Manic Miner – Remake

El "HowTo"
(o "Cómo se hizo, paso a paso")

Versión 0.12

Última versión en

http://nachocabanes.com/videojuegos/

Nacho Cabanes, 2011

00.- El motivo

El curso 2010/2011 llega casi a su mitad, y es el momento de aplicar conocimientos... ;-)

Como un juego, por sencillo que parezca, es un ejercicio de programación muy completo, a estas alturas de curso mis alumnos tienen ir que creando poco a poco su propio juego, apenas a un ritmo de una hora por semana. Esto debería ayudar a asentar los conocimientos básicos que ya tienen, y también a aplicar conocimientos nuevos que iremos viendo en clase.

Debe ser un juego de plataformas (o uno de un nivel de dicultad similar), que tenga elementos como:

- Un personaje que el usuario pueda controlar (mover a un lado u otro, hacer que salte o dispare, etc).
- Varios enemigos, de distintos tipos, que deberán estar en sus correspondientes clases (y subclases), y que deberán ser capaces de moverse "sólos" (aunque el usuario no pulse ninguna tecla).
- Un fondo repetitivo, creado usando arrays.
- Una pantalla de presentación, que incluya alguna animación (al menos, un texto o una imagen que rebote).
- Una lógica de juego "razonable": comprobar colisiones, perder energía o perder vidas, recoger objetos, etc.
- Una tabla de mejores puntuaciones, que se guarde en fichero y se lea desde fichero.
- ...

Para que tengan un punto de partida y algo con lo que comparar, yo iré realizando también un juego similar a lo que espero que ellos consigan. Ese juego quedará disponible en mi web por si ellos (o cualquier otra persona) quiere consultar la forma en la que he hecho cualquier cosa. Yo he optado por uno de los juegos que ellos no han escogido: Manic Miner, cuya apariencia original (en la versión para Amstrad CPC) era ésta:

Mi versión estará creada en C#, usando Tao.SDL como librería gráfica. He creado un esqueleto de juego, del que podrán partir mis alumnos y del que partiré yo. Este esqueleto es capaz de mostrar una pantalla de presentación, y desde ella acceder a una pantalla de créditos (nombre de los autores) y al juego en sí.

Si quieres saber más, sólo tienes que seguir leyendo...

01.- Esqueleto vacío

En este apartado puedes encontrar:

- Algunas ideas sobre el esqueleto del juego
- Aplicación a esta primer entrega
- Estructura de clases empleada
- Compiladores y otras herramientas necesarias
- ¿Y si tengo dudas?

Algunas ideas sobre el esqueleto del juego

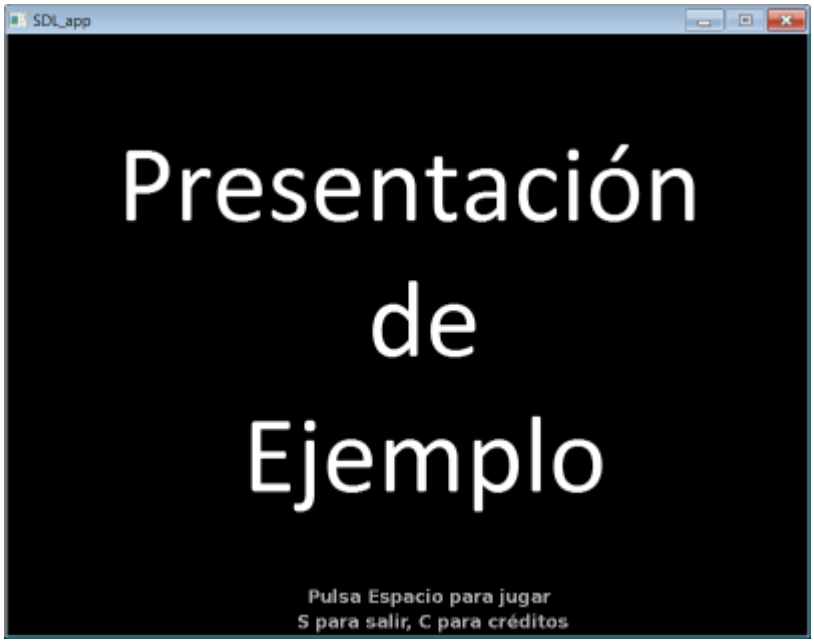
Para no necesitar conocer demasiado los detalles internos de SDL y de su adaptación como parte del "Tao Framework", el esqueleto incluye varias clases auxiliares, como:

- El "Hardware", para poder mostrar cosas en pantalla y leer de teclado.
- Un "Elemento Gráfico", que representa a una imagen en el juego (o incluso a un personaje que está representado por una secuencia de imágenes).
- Una "Fuente" (un tipo de letra), que nos permitirá escribir textos en pantalla.

Para evitar parpadeos, las imágenes y los textos se dibujan en una pantalla oculta (un "doble buffer"), que se muestra cuando todos los elementos están preparados, por lo que tanto nuestros textos como nuestros tipos de letra se dibujarán en dicha pantalla oculta y se mostrarán posteriormente.

Aplicación a esta primera entrega

La pantalla de presentación simplemente muestra una imagen como ésta, escribe un poco de texto aclaratorio y espera a que se pulse la tecla Espacio para jugar, C para ver los créditos o S para salir:



Y su fuente debería ser sencillo de seguir: apenas hay un método "Ejecutar", que se encarga de dibujar la imagen de ejemplo, escribir el texto, esperar a que se pulse una tecla (25 veces por segundo, para no saturar el sistema) y memorizar la tecla pulsada:

```
public  void Ejecutar()
{
    // Dibujo la imagen de la presentacion
    imagen.DibujarOculta(0,0);

    // Escribo avisos de las teclas utilizables
    Hardware.EscribirTextoOculta(
            "Pulsa Espacio para jugar",
            300, 550, 0xAA, 0xAA, 0xAA, fuenteSans18);
    Hardware.EscribirTextoOculta(
            "S para salir, C para créditos",
            290, 575, 0xAA, 0xAA, 0xAA, fuenteSans18);

    // Finalmente, muestro en pantalla
    Hardware.VisualizarOculta();

    //hasta que se pulse espacio (sin saturar la CPU)
    do {
      Hardware.Pausa(40);
    } while ((! Hardware.TeclaPulsada(Hardware.TECLA_ESP) )
```

```
            && (! Hardware.TeclaPulsada(Hardware.TECLA_S))
            && (! Hardware.TeclaPulsada(Hardware.TECLA_C)));

    opcionEscogida = OPC_PARTIDA;
    if (Hardware.TeclaPulsada(Hardware.TECLA_S))
        opcionEscogida = OPC_SALIR;
    if (Hardware.TeclaPulsada(Hardware.TECLA_C))
        opcionEscogida = OPC_CREDITOS;
}
```

(Si quieres ver más detalles, puedes ver el fuente completo en Google Code).

La pantalla de créditos solo escribe algo de texto:

Y su fuente es muy similar: poco más que un método "Ejecutar", que se encarga de escribir los textos de ejemplo y esperar a que se pulse la tecla ESC. Para que no sea totalmente igual a la presentación, en este caso se escribe el texto dentro de la parte repetitiva (algo que sserá totalmente necesario cuando queramos que el texto incluya alguna animación, pero que por ahora podríamos haber evitado, como hemos hecho con la presentación):

```
public  void Ejecutar()
{
  bool salir = false;
  while (! salir )
  {
      Hardware.BorrarPantallaOculta(0,0,0); // Borro en negro

      Hardware.EscribirTextoOculta(
          "XXX (Remake) - Creditos", 110, 100,
            0x77, 0x77, color, fuenteSans18);

      [...]

      Hardware.EscribirTextoOculta(
            "Pulsa ESC para volver a la presentación...",
            110,550,0xAA, 0xAA, 0xAA, fuenteSans12);
```

```
      Hardware.VisualizarOculta();
      Hardware.Pausa(40);

      salir = Hardware.TeclaPulsada (Hardware.TECLA_ESC);
  }
}
```

