03 - WinXP */
/*-----*/

```

```

#ifndef ElementoGraf_h
#define ElementoGraf_h

#include <allegro.h>

class ElementoGraf {

public:

    void moverA(int x, int y);
    void indicarAnchoAlto(int an, int al);
    void crearDesdeFichero(char *nombre);
    BITMAP* leerImagen();
    int leerX();
    int leerY();
    int leerAnchura();
    int leerAltura();

    bool colisionCon(ElementoGraf e2);
    bool colisionCon(int x, int y, int ancho, int alto);

protected:

    int posX;
    int posY;
    int anchura, altura;
    int anchuraOrig, alturaOrig;
    int colorTransp;

private:

    BITMAP* imagen;

};
#endif

```

Y su desarrollo:

```

/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes   */
/*                               */
/*  ElementoGraf.cpp:           */
/*  Clase "elemento gráfico"    */
/*  para Allegro                */
/*  Fichero de desarrollo      */
/*                               */
/*  Parte de "MiniMiner"       */
/*                               */
/*  Ejemplo:                    */
/*  Primer acercamiento a      */
/*  "MiniMiner"                 */
/*                               */
/*  Comprobado con:            */
/*  - DevC++ 4.9.9.2(gcc 3.4.2) */
/*  y Allegro 4.03 - WinXP     */
/*-----*/

#include "ElementoGraf.h"

#ifdef __cplusplus
    #include <cstdlib>
#else
    #include <stdlib.h>
#endif

#include <allegro.h>

/** moverA: cambia la posicion del elemento grafico
 * (actualiza tambien las coordenadas del centro)
 */
void ElementoGraf::moverA(int x, int y)
{
    posX = x;
    posY = y;
}

/** indicarAnchoAlto: indica el ancho y el alto del

```

```

    * elemento gráfico, para que se pueda calcular colisiones
    */
void ElementoGraf::indicarAnchoAlto(int an, int al)
{
    anchura = an;
    altura = al;
}

/** leerX: devuelve la coordenada X de la posicion
    */
int ElementoGraf::leerX()
{
    return posX;
}

/** leerY: devuelve la coordenada Y de la posicion
    */
int ElementoGraf::leerY()
{
    return posY;
}

/** leerAnchura: devuelve la anchura del elemento grafico
    */
int ElementoGraf::leerAnchura()
{
    return anchura;
}

/** leerAltura: devuelve la altura del elemento grafico
    */
int ElementoGraf::leerAltura()
{
    return altura;
}

/** leerImagen: devuelve la imagen (bitmap) del elemento grafico
    */
BITMAP* ElementoGraf::leerImagen()
{
    return imagen;
}

/** crearDesdeFichero: lee desde fichero el bitmap, y actualiza
    * su anchura y altura
    */
void ElementoGraf::crearDesdeFichero(char *nombre)
{
    imagen = load_bmp(nombre, NULL);
    if (!imagen) {
        set_gfx_mode(GFX_TEXT, 0, 0, 0, 0);
        allegro_message("No se ha podido abrir la imagen\n");
        exit( 1 );
    }
}

/** colisionCon: devuelve si hay colision del ElementoGraf con otro
    */
bool ElementoGraf::colisionCon(ElementoGraf e2)
{
    return colisionCon(e2.posX, e2.posY, e2.anchura, e2.altura);
}

/** colisionCon: devuelve si hay colision del ElementoGraf con
    * un rectangulo dado por sus coordenadas
    */
bool ElementoGraf::colisionCon(int x, int y, int ancho, int alto)
{
    if ((this->posX+this->anchura > x)
        && (this->posY+this->altura > y)
        && (x+ancho > this->posX)
        && (y+alto > this->posY))
        return true;
    else
        return false;
}

```

Y el cuerpo del programa, a pesar de que todavía tiene partes repetitivas (como eso de que haya una variable x para el personaje, pero también otra xEnemigo y otra xSuelo), ya ocupa unas 60 líneas menos:

```

/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  miner02.cpp                  */
/*                               */
/*  Ejemplo:                     */
/*  "MiniMiner" (version 0.02)   */
/*                               */
/*  Comprobado con:              */
/*  - DevC++ 4.9.9.2(gcc 3.4.2)  */
/*  y Allegro 4.03 - WinXP      */
/*-----*/

#include "Hardware.h"
#include "ElementoGraf.h"

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 640
#define ALTOPANTALLA 480

/* ----- Variables globales ----- */
Hardware hard;
ElementoGraf personaje, enemigo, presentacion, fragmentoSuelo;

int partidaTerminada;
int x = 200;
int y = 200;
int incrX = 4;
int incrY = 4;
int tecla;
int xEnemigo = 500;
int incrXEnemigo = 2;
int yEnemigo = 200;
int ySuelo = 232;

// Prototipos de las funciones que usaremos
void comprobarTeclas();
void moverElementos();
void comprobarColisiones();
void dibujarElementos();
void pausaFotograma();
void moverDerecha();
void moverIzquierda();
void lanzarPresentacion();
void moverEnemigo();
void dibujarFondo();

// --- Bucle principal del juego -----
void buclePrincipal() {
    partidaTerminada = false;
    do {
        comprobarTeclas();
        moverElementos();
        comprobarColisiones();
        dibujarElementos();
        pausaFotograma();
    } while (partidaTerminada != true);
}

// -- Comprobac de teclas para mover personaje o salir
void comprobarTeclas() {

    if (hard.comprobarTecla(TECLA_ESC))
        partidaTerminada = true;

    if (hard.comprobarTecla(TECLA_DCHA))
        moverDerecha();
    else if (hard.comprobarTecla(TECLA_IZQD))
        moverIzquierda();
}

```

```

// -- Intenta mover el personaje hacia la derecha
void moverDerecha() {
    x += incrX;
}

// -- Intenta mover el personaje hacia la izquierda
void moverIzquierda() {
    x -= incrX;
}

// -- Mover otros elementos del juego
void moverElementos() {
    moverEnemigo();
}

// -- Comprobar colisiones de nuestro elemento con otros, o disparos con enemigos, etc
void comprobarColisiones() {
    // Por ahora, no hay colisiones que comprobar
}

// -- Dibujar elementos en pantalla
void dibujarElementos() {

    hard.borrarOculta();
    dibujarFondo();
    enemigo.moverA(xEnemigo, yEnemigo);
    hard.dibujarOculta(enemigo);
    personaje.moverA(x, y);
    hard.dibujarOculta(personaje);
    hard.visualizarOculta();
}

// -- Pausa hasta el siguiente fotograma
void pausaFotograma() {
    // Para 25 fps: 1000/25 = 40 milisegundos de pausa
    hard.pausa(40);
}

// -- Funciones que no son de la logica juego, sino de
// funcionamiento interno de otros componentes

// -- Pantalla de presentacion
void lanzarPresentacion() {
    presentacion.moverA(0,0);
    hard.dibujarOculta(presentacion);
    hard.visualizarOculta();
    hard.esperarTecla();
}

// -- Mover el enemigo a su siguiente posicion
void moverEnemigo() {
    xEnemigo += incrXEnemigo;
    // Da la vuelta si llega a un extremo
    if ((xEnemigo > ANCHOPANTALLA-30) || (xEnemigo < 30))
        incrXEnemigo = -incrXEnemigo;
}

// -- Dibuja el fondo (por ahora, apenas un fragmento de suelo)
void dibujarFondo() {
    int i;
    int anchoImagen = 16;
    for (i=0; i<15; i++)
    {
        fragmentoSuelo.moverA(i*anchoImagen, ySuelo);
        hard.dibujarOculta(fragmentoSuelo);
    }
}

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    hard.inicializar(640,480);

    personaje.crearDesdeFichero("personaje.bmp");
}

```

```

    enemigo.crearDesdeFichero("enemigo.bmp");
    fragmentoSuelo.crearDesdeFichero("suelo.bmp");
    presentacion.crearDesdeFichero("miner.bmp");

    // Y termino indicando que no ha habido errores
    return 0;
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int i,j;

    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    lanzarPresentacion();
    buclePrincipal();

    hard.pausa(1000);
    return 0;
}
/* Termina con la "macro" que me pide Allegro */
END_OF_MAIN();

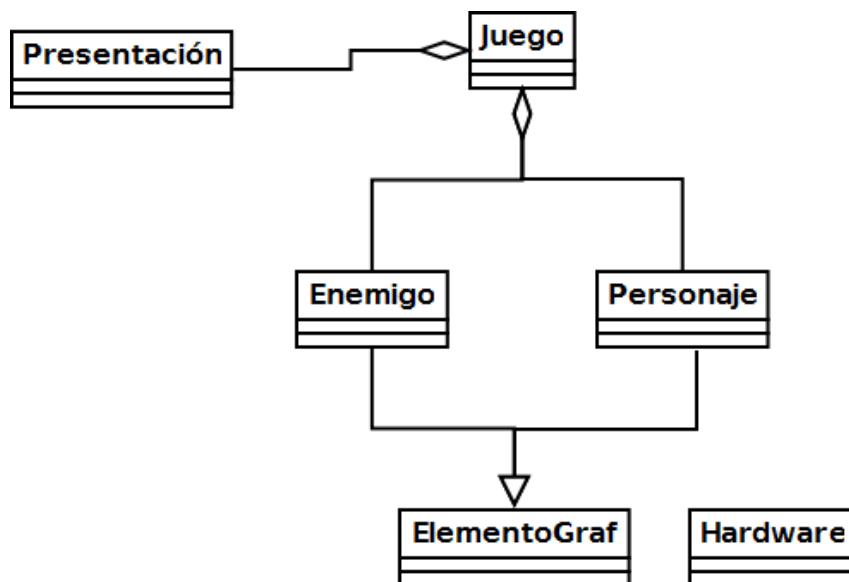
```

30. La aproximación orientada a objetos (3). MiniMiner 3: Personaje y enemigo como clases (*)

Ya habíamos creado las clases auxiliares "Hardware" y "ElementoGraf". Ahora vamos a empezar a descomponer la lógica de juego como una serie de objetos que cooperan entre sí. En este acercamiento vamos a incluir:

- Una clase "Presentacion", para representar a la pantalla de presentación del juego, y que tendrá solamente un método "mostrar()", que por ahora mostrará una imagen estática (aprovecharemos para que realmente sea un pantalla de presentación, y no la pantalla de juego).
- Una clase "Personaje", que heredará de "ElementoGraf" y añadirá sólo dos métodos: "moverDerecha()" y "moverIzquierda()", que desplazarán al personaje en un sentido u otro, para responder a las pulsaciones de teclas por parte del usuario del juego.
- Una clase "Enemigo", que también heredará de "ElementoGraf" y lo ampliará con un único método: "mover()", que calcule la siguiente posición del enemigo. En este juego se limitará a moverse a un lado y a otro, pero en juegos más avanzados, podría existir una cierta inteligencia en esta función "mover()", bien porque nos persiguiera siempre, bien porque lo hiciera sólo cuando estuviéramos cerca (y si estamos lejos "no nos ve"), bien, porque siguiese un patrón de movimiento más complejo.

El diagrama de clases (sin detallar los métodos ni los atributos de cada clase de objetos) podría ser:



Y ninguna de estas tres clases es complicada de crear...

El fichero de cabecera de Presentacion podría ser así:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* Presentacion.h: */
/* La "presentacion" de */
/* Miniminer (version 0.03) */
/* Fichero de cabecera */
/* */
/* Comprobado con: */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/* y Allegro 4.03 - WinXP */
/*-----*/

#ifndef Present_h
#define Present_h

#include "ElementoGraf.h"
#include "Hardware.h"

class Presentacion {

public:

    Presentacion();
    int mostrar(Hardware h);

private:
    ElementoGraf cartel;

};
#endif
  
