Práctica

Metodología de la Programación II

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

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

Incluye la descripción de la práctica obligatoria propuesta para la asignatura. El documento pdf así como todo el material necesario se encuentran disponibles en la página web de la asignatura.

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

Índice de contenido

1. Introducción

El objetivo de esta práctica es:

1. Resolver un problema en el que es necesaria la *modularización*. La resolución de esta práctica obliga a la creación de diversos archivos, su compilación y enlace para obtener ejecutables, así como el empleo de la orden `make` y ficheros `makefile`.
2. Familiarizar al alumno con la *abstracción* y el uso de *clases* en `C++`. El diseño de la solución deberá estar basado en nuevos tipos de datos creados mediante clases.
3. Practicar con el sistema de E/S de `C++`. El programa deberá leer y escribir información en o desde archivos en disco.

Los requisitos para llevarla a cabo son:

1. Conocer las clases en `C++`. En las soluciones no es necesario incluir herencia o polimorfismo, pero sí se exigirá el uso de clases.
2. Conocer la orden `make` y los ficheros `makefile`.
3. Conocer cómo se gestiona la memoria dinámica en `C++`.
4. Conocer el sistema de E/S de `C++` (en particular, el manejo de ficheros).
5. Manejar correctamente el módulo de gráficos que se ha facilitado en la asignatura.

El alumno de primer curso debe comenzar el desarrollo de la práctica después de estudiar el tema de abstracción y encapsulamiento con clases C++. Este aspecto es básico puesto que afecta de forma directa al diseño inicial de la solución.

El tema de E/S aparece en el temario más tarde, y por tanto el alumno no puede desarrollar esa parte. Sin embargo, puede ser obviado inicialmente, ya que gran parte del desarrollo se puede realizar de forma independiente.

Básicamente, la práctica que se propone consiste en desarrollar un programa para jugar al *TETRIS*. Este juego es ampliamente conocido, por lo que no es necesario describir los detalles de cómo funciona, aunque tendremos que exponer y aclarar las características de la solución deseada. Para superar la parte práctica de la asignatura, el alumno deberá realizar correctamente este programa.

Adicionalmente, se proponen una serie de mejoras o módulos adicionales que se podrán realizar de forma opcional. El objetivo de esta parte es ofrecer al alumno la posibilidad de mejorar la calificación en la parte práctica.

2. Programa a realizar: TETRIS

El juego se representa en una ventana (ver figura 1) que dividiremos en 3 partes:

1. *Acumulador o tablero*: donde se van apilando las piezas. En la imagen aparece en la parte izquierda con fondo de color blanco, con una pieza amarilla de forma cuadrada que está cayendo hacia la parte inferior, donde se encuentran acumuladas otras piezas.
2. *Cola*: donde aparecen, de forma ordenada, las piezas que se van a introducir en el acumulador. En la imagen aparece una cola con 4 piezas en la parte central de la ventana.
3. *Marcadores*: mensajes que informan de las puntuaciones y evolución del juego.

Una ejemplo gráfico de esta ventana se puede ver en la siguiente imagen:

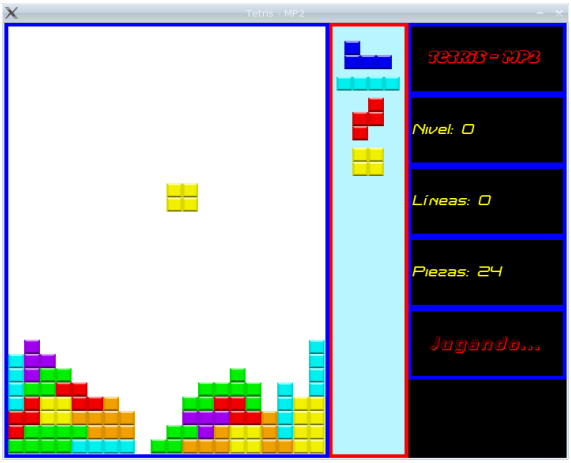

Figura 1: Juego del tetris

La dinámica del juego consiste en:

1. Mientras que el acumulador no esté “lleno”:
   a) Obtener la siguiente ficha para jugar:
      1. Se saca la primera ficha de la cola y se introduce en la columna central del acumulador.
      2. A continuación se genera una nueva ficha de manera aleatoria y se añade al final de la cola.
   b) Bajar o colocar la ficha:
      1. Mover la ficha que está descendiendo según el control del jugador.
      2. Si la ficha se ha depositado sobre las ya acumuladas:
         - Borrar las líneas completas, en caso de que existan, de forma que todas las líneas de encima bajan una posición para ocupar la línea eliminada.
         - Actualizar marcadores.

El objetivo del juego consiste en maximizar el número de líneas que se han completado. En la figura 1 se puede observar que en la fila más baja sólo se necesita rellenar un hueco para ser completada. Por ejemplo, si la ficha que está cayendo se situara en el hueco doble de la penúltima fila, provocaría un borrado de ésta y un descenso, de una posición, de todas las de encima.

2.1. El acumulador y las piezas

El acumulador es la zona donde el jugador tiene que controlar la caída de las piezas de forma que se depositen formando líneas completas. Podemos considerar que está formado por una matriz de casillas con *f* filas y c columnas.

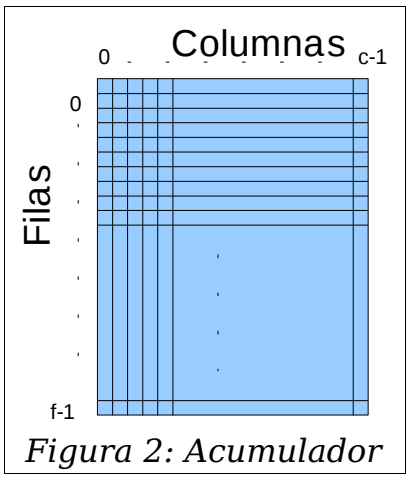

Figura 2: Acumulador

Estas casillas pueden estar libres u ocupadas, de forma que una pieza podrá descender por el acumulador mientras las posiciones en su camino de descenso estén vacías, y se quedará depositada cuando, debajo de ella, haya alguna casilla que le impida descender.