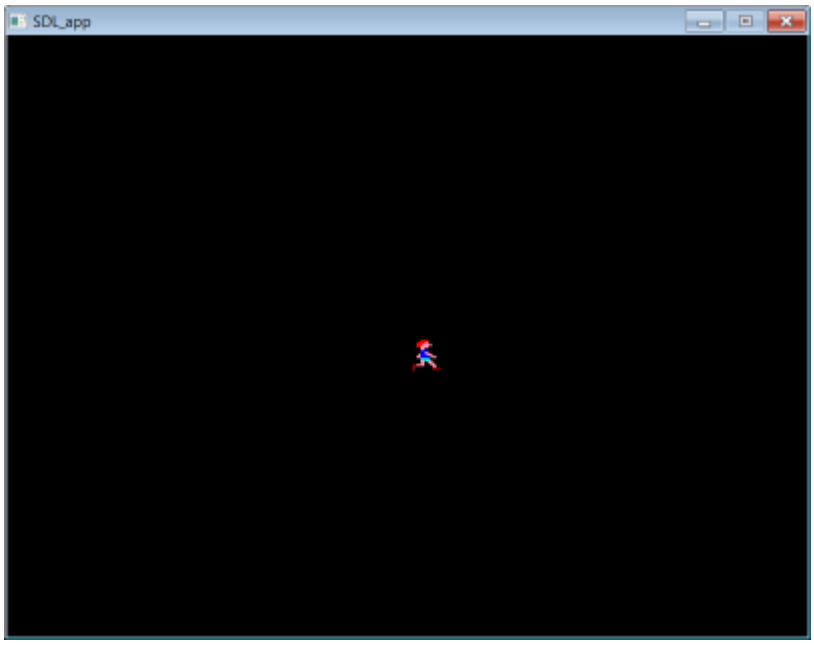
La clase "Juego" (un poco más adelante tienes toda la estructura de clases) se limita a coordinar a las demás: lanza la presentación y, según lo que se escoja en ella, muestra la pantalla de créditos o comienza una nueva partida (una sesión real de juego):

```
public class Juego
{
    private Presentacion presentacion;
    private Partida partida;
    private Creditos creditos;


    // Inicialización al comenzar la sesión de juego
    public Juego()
    {
        // Inicializo modo grafico 800x600 puntos, 24 bits de color
        bool pantallaCompleta = false;
        Hardware.Inicializar(800, 600, 24, pantallaCompleta);

        // Inicializo componentes del juego
        presentacion = new Presentacion();
        partida = new Partida();
        creditos = new Creditos();
    }


    // --- Comienzo de un nueva partida: reiniciar variables ---
    public void Ejecutar()
    {
        do
        {
            presentacion.Ejecutar();
            switch( presentacion.GetOpcionEscogida() )
            {
                    case Presentacion.OPC_CREDITOS:
                        creditos.Ejecutar();
                        break;
                    case Presentacion.OPC_PARTIDA:
                        partida.BuclePrincipal();
                        break;
            }
        }
        while (presentacion.GetOpcionEscogida() != Presentacion.OPC_SALIR );
    }


    // --- Cuerpo del programa -----
    public static void Main()
    {
        Juego juego = new Juego();
        juego.Ejecutar();
```

```
    }

} /* fin de la clase Juego */
```

Finalmente, la clase "Partida" es la que contiene la lógica "real" de una sesión de juego. La parte más importante es el "bucle principal", que repite la secuencia de pasos habituales en este tipo de juegos:

- Comprobar las teclas que pulse el usuario (o joystick, o ratón) y actuar en consecuencia (moviendo nuestro personaje a derecha e izquierda, por ejemplo).
- Mover los elementos que deban moverse sólos (como el enemigo, más adelante, o incluso nuestro propio personaje si está saltando o cayendo).
- Comprobar colisiones entre elementos (por ejemplo, si nuestro personaje toca un enemigo o recoge un objeto de la pantalla).
- Dibujar todos los elementos del juego en su estado actual.
- Hacer una pausa hasta el momento del siguiente fotograma, para que la velocidad de juego sea lo más estable posible, independientemente del ordenador en que se pruebe:



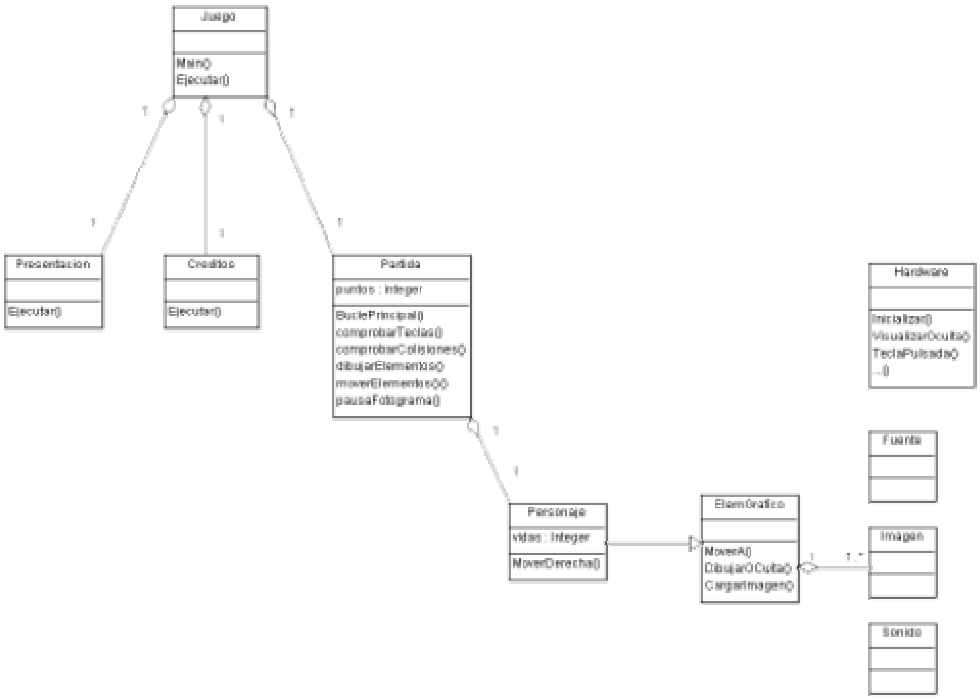

```
public class Partida
{
    // Componentes del juego
    private Personaje miPersonaje;

    // Otros datos del juego
    int puntos;             // Puntuacion obtenida por el usuario
    bool partidaTerminada;  // Si ha terminado una partida

    // Inicialización al comenzar la sesión de juego
    public Partida()
    {
        miPersonaje = new Personaje(this);
        puntos = 0;
        partidaTerminada = false;
    }


    // --- Comprobación de teclas, ratón y joystick -----
```

```
void comprobarTeclas()
{
      // Muevo si se pulsa alguna flecha del teclado
      if (Hardware.TeclaPulsada(Hardware.TECLA_DER) )
          miPersonaje.MoverDerecha();
      if (Hardware.TeclaPulsada(Hardware.TECLA_IZQ))
          miPersonaje.MoverIzquierda();

      if (Hardware.TeclaPulsada(Hardware.TECLA_ARR))
          miPersonaje.MoverArriba();

      if (Hardware.TeclaPulsada(Hardware.TECLA_ABA))
          miPersonaje.MoverAbajo();

      if (Hardware.TeclaPulsada(Hardware.TECLA_ESP))
          miPersonaje.Disparar();

      // Compruebo el Joystick
      if (Hardware.JoystickPulsado(0))
          miPersonaje.Disparar();

      int posXJoystick, posYJoystick;
      if (Hardware.JoystickMovido(out posXJoystick, out posYJoystick))
      {
          if (posXJoystick > 0) miPersonaje.MoverDerecha();
          else if (posXJoystick < 0) miPersonaje.MoverIzquierda();
          else if (posYJoystick > 0) miPersonaje.MoverAbajo();
          else if (posYJoystick < 0) miPersonaje.MoverArriba();
      }

      // Compruebo el raton
      int posXRaton = 0, posYRaton = 0;
      if (Hardware.RatonPulsado(out posXRaton, out posYRaton))
      {
          miPersonaje.MoverA(posXRaton, posYRaton);
      }


      // Si se pulsa ESC, por ahora termina la partida... y el juego
      if (Hardware.TeclaPulsada(Hardware.TECLA_ESC))
          partidaTerminada = true;
}


// --- Animación de los enemigos y demás objetos "que se muevan solos" -----
 void moverElementos()
{
    // Nada por ahora
}


// --- Comprobar colisiones de enemigo con personaje, etc ---
 void comprobarColisiones()
{
    // Nada por ahora
}


// --- Dibujar en pantalla todos los elementos visibles del juego ---
void dibujarElementos()
{
```

```
        // Borro pantalla
        Hardware.BorrarPantallaOculta(0,0,0);

        // Dibujo el personaje
        miPersonaje.DibujarOculta();

        // Finalmente, muestro en pantalla
        Hardware.VisualizarOculta();
    }


    // --- Pausa tras cada fotograma de juego, para velocidad de 25 fps -----
     void pausaFotograma()
    {
        Hardware.Pausa( 40 );
    }


    // --- Bucle principal de juego -----
    public void BuclePrincipal()
    {

        partidaTerminada = false;
        do {
            comprobarTeclas();
            moverElementos();
            comprobarColisiones();
            dibujarElementos();
            pausaFotograma();
        } while (! partidaTerminada);
    }


} /* fin de la clase Partida */
```

Estructura de clases empleada

Como clases "del juego" tenemos las ya comentadas: Juego, Presentacion, Creditos, Partida. Como clases auxiliares que ocultan el acceso al sistema: Hardware, Imagen, Fuente, Sonido y ElemGrafico (haz clic en la imagen para verla más grande):



Compiladores y otras herramientas necesarias

El proyecto compila en Windows, pero me temo que no en Linux (o al menos yo no lo he conseguido, la DLL de Tao.SDL no parece funcionar correctamente en este sistema operativo). Se puede utilizar como entorno de desarrollo Visual Studio: incluyo un proyecto para la versión Visual Studio 2008 Express.

En equipos más antiguos o menos potentes, en los que Visual Studio no se mueva con soltura, se puede usar SharpDevelop: incluyo también un proyecto para SharpDevelop 2.2 (o Visual Studio 2005), que sólo necesita la plataforma .Net versión 2, por lo que se podría usar cualquier ordenador que tenga Windows XP SP2.

También se podría usar como compilador Mono (www.mono-project.com), pero al tratarse de varios fuentes habría que crearse un fichero BAT para compilarlos todos juntos. No incluyo dicho fichero BAT (al menos por ahora), así que quien quiera usar Mono tendrá que buscarse la vida un poco más... o preguntar.

Como siempre, si quieres saber más, sólo tienes que seguir leyendo...