```

Y el desarrollo podría ser así:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* Presentacion.cpp: */
/* La "presentacion" de */
/* Miniminer (version 0.03) */
/* Fichero de desarrollo */
/* */
/* Comprobado con: */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/* y Allegro 4.03 - WinXP */
/*-----*/
  
```

```

#include "Hardware.h"
#include "ElementoGraf.h"
#include "Presentacion.h"

#ifdef __cplusplus
#include <cstdlib>
#else
#include <stdlib.h>
#endif
using namespace std;

/** constructor
 * y lee los records
 */
Presentacion::Presentacion() {
    cartel.crearDesdeFichero( "miner.bmp" );
}

/** bienvenida: muestra la pantalla de bienvenida
 * y espera una tecla
 */
int Presentacion::mostrar(Hardware h) {
    int tecla;
    // Primero muestro el cartel antiguo
    hard = h;
    hard.borrarOculto();
    cartel.moverA(0,0);
    hard.dibujarOculto(cartel);
    hard.visualizarOculto();

    hard.esperarTecla();
}

```

Del forma similar, el fichero de cabecera de Personaje podría ser:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* Personaje.h: */
/* El "personaje" de */
/* Miniminer (version 0.03) */
/* Fichero de cabecera */
/* */
/* Comprobado con: */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/* y Allegro 4.03 - WinXP */
/*-----*/

#ifndef Personaje_h
#define Personaje_h

#include "ElementoGraf.h"
#include "Hardware.h"

class Personaje: public ElementoGraf {

public:

    Personaje();

    void moverDerecha();
    void moverIzquierda();
};

#endif

```

Y su desarrollo:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* Personaje.cpp: */
/* El "personaje" de */
/* Miniminer (version 0.03) */

```



```

/*      Fichero de desarrollo      */
/*      *                          */
/* Comprobado con:                */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*   y Allegro 4.03 - WinXP      */
/*-----*/

#include "Hardware.h"
#include "ElementoGraf.h"
#include "Personaje.h"

#ifdef __cplusplus
#include <cstdlib>
#include <time.h>
#else
#include <stdlib.h>
#include <time.h>
#endif
using namespace std;

/** constructor
 * y lee los records
 */
Personaje::Personaje() {
    posX = 200;
    posY = 200;
    crearDesdeFichero( "personaje.bmp" );
}

/** moverDerecha: mueve el personaje hacia la derecha
 */
void Personaje::moverDerecha() {
    posX += 4;
}

/** moverIzquierda: mueve el personaje hacia la izquierda
 */
void Personaje::moverIzquierda() {
    posX -= 4;
}

```

Finamente, el fichero de cabecera de Enemigo sería:

```

/*-----*/
/* Intro a la programac de      */
/* juegos, por Nacho Cabanes    */
/*                               */
/* Enemigo.h:                   */
/* El "enemigo" de              */
/* Miniminer (version 0.03) */
/* Fichero de cabecera          */
/*                               */
/* Comprobado con:             */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*   y Allegro 4.03 - WinXP     */
/*-----*/

#ifndef Enemigo_h
#define Enemigo_h

#include "ElementoGraf.h"
#include "Hardware.h"

class Enemigo: public ElementoGraf {

public:

    Enemigo();
    void mover();

private:

    int incrX;
};

```

```
#endif
```

Y su desarrollo:

```
/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  Enemigo.cpp:                 */
/*  El "enemigo" de             */
/*  Miniminer (version 0.03)    */
/*  Fichero de desarrollo      */
/*                               */
/*  Comprobado con:            */
/*  - DevC++ 4.9.9.2(gcc 3.4.2) */
/*  y Allegro 4.03 - WinXP     */
/*-----*/

#include "Hardware.h"
#include "ElementoGraf.h"
#include "Enemigo.h"

#ifdef __cplusplus
#include <cstdlib>
#include <time.h>
#else
#include <stdlib.h>
#include <time.h>
#endif
using namespace std;

/** constructor
 * y lee los records
 */
Enemigo::Enemigo() {
    posX = 500;
    posY = 200;
    incrX = 2;
    crearDesdeFichero( "enemigo.bmp" );
}

/** mover: mueve el personaje según indique su lógica
 */
void Enemigo::mover() {
    posX += incrX;
    // Da la vuelta si llega a un extremo
    if ((posX > 570) || (posX < 30))
        incrX = -incrX;
}

```

Y el cuerpo del programa, que vuelve a reducir su tamaño (esta vez en 20 líneas) y a ganar en legibilidad quedaría así:

```
/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  miner03.cpp                 */
/*                               */
/*  Ejemplo:                    */
/*  "MiniMiner" (version 0.03)  */
/*                               */
/*  Comprobado con:            */
/*  - DevC++ 4.9.9.2(gcc 3.4.2) */
/*  y Allegro 4.03 - WinXP     */
/*-----*/

#include "Hardware.h"
#include "ElementoGraf.h"
#include "Personaje.h"
#include "Enemigo.h"
#include "Presentacion.h"

```

```

/* ----- Constantes globales ----- */
#define ANCHOPANTALLA 640
#define ALTOPANTALLA 480

/* ----- Variables globales ----- */
Hardware hard;
Personaje *personaje;
Enemigo *enemigo;
Presentacion *presentacion;
ElementoGraf *fragmentoSuelo;

int partidaTerminada;
int incrX = 4;
int incrY = 4;
int tecla;
int ySuelo = 232;

// Prototipos de las funciones que usaremos
void comprobarTeclas();
void moverElementos();
void comprobarColisiones();
void dibujarElementos();
void pausaFotograma();
void moverDerecha();
void moverIzquierda();
void lanzarPresentacion();
void moverEnemigo();
void dibujarFondo();

// --- Bucle principal del juego -----
void buclePrincipal() {
    partidaTerminada = false;
    do {
        comprobarTeclas();
        moverElementos();
        comprobarColisiones();
        dibujarElementos();
        pausaFotograma();
    } while (partidaTerminada != true);
}

// -- Comprobac de teclas para mover personaje o salir
void comprobarTeclas() {

    if (hard.comprobarTecla(TECLA_ESC))
        partidaTerminada = true;

    if (hard.comprobarTecla(TECLA_DCHA))
        personaje->moverDerecha();
    else if (hard.comprobarTecla(TECLA_IZQD))
        personaje->moverIzquierda();
}

// -- Mover otros elementos del juego
void moverElementos() {
    enemigo->mover();
}

// -- Comprobar colisiones de nuestro elemento con otros, o disparos con enemigos, etc
void comprobarColisiones() {
    // Por ahora, no hay colisiones que comprobar
}

// -- Dibujar elementos en pantalla
void dibujarElementos() {

    hard.borrarOculta();
    dibujarFondo();
    hard.dibujarOculta( *enemigo );
    hard.dibujarOculta( *personaje );
    hard.visualizarOculta();
}

```

```

// -- Pausa hasta el siguiente fotograma
void pausaFotograma() {
    // Para 25 fps: 1000/25 = 40 milisegundos de pausa
    hard.pausa(40);
}

// -- Funciones que no son de la logica de juego, sino de
// funcionamiento interno de otros componentes

// -- Pantalla de presentacion
void lanzarPresentacion() {
    presentacion->mostrar(hard);
}

// -- Dibuja el fondo (por ahora, apenas un fragmento de suelo)
void dibujarFondo() {
    int i;
    int anchoImagen = 16;
    for (i=0; i<15; i++)
    {
        fragmentoSuelo->moverA(i*anchoImagen, ySuelo);
        hard.dibujarOculta( *fragmentoSuelo );
    }
}

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    hard.inicializar(640,480);
    personaje = new Personaje();
    enemigo = new Enemigo();
    presentacion = new Presentacion();
    fragmentoSuelo = new ElementoGraf();

    fragmentoSuelo->crearDesdeFichero("suelo.bmp");

    // Y termino indicando que no ha habido errores
    return 0;
}

/* ----- */
/* ----- */
/* ----- Cuerpo del programa ----- */

int main()
{
    int i,j;

    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    lanzarPresentacion();
    buclePrincipal();

    hard.pausa(1000);
    return 0;
}

/* Termina con la "macro" que me pide Allegro */
END_OF_MAIN();

```

31. La aproximación orientada a objetos (4). MiniMiner 4: Una pantalla de juego real

Ahora, en vez de dibujar desde el cuerpo del programa un fondo sencillo formado por varias casillas repetitivas de suelo, tendremos una pantalla de fondo más compleja, formada por varios elementos de distinto tipo. En esta versión será un fondo estático, pero dentro de poco será capaz de indicar al personaje y los enemigos si se pueden mover a una cierta posición, y también será capaz de mostrar objetos en movimiento.

Nuestro diagrama de clases incluirá una nueva clase, que llamaremos "Nivel":

Así, esta clase nivel podría tener apenas dos métodos: un "leerDeFichero" y un "dibujarOculto". El fichero de cabecera podría ser así:

```

/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  Nivel.h:                     */
/*    La pantalla del nivel 1   */
/*    Miniminer (version 0.04)  */
/*    Fichero de cabecera       */
/*                               */
/*  Comprobado con:             */
/*  - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*    y Allegro 4.03 - WinXP    */
/*  - gcc 4.4.3 y Allegro 4.2   */
/*    en Ubuntu 10.04          */
/*-----*/

#ifndef Nivel_h
#define Nivel_h

#include <vector>
#include <string>
using namespace std;

#include "ElementoGraf.h"

class Nivel {

public:

    Nivel();
    void dibujarOculto(Hardware h);
    void leerDeFichero();

private:
    int nivelActual;

    /* El mapa que representa a la pantalla */
    /* 16 filas y 32 columnas (de 16x16 cada una) */
    #define MAXFILAS 16
    #define MAXCOLS 32

    char mapa[MAXFILAS][MAXCOLS];
    ElementoGraf *fragmentoNivel[MAXFILAS][MAXCOLS];

