Proyecto de programación en ensamblador de Estructura y Tecnología de Computadores

Julio de 2015

Práctica 1

PROYECTO DE PROGRAMACIÓN EN MIPS: TETRIS

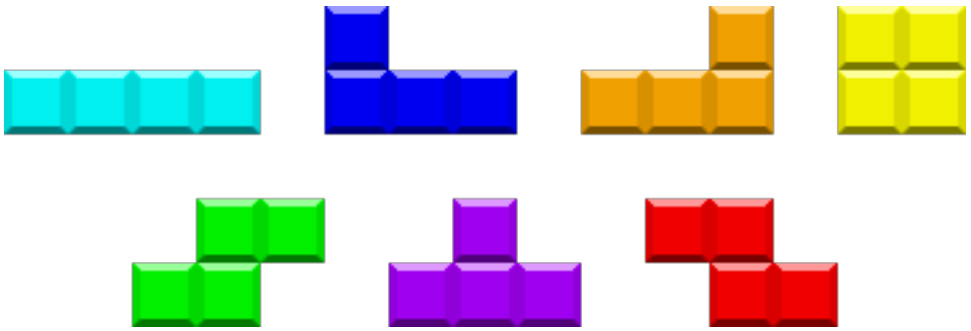
1.1 OBJETIVOS

El objetivo de la práctica es que el alumno demuestre que es capaz de entender el esqueleto de un programa en ensamblador desarrollado parcialmente, e implementar el código faltante a partir de una especificación funcional.

1.2 TETRIS

El programa que utilizaremos para realizar este proyecto, será una versión del famoso Tetris, lanzado al mercado en 1984 y que se convirtió en un fenómeno mundial, siendo considerado el mejor videojuego de todos los tiempos en más de una ocasión.

El juego está basado en el concepto matemático de los "pentominós" (conjunto de fichas que se pueden construir acoplando 5 cuadrados por sus lados), simplificándolo a 4 cuadrados en lugar de 5. De esta forma, en lugar de los 12 pentominós, obtenemos 7 tetraminós (http://es.wikipedia.org/wiki/Tetromino), incluyendo simetrías:



El funcionamiento es sencillo. Una pieza, seleccionada de forma aleatoria de entre las de un conjunto predeterminado, aparece en la parte superior de un campo de juego delimitado lateral e inferiormente, y comienza a caer a velocidad uniforme, hasta que tropieza con alguna de las otras piezas o el fondo del campo. Se permite rotar la pieza mientras cae, moverla a izquierda o derecha y acelerar su descenso. Una vez que la pieza se detiene, puede suceder que se haya completado una fila de cuadrados en el campo de juego. En ese caso, la fila desaparece y todo lo que estuviera sobre ella cae hasta encontrar una posición estable. Cada pieza puesta aumenta el marcador en una cierta cantidad de puntos. Si las piezas depositadas llegan a alcanzar una altura tal que no dejan que aparezcan más, el juego termina.

1.3 FASES DEL DESARROLLO DEL PROYECTO

En una primera fase partiremos de un esqueleto del programa, y cada grupo de prácticas, formado por un par de alumnos, desarrollará de forma incremental y guiada una versión simplificada que no elimina las filas completas. Durante esta primera fase, sólo será necesario realizar la traducción a ensamblador de varios procedimientos cuya implementación en C se proporciona.

Con este funcionamiento básico, aparecerá una pieza aleatoria que caerá a un cierto ritmo y tendremos que guiarla, rotándola y moviéndola, hasta que llegue al fondo del campo o tropiece con otras piezas que hayan caído previamente. Tampoco se llevará, de momento, control de la puntuación, ni comprobaremos si el juego debe terminar.

En una segunda fase, cada grupo de prácticas deberá desarrollar de forma autónoma una serie de procedimientos para implementar funcionalidades más avanzadas. Finalmente, se proponen también mejoras opcionales que el alumno podría llevar a cabo.

1.4 DESCRIPCIÓN DEL PROGRAMA INICIAL

El programa incompleto inicial se puede encontrar en el fichero `tetris-esqueleto.s`. Se ha desarrollado a partir de una versión original en C (`tetris.c`) que también se proporciona como guía. La versión en C es jugable (si la compilamos), y nos permite ver el aspecto final que tendrá nuestro programa.

Vamos a trabajar sobre la implementación en ensamblador de MIPS. Para su ejecución, deberemos utilizar la versión de MARS disponible en la sección de recursos del aula virtual.

Si se carga el programa en MARS, se ensambla y se ejecuta, aparecerá un menú inicial en el panel de entrada y salida de ejecución. Podremos terminar el juego o comenzar una partida.[1].