¿Y si tengo dudas?

Si te surgen dudas y eres alumno mío, ya sabes dónde preguntar ;-) pero si no eres alumno mío, tampoco lo tienes especialmente difícil: puedes localizarme en un foro de AprendeAProgramar.com que está dedicado a este proyecto.

Y si quieres descargar el proyecto para verlo con más detenimiento o para usarlo como base para cualquier proyecto tuyo, puedes hacerlo desde lapágina del proyecto en Google Code (ésta es la versión 0.01, claro).

02.- Añadir un enemigo que se mueva solo

¿Qué hará esta versión?

- Mostrar un enemigo, que se mueva solo de izquierda a derecha
- Incluir imágenes reales para el enemigo y el personaje

- Incluir una imagen real para la pantalla de presentación

Vamos a ver cómo se puede conseguir...

Mostrar un enemigo, que se mueva solo de izquierda a derecha

Por una parte, deberemos crear la clase Enemigo, descendiente de ElemGrafico, que tendrá apenas dos métodos: un constructor (que dará los valores iniciales a X, Y, su velocidad y su imagen) y un método Mover (que aumentará su X en cada pasada, y cambiará el signo del incremento cada vez que llegue a un extremo). Los demás métodos, como DibujarOculta, no es necesario crearlos porque se heredan de ElemGrafico:

```
public class Enemigo : ElemGrafico
{
  Partida miPartida;

  // Constructor
  public Enemigo(Partida p)
  {
    miPartida = p;   // Para enlazar con el resto de componentes
    x = 400;         // Resto de valores iniciales
    y = 400;
    incrX = 4;
    CargarImagen("imagenes/enemigo.png");
  }


  // Métodos de movimiento
  public void Mover()
  {
      x += incrX;

      if ((x < 100) || (x > 700))
        incrX = (short) (- incrX);
  }

} /* fin de la clase Personaje */
```

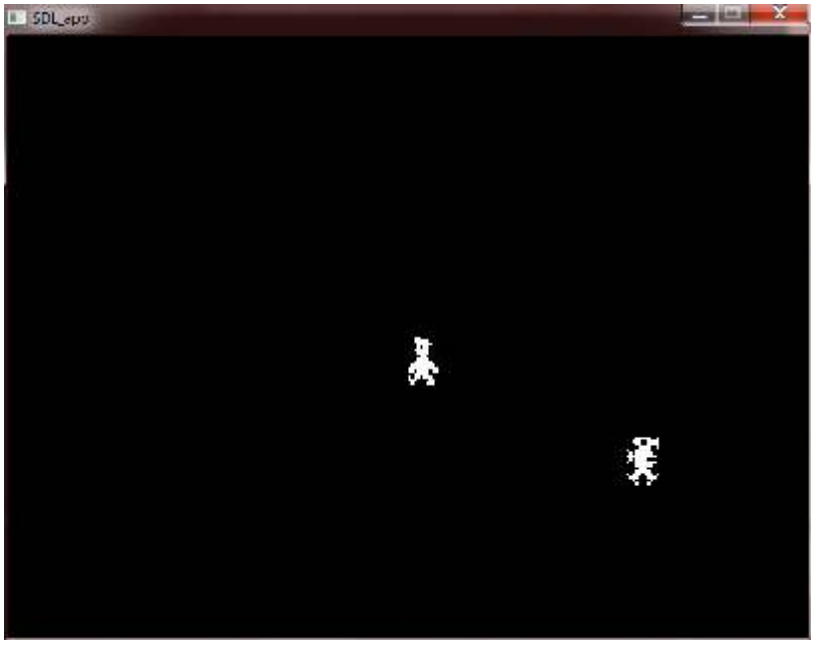
Por su parte, la partida no tendrá muchos cambios: declarar el Enemigo, inicializarlo en el constructor, y dibujarlo y moverlo en los métodos correspondientes, así:

```
public class Partida
{

    // Componentes del juego
    private Personaje miPersonaje;
    private Enemigo miEnemigo;
    [...]

    public Partida()
    {
        miPersonaje = new Personaje(this);
        miEnemigo = new Enemigo(this);
        [...]
    }


    void moverElementos()
    {
```

```
    miEnemigo.Mover();
}


void dibujarElementos()
{
    // Borro pantalla
    Hardware.BorrarPantallaOculta(0,0,0);

    // Dibujo el personaje y el enemigo
    miPersonaje.DibujarOculta();
    miEnemigo.DibujarOculta();
    [...]
```

Incluir imágenes reales para el enemigo y el personaje

Simplemente es cambiar la imagen "personaje.png" por otra que sí pertenezca a este juego, y añadir una imagen para el enemigo, con lo que el resultado podría ser así:



Incluir una imagen real para la pantalla de presentación

Basta con cambiar la imagen "present.png" por otra:

Si quieres descargar el proyecto, lo tienes en su página en Google Code(ésta es la versión 0.02).

03.- Añadir un fondo repetitivo, usando arrays.

¿Qué hará esta versión?

En los juegos de plataformas clásicos, era habitual que una pantalla de juego estuviera formada por casillas repetitivas, lo que simplifica la creación de niveles y además permite gastar menos memoria para almacenarla:

Para imitarlo, vamos a hacer lo siguiente en nuestro juego:

- Crear una clase Mapa, que represente una pantalla de fondo formada por casillas repetitivas

- Actualizar la clase Partida, para que use esta nueva clase

Vamos a ver cómo se puede conseguir...

Crear una clase Mapa, que represente una pantalla de fondo formada por casillas repetitivas

Por ahora, la clase Mapa contendrá poco más que:

- Un array que represente la distribución de casillas en la pantalla. Este podría ser un array bidimensional del enteros, o, más legible aún, de caracteres. La forma más cómoda de conseguirlo en C# puede ser usando un array de strings, de modo cada letra de cada línea corresponda a una casilla del fondo (por ejemplo, la L podría indicar dónde debe aparecer un ladrillo, y la V dónde debe haber una llave).
- un constructor, que cargará las imágenes de esas casillas que dibujaremos más adelante.
- Un método DibujarOculta, que se limitará a recorrer el array y dibujar una casilla u otra según corresponda, usando una orden "switch":

```
public class Mapa : ElemGrafico
{
    Partida miPartida;

    private int altoMapa = 16, anchoMapa = 32;
    private int anchoTile = 24, altoTile = 24;
    private int margenIzqdo = 20, margenSuperior = 100;

    ElemGrafico arbol, deslizante, ladrillo, ladrilloX, llave, puerta,
        sueloFino, sueloFragil, sueloGrueso, techo;

    string[] datosNivel =
    {"L        V T    T            V L",
     "L               V              L",
     "L                              L",
     "L                              L",
     "L                      VA  A   L",
     "LSSSSSSSSSSSSSFFFFSFFFFSSSSSSSSL",
     "L                             VL",
     "LSSS                          L",
     "L                LLL A        L",
     "LSSSS   DDDDDDDDDDDDDDDDDDD   L",
     "L                            SSL",
     "L                              L",
     "L            A      LLLFFFFFSSSL",
     "L    SSSSSSSSSSSSSS          PPL",
     "L                            PPL",
     "LSSSSSSSSSSSSSSSSSSSSSSSSSSSSSL"};

    // Constructor
    public Mapa(Partida p)
    {
        miPartida = p;    // Para enlazar con el resto de componentes

        arbol = new ElemGrafico("imagenes/arbol.png");
        deslizante = new ElemGrafico("imagenes/deslizante.png");
        ladrillo = new ElemGrafico("imagenes/ladrillo.png");
        ladrilloX = new ElemGrafico("imagenes/ladrillo2.png");
        llave = new ElemGrafico("imagenes/llave.png");
        puerta = new ElemGrafico("imagenes/puerta.png");
        sueloFino = new ElemGrafico("imagenes/suelo.png");
        sueloFragil = new ElemGrafico("imagenes/sueloFragil.png");
        techo = new ElemGrafico("imagenes/techo.png");
```

```
    }

    public void DibujarOculta()
    {
        // Dibujo el fondo
        for (int i = 0; i < altoMapa; i++)
            for (int j = 0; j < anchoMapa; j++)
            {
                int posX = j * anchoTile + margenIzqdo;
                int posY = i * altoTile + margenSuperior;
                switch (datosNivel[i][j])
                {
                    case 'A': arbol.DibujarOculta(posX, posY); break;
                    case 'D': deslizante.DibujarOculta(posX, posY); break;
                    case 'F': sueloFragil.DibujarOculta(posX, posY); break;
                    case 'L': ladrillo.DibujarOculta(posX, posY); break;
                    case 'P': puerta.DibujarOculta(posX, posY); break;
                    case 'S': sueloFino.DibujarOculta(posX, posY); break;
                    case 'T': techo.DibujarOculta(posX, posY); break;
                    case 'V': llave.DibujarOculta(posX, posY); break;
                }
            }
    }

} /* fin de la clase Mapa */
```

Actualizar la clase Partida, para que use esta nueva clase

Por su parte, la partida tendrá muy pocos cambios: declarar el Mapa, inicializarlo en el constructor, y dibujarlo en el método correspondientes, así:

```
public class Partida
{
    // Componentes del juego
    private Personaje miPersonaje;
    private Enemigo miEnemigo;
    private Mapa miPantallaJuego;
    [...]

    public Partida()
    {
        miPersonaje = new Personaje(this);
        miEnemigo = new Enemigo(this);
        miPantallaJuego = new Mapa(this);
        [...]
    }

    void dibujarElementos()
    {
        // Borro pantalla
        Hardware.BorrarPantallaOculta(0,0,0);

        // Dibujo todos los elementos
        miPantallaJuego.DibujarOculta();
        miPersonaje.DibujarOculta();
        miEnemigo.DibujarOculta();

        // Finalmente, muestro en pantalla
        Hardware.VisualizarOculta();
    }
```

```
[...]
```

El resultado podría quedar así:

Ésta es la versión 0.03 en la página del proyecto en Google Code.