};
#endif

```

Y su desarrollo:

```

/*-----*/
/*  Intro a la programac de      */
/*  juegos, por Nacho Cabanes    */
/*                               */
/*  Nivel.cpp:                   */
/*    La pantalla del nivel 1   */
/*    Miniminer (version 0.04)  */
/*    Fichero de desarrollo     */
/*                               */
/*  Comprobado con:             */
/*  - DevC++ 4.9.9.2 (gcc 3.4.2) */
/*    y Allegro 4.03 - WinXP    */
/*  - gcc 4.4.3 y Allegro 4.2   */
/*    en Ubuntu 10.04          */
/*-----*/

#include "Hardware.h"
#include "ElementoGraf.h"
#include "Nivel.h"

#ifdef __cplusplus
#include <cstdlib>

```

```

#include <cstdio>
#else
#include <stdlib.h>
#include <stdio.h>
#endif

#include <math.h>
#include <string>

#include <allegro.h>

Nivel::Nivel()
{
    nivelActual = 1;

    int i, j;
    int anchoImagen = 16, altoImagen = 16;
    int margenIzq = 64, margenSup = 48;

    for(i=0; i<MAXCOLS; i++)
        for (j=0; j<MAXFILAS; j++)
        {
            fragmentoNivel[j][i] = new ElementoGraf();
            fragmentoNivel[j][i]->moverA(
                margenIzq + i*anchoImagen, margenSup + j*altoImagen);
        }

    leerDeFichero();
}

/** dibujarOculto: muestra el nivel sobre el fondo
 * de pantalla, con su imagen correspondiente
 * segun el tipo de ladrillo del que se trate
 */

void Nivel::dibujarOculto(Hardware h)
{
    int i, j;

    h.borrarOculto();

    for(i=0; i<MAXCOLS; i++)
        for (j=0; j<MAXFILAS; j++)
            h.dibujarOculto( *fragmentoNivel[j][i] );
}

/** leerNivel: lee de fichero los datos del nivel actual
 */
void Nivel::leerDeFichero()
{
    // Si hay fichero de datos, lo leo
    FILE *fichDatos;
    char linea[MAXCOLS+2];
    char nombreFich[50];
    sprintf(nombreFich, "nivel%03d.dat", nivelActual);
    fichDatos = fopen(nombreFich, "rt"); //### Mas adelante: creciente
    if (fichDatos != NULL)
        for (int j=0; j<MAXFILAS;j++) {
            fgets(linea,MAXCOLS+1,fichDatos);
            if (strlen(linea) < 5) // Salto los avances de linea de Windows
                fgets(linea,MAXCOLS+1,fichDatos);
            for (int i=0; i<MAXCOLS;i++) {
                switch(linea[i]) {
                    case 'S':
                        fragmentoNivel[j][i]->crearDesdeFichero("suelo.bmp");
                        break;
                    case 'F':
                        fragmentoNivel[j][i]->crearDesdeFichero("sueloFragil.bmp");
                        break;
                    case 'L':
                        fragmentoNivel[j][i]->crearDesdeFichero("ladrillo.bmp");
                        break;
                    case 'V':
                        fragmentoNivel[j][i]->crearDesdeFichero("llave.bmp");
                        break;
                    case 'P':
                        fragmentoNivel[j][i]->crearDesdeFichero("puerta.bmp");

```



```

// --- Bucle principal del juego -----
void buclePrincipal() {
    partidaTerminada = false;
    do {
        comprobarTeclas();
        moverElementos();
        comprobarColisiones();
        dibujarElementos();
        pausaFotograma();
    } while (partidaTerminada != true);
}

// -- Comprobac de teclas para mover personaje o salir
void comprobarTeclas() {

    if (hard.comprobarTecla(TECLA_ESC))
        partidaTerminada = true;

    if (hard.comprobarTecla(TECLA_DCHA))
        personaje->moverDerecha();
    else if (hard.comprobarTecla(TECLA_IZQD))
        personaje->moverIzquierda();
}

// -- Mover otros elementos del juego
void moverElementos() {
    enemigo->mover();
}

// -- Comprobar colisiones de nuestro elemento con otros, o disparos con enemigos, etc
void comprobarColisiones() {
    // Por ahora, no hay colisiones que comprobar
}

// -- Dibujar elementos en pantalla
void dibujarElementos() {

    hard.borrarOculto();
    primerNivel->dibujarOculto(hard);
    hard.dibujarOculto( *enemigo );
    hard.dibujarOculto( *personaje );
    hard.visualizarOculto();
}

// -- Pausa hasta el siguiente fotograma
void pausaFotograma() {
    // Para 25 fps: 1000/25 = 40 milisegundos de pausa
    hard.pausa(40);
}

// -- Funciones que no son de la logica de juego, sino de
// funcionamiento interno de otros componentes

// -- Pantalla de presentacion
void lanzarPresentacion() {
    presentacion->mostrar(hard);
}

/* ----- Rutina de inicialización ----- */
int inicializa()
{
    hard.inicializar(640,480);
    personaje = new Personaje();
    enemigo = new Enemigo();
    presentacion = new Presentacion();
    primerNivel = new Nivel();

    // Y termino indicando que no ha habido errores
    return 0;
}

/* ----- */

```

```

/* ----- Cuerpo del programa ----- */
int main()
{
    int i,j;

    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    lanzarPresentacion();
    buclePrincipal();

    hard.pausa(1000);
    return 0;
}

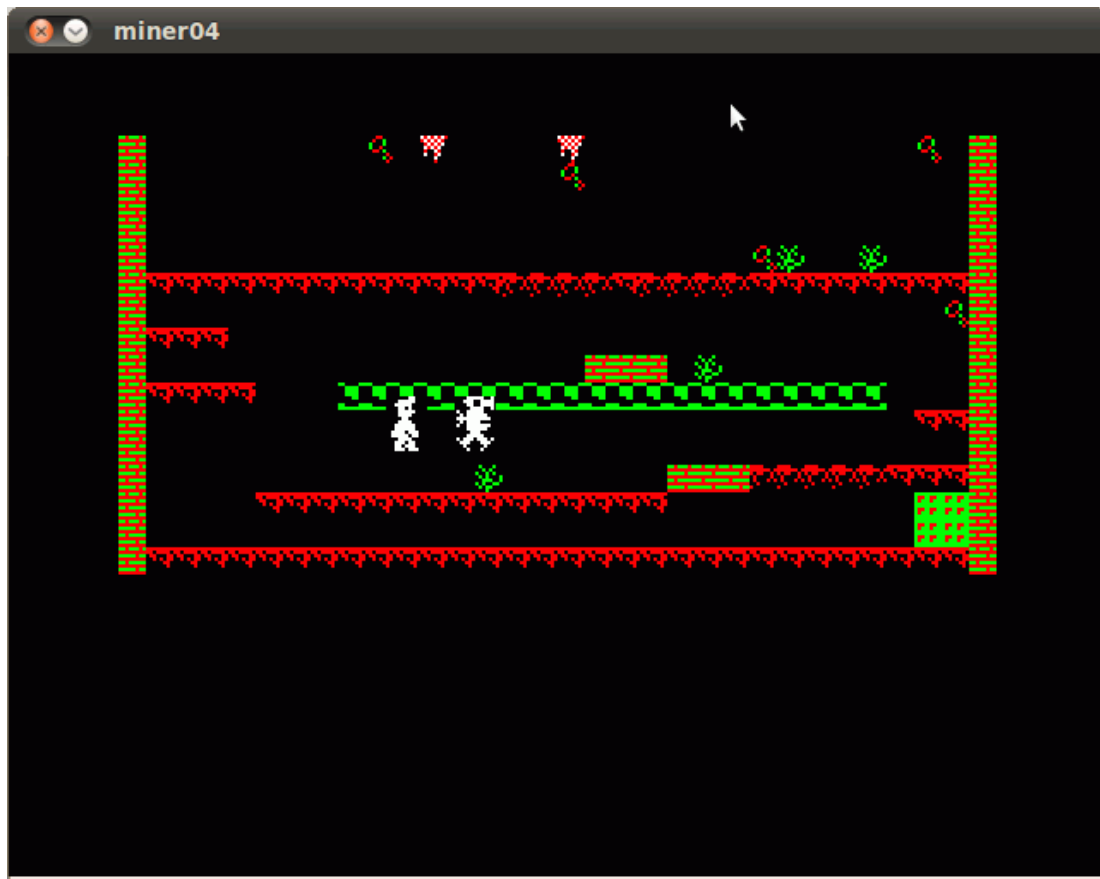
/* Termino con la "macro" que me pide Allegro */
END_OF_MAIN();

```

Y la apariencia resultante ya es mucho más cercana a la del juego original: en windows 7 se vería así:



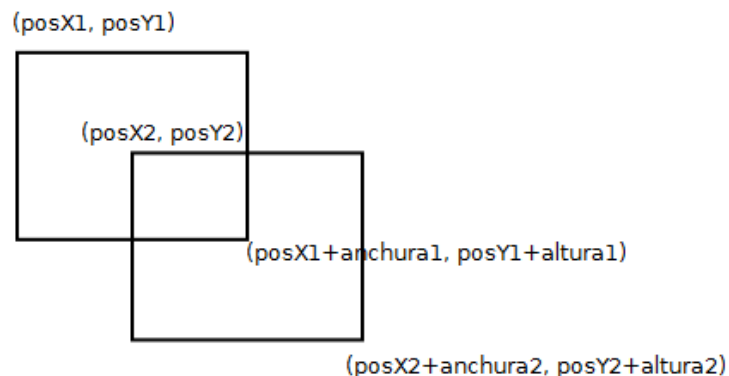
y en Ubuntu 10.04 así:



Puedes [descargar toda esta versión, en un fichero ZIP, que incluye todos los fuentes, las imágenes, el proyecto de Dev-C++ listo para compilar en Windows, y un fichero "compila.sh" para compilar en Linux.](#)

32. Colisiones con enemigos. Perder vidas. Aplicación a MiniMiner (versión 5) (*)

La comprobación de colisiones no es difícil, y además ya tenemos un esqueleto en la clase "Elemento Gráfico": la forma más sencilla es comprobar si se solapan el rectángulo que contiene un elemento gráfico y el rectángulo que contiene a otro:



El fragmento de fuente que se encarga de esto es simplemente así:

```
/** colisionCon: devuelve si hay colision del ElementoGraf con
 * un rectangulo dado por sus coordenadas
 */
bool ElementoGraf::colisionCon(int x, int y, int ancho, int alto)
{
    if ((this->posX+this->anchura > x)
        && (this->posY+this->altura > y)
        && (x+ancho > this->posX)
        && (y+alto > this->posY))
        return true;
    else
        return false;
}
```