La versión inicial, que se proporciona en ensamblador solamente, muestra el menú básico (jugar y salir) y, si se selecciona la opción de jugar, se sale inmediatamente al alcanzar la llamada a `imagen_init`, uno de los procedimientos que hay que implementar como parte de la práctica (una vez implementada la funcionalidad básica descrita en el apartado 1.5.1, la ejecución permanece en el bucle principal del programa hasta que se pulsa la tecla `x` para finalizar la partida).

El jugador interactúa con el programa a través del teclado, para lo cual hay que tener en cuenta que el panel de entrada y salida debe tener el foco para recibir las pulsaciones de teclas (normalmente es suficiente con hacer click con el ratón en el cuadro de texto). Sólo unas pocas teclas tienen alguna función asociada, como se puede ver en la tabla 1.1, y de ellas sólo la `x` está implementada en el programa inicial.

Tecla	Función
`i`	Rotar la pieza 90 grados
`j`	Mover la pieza hacia la izquierda
`l`	Mover la pieza hacia la derecha
`k`	Mover la pieza hacia abajo
`x`	Salir al menú

Tabla 1.1: Pulsaciones de teclas reconocidas por el programa `tetris.s`.

[1] Conviene ajustar el tamaño del panel de forma que quepa entero el campo de juego. Para ello, puede ser útil utilizar las pequeñas flechas situadas en la esquina superior izquierda del panel.

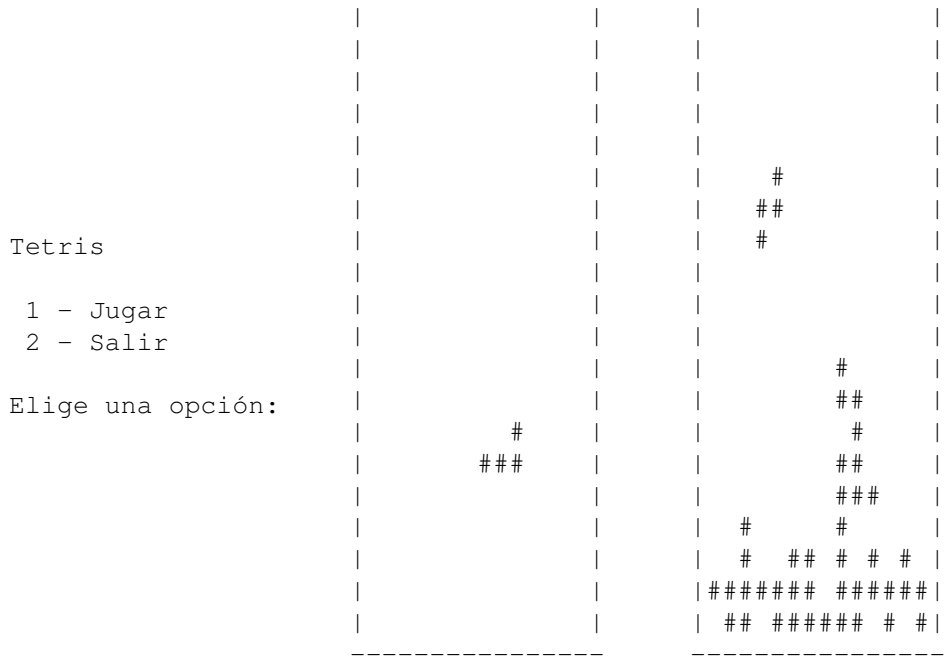


Figura 1.1: Ejemplos del aspecto de la salida del programa en tres momentos diferentes del juego.

La figura 1.1 muestra ejemplos de la ejecución del programa inicial. La pantalla se limpia y se vuelve a generar completamente cada vez que cambia el estado del juego. Los límites del campo se representan con los caracteres «|» y «_», y las fichas se construyen a partir de almohadillas «#».

1.4.1 Funcionamiento general

El programa, una vez elegida la opción de comenzar a jugar, dibuja el campo, elige una pieza aleatoriamente y la muestra en la parte superior. Entonces entra en un bucle donde se realizan las siguientes acciones:

1. Comprueba si se ha pulsado alguna tecla (mediante *polling*).
   - Si se ha pulsado la tecla de rotar, gira la pieza si no tropieza con nada, y se actualiza la pantalla.
   - Si se ha pulsado alguna tecla de mover a la derecha o izquierda, actualiza las coordenadas de la pieza si no se sale del campo de juego ni tropieza con nada, y se actualiza la pantalla.
   - Si se ha pulsado la tecla de bajar la pieza, cambia sus coordenadas comprobando antes que no tropieza con nada, y se actualiza la pantalla.
2. Obtiene la hora actual del sistema (en milisegundos) y la compara con la hora en la que apareció la última pieza o se realizó el último movimiento automático hacia abajo.
   - Si el tiempo transcurrido es mayor que una cierta cantidad, baja automáticamente la pieza actual, se imprime el nuevo estado del programa y se guarda la hora actual para futuras comparaciones.
3. Si al moverse la pieza hacia abajo, ya sea por efecto de la tecla de bajar o debido a la caída automática de las piezas, ésta choca con otra pieza o con el borde inferior del campo, se saca una nueva ficha por la parte superior.