Por otro lado, las piezas se pueden considerar, de forma similar, como una matriz de casillas donde algunas posiciones están ocupadas y otras vacías.

En el juego que vamos a desarrollar el acumulador, el número de piezas y las respectivas dimensiones serán variables. Por

tanto, el programa deberá ser capaz de aceptar distintas configuraciones (véase sección 2.5).

En la figura 3, se presentan las piezas clásicas del juego. Podemos ver que hay siete piezas: una de dimensiones *1x4 (*con todas las casillas ocupadas), una de dimensiones *2x3 (*con las posiciones *0,1* y *0,2* libres), etc.

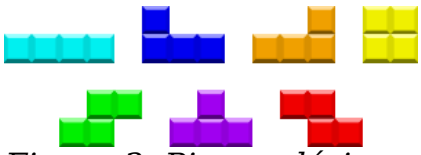

Figura 3: Piezas clásicas

Cada una de las casillas se representará como un hueco vacío o con la correspondiente imagen. En el caso de las piezas clásicas, podemos ver que cada posición se define con un pequeño bloque de un determinado color. Así, el acumulador o las piezas para este caso tendrá en cuenta 7 posibles imágenes con colores celeste, azul, naranja, amarillo, verde, violeta y rojo.

Durante una partida, el jugador tiene la opción de controlar la posición donde caerá cada una de las piezas mediante los siguientes controles:

- Tecla *flecha a la izquierda*. Si es posible, mueve la pieza que está cayendo una posición a la izquierda.
- Tecla *flecha a la derecha*. Si es posible, mueve la pieza una posición a la derecha.
- Tecla *flecha hacia abajo*. Si es posible, mueve la pieza una posición hacia abajo.
- Tecla *q*. Si es posible, rota la pieza hacia la izquierda.
- Tecla *w*. Si es posible, rota la pieza hacia la derecha.

2.2. La cola de piezas

El juego dispone de un visualizador de las piezas que están por salir. Así, en la figura 1 se presenta un ejemplo donde podemos observar una cola que contiene 4 piezas.

Cuando la pieza que está cayendo se deposite, la pieza azul se insertará en la posición central de la parte superior del acumulador, las otras 3 subirán una posición, y se seleccionará una nueva pieza aleatoria para ocupar la cuarta posición.

El programa debe admitir distintas longitudes de cola (este será un parámetro dado en la configuración del juego), aunque para una partida concreta, el número de piezas que se incluyen en la cola es constante.

2.3. Mensajes y marcadores

La última parte de la ventana del juego contiene los mensajes y marcadores. Se puede observar en la figura 1, en la parte de la derecha. Aparecen los siguientes mensajes de texto:

1. *Mensaje del título*: *"Tetris – MP2"*. Este mensaje será variable, ya que se podrá modificar según la configuración de la partida.
2. *Nivel*. Indica el nivel de juego en cada instante. El nivel inicial de una partida es el cero, pero se incrementará conforme aumente el número de líneas completadas (véase la sección 4.1).
3. *Líneas*. Indica el número de líneas completadas hasta el momento.
4. *Piezas*. Indica el número de piezas insertadas en el acumulador hasta el momento.
5. *Estado*. Indica si se está jugando, o si la partida ya ha terminado.

2.4. Márgenes, colores y tipo de letra

La ventana de juego es el resultado de presentar distintos elementos: un acumulador, una cola, y 5 visualizadores de texto. Cada uno de estos elementos se caracteriza por:

1. Un *color de fondo*. Por ejemplo, en la figura 1 el acumulador tiene un color de fondo blanco, mientras que los mensajes de texto tienen un fondo negro.
2. Un *color de borde*. Por ejemplo, en la figura indicada, el acumulador tiene un marco azul, mientras que la cola lo tiene de color rojo.
3. Unas *dimensiones*. Éstas las podemos concretar como:
   i. *Posición.* Fila y columna de la ventana donde se sitúa la esquina superior izquierda de la región.
   ii. *Marco*. Número de píxeles que determinan el grosor del borde.
   iii. *Ancho*. Número de columnas (en píxeles) de la región, incluyendo el marco.
   iv. *Alto*. Número de filas (en píxeles) de la región, incluyendo el marco.

Adicionalmente, los visualizadores de texto también se caracterizan por:

1. Un *tipo de letra*. Éste viene determinado por una fuente (archivo *ttf*), un tamaño (en píxeles), y un *estilo* (normal, negrita, itálica o negrita-itálica).
2. Un *color de letra*. Por ejemplo, en la figura 1 los marcadores numéricos se presentan de color amarillo.

Todos estos parámetros serán variables, es decir, el programa podrá aceptar distintas configuraciones, de forma que la presentación se pueda ajustar al gusto del usuario.

2.5. Ficheros de configuración

El programa final resultado de la práctica se deberá llamar desde la línea de órdenes con la siguiente sintaxis:

```
% tetris <fichero de configuración>
```

donde "*tetris*" es el nombre del archivo ejecutable, y el "*fichero de configuración*" corresponde a un archivo de texto en disco con todos los datos que componen la configuración de una partida.

Básicamente, la información que contiene un fichero de este tipo consiste en:

1. Parámetros relacionados con la dinámica del juego.
2. Parámetros relacionados con la presentación en pantalla.
3. Configuración de piezas.
4. Configuración inicial del tablero.