```

/** colisionCon: devuelve si hay colision del ElementoGraf con otro
 */
bool ElementoGraf::colisionCon(ElementoGraf e2)
{
    return colisionCon(e2.posX, e2.posY, e2.anchura, e2.altura);
}

```

Cuando veamos que colisionan el personaje y el enemigo (lo único que vamos a comprobar por ahora), deberemos hacer varias cosas:

- Restar una vida al personaje. Como hasta ahora no estábamos comprobando vidas, esto supone que añadamos esta posibilidad en la clase "Personaje", con todo lo que implica: un atributo privado "numVidas" y varios métodos para leer su valor ("getVidas"), para restar una vida cuando nos toque el enemigo ("disminuirVidas"), y para volver a las 3 vidas iniciales cuando termine una partida, algo que todavía no permitirá esta versión pero haremos muy pronto ("reiniciarVidas").
- Actualizar el "marcador", para que muestre el número de vidas actual. Eso supone que nos interesará crear una nueva clase "Marcador", que represente la parte de la pantalla que indica las vidas restantes, la puntuación actual y la mejor puntuación.
- Recolocar el personaje y el enemigo en su posición inicial, o de lo contrario volverían a chocar en el siguiente "fotograma" del juego, y perderíamos más de una vida. Para eso, crearemos un método "reiniciarPosicion" en ambas clases, que además en el Enemigo cambiará su "incrX" (incremento de X) para que vuelva a su posición inicial, moviéndose también en su misma dirección inicial.
- Comprobar si el personaje se ha quedado sin vidas, y en ese caso marcaremos la partida como terminada (y reiniciaremos su cantidad de vidas, para que vuelva a tener 3, para cuando podamos comenzar una nueva partida... algo que esta versión del juego todavía no va a permitir).

Eso se convertiría en una función como esta:

```

void comprobarColisiones() {
    // Colisiones de personaje con enemigo: recolocar y perder vida
    if (personaje->colisionCon( *enemigo ))
    {
        personaje->disminuirVidas();
        marcador->indicarVidas(personaje->getVidas());
        personaje->reiniciarPosicion();
        enemigo->reiniciarPosicion();
        hard.pausa(100);
        if (personaje->getVidas() == 0)
        {
            partidaTerminada = true;
            personaje->reiniciarVidas(); // Para la siguiente partida
        }
    }
}

```

La apariencia ahora sería así:



Y aquí puedes [descargar toda esta versión, en un fichero ZIP, que incluye todos los fuentes, las imágenes, el proyecto de Dev-C++ listo para compilar en Windows, y un fichero "compila.sh" para compilar en Linux.](#)

33. Volver a comenzar una partida. Una pantalla de presentación animada. Aplicación a MiniMiner (versión 6)

En este momento, nuestro juego permite agotar tres vidas, pero cuando acaban esas tres vidas, termina la partida y salimos del juego automáticamente.

Eso no es lo que suele ocurrir en un juego real: cuando agotamos nuestras vidas, deberíamos volver a la pantalla de presentación, y poder comenzar una nueva partida (con puntuación 0, con el personaje y los enemigos en su posición inicial, etc). No es difícil: basta con que la parte del juego que lanza la pantalla de presentación y luego el bucle principal se convierta en un bloque repetitivo: lo que antes era

```
int main()
{
    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    lanzarPresentacion();
    buclePrincipal();

    hard.pausa(1000);
    return 0;
}
```

ahora se repetirá hasta que en la presentación se escoja "Terminar":

```
int main()
{
    // Intento inicializar
    if (inicializa() != 0)
        exit(1);

    sesionTerminada = false;
    do
    {
        lanzarPresentacion();
```

```

    sesionTerminada = ( presentacion->escogidoFin() );
    if (! sesionTerminada)
    {
        buclePrincipal();
        reiniciarPartida();
    }
}
while (! sesionTerminada);

return 0;
}

```

Ese nuevo método "escogidoFin" de la clase Presentación es el que indicaría si se ha pulsado la tecla de Terminar la partida (por ejemplo, la tecla T).

Hacer que la **pantalla de presentación sea animada** también es sencillo: basta con que en vez de terminar con un "esperarTecla", se repita con un "mientras no se pulse tecla", y que esa parte repetitiva mueva una imagen en pantalla. Esta imagen podría ser la "figura imposible" que aparecía junto al cartel de presentación del juego, y la forma de moverla podría ser aumentando su X y su Y hasta que llegue a los bordes de la pantalla, y haciendo que "rebote" en ese caso:

```

int Presentacion::mostrar(Hardware h) {

    int x=100, y=100; // Coordenadas del cartel movil
    int incrX = 2, incrY = 2; // Velocidad del cartel movil

    // Absorbo pulsaciones de teclas que lleguen desde el juego
    while ( h.algunaTeclaPulsada() )
        h.esperarTecla();

    do
    {
        // Primero muestro el cartel antiguo
        h.borrarOculto();
        cartelFondo.moverA(0,0);
        h.dibujarOculto(cartelFondo);

        cartelMovil.moverA(x,y);
        h.dibujarOculto(cartelMovil);

        h.visualizarOculto();
        h.pausa(40);

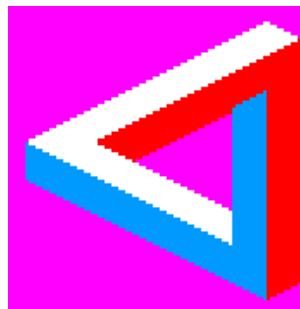
        x += incrX;
        y += incrY;

        // Invierto velocidad si llega al borde
        if ((x < 10) || (x > 640-10-174))
            incrX = - incrX;
        if ((y < 10) || (y > 480-10-179))
            incrY = - incrY;
    }
    while ( ! h.algunaTeclaPulsada() );

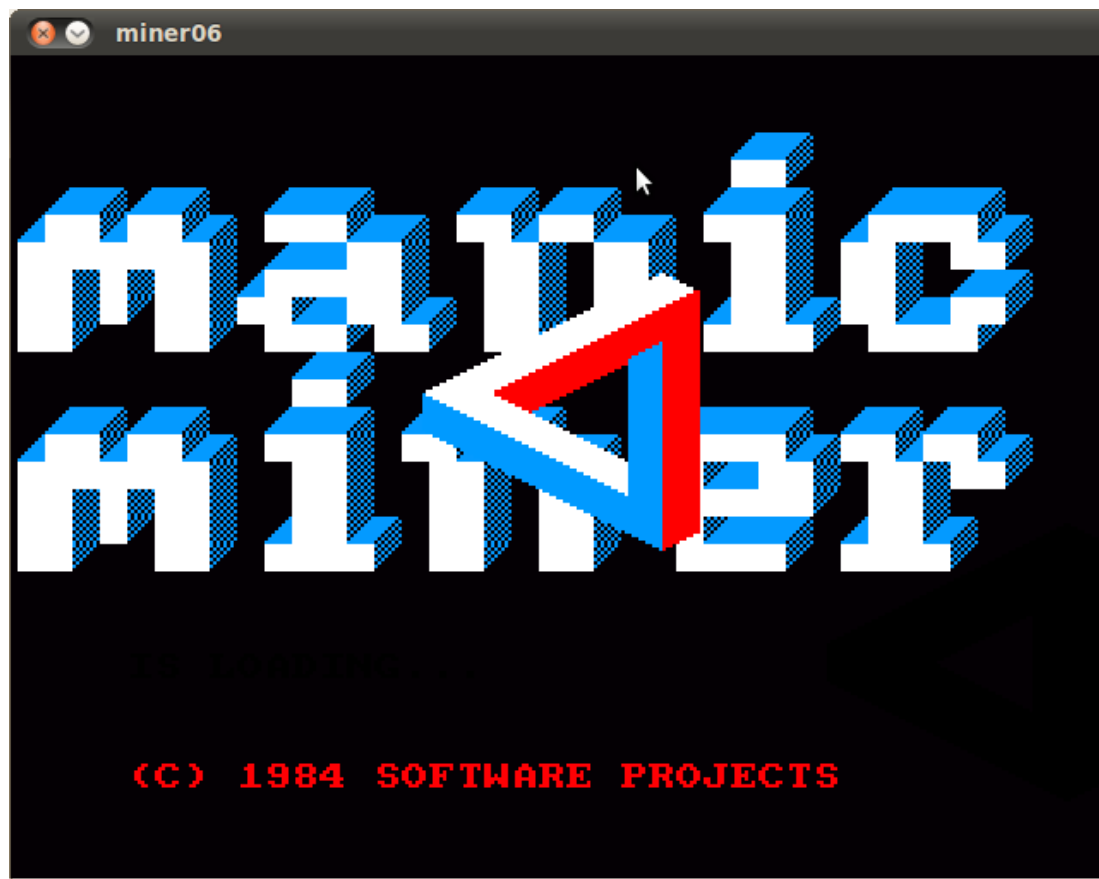
    tecla = h.esperarTecla();
}

```

Si queremos que la **imagen sea transparente** (para que no se borre el fondo al moverse sobre él), es algo que la mayoría de bibliotecas gráficas permiten hacer de una u otra forma. En el caso de Allegro, la función "draw_sprite" soporta transparencia de dos maneras distintas: o bien con el color 0 (si la imagen es de 256 colores) o bien el "rosa mágico" (255,0,255) si es una imagen "true color":



De modo que la presentación quedaría así:



Nuestro juego todavía no es jugable, pero no es conveniente hacer demasiados cambios en una misma versión, así que las mejoras de jugabilidad las dejamos para la siguiente entrega...

34. Añadiendo funcionalidades a "MiniMiner" (1): chocar con el fondo, saltar. (*)

En primer lugar, vamos a comprobar colisiones con el fondo, para que nuestro personaje se detenga cuando llegue a un obstáculo.

Nos puede bastar con añadir un método "esPosibleMover", que, a partir de las coordenadas de la nueva posición del personaje, nos digan si corresponden a una casilla "pisable" de nuestro mapa de juego. Basta con convertir de coordenadas de pantalla a coordenadas de mapa (restando el margen y dividiendo entre el ancho o el alto según corresponda), y luego comprobar qué símbolo aparecía en esa posición del mapa:

```
bool Nivel::esPosibleMover(int x1, int y1, int x2, int y2)
{
    int xCasilla1 = (x1 - margenIzq) / anchoImagen;
    int yCasilla1 = (y1 - margenSup) / altoImagen;
    int xCasilla2 = (x2 - margenIzq) / anchoImagen;
    int yCasilla2 = (y2 - margenSup) / altoImagen;

    if (mapa[yCasilla1][xCasilla1] != ' ')
        return false;

    if (mapa[yCasilla1][xCasilla2] != ' ')
        return false;

    if (mapa[yCasilla2][xCasilla2] != ' ')
        return false;

    if (mapa[yCasilla2][xCasilla1] != ' ')
        return false;

    return true;
}
```

Y entonces el personaje, antes de moverse a derecha o izquierda, deberá comprobar si es posible moverse a la nueva posición en la que debería quedar:

```
void Personaje::moverDerecha(Nivel n) {
    if (n.esPosibleMover(posX+4, posY,
        posX+anchura+4, posY+altura))
        posX += 4;
}
```

(ese planteamiento no es de todo fiable: comprobamos si colisiona alguna de las esquinas del personaje; en un personaje "alargado" como el nuestro, puede chocar también la parte central... cuando salte... y eso va a ocurrir pronto).

Hacer que el personaje salte es un poco más complicado, porque supone varios cambios:

- Por una parte, cuando haya comenzado a saltar deberá moverse solo, lo que implica que habrá un método "saltar", que comience la secuencia del salto, pero también un "mover", al igual que en el enemigo, que se encargará de continuar esa secuencia (cuando corresponda).
- Por tanto, el cuerpo del juego deberá llamar a "saltar" cuando se pulse la tecla correspondiente, pero también a "mover" en cada pasada por la función "moverElementos" del bucle de juego.
- Además, la secuencia de salto será más complicada que (por ejemplo) el rebote del cartel de presentación: en este juego (y en otros muchos), el personaje hace un movimiento parabólico. Vimos en el apartado 13 la forma básica de programar este tipo de movimientos, pero habrá que aplicarlo a nuestro personaje. Además, en el juego original que estamos imitando, el personaje saltaba en vertical si sólo se pulsaba la tecla de saltar, y lo hacía de forma parabólica a derecha o izquierda si se pulsa la tecla de salto junto con una de las teclas de dirección.

Una primera forma de conseguirlo sería usar realmente la ecuación de la parábola, con algo como:

```
void Personaje::mover(Nivel n) {
    if (! saltando)
        return;
    int xProxMov = posX + incrXSalto;
    int yProxMov = a*xProxMov*xProxMov + b*xProxMov + c;

    if (n.esPosibleMover(xProxMov, yProxMov,
        xProxMov+anchura, yProxMov+altura))
    {
        posX += incrXSalto;
        posY = a*posX*posX + b*posX + c;
    }
    else
        saltando = false;
}
```

Esa es la secuencia de salto casi completa: falta tener en cuenta que cuando el personaje deja de avanzar en forma parabólica bien porque realmente llegue a su destino o bien porque choque un obstáculo, deberíamos comprobar si tiene suelo por debajo, para que caiga en caso contrario.

Los valores de a, b y c se calcularían en el momento en que se da la orden de saltar:

```
void Personaje::saltar() {
    if (saltando)
        return;
    saltando = true;
    xInicialSalto = posX;
    yInicialSalto = posY;
    incrXSalto = 0;

    // Calculo a, b y c de la parábola
    float xVerticeParabola = posX+32;
    float yVerticeParabola = posY-40;
    float pParabola = 12;
    a = 1 / (2*pParabola);
    b = -xVerticeParabola / pParabola;
    c = ((xVerticeParabola*xVerticeParabola) / (2*pParabola) )
        + yVerticeParabola;
}
```

Pero esta aproximación tiene tres problemas:

- Supone hacer muchos cálculos cada vez que se pide que el personaje salte... a pesar de que todos los saltos son iguales.
- No es aplicable directamente a otros movimientos más complejos.
- Al ser una función matemática, una misma coordenada X no puede corresponder a varias Y distintas. Eso supone que este sistema no sirva para movimientos circulares (por ejemplo), pero tampoco (y eso es más grave en nuestro caso) a

movimientos verticales.

Por eso, podemos hacerlo de otra forma que, a pesar de que pueda parecer "menos natural", evita todos esos problemas: precalcular el movimiento, y guardar las coordenadas de cada nuevo paso en un array:

```
int pasosSaltoArriba[] = { -6, -6, -6, -6, -4, -4, -2, -2, 0,
                          0, 2, 2, 4, 4, 6, 6, 6, 6 };

void Personaje::mover(Nivel n) {
    if (saltando)
    {
        int xProxMov = posX + incrXSalto;
        int yProxMov = posY + pasosSaltoArriba[ fotogramaMvto ];

        // Si todavia se puede mover, avanzo
        if (n.esPosibleMover(xProxMov, yProxMov,
            xProxMov+anchura, yProxMov+altura))
        {
            posX = xProxMov;
            posY = yProxMov;
        }
        // Y si no, quizá esté cayendo
        else
        {
            saltando = false;
            cayendo = true;
        }

        fotogramaMvto ++;
        if (fotogramaMvto >= MVTOS_SALTO)
        {
            saltando = false;
            cayendo = true;
        }
    }
    else if (cayendo)
    {
        if (n.esPosibleMover(posX, posY+desplazVertical,
            posX+anchura, posY+desplazVertical+altura))
        {
            posY += desplazVertical;
        }
        else
            cayendo = false;
    }
}
```

En ese ejemplo, las coordenadas verticales cambian, para dar la impresión de un movimiento que va disminuyendo su velocidad al separarse del suelo, y vuelve a aumentar a medida que vuelve a acercarse al suelo; por el contrario, las coordenadas horizontales cambian siempre en un mismo valor, que es positivo si salta hacia la derecha, negativo si salta hacia la izquierda o 0 si salta en vertical.

(Nota: esa forma de declarar el array puede dar problemas en muchos compiladores, diciendo que ya se ha definido con anterioridad el valor de "pasosSaltoArriba". Para evitarlo, podríamos definir el array (sin detallar su contenido) en el fichero de cabecera de la clase:

```
class Personaje: public ElementoGraf {
public:
    ...

private:
    ...
    int pasosSaltoArriba[MVTOS_SALTO];
};
```

y rellenar el contenido del array en el constructor):

```
Personaje::Personaje() {
    ...
    pasosSaltoArriba[0] = -6; pasosSaltoArriba[1] = -6;
    pasosSaltoArriba[2] = -6; pasosSaltoArriba[3] = -6;
    pasosSaltoArriba[4] = -6; pasosSaltoArriba[5] = -4;
```

```

pasosSaltoArriba[6] = -4; pasosSaltoArriba[7] = -2;
pasosSaltoArriba[8] = -2; pasosSaltoArriba[9] = 0;
pasosSaltoArriba[10] = 0; pasosSaltoArriba[11] = 2;
pasosSaltoArriba[12] = 2; pasosSaltoArriba[13] = 4;
pasosSaltoArriba[14] = 4; pasosSaltoArriba[15] = 4;
pasosSaltoArriba[16] = 6; pasosSaltoArriba[17] = 6;
pasosSaltoArriba[18] = 6; pasosSaltoArriba[19] = 6;
}

```

La forma de hacer que caiga cuando termina el salto y no está sobre una plataforma (porque la plataforma acabe o porque choque con un obstáculo) sería:

```

void Personaje::mover(Nivel n) {
    [...]
    else if (cayendo)
    {
        if (n.esPosibleMover(posX, posY+desplazVertical,
            posX+anchura, posY+desplazVertical+altura))
        {
            posY += desplazVertical;
        }
        else
            cayendo = false;
    }
}

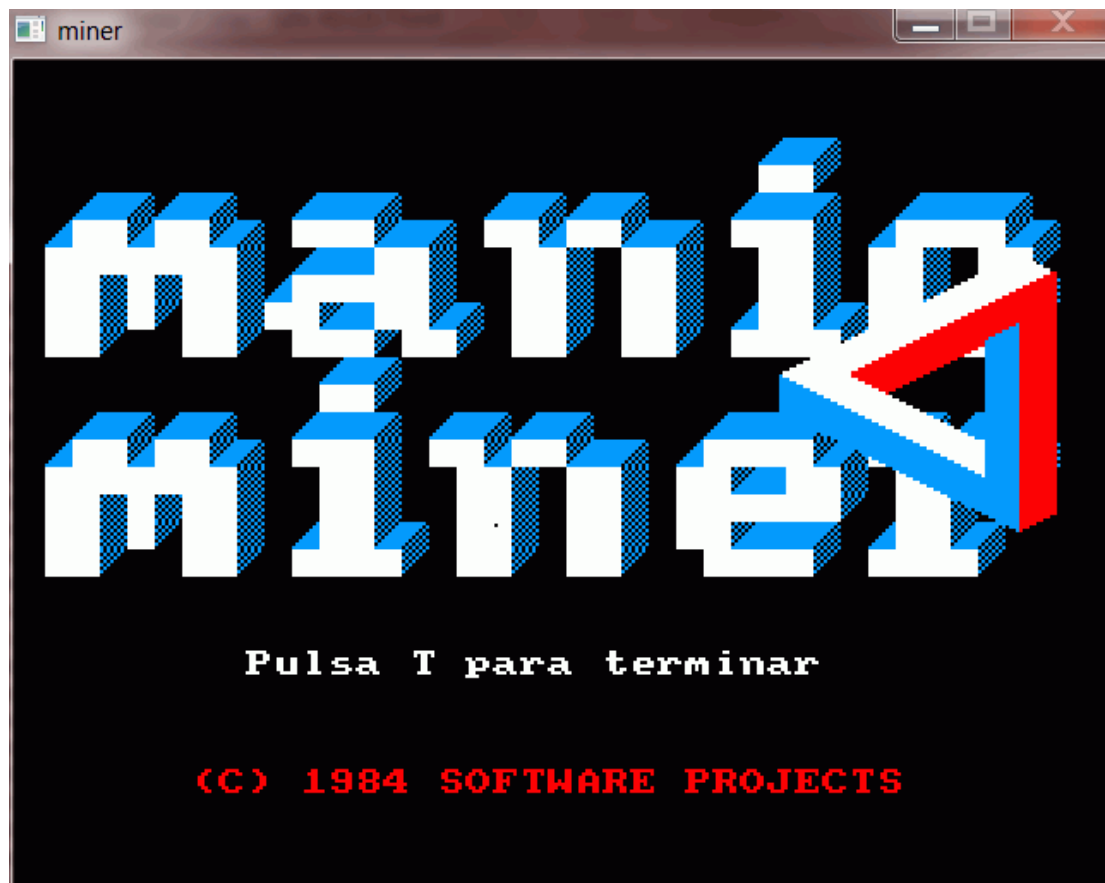
```

También podemos hacer que caiga después de andar normalmente si se termina la plataforma, simplemente añadiendo "cayendo = true;" al final de las funciones de movimiento a derecha o izquierda, de modo que tras cada paso comprobará si debe caer.

En la apariencia no hay cambios en esta versión, salvo por el hecho de que nuestro personaje se puede separar del suelo:



De paso, en la pantalla de bienvenida podemos añadir el texto "Pulse T para terminar", para que sea más amigable.



Y como es habitual, aquí puedes [descargar toda esta versión, en un fichero ZIP, que incluye todos los fuentes, las imágenes, el proyecto de Dev-C++ listo para compilar en Windows, y un fichero "compila.sh" para compilar en Linux.](#)

35. Una consola para depuración. (*)

En ocasiones nos encontraremos con errores difíciles de detectar. Cuando se trata de un programa en modo texto, podemos intentar detectarlos con la ayuda de depuradores como los que incluyen mucho entornos integrados de desarrollo (IDE) o con depuradores independientes, como "gdb". Éstos nos permiten avanzar paso a paso y ver los valores de las variables que nos interesen. Pero en un programa gráfico, como nuestros juegos, es mucho más difícil interrumpir el programa para ver el valor de una variable: algunos depuradores no permiten interrumpir un programa de este tipo, y otros resultan incómodos de manejar.

Si no tenemos necesidad de interrumpir el programa por completo, tenemos otra alternativa para ver el valor de una variable: mostrarlo en pantalla. Nuevamente, esto es fácil en programas en modo texto o incluso en programas con ventanitas, pero no tanto en los programas en modo gráfico. Siempre podemos reservar una zona de la pantalla, y mostrar el valor de una variable igual que mostraríamos las vidas o la puntuación.

Pero existe otra alternativa sencilla, y que permite analizar una mayor cantidad de información: ir volcando a fichero los mensajes que nos interese, de modo que podamos observar después el resultado con detenimiento.

Para eso, nos vamos a crear una clase "ConsolaDepuracion", que mantendrá un fichero de texto al que se irá añadiendo la información que digamos:

En esa clase podríamos crear un método "estático" (para que no sea necesario crear "objetos" de la clase "ConsolaDepuracion"), llamado "escribir". Así, en cualquier parte de nuestro fuente podríamos hacer algo como:

```
ConsolaDepurac::escribir("Ha chocado con el enemigo\n");
```

Y todos los mensajes como ése irían a parar a un fichero de texto, de modo que pudiéramos analizarlo después, para ayudarnos a comprobar que el programa está funcionando correctamente.

La clase "ConsolaDepuracion" puede ser muy simple (al menos por ahora, si sólo queremos escribir textos, no números que no hayamos convertido previamente a texto, ni ningún otro tipo de información). El fichero de cabecera podría ser así:

```
/*-----*/
/* Intro a la programac de */
```

```

/* juegos, por Nacho Cabanes */
/* */
/* ConsolaDepurac.h: */
/* Clase "ConsolaDepurac" */
/* Fichero de cabecera */
/* */
/* Comprobado con: */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/* y Allegro 4.03 - WinXP */
/* - gcc 4.4.3 y Allegro 4.2 */
/* en Ubuntu Linux 10.04 */
/*-----*/

#ifndef ConsolaDepurac_h
#define ConsolaDepurac_h

class ConsolaDepurac {

public:

    static void escribir(char* texto);

};
#endif

```

Y el desarrollo podría ser:

```

/*-----*/
/* Intro a la programac de */
/* juegos, por Nacho Cabanes */
/* */
/* ConsolaDepurac.cpp: */
/* Clase "ConsolaDepurac" */
/* Fichero de desarrollo */
/* */
/* Comprobado con: */
/* - DevC++ 4.9.9.2 (gcc 3.4.2) */
/* y Allegro 4.03 - WinXP */
/* - gcc 4.4.3 y Allegro 4.2 */
/* en Ubuntu Linux 10.04 */
/*-----*/

#include "ConsolaDepurac.h"
#include <fstream>
using namespace std;