1.4.2 Tipos definidos

En ensamblador no hay tipos estructurados, por lo que tendremos que gestionar manualmente el acceso a los datos. Vamos a trabajar con el equivalente a una estructura de datos en C llamada `Imagen`, compuesta por un array de píxeles (`data`) y un par de enteros (`ancho` y `alto`). Cada píxel ocupará un byte, en el que se almacenará el carácter usado para representarlo en la pantalla.

Este tipo de datos servirá para dos propósitos: en unos casos nos servirá como lugar donde componer el estado del juego, situando en su interior todo lo necesario para después imprimirlo de una vez; en otros contendrá una ficha, permitiendo operar con ella (rotarla, copiarla, etc). Para conocer más detalles sobre cómo acceder a una estructura, lo más adecuado es repasar la sección A.8.3 del texto guía.

Hay que tener en cuenta que cada campo de tipo entero de la estructura tiene que comenzar en una posición múltiplo de 4 para estar alineado. De no ser así, se produciría una excepción de *acceso no alineado* al tratar de utilizar la instrucción `lw` o `sw`. En nuestro caso es fácil, ya que los campos `ancho` y `alto` son dos enteros que ocupan 4 bytes cada uno, y los colocaremos al principio de la estructura, la cual nos aseguraremos de que siempre empiece en una dirección múltiplo de 4. Por otro lado, no es imprescindible que el tamaño del array `data` sea exactamente *alto* $\times$ *ancho*, sino que puede perfectamente ser mayor. Lo importante es que no sea menor que *alto* $\times$ *ancho*, ya que en ese caso se produciría un desbordamiento de buffer[2]. Gracias a los valores `alto` y `ancho` podemos reservar la cantidad de memoria adecuada para cada uso de la imagen. Por ejemplo, para contener una ficha no necesitaremos más de 8 píxeles (2×4), y aún nos sobrará espacio, mientras que para contener el campo de juego necesitaremos una imagen mucho mayor, donde se irán dibujando las fichas. Además, el menor tamaño de las fichas nos facilitará la tarea de depuración al aparecer agrupadas en la ventana de inspección de memoria[3].

Veamos un ejemplo. Si tuviéramos una variable de tipo `Imagen` tal y como lo hemos definido, quedaría en C como sigue:

```
typedef unsigned char Pixel;
struct ImagenRep {
    int ancho;
    int alto;
    Pixel data[1024];
};
typedef struct ImagenRep Imagen;

Imagen mi_imagen = { .ancho = 20, .alto = 40 };
```

En ensamblador habríamos definido la variable de la siguiente manera:

```
mi_imagen: .word 20     # ancho
           .word 40     # alto
           .space 1024  # data
```

Obsérvese que desaparecen los nombres de los campos (salvo en los comentarios, si es que queremos añadirlos). De hecho, en ensamblador no se definen los tipos `Imagen` ni `Pixel` en ningún sitio.

[2]Más conocido como «buffer overflow», y fuente de gran cantidad de fallos y problemas de seguridad en programas reales.

[3]En realidad en la versión en C del programa todas las variables de tipo `Imagen` ocupan la misma cantidad de memoria, pero no se ha traducido literalmente este aspecto en la versión en ensamblador

En este caso, hemos inicializado el ancho y el alto a 20 y 40 respectivamente. Por tanto, estrictamente hablando sólo necesitaríamos 800 bytes para los datos, aunque hemos reservado 1024.

