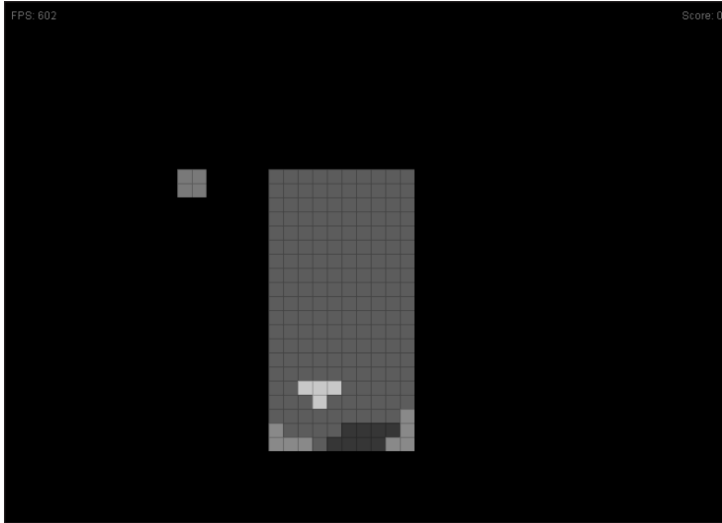


# Tetris

La mayoría de nosotros alguna vez jugamos al Tetris o, al menos, conocemos de qué se trata este juego. Por lo tanto, sin más preámbulo, desarrollaremos un juego del mismo estilo, en donde pondremos en práctica lo aprendido y abordaremos un nuevo concepto: el mapa.

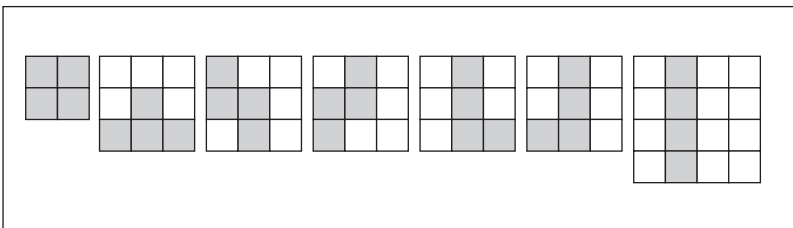
## COMPONENTES Y ESTRUCTURA

El **Tetris** es un juego del tipo puzzle donde irá descendiendo una figura (**pieza**) dentro de un rectángulo (**tablero o mapa**). Una vez que ésta colisiona con el borde inferior del rectángulo o contra otra figura, se *estampará* allí y se lanzará una nueva pieza. Éstas deberán ser ubicadas de manera tal de llenar filas para que sean eliminadas, evitando que lleguen hasta el extremo superior.



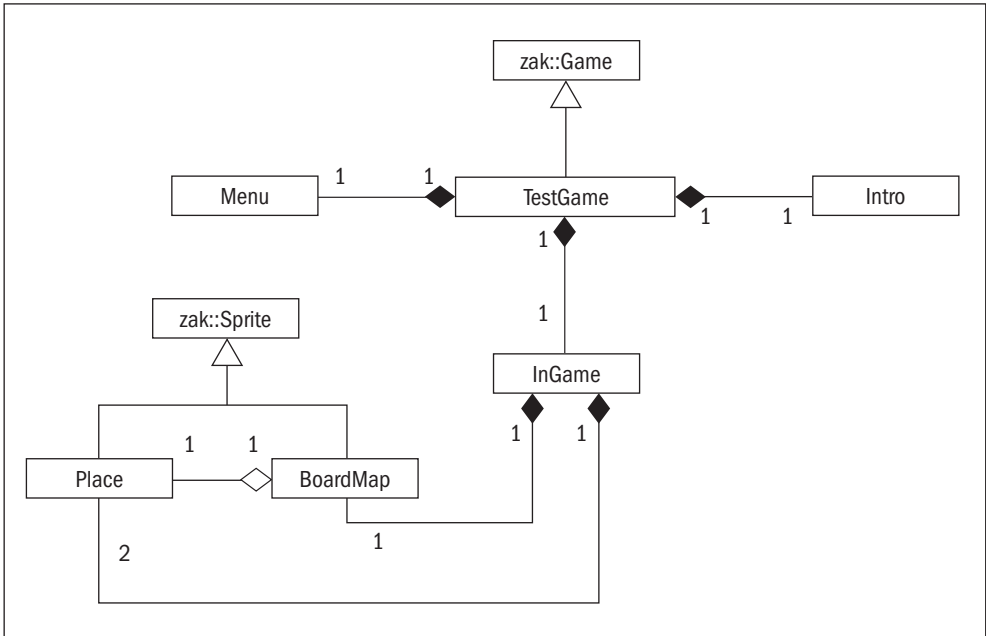
**Figura 1.** El Tetris es un clásico juego del tipo puzzle.