Además, podrá contener un número indeterminado de comentarios a fin de que sea más fácil de gestionar y mantener. Un comentario comienza con un carácter '#' y llega hasta el final de la línea.

Un ejemplo completo de configuración se puede consultar en el archivo "conf_clasico_basico.txt", disponible en el paquete asociado a la práctica.

En cualquier caso, y puesto que la parte de E/S no se estudia hasta bien avanzado el curso, para poder empezar la práctica, todos los parámetros de la configuración se podrán incluir directamente en el código fuente (asignando los valores) hasta que se disponga de los conocimientos necesarios para realizar la E/S con ficheros.

2.5.1. Parámetros de la dinámica del juego.

Un ejemplo de parámetros relacionados con la dinámica del juego es el siguiente:

```
# Semilla de generación de números aleatorios inicial (0=aleatoria)
0   # Con esto, todas las partidas serán distintas.
# Tiempo inicial de espera para bajar piezas (milisegundos)
500 # Se irá cambiando conforme se avanza de nivel
# Cada cuántas líneas se avanza de nivel
5   # afecta a la velocidad de bajada de piezas
# Número de piezas siguientes en cola (mayor que cero)
4
```

Como puede observar, hemos incluido comentarios como líneas independientes e incluso acompañando los valores a modo de aclaración. El significado de cada uno de ellos es:

- *Semilla*. Indica la semilla inicial que se introducirá en el generador de números aleatorios. En caso de que valga cero, se desea una semilla "indeterminada" que se puede obtener llamando a *time(0)* (incluida en *ctime*). Para más detalles, véase [FUN], página 182 y siguientes.
- *Tiempo inicial*. Establece el tiempo que tarda una pieza en bajar una casilla.
- *Líneas para subir de nivel*. Cada vez que se completa este número de líneas, el nivel aumenta en una unidad (véase sección 2.5 para más detalles).
- Número de piezas en cola. Es sólo un valor entero que indica el número de piezas que se visualizan en la cola, es decir, el tamaño de la cola.

2.5.2. Parámetros de la presentación en pantalla

En esta parte aparecen todos los datos que definen los detalles del contenido de la ventana. Los datos, según el orden de aparición, son los siguientes:

- *Imágenes del acumulador*. Para poder dibujar la parte izquierda, tanto casillas como piezas, es necesario especificar cada una de los bloques (recuerde la figura 3) mediante su nombre de archivo de imagen. Un ejemplo es:

  ```
  # Número de imágenes
  7
  # Directorio con imágenes
  data/piezas1
  # Ficheros con imágenes (tantas como las indicadas antes)
  rojo.bmp
  verde.bmp
  azul.bmp
  naranja.bmp
  amarillo.bmp
  violeta.bmp
  cyan.bmp
  ```

  Como puede ver, se indica el número de imágenes, el directorio donde se encuentran, y cada nombre. En las secciones 2.5.3 y 2.5.4 veremos cómo se establece la relación con el tablero y cada una de las piezas.

- *Posición y dimensiones del tablero*. Un ejemplo de estos parámetros es:

  ```
  # Posición en la ventana: Fila, Columna
  0 0
  # Ancho, Alto, marco. El tablero se dibuja centrado.
  410 610 5 # El dibujo dispone de un ancho de 400 y alto de 600
  # Color de borde y fondo
  0 0 255
  255 255 255
  ```

  Donde las posiciones y dimensiones se expresan en píxeles y los colores como valores

RGB (véase [GRF] para más detalles sobre el modelo *RGB*).

- *Posición y dimensiones de la cola*. Un ejemplo de estos parámetros es:

```
# Posición en la ventana: Fila, Columna
0 410
# Ancho, Alto, marco
100 610 5
# Color de borde y fondo
255 0 0
183 246 255
```

  Como podemos observar, los parámetros son similares a los anteriores del tablero.

- *Mensaje del título*. Corresponde a la cadena de caracteres que se presentará como título (véase sección 2.3). Un ejemplo es:

```
# mensaje del título ------------------------------------------------------
Tetris – MP2
```

  Observe que corresponde a la primera línea "completa" que no es un comentario.

- *Visualizadores de texto*. A continuación se presentan los parámetros para todos los mensajes de texto (incluido el título). Por ejemplo, los parámetros para el visualizador del título pueden ser los siguientes:

```
 # Configuración Visualizador: Título ----------------------------------------
# Tipo de letra del título, tamaño, estilo
# Estilo: 0 normal, 1 negrita, 2 italica, 3 negrita-italica
data/fuentes/Disko.ttf
25
0
# Color letra
255 0 0
# Posición en la ventana: Fila, Columna
0 510
# Ancho, Alto y marco
200 100 5
# Color de borde y fondo
0 0 255
0 0 0
```

  donde podemos ver que se usará la fuente "*Disko.ttf*", con un tamaño 25, con estilo "*normal*", y en color rojo. El resto de parámetros definen el marco donde se dibuja, con un esquema idéntico a los apartados anteriores.

  Adicionalmente, se definen los visualizadores de *nivel, líneas, piezas* y *estado* con una estructura idéntica al de *título*.

2.5.3. Parámetros de configuración de piezas

En esta parte se detallan las piezas que podrán aparecer en el juego. Para ello, tenga en cuenta que una pieza se puede ver como una matriz de bloques o imágenes (véase sección 2.1) y que el número de imágenes, así como los archivos donde se encuentran, se han especificado en la parte anterior del archivo de configuración.