Para acceder a cada campo en el código, no se usarán sus nombres (que no significan nada en ensamblador). En su lugar, se usan desplazamientos con respecto al comienzo de la estructura, denotado por la etiqueta `mi_imagen`. Obsérvese que para este fin es muy útil la constante que se suma al registro base en el modo de direccionamiento *base más desplazamiento* usado en las cargas y almacenamientos:

```
la $t0, mi_imagen  # $t0 ← dirección de inicio de mi_imagen
lw $t1, 0($t0)     # $t1 ← ancho
lw $t2, 4($t0)     # $t2 ← alto
addi $t3, $t0, 8   # $t3 ← dirección del array data
```

1.4.3 Datos del programa

En la sección de datos del programa (`.data`) se encuentran ya definidas las siguientes variables globales. Estas variables serán suficientes para realizar todas las modificaciones de la sección 1.5.1, mientras que puede ser necesario añadir nuevas variables globales para el resto de ejercicios:

`pantalla`: De tipo `Imagen`, contendrá todo lo que se imprimirá a la hora de refrescar el juego en pantalla.

Esta variable permite separar la zona donde se produce la interacción principal (la imagen `campo`) de otros elementos externos al campo de juego que también se deben mostrar por pantalla, como los bordes del campo o un posible marcador.

`campo`: De tipo `Imagen`, contendrá todo lo que se muestra en la zona de juego relativo a las piezas que ya han caído.

`pieza_actual`: De tipo `Imagen`. Contiene la pieza que se encuentra cayendo en este momento con la orientación que tenga, ya que puede haber sido rotada. La forma de representar las piezas es mediante almohadillas:

```
####     ##       ##     ###     #         #     ##
           ##     ##       #     ###     ###     ##
```

Usamos una almohadilla para indicar que ese cuadro está ocupado y, como en todas las imágenes del programa, un carácter nulo (valor 0, `'\0'` en C) para indicar que está libre (transparente).

`pieza_actual_x`, `pieza_actual_y`: Enteros. Coordenadas respecto al origen de la imagen `campo` de la pieza que está cayendo actualmente.

`imagen_auxiliar`: De tipo `Imagen`. Sirve para contener la pieza actual rotada de forma temporal en el procedimiento `intentar_rotar_pieza_actual`, para poder comprobar si dicha rotación es posible antes de volcarla a `pieza_actual`.

`pieza_jota`, `pieza_ele`, `pieza_zeta`, `pieza_ese`, `pieza_barra`, `pieza_cuadro`, `pieza_te`: De tipo `Imagen`. Piezas predefinidas del juego. Estas variables se consideran de sólo lectura.

```
        .align    2     # forzamos que el siguiente dato quede
                        #   alineado en un entero (4 bytes)
    pieza_jota:
        .word     2     # ancho
        .word     3     # alto
        .ascii    "\0#\0###\0\0"  # los caracteres que forman la pieza
        .align 2

        ...

        .align    2
    pieza_te:
        .word     3   ; ancho
        .word     2   ; alto
        .asciiz         "\0#\0###\0\0"
```

piezas: Array de punteros a estructuras `Imagen`. Contiene 7 elementos, uno por cada pieza posible anteriormente definida, indicando la dirección donde se encuentra la pieza en cuestión.

```
        .align    2
    piezas:
        .word     pieza_jota
        .word     pieza_ele
        .word     pieza_zeta
        .word     pieza_ese
        .word     pieza_barra
        .word     pieza_cuadro
        .word     pieza_te
```

En el código anterior podemos ver cómo se define dicho array de piezas. Ahora para elegir una pieza, sólo tenemos que elegir un número entre 0 y 6 y usarlo como índice para acceder al array `piezas`, multiplicándolo por el tamaño de un puntero, que es 4 bytes, y sumándoselo a la etiqueta `piezas` para obtener la dirección en la que está almacenada la dirección de la imagen de la pieza elegida.

acabar_partida: Esta variable controla el bucle principal del juego. Vale 0 durante la partida, y se debe poner a 1 cuando se desee finalizar (ya sea por que se pulse la tecla x o porque la partida acabe por cualquier otra razón).

procesar_entrada.opciones: Este array se utiliza en el procedimiento `procesar_tecla`. Cada elemento de este array ocupa 8 bytes, de los cuales el primero almacena una tecla y los cuatro últimos almacenan la dirección de un procedimiento, el cual debe ser llamado cada vez que se pulse la tecla. Los 3 bytes restantes no se utilizan, pero son necesarios para asegurar que el puntero a la función quede alineado adecuadamente.

str000, str001, str002: Cadenas que almacenan los mensajes utilizados por el programa.

1.4.4 Procedimientos ya implementados del programa

En el esqueleto del programa ensamblador que se proporciona se encuentran una serie de procedimientos y funciones ya implementados con aspectos básicos del programa, como por ejemplo leer un píxel de una imagen, leer una tecla, imprimir la pantalla del juego, obtener un número aleatorio, etc. Su propósito es servir de base y de muestra a la hora de implementar el resto. Partiendo de éstos, se escribirán otros similares o algo más elaborados.