04.- Comprobar colisiones con el fondo (no cruzar paredes ni suelos)

¿Qué hará esta versión?

- El mapa será capaz de informar de si es posible moverse a unas ciertas coordenadas dentro de él
- El personaje comprobará si se puede mover a una posición antes de hacerlo

Vamos a ver cómo se puede conseguir...

El mapa será capaz de informar de si es posible moverse a unas ciertas coordenadas dentro de él

Podemos crear una función "EsPosibleMover", que reciba las coordenadas límite de un rectángulo (el que ocuparía el personaje después de moverse, por ejemplo), y nos devuelva "true" si es una posición válida, o "false" cuando no lo sea (porque se solape con alguna casilla del fondo):

```
public bool EsPosibleMover(int x, int y, int xmax, int ymax)
{
   // Compruebo si choca con alguna casilla del fondo
   for (int fila = 0; fila < altoMapa; fila++)
          for (int colum = 0; colum < anchoMapa; colum++)
          {
              int posX = colum * anchoTile + margenIzqdo;
              int posY = fila * altoTile + margenSuperior;
              // Si no es espacio blanco
              if ((datosNivel[fila][colum] != ' ')
                  // Y se solapa con la posic a la que queremos mover
                  && (posX+anchoTile > x) && (posY+altoTile > y)
                  && (xmax> posX) && (ymax > posY))
```

```
                {
                    return false;
                }
            }

    return true;
}
```

El personaje comprobará si se puede mover a una posición antes de hacerlo

Cuando se le pida mover a la derecha (o a cualquier otra dirección), comprobará primero si terminaría en una posición del mapa que realmente sea pisable:

```
public void MoverDerecha()
{
    if (miPartida.GetMapa().EsPosibleMover(x + incrX, y,
            x + ancho + incrX, y + alto))
        x += incrX;
}
```

Ésta es la versión 0.04, que puedes descargar en la página del proyecto en Google Code.

05.- Salto del personaje (sin gravedad)

¿Qué hará esta versión?

- Cuando se pulse Espacio, el personaje comenzará la secuencia de salto
- Si está en medio de la secuencia de salto, el personaje la continuará aunque no se pulse ninguna tecla

Vamos a ver cómo se puede conseguir...

Cuando se pulse Espacio, el personaje comenzará la secuencia de salto

Por una parte, debemos definir la secuencia de movimientos que hará durante el salto. Se puede conseguir con un "array", que almacene cuanto deberá aumentar la coordenada "y" de nuestro personaje en cada instante del salto (aumentará mucho al principio, luego aumentará más despacio, y finalmente repetirá los mismos pasos, pero disminuyendo, para la secuencia de caida):

```
int[] pasosSaltoArriba = {-14, -14, -11, -8, -6, -4, -2, 0,
                          0, 2, 4, 6, 8, 11, 14, 14 };
```

Y deberemos poner en marcha la secuencia de salto cuando se pulse cierta tecla (por ejemplo, Espacio). Esto se limitará a cambiar el valor de una variable booleana, porque después el resto del movimiento del salto debe proseguir aunque ya no se esté pulsando esa tecla.

```
// Comienza la secuencia de salto
public void Saltar()
{
    if (saltando)
        return;
    saltando = true;
    fotogramaMvto = 0;
    incrXSalto = 0;
}
```

De la misma forma, podemos hacer que salte hacia los lados, si existe un incremento que aplicar a la coordenada X, ya sea positivo o negativo:

```
// Comienza la secuencia de salto hacia la derecha
public void SaltarDerecha()
{
  Saltar();
  incrXSalto = incrX;
}

// Comienza la secuencia de salto hacia la izquierda
public void SaltarIzquierda()
{
  Saltar();
  incrXSalto = -incrX;
}
```

Por supuesto, esto se tiene que reflejar también en la clase "Partida", que debe permitir que se pulse Espacio (posiblemente, a la vez que otra tecla):

```
void comprobarTeclas()
{
      [...]
      if (Hardware.TeclaPulsada(Hardware.TECLA_ESP))
      {
          if (Hardware.TeclaPulsada(Hardware.TECLA_DER))
            miPersonaje.SaltarDerecha();
          else if (Hardware.TeclaPulsada(Hardware.TECLA_IZQ))
              miPersonaje.SaltarIzquierda();
          else
              miPersonaje.Saltar();
      }
      [...]
}
```

Si está en medio de la secuencia de salto, el personaje la continuará aunque no se pulse ninguna tecla

Deberemos crear una nueva función (por ejemplo, "Mover"), que no haga nada cuando no se está saltando, y que si se está en medio de un salto, calcule cual sería la siguiente posición a la que hay que desplazarse. Si ya se han dado todos los pasos que forman la secuencia de salto, cambiará el valor de "saltando" a "false" para dar la secuencia por terminada:

```
  // Para cuando deba moverse solo, p.ej. saltando
  public new void Mover()
  {
      if (saltando)
      {
          // Calculo las siguientes posiciones y veo si son validas
          short xProxMov = (short) (x + incrXSalto);
          short yProxMov = (short) (y + pasosSaltoArriba[fotogramaMvto]);
          bool subiendoSalto = (pasosSaltoArriba[fotogramaMvto] < 0);

          // Si todavía se puede mover, avanzo
```

```
            if (miPartida.GetMapa().EsPosibleMover(
                xProxMov, yProxMov + alto - 4,
                xProxMov + ancho, yProxMov + alto)
                || subiendoSalto)
            {
                x = xProxMov;
                y = yProxMov;
            }
            // Y si no, quizá esté cayendo
            else
                saltando = false;

            fotogramaMvto++;
            if (fotogramaMvto >= cantidadMovimientoSalto)
                saltando = false;
        }

    }
```

Obviamente, esto falla cuando saltamos en el extremo de una plataforma, porque el personaje todavía no "cae" sino que se queda "en el aire", aunque no tenga nada debajo. Los cambios para que caiga son pequeños, pero los veremos un poco más adelante.

Esta versión, que no tiene ningún cambio en cuanto a apariencia, es la 0.05, que puedes descargar en la página del proyecto en Google Code.

06.- Pantalla de créditos con texto real. Presentación con animación.

¿Qué hará esta versión?

- Que los créditos muestren los nombres de los autores reales
- Que la presentación no sea totalmente estática, sino que tenga una pequeña animación

Vamos a ver cómo se puede conseguir...

Que los créditos muestren los nombres de los autores reales

Ninguna dificultad: simplemente cambiar los "EscribirTextoOculta" que tenían datos de ejemplo por otros con datos reales...

```
public class Creditos
{
    [...]

    public  void Ejecutar()
    {
      bool salir = false;

      while (! salir )
      {
          [...]
          Hardware.EscribirTextoOculta("Programa original:", 200, 240,
                color, color, 0, fuenteSans18);
          Hardware.EscribirTextoOculta("Matthew Smith", 200, 270,
                color, color, 0, fuenteSans18);
          Hardware.EscribirTextoOculta("(c) Software Projects, 1984", 200, 300,
                color, color, 0, fuenteSans18);
```

```
            [...]
        }
    }
} /* fin de la clase Creditos */
```

Que la presentación no sea totalmente estática, sino que tenga una pequeña animación

Tampoco tiene por qué ser difícil: por ejemplo, podemos hacer que una imagen de pequeño tamaño se mueva, rebotando cada vez que llegue a los extremos de la pantalla.

Tampoco tiene por qué ser difícil: por ejemplo, podemos hacer que una imagen de pequeño tamaño se mueva, rebotando cada vez que llegue a los extremos de la pantalla.

La idea de cómo hacer que algo rebote en los extremos ya la vimos cuando incluimos el Enemigo, en la versión 0.02. Para la imagen que rebota, podemos "extraer" a una nueva imagen el triángulo que aparecía en la presentación de esa misma entrega 0.02. Los cambios en el fuente serían pocos:

```
public class Presentacion
{
    [...]
    public  void Ejecutar()
    {
        int x = 100, y = 100; // Coordenadas del cartel movil
        int incrX = 4, incrY = 4; // Velocidad del cartel movil

        //hasta que se pulse espacio (sin saturar la CPU)
        do
        {
            // Dibujo la imagen de la presentacion
            imagenFondo.DibujarOculta(0, 0);

            [...]

            // Dibujo y desplazo el cartel móvil
            cartelMovil.DibujarOculta(x, y);
            x += incrX;
            y += incrY;

            // Invierto velocidad si llega al borde
            if ((x < 10) || (x > 800 - 10 - 192))
                incrX = -incrX;
            if ((y < 10) || (y > 600 - 10 - 196))
                incrY = -incrY;

            [...]
```

Y la apariencia sería similar a la de antes... pero ahora al menos con algo de movimiento:

Esta versión es la 0.06, que puedes descargar en la página del proyecto en Google Code.