El número de piezas y su definición aparecen, por ejemplo, de la siguiente forma:

```
# Número de piezas (incluidas repetidas).
# Se pueden repetir para cambiar probabilidad de aparición
7
# Pieza 1 (filas columnas) seguidas de matriz indicando imagen (cero es vacío)
1 4
7 7 7 7
# Pieza 2 (filas columnas) seguidas de matriz indicando imagen (cero es vacío)
2 3
```

```
3 0 0
3 3 3
# Pieza 3 (filas columnas) seguidas de matriz indicando imagen (cero es vacío)
2 3
0 0 4
4 4 4
# Pieza 4 (filas columnas) seguidas de matriz indicando imagen (cero es vacío)
2 2
5 5
5 5
# Pieza 5 (filas columnas) seguidas de matriz indicando imagen (cero es vacío)
2 3
0 2 2
2 2 0
# Pieza 6 (filas columnas) seguidas de matriz indicando imagen (cero es vacío)
2 3
0 6 0
6 6 6
# Pieza 7 (filas columnas) seguidas de matriz indicando imagen (cero es vacío)
2 3
1 1 0
0 1 1
```

Observe que cada una de ellas se define como una matriz de enteros. Cada entero indica una imagen (desde la 1 a la 7), o con un cero para indicar vacío. Así, estos valores se corresponden con las piezas que se presentan en la figura 3.

Por otro lado, tenga en cuenta que las piezas se generarán en el juego con igual probabilidad, es decir, cada vez que haya que introducir una nueva pieza en la cola, deberemos escoger una de ellas con una probabilidad de *1/n*, siendo *n* el número de piezas.

2.5.4. Parámetros de configuración del acumulador

Finalmente, también tenemos que indicar la forma del acumulador. En este caso, no sólo indicamos sus dimensiones, sino también la configuración inicial. Esta inicialización nos permite crear configuraciones donde el tablero pueda contener, inicialmente, algunas casillas ocupadas. Un ejemplo es:

```
# Tablero (filas columnas) seguidas de matriz indicando imagen (cero es vacío)
20 15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Como puede observar, también se especifica como una matriz de enteros, indicando cada uno de los bloques o el vacío (con un cero).

3. Ampliaciones opcionales

Una vez que se ha presentado la práctica obligatoria, presentamos una serie de sugerencias que, de forma opcional, el alumno puede incluir a fin de mejorar su calificación.

Por otro lado, resulta conveniente enfatizar la importancia de un buen diseño a la hora de resolver la parte obligatoria. Así, el nivel de dificultad para incluir la mayoría de las ampliaciones está relacionado, en gran medida, con la calidad de dicha solución. Recuerde que un buen diseño facilita el mantenimiento del software. Además, contemplar posibles modificaciones en un programa, puede sugerir opciones de diseño que mejorarán la solución final.

3.1. Disponer de “respiros”

En este juego no tiene sentido disponer de “*vidas*” o “*créditos*”. Sin embargo, podemos añadir una característica que permita disponer de lo que llamamos “*respiros*”. Un “*respiro*” consiste en que el jugador puede usar una tecla – por ejemplo, la *flecha arriba*- para que el programa eleve la ficha que está cayendo hasta la primera fila, a fin de que vuelva a caer de nuevo desde esa posición.

Para hacerlo más interesante, podemos hacer que el movimiento de la ficha se realice poco a poco, es decir, moviéndose a lo largo de todas las filas hacia arriba, de una en una. Así, si existe una casilla ocupada en el trayecto hasta la primera fila, la ficha llegará sólo hasta esa posición.

Añadir esta opción implica que el fichero de configuración incluye también parámetros asociados con los “*respiros*”. Concretamente:

1. Un número de “*respiros*” inicial. Mientras el número sea mayor que cero, el jugador podrá usar la tecla asociada para elevar la ficha. Este número se puede incluir en la primera parte del archivo de configuración -dinámica del juego-, antes del tamaño de la cola.
2. Un visualizador que informa del número de respiros disponibles. Por ejemplo, se puede configurar antes del visualizador de “*estado*”.

Un ejemplo completo de configuración con estas modificaciones se puede consultar en el archivo “*conf_clasico_respiros.txt*”, disponible en el paquete asociado a la práctica.

3.2. Introducir líneas aleatorias

Para crear mayor dificultad, podemos hacer que el programa inserte líneas aleatorias desde abajo. La idea consiste en que, a partir de cierto momento en la partida, el programa mueva todas las líneas una posición hacia arriba y rellene la última línea con huecos y bloques.

La activación e introducción de líneas se controla por medio del número de líneas completadas hasta ese momento. Concretamente, podemos incluir dos parámetros al final de la primera parte -dinámica del juego- del archivo de configuración, como sigue:

```
# Número de líneas terminadas para activar introducción de líneas
10
# Cada cuántas líneas terminadas se introduce una nueva
5
```

De forma que proporcionamos dos números, el número de líneas completadas para que se active y cada cuántas se introduce una nueva. Así, en el ejemplo anterior, la primera línea se introducirá cuando se alcancen 15 líneas completadas.

Además, podemos hacer que el mensaje que aparece en el visualizador de estado cambie en el momento en que se activa la introducción de líneas. Por ejemplo, con los parámetros anteriores, podemos hacer que el estado refleje un mensaje “*¡Líneas!*” cuando llegamos a 10

líneas completadas.

Un ejemplo completo de configuración con estas modificaciones se puede consultar en el archivo "*conf_clasico_lineas.txt*", disponible en el paquete asociado a la práctica.

Por otro lado, si desea conocer un posible algoritmo para crear una línea aleatoria, consulte la sección 4.2.

3.3. Piezas animadas

Una mejora visual del juego consiste en presentar un tablero con imágenes animadas. Básicamente, la idea es sustituir cada una de las imágenes que hemos presentado en las secciones anteriores por una animación. Por ejemplo, en la figura 4 presentamos una animación de un disco que da la vuelta para mostrar su otra cara.

Figura 4: "Imagen animada"

Para modificar el programa, en primer lugar, tenemos que modificar la primera parte de los parámetros relacionados con la presentación en pantalla, concretamente, las imágenes del acumulador. El esquema que ahora proponemos lo podemos mostrar con el siguiente ejemplo:

```
# Configuración de las imágenes que componen las piezas-----------------------
# Número de imágenes y tiempo entre frames en la animación (milisegundos)
5 60
# Directorio con imágenes
data/piezas3
# Ficheros con imágenes número inicial indica número imágenes animadas

4 img1frame1.bmp img1frame2.bmp img1frame3.bmp img1frame4.bmp
4 img2frame1.bmp img2frame2.bmp img2frame3.bmp img2frame4.bmp
4 img3frame1.bmp img3frame2.bmp img3frame3.bmp img3frame4.bmp
4 img4frame1.bmp img4frame2.bmp img4frame3.bmp img4frame4.bmp
4 img5frame1.bmp img5frame2.bmp img5frame3.bmp img5frame4.bmp
```

En primer lugar, hemos añadido un parámetro correspondiente al tiempo entre cuadros en la animación. En este caso hemos indicado 60 milisegundos. Esto significa que el programa debe cambiar a la siguiente imagen cada 60 mseg. aproximadamente. Lógicamente, cuando llegamos a la última imagen el programa debe seguir por la primera.

Por otro lado, en lugar de un único nombre de archivo, hemos modificado cada imagen por un índice -número de imágenes- seguido por cada uno de los archivos (tantos como indique ese índice). Así, en el ejemplo anterior las animaciones consisten en 4 cuadros, es decir, cada 240 mseg. aproximadamente, volverá a repetirse "el vídeo".

Tenga en cuenta que si la animación se ha creado de forma que el último cuadro continúa de forma natural con el primero (es algo cíclico), entonces el programa mostrará un movimiento continuo. Por ejemplo, en la figura 5 podemos ver cuatro secuencias cíclicas de 16 cuadros.

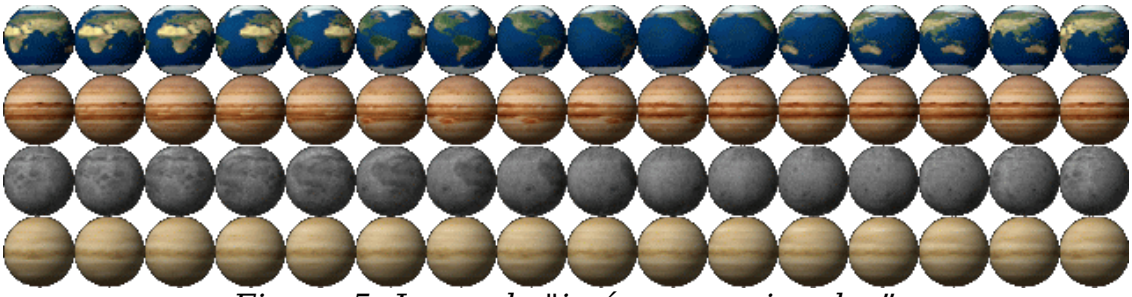

Figura 5: Juego de "imágenes animadas"

Observe que para implementar la animación tal como hemos indicado, debemos tener cada uno de los cuadros como un archivo "*bmp*" independiente. Por ejemplo, el archivo "*conf_planetas_animado.txt*" contiene la configuración correspondiente a las animaciones de la figura 5.

Por último, es interesante indicar que un programa que incluya animación puede presentar una versión estática sin más que definir las animaciones con una única imagen. Por ejemplo, el archivo "*conf_clasico_respiros_animado.txt*" visualiza la versión clásica -sin animación- pero como archivo que contiene animaciones.

3.4. Control de puntuaciones

El control de máximas puntuaciones nos permite proponer una ampliación ideal para practicar con el tema de E/S. El objetivo es controlar las puntuaciones obtenidas en todas las partidas que se han jugado.

En primer lugar, debemos tener en cuenta que las puntuaciones obtenidas dependen en gran medida del archivo de configuración usado. Por ejemplo, si fijamos un tiempo inicial de bajada de fichas muy alto, los valores obtenidos serán mucho más elevados. Por tanto, el control se realizará dependiendo del archivo que hayamos usado.

Para almacenar la información relacionada se usarán dos archivos:

1. Archivo "*puntuaciones.dat", que m*antiene una lista con las puntuaciones -número de líneas completadas- obtenidas en cada una de las partidas. Un ejemplo es el siguiente:

Nombre	**Configuración**	**Puntuación**
Pepe Tetris	1	3475
Fulano	2	225
Zutano	1	0
...	...	...

2. Archivo "*configuraciones.dat*", que mantiene una lista de todas las configuraciones que se han usado. Un ejemplo es el siguiente:

Número	**Nombre**	**Control**
1	conf_basico.txt	dk4/sd"#kddff34jkd3.
2	conf_basico_lineas.txt	leu83=1k,mkdor3.m3(/
3	conf_planetas_animado.txt	&5%dji@ldmkj$35.lki(
4	conf_basico.txt	dkrui$3#89klie4kd"|ñ
...	...	...

Observe que este segundo archivo contiene una serie de caracteres de control. El objetivo de éstos es poder distinguir entre archivos con el mismo nombre pero con distinto contenido. Por ejemplo, la línea 1 y 4 se refieren al archivo "*conf_basico.txt*", pero al tener distinto valor en el campo de "*control*", los dos archivos no tienen el mismo contenido (parámetros). Así, si conseguimos una serie de puntuaciones en un archivo, y lo modificamos, las nuevas puntuaciones se clasificarán como una ejecución distinta.

Lógicamente, nuestro programa tendrá que ser capaz de crear una secuencia de control a partir del contenido de un archivo de configuración. Presentar una buen algoritmo se sale de los objetivos de esta práctica[1], así que proponemos una operación muy simple que, aunque poco recomendable, resulta interesante como parte de esta práctica[2]. Concretamente, la secuencia de control se compondrá de 20 caracteres que codifican, a nivel de bit, un código

1 *El problema consiste en crear una buena función hash (véase [ABS], página 347 y siguientes).*

de paridad. Para ello, dividimos el archivo de entrada en *n* bloques de 20 caracteres:

Bloque	**160 bits (20 caracteres)**
1	0101011100101001 ... 00111010
2	0110000011010001 ... 00110000
...	...
n	0000010010111101 ... 00000000
Código	0101000110011100 ... 10010101

de forma que podemos obtener 160 bits de paridad a partir de cada una de las columnas que obtenemos. Una forma simple de llevar a cabo esta operación es usar la operación de "*O exclusivo*" (véase [FUN], página 259 y siguientes), que nos permite calcular el bit necesario para obtener una secuencia de paridad "*par*".

Finalmente, es necesario determinar la codificación concreta de los dos archivos propuestos:

1. Archivo "*puntuaciones.dat*". Se almacerá en formato binario con registros de longitud fija. La longitud vendrá determinada por:
   a) Un nombre, con una longitud máxima de 80 caracteres (incluido carácter cero de final de cadena).
   b) La longitud de un entero para almacenar el número de configuración.
   c) La longitud de un entero para almacenar la puntuación obtenida.
2. Archivo "*configuraciones.dat*". Se almacenará en formato binario con registros de longitud variable. Dado que el número es un entero fijo (empieza en 1 y aumenta de 1 en 1), nos ahorraremos este campo para quedarnos con:
   a) Un nombre de archivo terminado en un carácter cero de final de cadena.
   b) Un código de control con una longitud de 20 caracteres.

3.4.1. Interfaz con el programa "tetris"

La llamada se realizará desde la línea de órdenes con la siguiente sintaxis:

```
% tetris <fichero configuración> [<nombre jugador>]
```

donde el segundo parámetro es opcional y corresponde al nombre del jugador. En caso de que no aparezca, el programa asumirá un nombre de longitud cero indicando que es de identidad desconocida.

Los archivos de puntuaciones se encuentran en el mismo directorio que el archivo de configuración. Por ejemplo, si ejecutamos:

```
% tetris ../data/conf_basico.txt
```

el programa asume un nombre desconocido (nombre vacío), y que los archivos de puntuaciones son "*../data/puntuaciones.dat*" y "*../data/configuraciones.dat*". Al comenzar la partida, el programa mostrará un mensaje en la salida estándar, indicando el nombre del jugador.

Por otro lado, una vez que acaba la partida, el programa añade una nueva entrada en el archivo de puntuaciones con el valor obtenido en esa ejecución, y si es necesario, añadirá la nueva configuración en el segundo archivo.

3.4.2. Visualización de puntuaciones

La parte más interesante en el almacenamiento de puntuaciones radica en el análisis de los datos, para lo que son necesarios programas que puedan visualizar los resultados.

2 *Si está interesado en consultar una opción más práctica, puede buscar información sobre programas como md5sum, sha1sum, etc.*

Dado que disponemos de múltiples archivos de configuración, los cuales implican distinta dificultad, una forma lógica de realizar la visualización es mostrar los resultados para un fichero concreto. Para ello se propone el desarrollo de un programa para visualizar puntuaciones a partir de un archivo de configuración.

La llamada se realizará desde la línea de órdenes con la siguiente sintaxis:

```
% puntuaciones <fichero configuración>
```

que mostrará en la salida estándar todas las puntuaciones obtenidas, ordenadas desde la más alta a la más baja. Cada línea de salida contendrá un entero indicando el orden (comenzando con el 1), el nombre del jugador, y la puntuación obtenida. Además, la salida estará formateada para que los nombres comiencen en la misma columna y la puntuación se alinee a la derecha. Por ejemplo,

```
% puntuaciones ../data/conf_basico.txt
N.    Nombre                  Puntuación
----------------------------------------
1.-   Pepe Tetris               3475
2.-   Fulano                    345
3.-   Mengano                    33
4.-   Zutano                      2
```

Además, se propone que el programa acepte más de un archivo de configuración, es decir, la sintaxis:

```
% puntuaciones <f.conf1> <f.conf2> ... <f.confn>
```

donde *n* es un mayor o igual que 2. En este caso, que podría corresponder a ficheros de configuración de la misma dificultad, se pretende mostrar todas las puntuaciones de forma similar al caso anterior, pero incluyendo también el nombre de archivo. Por ejemplo,

```
% puntuaciones ../data/conf_basico.txt ../data/conf_basico2.txt
N.    Nombre                      Fichero                   Puntuación
-------------------------------------------------------------------------------
1.-   Pepe Tetris                 conf_basico.txt             3475
2.-   Menganito                   conf_basico2.txt            1243
3.-   Fulano                      conf_basico.txt              345
4.-   Mengano                     conf_basico.txt               33
5.-   Fulanito                    conf_basico2.txt              10
6.-   Zutano                      conf_basico.txt                2
```

En esta caso tenga en cuenta que los archivos de configuración se deben encontrar en el mismo directorio, ya que deben disponer de datos de puntuaciones comunes.

Finalmente, y dado que es interesante tener alguna forma simple de comparar archivos de configuración, se propone un programa que, aunque estadísticamente muy simple, nos da alguna información sobre ello. El programa tiene la siguiente sintaxis:

```
% puntuaciones <directorio configuraciones>
```

y genera una lista de todas las configuraciones, ordenadas según la media de puntuación en cada una de ellas. Concretamente, el listado tiene la siguiente forma:

```
% configuraciones ../data
N.    Fichero              N. Datos   Puntuación Media   Desviación típica
-------------------------------------------------------------------------
1.-  conf_basico.txt         4           963.75              798.43
2.-  conf_basico2.txt        2           626.5               616.5
```

Observe que en ningún caso ha habido que informar sobre el código de control de un archivo, ni ha habido que mostrar dos archivos que, teniendo el mismo nombre, tengan distintos valores en el campo de control. Esto se debe a que los resultados se darán siempre sobre archivos que realmente existan en el directorio. Por ejemplo, si hay información sobre "*conf_basico.txt*" con dos códigos de control, sólo aparecerán los del código "vigente", es decir, los datos cuyo campo de control corresponda al actual "*conf_basico.txt*".

3.4.3. Mezcla de puntuaciones

Para completar las herramientas descritas en la sección 3.4.2, vamos a desarrollar un programa para mezclar distintos archivos de configuración. La sintaxis de ejecución es:

```
% puntuaciones <directorio 1> <directorio 2> <directorio 3>
```

y realiza una mezcla de las puntuaciones que hay en los directorios 1 y 2, almacenando el resultado en el directorio 3. Concretamente, tendrá que:

1. Generar archivos "*puntuaciones.dat*" y "*configuraciones.dat*", en el directorio de salida, con los datos que hay en los dos primeros directorios.
2. Copiar todos los archivos de configuración de los directorios de entrada al directorio de salida. Si hay dos archivos con el mismo nombre, sólo aparecerá el que está situado en el segundo directorio.

4. Sugerencias de diseño

En un primer momento, un alumno con poca experiencia puede considerar que la práctica es difícil de abordar. Por ello, en esta sección vamos a mostrar algunas ideas referidas al diseño que se podría crear a fin de resolver este problema. Estas ideas deberían considerarse simplemente una sugerencia, ya que el alumno es el que finalmente debe optar por la solución que considere más adecuada.

4.1. Gestión del tiempo

El algoritmo principal del programa debería ser un algoritmo iterativo que realiza una serie de tareas de forma síncrona con el paso del tiempo. Por ello, para un programador sin experiencia puede resultar conveniente mostrar un esquema de cómo se podría implementar esa simulación.

En primer lugar, es interesante leer la documentación de [GRF], donde aparecen las funciones que nos permitirán consultar un reloj de tiempo real, y donde ya se muestran algunos algoritmos que funcionan con el paso del tiempo.

Un programa que implementa una simulación de un sistema -como por ejemplo, un juego- que muestra un avance del tiempo de forma continua, no puede garantizar el procesamiento de cada uno de los sucesos en el instante teórico de ocurrencia. Sin embargo, la velocidad de ejecución es tan alta que en la práctica el programa se comporta como si así fuera. Así, un algoritmo como el siguiente:

```
CronometroInicio();
while (!fin) {
    Procesar sucesos con tiempo <= CronometroValor()
    Mostrar estado actual del sistema
}
```

puede gestionar un sistema con avance de tiempo continuo. Observe que si el ordenador es infinitamente rápido, este algoritmo procesa el sistema en tiempo real.

Lógicamente, si el sistema permanece sin cambios durante un período de tiempo bastante alto, este programa desperdiciaría una gran cantidad de CPU, ya que se dedicaría continuamente a comprobar que no hay nada que hacer. En este sentido, se puede optar por un esquema como el siguiente:

```
CronometroInicio();
while (!fin) {
    Esperar(5); // Por ejemplo
    Procesar sucesos con tiempo <= CronometroValor()
    Mostrar estado actual del sistema
}
```

En este caso, lo que hemos hecho es modificar el comportamiento de nuestro sistema, ya que los sucesos los hemos trasladado a instantes concretos de tiempo. Por ejemplo, si mostramos el estado del sistema en el momento "*1 segundo*", no volveremos a procesar ningún cambio

antes del tiempo "*1.005 segundos*". Si el intervalo fijado -en este caso 5- es suficientemente pequeño, el comportamiento es prácticamente igual al caso anterior.

4.1.1. Tiempo y nivel de dificultad

Cuando avanza la partida, el juego avanza también de "nivel". La velocidad de caída de las fichas debe incrementarse en este caso. La forma obvia de realizarlo es hacer que el sistema programe un nuevo suceso "bajar posición" con una diferencia de tiempo cada vez más pequeña.

El tiempo entre bajadas debe ser función del parámetro de "*tiempo de bajada inicial*" (véase sección 2.5.1, en la página 7), así como del nivel de dificultad. Se puede crear una función decreciente con estos parámetros. Por ejemplo,

$$\frac{t_{bajada}}{1+\frac{nivel}{4}}$$

Lógicamente, se puede optar por otras funciones, siempre que sean decrecientes y permitan que el tiempo vaya disminuyendo de una forma razonable.

4.2. Introducción de una línea

Como hemos propuesto anteriormente, el alumno tiene la opción de añadir una dificultad adicional al juego por medio de la introducción de líneas aleatorias cada cierto tiempo. Ahora bien, ¿qué algoritmo podemos usar para rellenar una línea?

Tenemos múltiples opciones, aunque podemos crear un algoritmo muy simple que garantiza que se crea una línea válida: primero creamos la línea vacía, y después generamos *c/2* números aleatorios – es decir, columnas- que rellenamos como casilla ocupada. Observe que el nivel de ocupación de la línea garantiza que está por completar, y dependerá de los números generados.

4.3. Modularización del problema

Aunque el alumno es libre de implementar los módulos que considere necesarios, en este apartado se sugieren algunas ideas sobre posible clases que se pueden desarrollar. Independientemente de que el alumno siga o no estas sugerencias, la solución final deberá estar basada en una correcta modularización del problema.

A título orientativo, éstas podrían ser algunas de las clases que se pueden implementar:

- Clase *Pieza*. Permite gestionar las piezas individuales del juego. Almacena la configuración de las piezas y permitirá realizar acciones sobre ellas tales como construirlas, rotarlas, consultar sus dimensiones, etc.
- Clase *Tablero*. Permite gestionar el tablero de juego o acumulador. Sobre ella se realizarán operaciones como por ejemplo, consultar dimensiones, ver si una pieza encaja o no, añadirle una línea aleatoria, borrar una línea completa, comprobar si hay líneas completas y cuántas hay, etc.
- Clase *Imagen*. Para almacenar los bitmaps. Permitirá cargarlos desde un fichero en disco, dibujarlos en pantalla, etc.
- Clase *Cola de piezas*. Para disponer de la cola con las piezas que irán colocándose en el acumulador. Dispondrá de operaciones tales como obtener la siguiente pieza de la cola, consultar cual es la pieza n-ésima, añadir una nueva pieza al final de la cola, etc.

Aunque estas son algunas de las clases que se nos pueden ocurrir de forma más o menos directa a partir del enunciado del problema, debemos tener en cuenta que un buen diseño podría incluir otras clases. Por ejemplo, desde el punto de vista de su representación interna, tanto un tablero como una pieza contienen una estructura de datos de tipo matricial, por lo que se podría pensar en tener un módulo genérico que permita trabajar con matrices y que

sea utilizado por ambas clases. Obviamente, aunque su representación interna pueda coincidir en mayor o menor medida, lo que las diferenciará será su interfaz. De esta forma, dos clases pueden ser muy distintas desde un punto de la abstracción, aunque su representación interna sea muy similar o incluso igual.

5. Normas de elaboración y entrega

5.1. Estructura de ficheros y directorios

La solución deberá contener, al menos, los siguientes ficheros y directorios:

```
.
|-- Makefile                Makefile para la generación del proyecto completo (graficos y tetris)
|-- graficos                Carpeta correspondiente al módulo de gestión gráfica
|   |-- Makefile            Makefile para crear la biblioteca gráfica
|   |-- bin                 Ejecutables de prueba para el módulo gráfico
|   |-- data
|   |   `-- fuentes         Fuentes True Type
|   |-- doc                 Documentación del módulo
|   |   |-- doxys           Ficheros auxiliares para doxygen
|   |-- include             Ficheros de cabecera (ficheros .h)
|   |-- lib                 Carpeta con la biblioteca gráfica (ficheros libXXX.a)
|   |-- obj                 Ficheros objeto (ficheros .o)
|   `-- src                 Código fuente (ficheros.cpp)
|
`-- tetris
    |-- Makefile            Fichero makefile para generación de los ejecutables del tetris
    |-- bin                 Ejecutables
    |-- data                Ficheros de datos adicionales
    |   `-- fuentes         Tipos de letra de ejemplo (incluye ficheros .ttf)
    |-- doc                 Documentación
    |   `-- Doxyfile
    |-- include             Ficheros de cabecera (ficheros .h)
    |-- lib                 Bibliotecas (ficheros libXXX.a)
    |-- obj                 Código objeto (ficheros .o)
    |-- src                 Código fuente (ficheros .cpp)
    `-- LEEME               Breve descripción para compilar y ejecutar el programa.
```

El fichero Makefile, que está en el nivel superior, debe permitir que se generen tanto el módulo de gráficos como el módulo del tetris.

5.2. Normas generales

Deberán observarse también las siguientes normas generales:

1. Los programas deberán estar correctamente documentados[3].
2. Las entradas de datos han de estar convenientemente depuradas.
3. Se valorará la calidad de la documentación entregada, incluyendo la documentación interna del software.
4. En el fichero LEEME se debe dejar indicado qué partes de las opcionales han sido incluidas. Además, se deberán incluir ficheros de configuración adecuados según esas partes opcionales.
5. Se valorará la calidad del diseño realizado, no sólo si el programa funciona o no. Así, se

3 *Se debe utilizar la herramienta "doxygen" para ello.*

tendrán en cuenta características como la modularización y el uso adecuado de bibliotecas.

6. Para realizar esta práctica se proporcionan varios documentos. Es tarea del alumno, leer y entender todos y cada uno de ellos sin excepción. No se responderán preguntas cuya respuesta sea obvia o esté escrita explícitamente en dichos documentos.
7. La elaboración de la práctica es individual. Cualquier elaboración conjunta (voluntaria o involuntariamente) supondrá el suspenso en la práctica.
8. Cualquier copia total o parcial de soluciones supondrá el suspenso en la práctica. El alumno debe velar por la privacidad y seguridad de sus datos.
9. No se admitirán reclamaciones por virus, borrados accidentales, etc, causados por descuido del alumno.
10. Las condiciones de la entrega se publicarán con tiempo suficiente en la página web de la asignatura.

6. Referencias

- [TET] *Tetris*. En la Wikipedia: http://es.wikipedia.org/wiki/Tetris
- [FUN] "*Fundamentos de programación en C++*". A Garrido. Delta publicaciones, 2006.
- [ABS] "Abstracción y estructuras de datos en C++". A Garrido y J. Fdez-Valdivia. Delta publicaciones, 2006.
- [GRF] "Módulo de gestión gráfica". J. Martínez Baena y A. Garrido. Documento de apoyo a MP2. 2008.