El código de estos procedimientos **se debe examinar en el fichero `tetris-esqueleto.s` que se adjunta** con el material de prácticas. Además, corresponden con las funciones del código C proporcionado. En algunos casos sencillos también se muestra el código aquí por comodidad.

`imagen_pixel_addr`: Devuelve la dirección de un píxel dentro de una imagen. Recibe la dirección de una imagen y dos coordenadas enteras (x e y). Hay que tener en cuenta que las coordenadas se buscan dentro del array de datos suponiendo que la coordenada (0,0) se encuentra en la esquina superior izquierda y la coordenada (ancho-1, alto-1) en la esquina inferior derecha.

```
imagen_pixel_addr:          # ($a0, $a1, $a2) = (imagen, x, y)
                            # pixel_addr = &data + y*ancho + x
    lw     $t1, 0($a0)      # $a0 = dirección de la imagen
                            # $t1 ← ancho
    mul    $t1, $t1, $a2    # $a2 * ancho
    addu   $t1, $t1, $a1    # $a2 * ancho + $a1
    addiu  $a0, $a0, 8      # $a0 ← dirección del array data
    addu   $v0, $a0, $t1    # $v0 = $a0 + $a2 * ancho + $a1
    jr     $ra
```

`imagen_get_pixel`: Lee un píxel de una imagen. Recibe la dirección de la imagen y dos coordenadas enteras (x, y). Devuelve el píxel que hay en esa posición (un byte).

```
imagen_get_pixel:           # ($a0, $a1, $a2) = (img, x, y)
    addiu  $sp, $sp, -4     # espacio para un entero ($ra)
    sw     $ra, 0($sp)      # guardamos $ra porque haremos un jal
    jal    imagen_pixel_addr # (img, x, y) ya en ($a0, $a1, $a2)
    lbu    $v0, 0($v0)      # lee el pixel a devolver
    lw     $ra, 0($sp)      # restaura $ra
    addiu  $sp, $sp, 4      # libera el espacio de la pila
    jr     $ra
```

`imagen_print`: Imprime una imagen por pantalla carácter a carácter, terminando cada fila con un retorno de carro. Recibe la dirección de la imagen a imprimir[4].

`pieza_aleatoria`: Devuelve la dirección de la imagen de la siguiente pieza que va salir. Para ello, llama a la función `random_int_range` que también se proporciona como librería básica y utiliza el resultado para acceder al array `piezas`.

[4]Como ejercicio de prueba, es aconsejable escribir un pequeño código para comprobar que este procedimiento funciona y que puede imprimir, por ejemplo, una ficha de las predefinidas.

probar_pieza: Devuelve 1 si una pieza puede ponerse en una posición concreta de `campo` y 0 si no. Recibe la dirección de la imagen de la pieza y las coordenadas x e y de la posición. Para comprobarlo habrá que ver si alguno de los píxeles con almohadilla de pieza pasada choca con una pieza ya puesta en `campo`, o si estamos tan cerca de uno de los bordes que no puede ponerse.

actualizar_pantalla: Muestra por pantalla el estado actual del juego. Utiliza la imagen `pantalla` para dibujar en ella todo lo que se mostrará por pantalla. Para ello, se limpia la imagen, se dibujan los bordes del campo, se copia la imagen `campo` en la imagen `pantalla` y se dibuja la pieza actual en su posición. Finalmente, se limpia la pantalla y se imprime la imagen `pantalla` en ella.

procesar_entrada: Este procedimiento lee una tecla y la procesa, llamando al procedimiento que implementa la funcionalidad asociada a la tecla pulsada. Para ello, utiliza el array `procesar_entrada.opciones`, el cual se recorre y, si se encuentra una entrada correspondiente a la tecla pulsada, se llama al procedimiento asociado.

tecla_salir, tecla_izquierda, tecla_derecha tecla_abajo, tecla_rotar: Estos pequeños procedimientos realizan la acción asociada a cada tecla. En general, se limitan a llamar a otras funciones que hacen el trabajo real.

jugar_partida: Este es el procedimiento principal del juego, que implementa el bucle descrito en la sección 1.4.1. La salida del bucle está controlada por la variable `acabar_partida`.

main: Este es el procedimiento que se ejecuta al inicio del programa. Consiste en un bucle del cual el programa nunca sale (hasta que se realiza la llamada al sistema para terminar el programa). El bucle muestra un menú que permite comenzar una partida o salir del programa.