07.- Colisiones con enemigo, perder vidas, reiniciar partida

¿Qué hará esta versión?

- Cambios en personaje y enemigo: ancho y alto
- Cambios en Partida: colisiones y reiniciar
- Cambios menores en Juego, Presentacion, Opciones

Veamos cómo hacerlo...

Cambios en personaje y enemigo: ancho y alto

Casi ninguno: la rutina de colisiones (básica, por solapamiento de rectángulos) ya está definida en la clase ElemGrafico, así que basta con asegurarnos de que hayamos indicado el ancho y el alto de cada uno de los elementos que puede chocar, para que la comprobación de estas colisiones sea correcta.

Lo haríamos en el constructor de cada una de estas clases: para enemigo

```
public class Enemigo : ElemGrafico
{
  // Constructor
  public Enemigo(Partida p)
  {
    [...]
    SetAnchoAlto(36, 48);
  }

  [...]
```

y para personaje

```
public class Personaje : ElemGrafico
{
  [...]
```

```
  // Constructor
  public Personaje(Partida p)
  {
    [...]
    SetAnchoAlto(30, 48);
  }
  [...]
```

Cambios en Partida: colisiones y reiniciar

La rutina de comprobación de colisiones, que hasta ahora estaba vacia, deberá comprobar si tocamos al enemigo, para restar una vida, y dar la partida por terminada cuando no nos queden vidas. De paso, podemos hacer que la rutina de dibujar elementos al menos avise de cuantas vidas nos quedan:

```
// --- Comprobar colisiones de enemigo con personaje, etc ---
 void comprobarColisiones()
{
    if (miPersonaje.ColisionCon(miEnemigo))
    {
        miPersonaje.Morir();
        miPersonaje.Reiniciar();
        miEnemigo.Reiniciar();
    }

     if (miPersonaje.GetVidas() == 0)
         partidaTerminada = true;
}


// --- Dibujar en pantalla todos los elementos visibles del juego ---
void dibujarElementos()
{
    // Borro pantalla
    Hardware.BorrarPantallaOculta(0,0,0);

    // Dibujo todos los elementos
    miPantallaJuego.DibujarOculta();
    miPersonaje.DibujarOculta();
    miEnemigo.DibujarOculta();

    // Muestro vidas (pronto será parte del marcador)
    Hardware.EscribirTextoOculta("Vidas: "+miPersonaje.GetVidas(),
        280, 550, 0xAA, 0xAA, 0xAA, fuenteSans18);

    // Finalmente, muestro en pantalla
    Hardware.VisualizarOculta();
}
```

Cambios menores en Juego, Presentacion, Opciones

En Juego no es necesario ningún cambio, pero podemos aprovechar para hacer una pequeña mejora: si usamos la tecla Espacio para entrar al juego y también para saltar, la partida comenzará con nuestro personaje saltando, sí que podemos hacer que se comience a jugar con la tecla J.

La Presentación tampoco necesita cambios... salvo avisar de esa tecla en su pequeña zona de ayuda.

De paso, podemos preparar una pantalla de Opciones, que más adelante nos permita escoger cosas como imágenes de mayor o menor resolución, jugar en ventana o pantalla completa, etc. De momento se limitará a escribir texto, por lo que será casi igual que la pantalla de Creditos.

No hay apenas cambios en la apariencia. Esta versión es la 0.07, que puedes descargar en la página del proyecto en Google Code.

08.- Gravedad (caer al final de una plataforma). Mejoras gráficas.

¿Qué hará esta versión?

- Gravedad (caer al final de una plataforma)
- Mejoras gráficas
- Cambios menores en Mapa y Partida

Veamos cómo hacerlo...

Gravedad (caer al final de una plataforma)

No son grandes cambios: por una parte, después de cada movimiento, dejaremos "marcado" que ahora se deberá comprobar si hay que caer:

```
public void MoverDerecha()
{
    if (saltando || cayendo) return; // No debe moverse mientras salta

    if (miPartida.GetMapa().EsPosibleMover(x + incrX, y + alto - 4,
              x + ancho + incrX, y + alto))
          x += incrX;

    cayendo = true;
}
```

y la rutina de "Mover" el personaje, si tiene marcado que quizá haya que caer, deberá comprobar si hay hueco debajo, y moverlo en ese caso:

```
// Para cuando deba moverse solo, p.ej. saltando
public new void Mover()
{
    if (saltando)
    {
        [...]
    }
    else if (cayendo)
    {
        if (miPartida.GetMapa().EsPosibleMover(
            x, y + incrY + alto - 4,
           x + ancho, y + incrY + alto))
        {
            y += incrY;
        }
        else
            cayendo = false;
    }
}
```

Mejoras gráficas

Ya que estamos, vamos a usar un nuevo juego de imágenes, partiendo de las que en su día creó Andy Noble. Las originales seguirán estando disponibles, pero en una carpeta llamada "originales", para que más adelante se pueda optar entre jugar con la apariencia clásica o la mejorada.

Ahora el juego se vería así:

Cambios menores en Mapa y Partida

Ahora en el mapa deberemos hacer algún "agujero", porque si no nuestro personaje ya no podrá bajar desde las zonas más altas cuando están completamente cerradas (el juego original tenía zonas que se hundían, pero nuestra versión todavía no las tiene).

Y en Partida podemos eliminar ya las opciones de mover hacia arriba y hacia abajo, que no existían en el juego original, y que nos eran cómodas para hacer pruebas preliminares, pero que ahora ya no son (casi) necesarias.

Puedes descargar la versión 0.08 completa en la página del proyecto en Google Code.

09.- Recoger objetos del fondo (y obtener puntos).

¿Qué hará esta versión?

- Mejora en la clase Mapa: obtener puntos
- Cambios en la clase "Partida": ampliar ComprobarColisiones

Veamos cómo hacerlo...

Mejora en la clase Mapa: obtener puntos

Por una parte, tenemos que ampliar la clase Mapa. Va a afectar a dos cosas: podremos "pisar" las casillas que contengan ciertos objetos, como las llaves, y además esas casillas deberían darnos puntos.

Para poder "pisar" esas casillas basta ampliar un poco la función "EsPosibleMover", para que contemple más tipos de casillas:

```
public bool EsPosibleMover(int x, int y, int xmax, int ymax)
{

    // Compruebo si choca con alguna casilla del fondo
    for (int fila = 0; fila < altoMapa; fila++)
```

```
            for (int colum = 0; colum < anchoMapa; colum++)
            {
                int posX = colum * anchoTile + margenIzqdo;
                int posY = fila * altoTile + margenSuperior;
                // Si se solapa con la posic a la que queremos mover
                if ((posX+anchoTile > x) && (posY+altoTile > y)
                    && (xmax> posX) && (ymax > posY))
                    // Y no es espacio blanco, llave, puerta o arbol
                    if ((datosNivel[fila][colum] != ' ')
                        && (datosNivel[fila][colum] != 'V')
                        && (datosNivel[fila][colum] != 'P')
                        && (datosNivel[fila][colum] != 'A'))
                    {
                        return false;
                    }
             }

    return true;
}
```

Y para obtener puntos, podemos crear una nueva función "ObtenerPuntosPosicion", muy similar a la anterior, pero que en vez de devolver "true" o "false", nos devuelva la cantidad de puntos que obtendríamos al llegar a cierta posición (por ejemplo, 10 puntos si tocamos una llave). Además de devolvernos esos 10 puntos, esta misma función se podría encargar de "borrar" la llave (reemplazando la V del array por un espacio en blanco).

La comprobación de si hemos llegado a una puerta sería similar, pero la puerta sólo debe abrirse cuando hallamos recogido todas las llaves, para lo que podemos usar un contador de llaves restantes.

De paso, podemos comprobar también si hemos tocado algún objeto "mortal", como los arbolitos o las estalactitas del techo, y hacer que devuelvan una puntuación negativa (por ejemplo, -1), que nos indique que debemos perder una vida.

Si controlamos esos 3 casos desde nuestra función "ObtenerPuntosPosicion", ésta podría quedar así:

```
public int ObtenerPuntosPosicion(int x, int y, int xmax, int ymax)
{

    // Compruebo si choca con alguna casilla del fondo
    for (int fila = 0; fila < altoMapa; fila++)
        for (int colum = 0; colum < anchoMapa; colum++)
        {
            int posX = colum * anchoTile + margenIzqdo;
            int posY = fila * altoTile + margenSuperior;

            // Si choca con la casilla que estoy mirando
            if ( (posX + anchoTile > x) && (posY + altoTile > y)
                && (xmax > posX) && (ymax > posY))
            {
                // Si choca con el techo o con un arbol
                if ((datosNivel[fila][colum] == 'T')
                          || (datosNivel[fila][colum] == 'A'))
                    return -1; // (puntuacion -1: perder vida)

                // Si toca una llave
                if (datosNivel[fila][colum] == 'V')
                {
                    // datosNivel[fila][colum] = ' '; (No valido en C#: 2 pasos)
                    datosNivel[fila] = datosNivel[fila].Remove(colum, 1);
```

```
                    datosNivel[fila] = datosNivel[fila].Insert(colum, " ");
                    return 10;
                }

                // Si toca la puerta y no quedan llaves, 50 puntos
                // y (pronto) pasar al nivel siguiente
                if ((datosNivel[fila][colum] == 'P')
                    && (llavesRestantes == 0))
                {
                    return 50;
                }

            } // Fin de comprobacion de casilla actual
        } // Fin de "for" que revisa todas las casillas

    // Si llego hasta aqui, es que no hay puntos que devolver
    return 0;

}
```

Cambios en la clase "Partida": ampliar ComprobarColisiones

Esto se refleja en dos cambios en la clase partida, uno pequeño y otro más grande.