void ConsolaDepurac::escribir(char* texto)
{
    fstream ficheroDepuracion;
    ficheroDepuracion.open ( "depuracion.txt" , ios::out | ios::app);
    ficheroDepuracion << texto;
    ficheroDepuracion.close();
}

```

Esta vez no incluyo fichero ZIP: para crear la octava entrega del Miner, basta con copiar estos dos ficheros en la carpeta del proyecto, y añadirlos al proyecto de DevC++ si usamos Windows, o añadir el fichero .cpp al "compila.sh" si usamos Linux. Como debería ser fácil, lo dejo propuesto como ejercicio.

En la siguiente entrega, continuaremos ampliando las funcionalidades del juego...

36. Añadiendo funcionalidades a "MiniMiner" (2): mejora del movimiento, recoger objetos (*)

En la séptima entrega de nuestro Miner vimos cómo hacer un movimiento de salto, tanto vertical como parabólico. Es un tipo de movimiento aplicable a muchos juegos de plataformas, pero que no funciona bien en nuestro caso, por un par de detalles:

- El personaje es más alto que las casillas del fondo, de modo que si comprobamos sólo las 4 esquinas del personaje, se nos pueden escapar choques con la parte central del cuerpo, y eso da lugar a efectos no deseados cuando saltamos

cerca de un plataforma.

- No hacemos realmente lo mismo que en el juego original: nuestro personaje se para en cuanto toca una casilla del fondo (exceptuando el fallo indicado en el punto anterior), pero en el juego original, podía seguir subiendo si saltaba cuando estaba delante de una plataforma roja, y se detenía si chocaba con ella al caer (pero no al subir). En cambio, los ladrillos sí detienen su movimiento por completo.

Vamos a intentar mejorar ese comportamiento, y, de paso, recoger objetos en la pantalla...

(Pronto disponible)

37. Añadiendo funcionalidades a "MiniMiner" (3): avanzar a otra pantalla. (*)

Normalmente, en un juego de plataformas como este, podremos avanzar por diferentes pantallas. Hay jugos en los que podemos avanzar de una pantalla a otra en cuanto lleguemos a un extremo, o cuando pasemos por una puerta, y otros juegos (como éste) en los que antes debemos recoger una serie de objetos para que así se abra la puerta.

Por otra parte, hay juegos en los que se nos felicita (y termina la partida) cuando recorremos todos los niveles, y otros juegos en los que tras recorrer el último nivel volvemos al primero y seguimos acumulando puntos.

En el caso concreto de nuestro Miner, la lógica del sistema de niveles es la siguiente

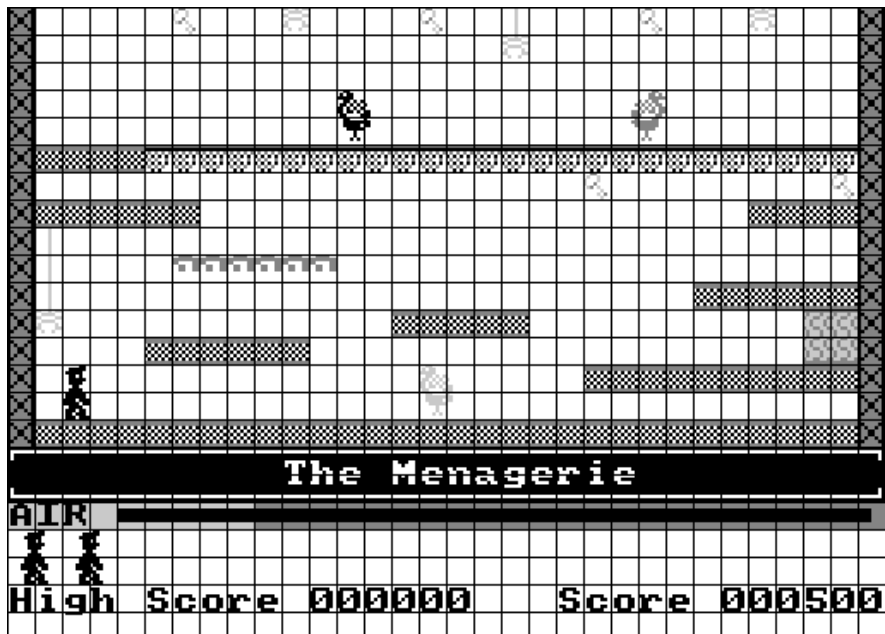
- Hay que superar 20 niveles, y tras completar el último se vuelve a comenzar por el primero (conservando al puntuación conseguida, claro).
- En cada nivel hay que recoger varias "llaves". Cuando recogemos todas, se abre la puerta que nos permite pasar al siguiente nivel.
- Si perdemos una vida (porque nos toque un enemigo, o choquemos con alguno de los obstáculos que son capaces de matarnos, o caigamos desde mucha altura), volvemos a comenzar el nivel, y todas las llaves aparecen en su posición inicial y la puerta aparece cerrada.

Esa serie de niveles se representarán típicamente como un "array" de niveles, y llevaremos cuenta del número de nivel en el que nos encontramos, para poder volver al nivel 1 cuando completemos el último.

Para cada uno de los niveles que componen el juego, deberemos diseñarlo (si estamos haciendo un juego desde cero) o crearlo a partir del original (si estamos "imitando un clásico"). Por ejemplo, en el caso de nuestro Miner, el tercer nivel es así:



Y deberíamos descomponerlo en componentes, como ya hicimos en la entrega 4 para el primer nivel:

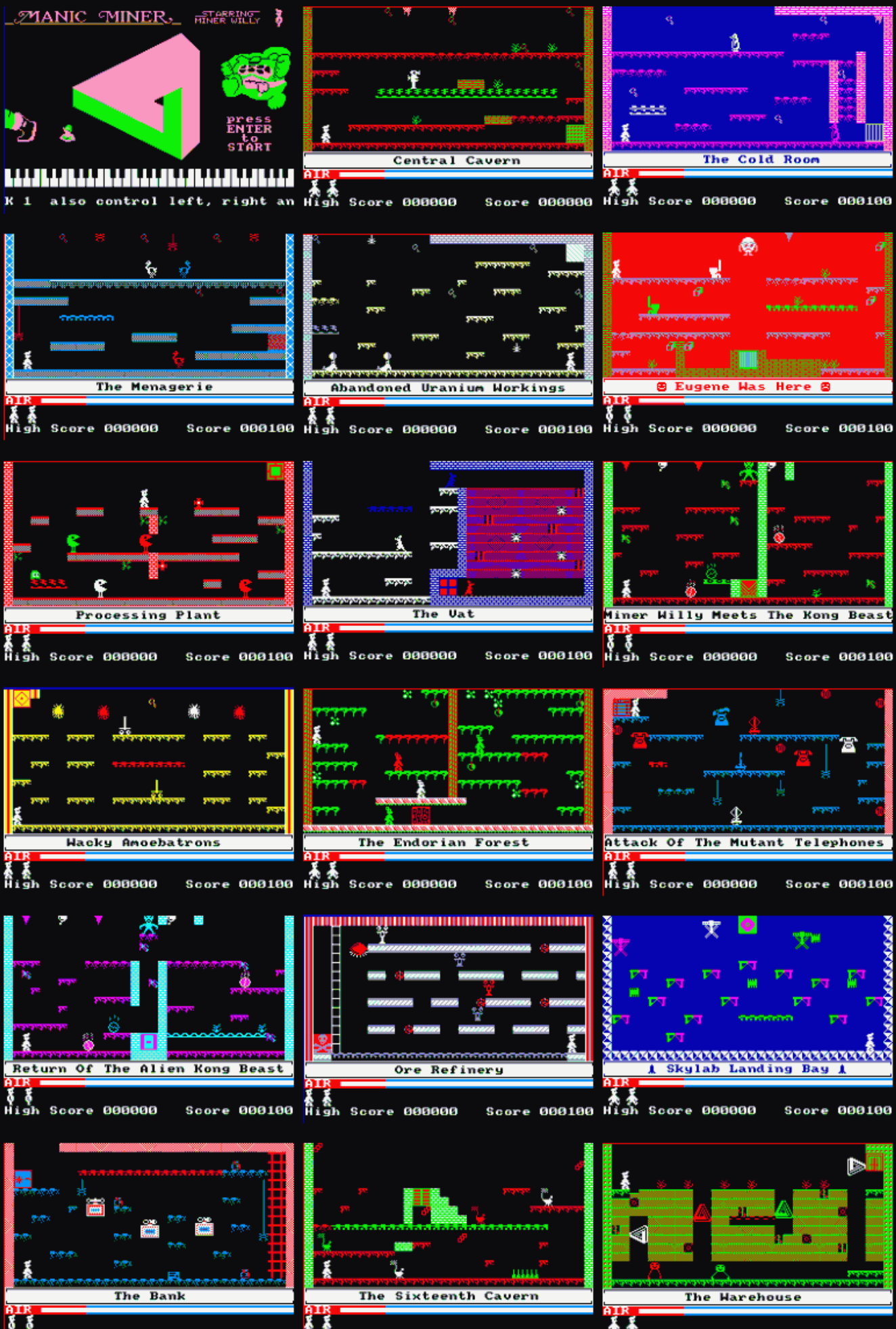


Como se ve en este ejemplo, es habitual que aparezcan nuevos elementos (distintos tipos de suelo, obstáculos, enemigos) a medida que avanzamos por los niveles.

Un consejo para la hora de implementar todo esto: normalmente será conveniente no modificar directamente el array cada vez que recojamos un objeto, sino trabajar sobre una copia. Así, cuando perdamos una vida (en este juego) o si volvamos a comenzar una partida (en cualquier juego), restauraremos el nivel (o todos ellos) a su estado inicial, de modo que los objetos vuelvan a aparecer en sus posiciones iniciales.

Nosotros crearemos sólo dos niveles (y tras el segundo se volverá al primero), porque añadir más niveles supone emplear más tiempo, pero no debería suponer dificultad extra. Por si alguien quiere crear más niveles, éstos eran los que componían el juego original:

Niveles Manic Miner (Amstrad CPC)



(Pronto disponible)

38. Una lista con las mejores puntuaciones. Cómo guardarla y recuperarla. (*)

En casi cualquier juego existe una "tabla de records", que guarda las mejores puntuaciones. Las características habituales son:

- Se guarda una cantidad prefijada de puntuaciones (típicamente 10).
- Para cada puntuación, se guarda también el nombre (o las iniciales) del jugador que la consiguió.
- Las puntuaciones (y sus nombres asociados) deben estar ordenadas, desde la más alta a la más baja.
- Las puntuaciones deben guardarse en fichero automáticamente cuando termina la partida (o incluso antes, cuando se introduzca un nuevo record).
- Las puntuaciones se deben leer de fichero cada vez que el juego se pone en marcha.
- Cada puntuación nueva que se intenta introducir deberá quedar en su posición correcta, desplazando "hacia abajo" a las que sean inferiores, o incluso no entrando en la tabla de records si se trata de una puntuación más baja que todas las que ya existen en ella.

Para conseguirlo, podemos crear una clase auxiliar, como hicimos con la consola de depuración, que también se podrá aplicar a cualquier otro juego. Esa clase debería tener un método para "Cargar" desde fichero (o el constructor se podría encargar), otro para "Grabar" en fichero, otro para "Introducir" un nuevo record (su puntuación y su nombre asociado) y otro para "Obtener" un record (tanto la puntuación como el nombre, de modo que las podamos mostrr en pantalla).

(Pronto disponible)

39. Manejo del joystick. (*)

La mayoría de bibliotecas para juegos nos permiten acceder al estado del joystick. Típicamente podremos saber si hay algún joystick conectado, si se ha pulsado algún botón de disparo y cómo de inclinado está un "eje" (por ejemplo, si está centrado, un poco inclinado hacia la derecha o muy inclinado hacia la derecha). Habitualmente, para los ejes que tienen dos sentidos, como el derecha-izquierda y el arriba-abajo, obtendremos un valor entre -128 y +127 (0 sería el centro, los valores negativos representarían un sentido y los valores positivos indicarían el sentido contrario); en los ejes que sólo tienen un sentido, como el acelerador de un volante, obtendríamos un valor entre 0 y 255.

Esto es así para los joystick (o gamepad) analógicos, en los que la inclinación puede tener valores intermedios. En un joystick (o gamepad) digital, en los que en cada dirección sólo existe dos posibilidades (inclinado o no inclinado), puede que obtengamos directamente el valor máximo (-128, por ejemplo) o puede que tengamos la posibilidad de acceder al estado (inclinado o no) igual que si se tratara de botones de disparo.

Como nuestros juegos serán sencillos, y no necesitaremos medir la inclinación del joystick para que el personaje se mueva más deprisa, sino que nos bastará con saber en qué dirección se ha inclinado, para desplazarlo en esa dirección, ampliaremos la clase "Hardware" con funciones que nos digan si el joystick se ha inclinado a la derecha o a la izquierda, si se ha pulsado un botón, etc.

(Pronto disponible)

Próximas mejoras previstas:

La mayoría estarán detalladas como próximos apartados en la [página de contenido](#), aunque también es posible que haya otras mejoras "de otros tipos", como la ampliación del índice alfabético, la creación de una versión en formato PDF, la conversión de los fuentes a Pascal y Java, etc.

Mi intención, si mi poco tiempo libre lo permite, es lanzar una nueva versión cada 15-20 días, siempre y cuando vea que hay gente interesada en seguir el curso.

Cambios a partir de 2009

Nuevamente, he recibido algún mensaje de gente interesada en que siga el curso. Iremos retocando...

- 0.42, de 08-Ago-2010.** Incluye el planteamiento (pero todavía no la resolución) del [tema 36](#), [tema 37](#), [tema 38](#), [tema 39](#), [tema 27](#). Creada una nueva versión en formato PDF para poder consultar sin necesidad de estar conectado a Internet.
- 0.41, de 05-Ago-2010.** Creado el [tema 35](#), con la consola de depuración. Ampliado el [tema 34](#).
- 0.40, de 30-Jul-2010.** Creado el [tema 34](#), con la séptima entrega de MiniMiner.
- 0.39, de 25-Jul-2010.** Creado el [tema 33](#), con la sexta entrega de MiniMiner.
- 0.38, de 17-Jul-2010.** Creado el [tema 32](#), con la quinta entrega de MiniMiner.
- 0.37, de 13-Jul-2010.** Creado el [tema 31](#), con la cuarta entrega de MiniMiner. Creada una nueva versión en formato PDF para poder consultar sin necesidad de estar conectado a Internet.
- 0.36, de 09-Jul-2010.** Creado el [tema 30](#), con la tercera entrega de MiniMiner.
- 0.35, de 05-Jul-2010.** Creado el [tema 29](#), con la segunda entrega de MiniMiner. Revisados los temas 2 y 3.
- 0.34, de 30-Jun-2010.** Creado el [tema 28](#), con la primera entrega de MiniMiner. Revisados y reenumerados los apartados del tema 1. Creado el apartado 1.5, sobre FreePascal para Linux.
- 0.33, de 05-Oct-2009.** Completado el [tema 26](#), con el resto de la lógica de juego de Columnnas.
- 0.32, de 12-Ago-2009.** Completado el [tema 25](#), con parte de la lógica de juego de Columnnas.
- 0.31, de 04-Ago-2009.** Completado el [tema 24](#), con la apariencia típica de un "bucle de juego", aplicado a una primera aproximación a "Columnnas". Corregida alguna errata de poca importancia en varios apartados: opción, ordeadores, porograma, tendríamos, etc. Actualizada la versión PDF, que ya se puede descargar desde la web.
- 0.30, de 31-Jul-2009.** Incluido el [tema 23](#) (sin fuente, que queda propuesto como ejercicio) y la idea básica del [tema 24](#). Incluidas instrucciones de cómo [compilar desde Linux usando C y Allegro](#). El índice alfabético, que incluye 10 entradas más (19 en total). Creada una versión en formato PDF.
- 0.29, de 26-Jul-2009.** Incluido el [tema 22](#), con el fuente en C. Más detalles sobre [la instalación y uso de Dev-C++](#). Mejoras internas en la estructura del apartado 7. Mejoras internas en la estructura del índice alfabético, que incluye las primeras 9 entradas.
- 0.28, de 19-Jul-2009.** Incluido el [tema 21](#), con el fuente en C. Mejoras internas en la estructura de los apartados 5 y 6. Corregido el fuente del apartado 6, en el que faltaba la última línea.
- 0.27, de 15-Jul-2009.** Incluido el [tema 20](#), con el fuente en C. Revisada y reenumerada la lista de los siguientes temas previstos.
- 0.26.003, de 12-Jul-2009.** Ampliado el [tema 19](#), con el fuente en C.