1.5 EJERCICIOS

1.5.1 Ejercicios de traducción *(4 puntos)*

Para llegar a una versión funcional del programa necesitaremos implementar una serie de procedimientos básicos a partir de los cuales construir el resto. Va a ser labor del alumno esta implementación inspirándose en los ejemplos anteriores.

El objetivo es ir implementando los procedimientos propuestos en cada paso, comprobando que cada uno funciona correctamente antes de continuar. Será labor del alumno escribir el código para probar su corrección, y explicar en la memoria del proyecto cómo se ha realizado esa comprobación.

Con el programa proporcionado, no se podrán probar directamente los procedimientos traducidos, ya que el programa necesita que estén prácticamente terminados todos antes de poder empezar a jugar. Sin embargo, es posible sustituir el procedimiento `main` del juego por uno simple de prueba que se limite a llamar a los procedimientos que queremos probar y que muestre el resultado para poder comprobar su corrección. Por ejemplo, la figura 1.2 muestra un procedimiento `main` que serviría para probar `imagen_init`, `imagen_clean` e `imagen_set_pixel`. Si se descomentan las líneas de la 13 a la 17 serviría también para probar `imagen_dibuja_imagen`.

Pasamos a especificar la interfaz y función de cada procedimiento. Si se considera más sencillo, o de forma complementaria, se puede consultar la implementación en C en `tetris.c`. En cualquier caso es importante comprobar la corrección de cada procedimiento antes de continuar con el siguiente. Realizando correctamente estos ejercicios se puede optar a 4 puntos sobre 10 en la valoración del proyecto.

```
main:                       # ($a0, $a1) = (argc, argv)
    addiu $sp, $sp, -4
    sw    $ra, 0($sp)

    # prueba de imagen_init, imagen_clean e imagen_set_pixel
    la    $a0, pantalla
    li    $a1, 10
    li    $a2, 8
    li    $a3, 'x'
    jal   imagen_init

    # prueba imagen_dibuja_imagen
    #la    $a0, pantalla
    #la    $a1, pieza_ese
    #li    $a2, 2
    #li    $a3, 3
    #jal   imagen_dibuja_imagen

    la     $a0, pantalla
    jal    imagen_print

    jal    mips_exit

    lw     $ra, 0($sp)
    addiu  $sp, $sp, 4
    jr     $ra
```

Figura 1.2: Ejemplo de procedimiento `main` para probar algunos preocedimientos traducidos.

`imagen_set_pixel`: Modifica un píxel de una imagen. Recibe la dirección de la imagen, dos coordenadas enteras (x, y) y el byte que se debe almacenar en esa posición.

`imagen_clean`: Rellena una imagen con un valor. Recibe la dirección de la imagen y un byte de relleno.

Tras implementarlo, comprueba que funciona usando el visor de la memoria de MARS.

`imagen_init`: Inicializa o reinicializa una imagen, asignándole nuevas dimensiones y rellenándola con un valor. Recibe la dirección de la imagen, dos enteros con los nuevos valores ancho y alto y un byte con el valor de relleno. Esta función no comprueba si el array de la imagen tiene un tamaño suficiente para las dimensiones que se pasan. Si usáramos una imagen sin espacio suficiente se sobreescribirían los datos que hubieran almacenados a continuación de la imagen en la memoria.

Tras implementarlo, comprueba que funciona usando el visor de memoria de MARS.

`imagen_copy`: Copia una imagen en otra, incluyendo las dimensiones. Recibe una dirección de imagen destino y otra dirección de imagen fuente. La variable destino tiene que tener espacio suficiente reservado como para contener la imagen fuente, ya que no se comprueba. La copia se realiza llamando repetidamente a `imagen_set_pixel` e `imagen_get_pixel`. Tras implementarlo, comprueba que funciona usando el visor de memoria de MARS.

`imagen_dibuja_imagen`: Superpone una imagen en otra, a partir de la coordenada (x, y) de la imagen destino, con la peculiaridad de que si el byte origen vale 0 no modifica el destino

(es decir, el píxel se interpreta como transparente). Recibe dos direcciones de imagen y las coordenadas del píxel de la imagen de destino a partir del cual se dibuja la imagen. Se asume que la imagen fuente cabe en la imagen destino a partir de la posición indicada.

imagen_dibuja_imagen_rotada: Superpone una imagen en otra, a partir de la coordenada (x, y) del destino, rotándola previamente 90 grados en sentido de las agujas del reloj. Si el píxel origen vale 0, no modifica el destino (es decir, se interpreta como transparente). Es aconsejable utilizar como guía la implementación del procedimiento `imagen_dibuja_imagen`. Para rotar una ficha tal y como se indica, se hace la siguiente transformación en las coordenadas destino: $(x,y) \rightarrow (alto-1-y,x)$. Recibe dos direcciones de imagen y las coordenadas del punto a partir del cual se dibuja.