El cambio pequeño es que por fin empezaremos a usar esa variable "puntos", por lo que deberíamos mostrar su valor desde "DibujarElementos", y deberemos volver a dejarlo con valor 0 cuando termine una partida.

El otro cambio es que "ComprobarColisiones" ya no sólo mirará si chocamos con el enemigo, sino también los puntos que obtendremos en la nueva posición, para sumarlos al marcador (si son positivos) o quitarnos una vida (si son negativos):

```
// --- Comprobar colisiones de enemigo con personaje, etc ---
 void comprobarColisiones()
{
    // Colisiones de personaje con fondo: obtener puntos o perder vida
    int puntosMovimiento = miPantallaJuego.ObtenerPuntosPosicion(
      miPersonaje.GetX(),
      miPersonaje.GetY(),
      miPersonaje.GetX() + miPersonaje.GetAncho(),
      miPersonaje.GetY() + miPersonaje.GetAlto());

    // Si realmente ha recogido un objeto, sumamos los puntos en el juego
    if (puntosMovimiento > 0)
    {
        puntos += puntosMovimiento;

        // Si ademas es una puerta, avanzamos de nivel
        if (puntosMovimiento == 50)
            //avanzarNivel()
            ;
    }

    // Y si es -1, ha chocaco con el fondo: igual caso que las
    // colisiones de personaje con enemigo: recolocar y perder vida
    if ((puntosMovimiento <0 ) || miPersonaje.ColisionCon(miEnemigo))
    {
        miPersonaje.Morir();
        miPersonaje.Reiniciar();
```

```
            miEnemigo.Reiniciar();
        }

        if (miPersonaje.GetVidas() == 0)
            partidaTerminada = true;
}
```

Como siempre, puedes descargar la versión 0.09 completa en la página del proyecto en Google Code.

10.- Añadir clase Marcador, que muestre vidas, puntos, aire

¿Qué hará esta versión?

- Crear la clase Marcador
- Cambios en "Partida": usar el marcador
- Otros cambios menores

Veamos cómo hacerlo...

Crear la clase Marcador

Queremos sacar todo lo relativo al "marcador" (vidas, puntos, energía, etc.) a una clase nueva, para que no se vaya complicando innecesariamente la función "DibujarElementos" de la partida.

De paso, aprovecharemos para mejorar un poco la apariencia: no sólo tendrá los métodos SetVidas, SetAire, SetPuntuacion que deben valor a estos atributos, sino que podemos incluir imágenes repetitivas para "dibujar" la cantidad de vidas restantes y (más adelante) el aire que queda, de modo que la apariencia sea:

Y el fuente que lo hace podría ser así:

```
public class Marcador
{
    private ElemGrafico imgVidas, imgEnergia;
    private ElemGrafico imgAireRojo, imgAireRojoVacio, imgAireVerde, imgAireVerdeVacio;
```

```
private ElemGrafico imgFondoMetal;
private int aire;
private int puntuacion;
private int mejorPunt;
private int vidas;

private Partida miPartida;
Fuente tipoDeLetra;

public void SetVidas( int valor )
{
  vidas = valor;
}

public int GetVidas()
{
  return vidas;
}

public Marcador(Partida p)
{
    miPartida = p;
    tipoDeLetra = new Fuente("FreeSansBold.ttf", 18);
    imgVidas = new ElemGrafico("imagenes/personaje.png");
    imgAireRojo = new ElemGrafico("imagenes/aireRojo.png");
    imgAireRojoVacio = new ElemGrafico("imagenes/aireRojoV.png");
    imgAireVerde = new ElemGrafico("imagenes/aireVerde.png");
    imgAireVerdeVacio = new ElemGrafico("imagenes/aireVerdeV.png");
    imgFondoMetal = new ElemGrafico("imagenes/metal.png");
}


/// Devuelve el valor del aire
public  int GetAire()
{
  return aire;
}


/// Cambia el valor del aire
public  void SetAire(int valor)
{
  aire = valor;

}

/// Incrementa el valor del aire
public void IncrAire(int valor)
{
    aire += valor;
}


/// Devuelve el valor de la puntuación
public  int GetPuntuacion()
{
  return puntuacion;
}


/// Cambia el valor de la mejor puntuación
```

```
    public  void SetMejorPuntuacion(int valor)
    {
      mejorPunt = valor;
    }


    /// Cambia el valor de la puntuación
    public  void SetPuntuacion(int valor)
    {
      puntuacion = valor;
    }


    /// Incrementa el valor de la puntuación
    public  void IncrPuntuacion(int valor)
    {
      puntuacion += valor;
    }


    public  void DibujarOculta()
    {

      // Carteles
      //Hardware.EscribirTextoOculta("Vidas: " + miPartida.GetPersonaje().GetVidas(),
      //   280, 550, 0xAA, 0xAA, 0xAA, tipoDeLetra);

     Hardware.EscribirTextoOculta("Mejor puntuación: 000000",
         200, 520, 0xFF, 0xFF, 0x00, tipoDeLetra);
     Hardware.EscribirTextoOculta("Puntos: " + puntuacion.ToString("000000"),
         550, 520, 0xFF, 0xFF, 0x00, tipoDeLetra);

     // Borde superior e inferior y nombre de nivel (prefijado por ahora)
     imgFondoMetal.DibujarOculta(0, 0);
     imgFondoMetal.DibujarOculta(0, 420);
     Hardware.EscribirTextoOculta("Central Cavern",
         350, 430, 0,0,0, tipoDeLetra);

     // Medidor de aire
     int i;
     for (i = 0; i < 200; i++)
     {
         if (i < 25) imgAireRojo.DibujarOculta(i * 4, 460);
         else if (i < 60) imgAireRojoVacio.DibujarOculta(i * 4, 460);
         else if (i < 175) imgAireVerdeVacio.DibujarOculta(i * 4, 460);
         else imgAireVerde.DibujarOculta(i * 4, 460);
     }
     Hardware.EscribirTextoOculta("Aire",
      10, 464, 0, 0, 0, tipoDeLetra);

      // Vidas, como varias imagenes (maximo 5)
     int vidasMostrar = vidas;
     if (vidasMostrar > 5) vidasMostrar = 5;
     for (i = 0; i < vidasMostrar; i++)
     {
        imgVidas.DibujarOculta(20 + 40*i, 500);
     }

    }
} /* end class Marcador */
```

Cambios en "Partida": usar el marcador

Ahora la rutina de "DibujarElementos" ya no mostrará las vidas y los puntos directamente, sino que pasará esta información al Marcador, y dibujará el marcador:

```
// --- Dibujar en pantalla todos los elementos visibles del juego ---
void dibujarElementos()
{
    // Borro pantalla
    Hardware.BorrarPantallaOculta(0,0,0);

    // Dibujo todos los elementos
    miPantallaJuego.DibujarOculta();
    miPersonaje.DibujarOculta();
    miEnemigo.DibujarOculta();

    // Y el marcador
    miMarcador.SetVidas(miPersonaje.GetVidas());
    miMarcador.SetPuntuacion(puntos);
    miMarcador.DibujarOculta();

    // Finalmente, muestro en pantalla
    Hardware.VisualizarOculta();
}
```

(Por supuesto, también deberemos crear el Marcador en el constructor, etc).

Otros cambios menores

Hemos cambiado el tamaño del margen superior del juego. Deberemos reflejarlo en la coordenada Y inicial del personaje, del enemigo, y la posición inicial de nuestro mapa de casillas. Es un cambio muy pequeño, pero que no debemos pasar por alto.

Como en las entregas anteriores, puedes descargar la versión 0.10 completa en la página del proyecto en Google Code.

11.- Añadir dos niveles más

Novedades en esta versión

- Añadir dos niveles más
- Abrir la puerta que permite pasar de un nivel a otro cuando se recogen todas las llaves
- Pasar de un nivel a otro pulsando T+N (truco)
- Otros cambios menores

Vamos con ello...

Añadir dos niveles más

Vamos a ampliar un poco la estructura del juego: ahora el Mapa ya no tendrá definidos los datos de un único nivel, sino que crearemos una nueva clase "Nivel" y el mapa contendrá un array de niveles.

El juego que imitamos tenía 20 niveles. Para los detalles de cada nivel concreto, no crearemos un nuevo objeto de la clase Nivel, que sería suficiente para este juego, sino que tomaremos la alternativa "avanzada", que es más trabajosa pero también mucho más versátil: crearemos una clase "Nivel01", otra "Nivel02" y así sucesivamente, lo que nos permitiría poder llegar a conseguir que la "lógica" de un

nivel fuera totalmente distinta de la de otro (hasta el extremo de que un nivel podría ser de vista lateral, y otro de vista superiior, y otro en 3D...)