- 0.26.002, de 09-Jul-2009.** Ampliado el tema 18, con el fuente en C. Incluido Dev-C++ como entorno de desarrollo para Windows, en vez de MigGW Developer Studio, que parece abandonado. Creada una descarga rápida, que incluye el compilador, la biblioteca Allegro y alguna librería adicional que quizá usemos en el curso, como SDL y OpenGL. Mejoras internas en la estructura de los apartados 8 al 14.
- 0.26.001, de 27-Jun-2009.** Ampliado el tema 17, con el fuente en C y la imagen en formato PCX. Los enlaces del apartado 16 eran incorrectos; corregidos. Mejoras internas en la estructura de los apartados 2, 3 y 4.

Cambios a partir de 2008

Contra todo pronóstico, y después de meses sin que nadie mostrara interés, he recibido algún mensaje de gente interesada en que retomara el curso. Vamos con ello...

0.25.013, de 23-Jun-2008. Incluidos los fuente de ejemplo en los temas 12, 13 y 14. Sintaxis en colores en el fuente del tema 11. Mejorada la apariencia de las expresiones matemáticas en el tema 13.

0.25.012, de 18-May-2008. Ligeramente ampliado el tema 15. Incluida parte de los temas 16, 17, 18 y 19.

0.25.011, de 04-Abr-2008. Incluida parte de los temas 12, 13, 14 y 15.

0.25.010, de 28-Mar-2008. Revisados los apartado 8, 9 y 10.

0.25.007, de 21-Mar-2008. Revisados los apartado 6 y 7.

0.25.005, de 05-Mar-2008. Revisados los apartado 4 y 5.

0.25.003, de 02-Mar-2008. Revisado el apartado 3.

0.25.002, de 29-Feb-2008. Revisado el apartado 2.

0.25.001, de 24-Feb-2008. Cambiado el estilo del curso por otro algo más actual y más legible. Corregidos algunos enlaces huérfanos. Comienza la revisión (por ahora: apartado 1).

Cambios entre versiones recientes del curso, desde la remodelación:

(Mejoras por ahora sobre la versión 0.21, a pesar de tener todavía menos contenido: instrucciones sobre cómo compilar en C para Windows con MinGW Developer Studio ([apartado 1.1](#)), incluido un juego sencillo de adivinar números ([apartado 4](#)), los apartados 7 y 8 incluyen una versión en Java.

0.24.021, de 07-Abr-2007. Ha habido gente (poca) que ha mostrado interés en el curso, así que lo retomo para seguirlo ampliando. Incluida un tema sobre la paleta de colores (apartado 10) y la versión "casi completa" del juego de la serpiente (apartado 11), en su versión en C.

0.24.012, de 11-Sep-2005. Incluida la tercera aproximación al juego de la serpiente: "MiniSerpiente 3" (apartado 9), en su versión en C. Incluido un primer índice alfabético, aunque todavía es sólo el esqueleto, sin incluir información sobre ningún tema.

0.24.011, de 05-Sep-2005. Incluida la segunda aproximación al juego de la serpiente: "MiniSerpiente 2" (apartado 8), en su versión en Java.

0.24.010, de 02-Sep-2005. Incluida la segunda aproximación al juego de la serpiente: "MiniSerpiente 2" (apartado 8), en sus versiones en C y en Pascal.

0.24.009, de 30-Ago-2005. Incluida la primera aproximación al juego de la serpiente: "MiniSerpiente 1" (ap.7) en Java.

0.24.008, de 29-Ago-2005. Incluida la primera aproximación al juego de la serpiente: "MiniSerpiente 1" (apartado 7), en sus versiones en Pascal y en Java.

0.24.007, de 26-Ago-2005. Incluido un tercer juego, el de las "Motos de luz", apartado 6. Actualizado el índice de contenidos para que refleje mejor los próximos apartados previstos.

0.24.006, de 25-Ago-2005. Incluido un segundo juego, el de "El Ahorcado". Es el apartado 5. Incluida también información sobre como compilar en C para Windows usando Dev-C++ o bien MinGW sin entorno de desarrollo.

0.24.005, de 24-Ago-2005. Incluido un primer juego, más sencillo que el que había en la entrega inicial del curso. Se trata de un juego de adivinar números. Es el apartado 4.

0.24.004, de 18-Ago-2005. Revisado e incluido el apartado 2 (entrando a modo gráfico y dibujando). Incluido también el apartado 3 (Leyendo del teclado y escribiendo texto).

0.24.003, de 17-Ago-2005. Mucho más detallada la instalación de MinGW, Allegro y un entorno de desarrollo, para crear juegos para Windows (29 imágenes del proceso).

0.24.002, de 16-Ago-2005. El nuevo tema 1 incluirá la forma de instalar y probar todos los compiladores que se usarán en el curso. Esta entrega tiene la instalación de Free Pascal para Windows y del JDK para Windows, con muchas imágenes (27!) para que nadie tenga problemas a la hora de instalarlos (esperemos).

0.24.001, de 08-Ago-2005. Comenzada la reestructuración del curso, que estará disponible también para consulta en línea.

El curso está siendo revisado desde el primer tema y procurando que todos los ejemplos (o casi) funcionen en C, Pascal y Java, ya desde el primer apartado. Esta entrega sólo incluye la introducción.

Cambios entre anteriores versiones de este curso:

0.21, de 18-Mar-2005. Ampliado el apartado 28, con varias animaciones sencillas para la pantalla de presentación: texto parpadeante, texto que rebota y texto que se escribe secuencialmente. Añadidas instrucciones sobre cómo compilar fuentes para Windows usando Dev-C++. Añadida información sobre el formato de los ficheros PCX y sobre cómo mostrarlos desde Pascal.

0.20, de 06-Mar-2005. Visto que más de una persona me ha escrito interesándose por la continuidad del curso, démosle un pequeño empujón... Añadidas instrucciones sobre cómo compilar los ejemplos de los apartados 24b y 24c. Comenzado el apartado 28 (creación de la pantalla de presentación). Muy pronto habrá más...

0.19, de 29-Dic-2004. Añadida la versión en Java del tercer y el cuarto apartado. Incluida la introducción del apartado 25 (clases a utilizar en el matamarcianos orientado a a objetos) y el apartado 25a (pseudocódigo del matamarcianos).

0.18, de 24-Dic-2004. El segundo apartado incluye también una versión en Java. Añadidas imágenes a la versión en Pascal y en C del segundo apartado. El apartado 25 en C++ está casi listo, pero queda para la siguiente entrega.

0.17, de 20-Nov-2004. Resuelta la primera aproximación orientada a objetos al juego de marcianos con muchos enemigos (apartados 24 a 24d).

0.16, de 24-Oct-2004. Resuelta la segunda aproximación al juego de marcianos con muchos enemigos, que elimina los problemas de parpadeo (apartado 23b).

0.15, de 12-Oct-2004. Resuelta la primera aproximación al juego de marcianos con muchos enemigos (apartado 23a).

0.14, de 16-Sep-2004. Resuelta la cuarta aproximación al juego de marcianos (apartado 22d).

0.13, de 13-Jul-2004. Resuelta la tercera aproximación al juego de marcianos (apartado 22c). Creada una versión en formato PDF totalmente actualizada. Corregida alguna errata de poca importancia por culpa de teclear rápido: "niguna", "bibloteca", "porporciona", "cantidd"...

0.12, de 02-Jun-2004. Resuelta la segunda aproximación al juego de marcianos (apartado 22b). Previstos dos próximos apartados: Cómo redefinir las teclas con las que jugaremos (26), Creando una pantalla de presentación sencilla (27). Ligeramente cambiado el orden previsto de los próximos apartados, para dejar hueco cuanto antes a un juego "completo".

0.11, de 21-May-2004. Resuelta la primera aproximación al juego de marcianos (apartado 22a). Añadidos sonidos de ejemplo para el juego de "Simon", por si alguien no tiene ningún editor MIDI o prefiere el trabajo hecho. Incluida la versión en Pascal del juego de MiniSerpiente, como ejemplo básico de Sprites en Pascal.

0.10, de 21-Abr-2004. Añadida la solución del juego de "Simon" (apartado 19). Añadidos más detalles sobre cómo resolver la primera aproximación al juego de marcianos (apartados 22a a 22c). Incluida la versión en Pascal del apartado 3. Incluida la versión en Pascal del juego de Motos de Luz. La versión en Pascal del juego del ahorcado estaba lista pero no había sido incluida; ahora sí. El índice incluye también el apartado 5 y el 6.

0.09, de 10-Abr-2004. Añadidos dos nuevos apartados: *Formatos de ficheros de imágenes más habituales* (20), *Cómo leer imágenes desde ficheros* (21). Propuesto el ejercicio correspondiente al apartado 22. Una ligera corrección al juego del ahorcado, incluyendo "ctype.h" para que funcione en otros compiladores (DevC++, gracias a Rafael Muñoz por el aviso). Ampliado el apartado 4, para detallar más sobre la generación de números al azar. El índice incluye también el apartado 3 y el 4.

0.08, de 19-Mar-2004. Añadidos tres nuevos apartados: *Distintas resoluciones sin cambiar el programa* (17), *Cómo reproducir sonidos* (18), *Séptimo juego: SimonDice* (19) (este tercer apartado está sin completar). Incluida la versión en Pascal del juego del ahorcado (apartado 4). Incluida una primera versión del índice (por ahora sólo cubre los temas 1 y 2). La página principal (contenido) ya no habla de "entregas" del curso, sino sólo de los temas que se van a a tratar en las próximas actualizaciones. Añadida información sobre mí al apartado "Autor".

0.07, de 28-Feb-2004. Mencionada la posibilidad de usar también el lenguaje Java si hay gente interesada. Añadidos tres nuevos apartados: *Temporizadores con Allegro* (14), *Un poco de matemáticas para juegos* (15), *Sexto juego: TiroAlPlato* (16). Reducido ligeramente el espacio ocupado por algunas imágenes (24 Kb).

0.06, de 29-Ene-2004. Retomado el proyecto, gracias a los nuevos mensajes de usuarios interesados. Añadidos tres nuevos apartados: *Cómo compilar programas para Linux (11)*, *Manejo del ratón (12)*, *Quinto juego: puntería (13)*. Ligeramente ampliado el apartado 6. En el apartado sobre "el autor", eliminada la referencia a mi apartado de Correos, que deja de existir en enero de 2004 por falta de uso. Corregido un enlace incorrecto a la página web de Free Pascal.

0.05, de 30-Jul-2003. Añadido un nuevo apartado: *Cuarto juego: serpiente (10)*.

0.04, de 29-Jun-2003. Retomado el proyecto, después de ver que sí hay gente interesada y que ahora tengo algo más de tiempo libre. Incluye un nuevo apartado: *Más sobre la paleta de colores (9)*.

0.03, de 21-Feb-2003. Incluye dos nuevos apartados: *Cómo crear figuras multicolor que se muevan (7)*, *Mapas, Tercer juego: MiniSerpiente. (8)*. En el apartado sobre cómo compilar para Windows (5), añadido un comentario sobre el fichero DLL que es necesario distribuir. Comenzado a traducir algún fuente a Pascal, para la gente que prefiera usar este lenguaje; esta traducción afecta por ahora sólo al apartado 2 (*entrando a modo gráfico y dibujando*). En los "enlaces", añadidas las páginas oficiales de los compiladores usados en el curso.

0.02, de 30-Ene-2003. Incluye dos nuevos apartados: *Creando programas para Windows (5) - Evitemos esperar al teclado. Segundo juego: motos de luz (6)*. Cambiado ligeramente el orden de los apartados previstos.

0.01, de 15-Ene-2003. Primera versión disponible. Incluye hasta el apartado 4: *¿Por qué este curso? Condiciones de uso, ¿Qué herramientas emplearemos?, Preparando las herramientas, Entrando a modo gráfico y dibujando, Leyendo del teclado y escribiendo texto, Nuestro primer juego: Ahorcado.*



Nacho Cabanes, 2003-2009

Página actualizada: 07-08-2010

Última versión en www.nachocabanes.com