nueva_pieza_actual: Determina cuál es la siguiente pieza en caer, llamando a la función `pieza_aleatoria`, y la copia a la imagen `pieza_actual`, asignando la posición inicial de salida (`pieza_actual_x`, `pieza_actual_y`) para que sea el centro de la primera línea del campo. No recibe parámetros y no devuelve nada.

intentar_movimiento: Trata de colocar la pieza actual en una nueva posición. Para comprobarlo, tendrá que recorrer las coordenadas de la pieza y las de la imagen donde va a situarse y ver si se sale de las dimensiones de la imagen destino, o bien un píxel ocupado de la pieza cae sobre un píxel ocupado del campo. Devuelve 0 si no se puede encajar la pieza actual en la posición dada, y en el caso de que sí se pueda encajar, actualiza la posición de la pieza actual (`pieza_actual_x`, `pieza_actual_y`) a esa nueva y devuelve 1. Recibe la coordenada (x, y) donde se desea colocar la pieza.

intentar_rotar_pieza_actual: Trata de rotar la pieza actual 90 grados. Para ello, se utiliza la variable `imagen_auxiliar`, la cual se inicializa con ceros y al tamaño adecuado para copiar en ella una versión de `pieza_actual` rotada usando `imagen_dibuja_imagen_rotada`. Se prueba si la pieza rotada encaja en el campo, llamando a `probar_pieza`, y si es así se copia `imagen_auxiliar` en `pieza_actual`. No recibe parámetros ni devuelve nada.

bajar_pieza_actual: Utiliza `intentar_movimiento` para intentar poner la pieza actual en la posición (`pieza_actual_x`,`pieza_actual_y+1`) del campo. Si no se puede es porque la pieza actual ha llegado al suelo o choca con otra pieza. Si es así, se dibuja la pieza en el `campo` en la posición en la que se encuentra, de forma que queda fijada, y se llama a `nueva_pieza_actual` para que comience a caer una nueva pieza.

1.5.2 Ejercicios de implementación *(6 puntos)*

Tras implementar las funciones anteriores, ya tenemos una versión jugable de nuestro Tetris. Ahora el objetivo es añadir nuevas funcionalidades que le den un aspecto más vistoso. Para ello, proporcionamos una lista de funcionalidades que deberán implementarse en orden, ya que en algún caso una mejora puede depender de las anteriores.

Esta parte permite obtener los 6 puntos restantes del proyecto (sobre 10).

Marcador de puntuación: Cada nueva pieza que aparezca proporcionará 1 punto. La puntuación hasta el momento se mostrará sobre el campo de juego, usando el formato «Puntuación: X» (donde X es la puntuación actual).

Se deberá implementar un procedimiento llamado `integer_to_string`, que reciba como parámetros un valor entero y una dirección de memoria a partir de la cual dejar la cadena de

caracteres que contiene la representación de los dígitos del entero. La cadena de caracteres, como todas las usadas en el programa, debe terminar con un carácter '\0'. Se puede suponer que el buffer de destino siempre tiene un tamaño suficiente. Nótese que este procedimiento fue desarrollado en la segunda práctica de ensamblador.

También habrá que implementar un procedimiento llamado `imagen_dibuja_cadena` que recibe como parámetros la dirección de la imagen donde se dibujará la cadena, las coordenadas dentro de la imagen en las que comenzar a escribir y la dirección donde se encuentra la cadena que se desea escribir. Se recomienda hacer uso de la función `imagen_set_pixel` para guardar los diferentes caracteres de la cadena en la imagen.

`Final de la partida:` En la versión inicial, la partida acaba cuando se pulsa la tecla `x`. Sin embargo, puede suceder que las piezas acumuladas alcancen tal altura que no permitan que una nueva pieza salga. En ese momento, la partida debe también terminar, mostrándose los mensajes «FIN DE PARTIDA» y «Pulse una tecla» uno bajo el otro, en el centro de la pantalla y rodeados de espacios y de un recuadro (dibujado con los caracteres « | », «−» y «+») para que se puedan leer con facilidad. Se esperará a que se pulse cualquier tecla y se volverá al menú principal. Este mensaje no puede aparecer cuando la partida acabe porque el usuario pulse la tecla `x` (aunque sí podría aparecer un mensaje distinto en ese caso).Se puede utilizar la función `probar_pieza` para comprobar si la partida debe terminar.