En nuestro caso "real", los niveles se diferenciarán unos de otros (por ahora) sólo por el contenido del array que lo describe, y por su nombre. Por ejemplo, el segundo nivel podría ser así:

```
public class Nivel02 : Nivel
{
    public Nivel02()
    {
        nombre = "The Cold Room";
        datosNivelIniciales[ 0] = "L                  LLLLLLLLLLLL";
        datosNivelIniciales[ 1] = "L      V                V     TL";
        datosNivelIniciales[ 2] = "L                             L";
        datosNivelIniciales[ 3] = "L                    FFFF     L";
        datosNivelIniciales[ 4] = "L                             L";
        datosNivelIniciales[ 5] = "LSSSSSSSSSSSSSSSSSS        L  L";
        datosNivelIniciales[ 6] = "L                    SSSSLFFL  L";
        datosNivelIniciales[ 7] = "LSFFFFF                  LV L  L";
        datosNivelIniciales[ 8] = "L                        LFFL  L";
        datosNivelIniciales[ 9] = "L  V     SSSSSSS         LFFL  L";
        datosNivelIniciales[10] = "L                  FFFF  LFFL  L";
        datosNivelIniciales[11] = "L  DDDD                  LFFL  L";
        datosNivelIniciales[12] = "L             SSSS V     LFFL  L";
        datosNivelIniciales[13] = "L       FFFF                PPL";
        datosNivelIniciales[14] = "L                           PPL";
        datosNivelIniciales[15] = "LSSSSSSSSSSSSSSSSSSSSSSSSSSSSL";

        Reiniciar();
    }
} /* fin de la clase Nivel02 */
```

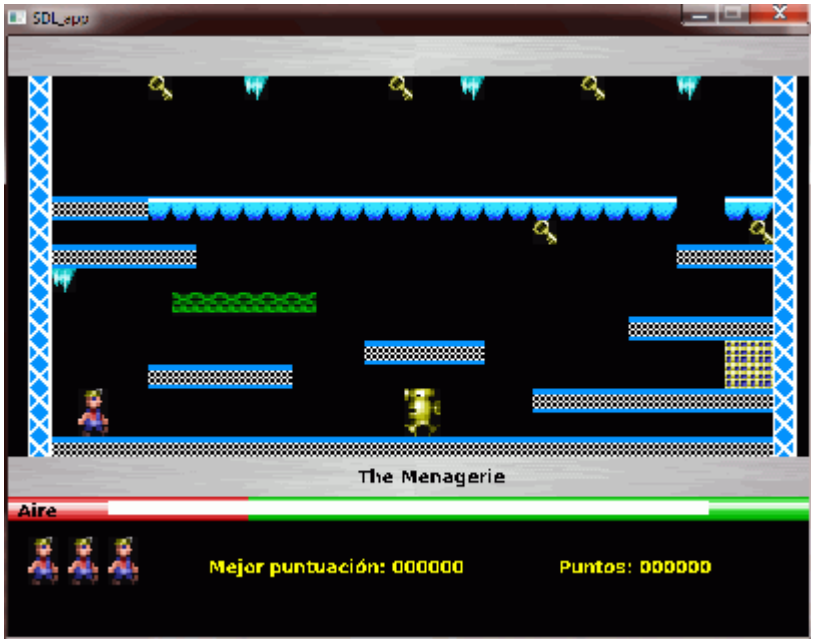
Y se vería así:

Y el tercer nivel así:

```
public class Nivel03 : Nivel
```

```
{
    public Nivel03()
    {
        nombre = "The Menagerie";
        datosNivelIniciales[ 0] = "M    V   T     V  T    V   T   M";
        datosNivelIniciales[ 1] = "M                                M";
        datosNivelIniciales[ 2] = "M                                M";
        datosNivelIniciales[ 3] = "M                                M";
        datosNivelIniciales[ 4] = "M                                M";
        datosNivelIniciales[ 5] = "MNNNNOOOOOOOOOOOOOOOOOOOOOO  OOM";
        datosNivelIniciales[ 6] = "M                      V         VM";
        datosNivelIniciales[ 7] = "MNNNNNNN                     NNNNM";
        datosNivelIniciales[ 8] = "MT                               M";
        datosNivelIniciales[ 9] = "M     DDDDDD                     M";
        datosNivelIniciales[10] = "M                           NNNNNNM";
        datosNivelIniciales[11] = "M             NNNNN           PPM";
        datosNivelIniciales[12] = "M    NNNNNN                   PPM";
        datosNivelIniciales[13] = "M                       NNNNNNNNNM";
        datosNivelIniciales[14] = "M                                M";
        datosNivelIniciales[15] = "MNNNNNNNNNNNNNNNNNNNNNNNNNNNNNM";

        Reiniciar();
    }
} /* fin de la clase Nivel03 */
```

Que se vería:



Ahora la nueva clase "Nivel" sería la que tendría la lógica común a todos los niveles. Básicamente contendrá todo lo que antes estaba en "Mapa", junto con un nuevo "GetNombre" que permita saber el nombre del nivel actual:

```
public class Nivel
{

    // Datos del mapa del nivel actual
    Partida miPartida;
```

```
 private int altoMapa = 16, anchoMapa = 32;
 private int anchoTile = 24, altoTile = 24;
 private int margenIzqdo = 20, margenSuperior = 40;
 private int llavesRestantes = 4;   // Valor arbitrario (no 0: final de partida)

 ElemGrafico arbol, deslizante, ladrillo, ladrilloX, llave, puerta,
     sueloFino, sueloFragil, sueloFragil2, sueloGrueso, techo;


 protected string nombre = "(Nonamed)";

 string[] datosNivel;  // Datos en el momento de juego

 protected string[] datosNivelIniciales =  // Datos para reiniciar
 {"LLLLLLLLLLLLLLLLLLLLLLLLLLLLLL",
  "L                             L",
  "L                             L",
  "L                             L",
  "L                             L",
  "L                             L",
  "L                             L",
  "L                             L",
  "L                             L",
  "L                             L",
  "L                             L",
  "L                             L",
  "L                           PPL",
  "L                           PPL",
  "LSSSSSSSSSSSSSSSSSSSSSSSSSSSSL"};

 // Constructor
 public Nivel()
 {
     //miPartida = p;   // Para enlazar con el resto de componentes

     arbol = new ElemGrafico("imagenes/arbol.png");
     deslizante = new ElemGrafico("imagenes/deslizante.png");
     ladrillo = new ElemGrafico("imagenes/ladrillo.png");
     ladrilloX = new ElemGrafico("imagenes/ladrillo2.png");
     llave = new ElemGrafico("imagenes/llave.png");
     puerta = new ElemGrafico("imagenes/puerta.png");
     sueloFino = new ElemGrafico("imagenes/suelo.png");
     sueloGrueso = new ElemGrafico("imagenes/suelo2.png");
     sueloFragil = new ElemGrafico("imagenes/sueloFragil.png");
     sueloFragil2 = new ElemGrafico("imagenes/sueloFragil2.png");
     techo = new ElemGrafico("imagenes/techo.png");

     datosNivel = new string[altoMapa];
     Reiniciar();
}

 public void Reiniciar()
 {
     for (int fila = 0; fila < altoMapa; fila++)
         datosNivel[fila] = datosNivelIniciales[fila];
 }

 public void DibujarOculta()
 {
    // Dibujo el fondo
```

```
    llavesRestantes = 0;
    for (int fila = 0; fila < altoMapa; fila++)
        for (int colum = 0; colum < anchoMapa; colum++)
        {
            int posX = colum * anchoTile + margenIzqdo;
            int posY = fila * altoTile + margenSuperior;
            switch (datosNivel[fila][colum])
            {
                case 'A': arbol.DibujarOculta(posX, posY); break;
                case 'D': deslizante.DibujarOculta(posX, posY); break;
                case 'F': sueloFragil.DibujarOculta(posX, posY); break;
                case 'L': ladrillo.DibujarOculta(posX, posY); break;
                case 'M': ladrilloX.DibujarOculta(posX, posY); break;
                case 'N': sueloGrueso.DibujarOculta(posX, posY); break;
                case 'O': sueloFragil2.DibujarOculta(posX, posY); break;
                case 'P': puerta.DibujarOculta(posX, posY); break;
                case 'S': sueloFino.DibujarOculta(posX, posY); break;
                case 'T': techo.DibujarOculta(posX, posY); break;
                case 'V': llave.DibujarOculta(posX, posY);
                          llavesRestantes++;  break;
            }
        }
}

public bool EsPosibleMover(int x, int y, int xmax, int ymax)
{

    // Compruebo si choca con alguna casilla del fondo
    for (int fila = 0; fila < altoMapa; fila++)
           for (int colum = 0; colum < anchoMapa; colum++)
           {
               int posX = colum * anchoTile + margenIzqdo;
               int posY = fila * altoTile + margenSuperior;
               // Si se solapa con la posic a la que queremos mover
               if ((posX+anchoTile > x) && (posY+altoTile > y)
                   && (xmax> posX) && (ymax > posY))
                   // Y no es espacio blanco, llave, puerta o arbol
                   if ((datosNivel[fila][colum] != ' ')
                       && (datosNivel[fila][colum] != 'V')
                       && (datosNivel[fila][colum] != 'P')
                       && (datosNivel[fila][colum] != 'A'))
                   {
                       return false;
                   }
            }

    return true;
}

public int ObtenerPuntosPosicion(int x, int y, int xmax, int ymax)
{

    // Compruebo si choca con alguna casilla del fondo
    for (int fila = 0; fila < altoMapa; fila++)
        for (int colum = 0; colum < anchoMapa; colum++)
        {
            int posX = colum * anchoTile + margenIzqdo;
            int posY = fila * altoTile + margenSuperior;

            // Si choca con la casilla que estoy mirando
            if ( (posX + anchoTile > x) && (posY + altoTile > y)
```

```
                    && (xmax > posX) && (ymax > posY))
                {
                    // Si choca con el techo o con un arbol
                    if ((datosNivel[fila][colum] == 'T')
                            || (datosNivel[fila][colum] == 'A'))
                        return -1; // (puntuacion -1: perder vida)

                    // Si toca una llave
                    if (datosNivel[fila][colum] == 'V')
                    {
                        // datosNivel[fila][colum] = ' '; (No valido en C#: 2 pasos)
                        datosNivel[fila] = datosNivel[fila].Remove(colum, 1);
                        datosNivel[fila] = datosNivel[fila].Insert(colum, " ");
                        return 10;
                    }

                    // Si toca la puerta y no quedan llaves, 50 puntos
                    // y (pronto) pasar al nivel siguiente
                    if ((datosNivel[fila][colum] == 'P')
                        && (llavesRestantes == 0))
                    {
                        return 50;
                    }

                } // Fin de comprobacion de casilla actual
            } // Fin de "for" que revisa todas las casillas

        // Si llego hasta aqui, es que no hay puntos que devolver
        return 0;

    }

    public string LeerNombre()
    {
        return nombre;
    }

} /* fin de la clase Mapa */
```