`Completando líneas:` Conforme van cayendo piezas y se van acumulando unas sobre otras es inevitable que terminen alcanzando el techo, finalizando la partida y haciendo que el juego sea bastante corto. Para darle más emoción, empezaremos haciendo que cada vez que al insertar una pieza se complete una línea horizontal, se le sumen 10 puntos al jugador.

`Eliminando líneas:` Para aumentar la duración del juego y que sea más interesante, vamos a eliminar las líneas horizontales completadas que se contabilizaron en el apartado anterior. Cada vez que una ficha al encajar llene una o varias líneas horizontales, además de sumarse los puntos, se eliminarán las líneas completadas.

`Mostrar la siguiente pieza:` Una forma de hacer más fácil nuestro juego, es conocer cuál será la próxima pieza que caerá. De esta manera podemos seleccionar dónde poner la actual para encajar mejor la siguiente. Queremos que la siguiente pieza se muestre en la parte superior del campo de juego, a la derecha del campo actual y rodeada por un recuadro.

1.5.3 Funcionalidad opcional *(hasta 1,5 puntos adicionales)*

A continuación se describen dos ejercicios opcionales que permiten obtener hasta 1,5 puntos adicionales. Estos ejercicios sólo se contarán si se han realizado previamente todos los ejercicios de las secciones 1.5.1 y 1.5.2. El alumno podrá también optar por desarrollar sus propias extensiones, previa autorización por parte del profesor correspondiente. En cualquier caso, es importante notar que la puntuación máxima que puede obtenerse a través de todas estas mejoras extras está limitada a 1,5 puntos.

`Configuración:` Ampliaremos el menú principal con una opción que nos llevará a un menú de configuración, donde permitiremos cambiar varias opciones del programa (tamaño del campo, velocidad inicial, teclas para el movimiento, etc.). Habrá que hacer todas las comprobaciones adecuadas de que los valores introducidos sean válidos. Por ejemplo, no permitiremos dar unas dimensiones al campo de juego que hagan que nos salgamos de la memoria asignada a la imagen (o de la pantalla).

`Ritmo de caída:` Cada 20 puntos aumentará la velocidad de caída de las piezas. El ritmo de caída actual viene marcado por la variable `pausa` de la función `jugar_partida`, que se ha traducido inicialmente como una constante. Se deberá crear una variable global para almacenar este valor, que será decrementado cada 20 puntos en un 10 % hasta un mínimo de 300.

1.6 CRITERIOS DE EVALUACIÓN

A la hora de evaluar el trabajo, se consideraran (entre otros) los siguientes aspectos:

- Funcionamiento correcto del programa con las mejoras realizadas.
- Claridad del código entregado.
- Uso correcto de todos los convenios de programación vistos en la asignatura. No será posible aprobar la práctica si no se siguen correctamente los convenios de programación explicados en clase.
- Completitud, claridad y concisión de la memoria entregada.

Para obtener una calificación de 5 puntos (el mínimo para aprobar), es imprescindible entregar un programa que funcione correctamente y haber implementado correctamente todas las funciones descritas en la sección 1.5.1 y al menos un ejercicio de los descritos en la sección 1.5.2. Para obtener los 10 puntos será necesario, además, haber realizado correctamente los ejercicios de la sección 1.5.2. Adicionalmente, el alumno podrá optar por desarrollar alguno de los ejercicios opcionales de la sección 1.5.3 para sumar nota.

Para que se considere que un ejercicio se ha realizado correctamente, además de cumplir con los requisitos funcionales de la especificación, se deberán evitar los siguientes errores graves:

1. No seguir correctamente los convenios del ABI de MIPS explicado en clase, en cualquiera de sus aspectos (uso de registros, uso de la pila, llamadas a procedimietos...).
2. Uso de saltos no locales (saltos desde el cuerpo de un procedimiento a otro procedimiento distintos de las llamadas normales entre procedimientos).

La comisión de fallos graves dará lugar a una calificación automática de **suspenso**.

1.7 REQUISITOS DE ENTREGA

La práctica se entregará por medio del aula virtual empaquetada en un archivo «.tar.gz» o «.zip». El archivo debe contener al menos:

- Información suficiente para identificar de forma sencilla e inequívoca a los miembros del grupo.
- Memoria explicativa de la práctica en formato PDF.
- Código fuente de la práctica.

Se debe entregar un solo archivo por cada grupo de prácticas. El nombre del archivo debe seguir el siguiente formato:

`tetris-`*`dniA`*`-`*`dniB`*`.`*`extensión`*

Donde *dniA* y *dniB* son los DNI de los integrantes del grupo ordenados ascendentemente y *extensión* es «`tar.gz`» o «`zip`».