Y la clase "Mapa" ya no hace casi nada por ella misma: se limita a crear un array de niveles, y a preguntar al nivel actual cada dato que se le pida desde el cuerpo del programa, y también tendrá un método "Avanzar" que pase al siguiente nivel:

```
public class Mapa
{
    Nivel nivelActual;
    Nivel[] listaNiveles;
    const int MAX_NIVELES = 3;
    int numeroNivelActual = 0;
    Fuente fuente18;   // Tipo de letra para mensajes

    // Constructor
    public Mapa(Partida p)
    {
        listaNiveles = new Nivel[MAX_NIVELES];
        listaNiveles[0] = new Nivel01();
        listaNiveles[1] = new Nivel02();
        listaNiveles[2] = new Nivel03();
```

```
        nivelActual = listaNiveles[ numeroNivelActual ];
        fuente18 = new Fuente("FreeSansBold.ttf", 18);
    }

    public void Reiniciar()
    {
        nivelActual.Reiniciar();
    }

    public void DibujarOculta()
    {
        nivelActual.DibujarOculta();
    }

    public bool EsPosibleMover(int x, int y, int xmax, int ymax)
    {
        return nivelActual.EsPosibleMover(x, y, xmax, ymax);
    }

    public int ObtenerPuntosPosicion(int x, int y, int xmax, int ymax)
    {
        return nivelActual.ObtenerPuntosPosicion(x, y, xmax, ymax);
    }

    public void Avanzar()
    {
        numeroNivelActual++;
        if (numeroNivelActual >= MAX_NIVELES)
            numeroNivelActual = 0;

        // Rectángulo de fondo
        Hardware.RectanguloRellenoRGBA(
           200, 100, 600, 300,  // Posicion, ancho y alto de la pantalla
           200, 200, 200,   // Gris claro
           200);         // Con algo de transparencia

        // Y texto de aviso
        Hardware.EscribirTextoOculta(
                "Pasando al nivel "+(numeroNivelActual+1),
                300, 200, 0, 0, 0, fuente18);

        Hardware.VisualizarOculta();
        Hardware.Pausa(2000);

        nivelActual = listaNiveles[numeroNivelActual];
        nivelActual.Reiniciar();
    }

    public string GetNombre()
    {
        return nivelActual.LeerNombre();
    }

} /* fin de la clase Mapa */
```

Abrir la puerta que permite pasar de un nivel a otro cuando se recogen todas las llaves

Ya teníamos la infraestructura preparada: desde la función de "ComprobarColisiones" de la "Partida" ya mirábamos si se habían recogido todas las llaves y se tocaba la puerta, pero nos limitábamos a dar 50 puntos al jugador. Ahora también avisaremos de que hay que avanzar al siguiente nivel:

```
// Si ademas es una puerta, avanzamos de nivel
if (puntosMovimiento == 50)
    miPantallaJuego.Avanzar();
```

Pasar de un nivel a otro pulsando T+N (truco)

Sencillo: una vez que tenemos preparada la orden que nos permite avanzar de nivel, basta comprobar si se pulsa cierta combinación de teclas para activarla:

```
if ( (Hardware.TeclaPulsada(Hardware.TECLA_T)) &&
    (Hardware.TeclaPulsada(Hardware.TECLA_N)) )
  miPantallaJuego.Avanzar();
```

Otros cambios menores

No necesitamos casi nada más, pero ya que hemos añadido un "nombre" a cada nivel, podemos ampliar un poco la clase Marcador para que lo muestre en pantalla.

Como siempre, puedes encontrar la versión 0.11 completa en la página del proyecto en Google Code.

12.- Personaje y enemigo animados

- Un enemigo animado
- Y un personaje animado

Vamos allá...

Un enemigo animado

No es nada difícil: ya está previsto en la clase "ElemGrafico": basta con cargar una secuencia de imágenes en vez de una única imagen, y llamar a "SiguienteFotograma()" cuando queramos que se cambien de un fotograma de la animación al siguiente.

En nuestro caso, cargaremos dos secuencias: una para cuando el enemigo se mueva hacia la derecha y otra para cuando se mueva hacia la izquierda. Indicaremos en qué dirección queremos que se mueva inicialmente, y volveremos a llamar a "CambiarDireccion" cada vez que tenga que dar la vuelta.

El fuente completo quedaría así

```
public class Enemigo : ElemGrafico
{

  // Datos del enemigo
  Partida miPartida;

  // Constructor
  public Enemigo(Partida p)
  {
    miPartida = p;    // Para enlazar con el resto de componentes
    MoverA(400,352);  // Resto de valores iniciales
    SetVelocidad(4,0);
    CargarSecuencia(DERECHA,
                new string[] {"imagenes/enemigoN01D01.png",
                          "imagenes/enemigoN01D02.png",
```

```
                        "imagenes/enemigoN01D03.png",
                        "imagenes/enemigoN01D04.png",
                        "imagenes/enemigoN01D05.png",
                        "imagenes/enemigoN01D06.png",
                        "imagenes/enemigoN01D07.png",
                        "imagenes/enemigoN01D08.png"});
    CargarSecuencia(IZQUIERDA,
                new string[] {"imagenes/enemigoN01I01.png",
                        "imagenes/enemigoN01I02.png",
                        "imagenes/enemigoN01I03.png",
                        "imagenes/enemigoN01I04.png",
                        "imagenes/enemigoN01I05.png",
                        "imagenes/enemigoN01I06.png",
                        "imagenes/enemigoN01I07.png",
                        "imagenes/enemigoN01I08.png"});
    direccion = DERECHA;
    SetAnchoAlto(36, 48);
  }


  // Métodos de movimiento
  public new void Mover()
  {
      x += incrX;
      SiguienteFotograma();

      if ((x < 100) || (x > 700))
      {
          incrX = (short)(-incrX);
          if (incrX < 0)
              CambiarDireccion(IZQUIERDA);
          else
              CambiarDireccion(DERECHA);
      }
  }

} /* fin de la clase Enemigo */
```

Y un personaje animado

La idea es exactamente la misma: cargaremos una secuencia de imágenes en el constructor y cambiaremos de fotograma cuando corresponda, en los métodos encargados de mover (en el caso del personaje: MoverDerecha, MoverIzquierda y Mover).

Sólo dos detalles más:

- Si vemos que el personaje (o cualquier otro elemento) se mueve demasiado rápido, se puede usar una solución sencilla: cargar cada imagen dos veces, de modo que el efecto visual será que se cambia de fotograma con menos frecuencia (realmente no ocurre eso, sino que se dibuja cada imagen durante 2 fotogramas).
- Por una limitación (todavía) de la clase Hardware, si cargamos una secuencia, deberemos indicar después el ancho y el alto, o se quedará con los valores por defecto (32), de modo que quizá se calculen incorrectamente las colisiones.

Los cambios son:

```
public class Personaje : ElemGrafico
{
```

```
// Constructor
public Personaje(Partida p)
{
  [...]
  CargarSecuencia(DERECHA,
          new string[] {"imagenes/personajeD01.png", "imagenes/personajeD01.png",
                        "imagenes/personajeD02.png", "imagenes/personajeD02.png",
                        "imagenes/personajeD03.png", "imagenes/personajeD03.png",
                        "imagenes/personajeD04.png", "imagenes/personajeD04.png",
                        "imagenes/personajeD05.png", "imagenes/personajeD05.png",
                        "imagenes/personajeD06.png", "imagenes/personajeD06.png",
                        "imagenes/personajeD07.png", "imagenes/personajeD07.png",
                        "imagenes/personajeD08.png", "imagenes/personajeD08.png"});
  [...]
  direccion = DERECHA;
  SetAnchoAlto(30, 48); // Si se carga secuencia, ancho y alto deben ir detras
}


// Métodos de movimiento
public void MoverDerecha()
{
    if (saltando || cayendo) return; // No debe moverse mientras salta

    CambiarDireccion(DERECHA);
    if (miPartida.GetMapa().EsPosibleMover(x + incrX, y + alto - 4,
              x + ancho + incrX, y + alto))
    {
        x += incrX;
        SiguienteFotograma();
    }
    cayendo = true;
}
```

Y ya sabes: la versión 0.12 completa está en la página del proyecto en Google Code.