El origen del nombre del juego proviene de la palabra **tetra** (cuatro) dado que las piezas posibles están conformadas por las combinaciones de cuatro cuadrados, como nos muestra la **Figura 2**.



**Figura 2.** Las piezas están conformadas por todas las combinaciones posibles de cuatro cuadrados.

Como vimos en el capítulo anterior, si imaginamos el juego terminado, podemos deducir qué elementos necesitamos. Observando la descripción, deducimos que necesitaremos como mínimo una pieza y un tablero donde estamparla. Por lo tanto, el diagrama de clases se verá como nos muestra la **Figura 3**.



**Figura 3.** Observemos que necesitamos, al menos, una clase que manipule la pieza que irá descendiendo y otra para el mapa.

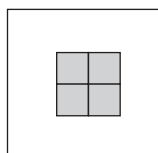
La clase **Piece** se encargará de almacenar, trasladar, rotar y mostrar la pieza en escena. Luego, la clase **BoardMap** manipulará el mapa chequeando colisiones de la pieza con los bordes y con las piezas ya estampadas en él.

Veamos, a continuación, las clases en detalle con su código fuente.

## LA CLASE PIECE

Como ya se dijo, la clase **Piece** se encargará de almacenar, trasladar, rotar y mostrar la pieza en escena.

Cada pieza estará representada por un arreglo cuyo primer elemento corresponderá al tamaño del lado de la pieza, y el resto, a la forma.

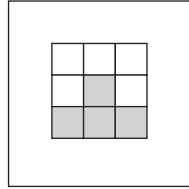


**Figura 4.** La pieza cuadrada del juego.

En la **Figura 4**, apreciamos la pieza cuadrada. Ésta posee dos casilleros por lado, por lo tanto, el arreglo que la define tendrá la siguiente información:

{2,1,1,1,1}

El elemento 2 indica que posee dos casilleros por lado; los unos siguientes representan la forma de la pieza y serán interpretados como una matriz de 2x2. Veamos otro ejemplo:

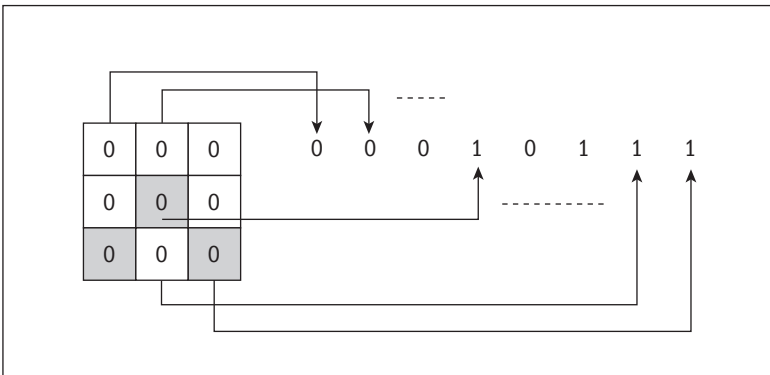


**Figura 5.** Esta pieza posee tres casilleros por lado.

La pieza que muestra la **Figura 5** posee tres casilleros por lado y, por lo tanto, el arreglo que la representa se verá del siguiente modo:

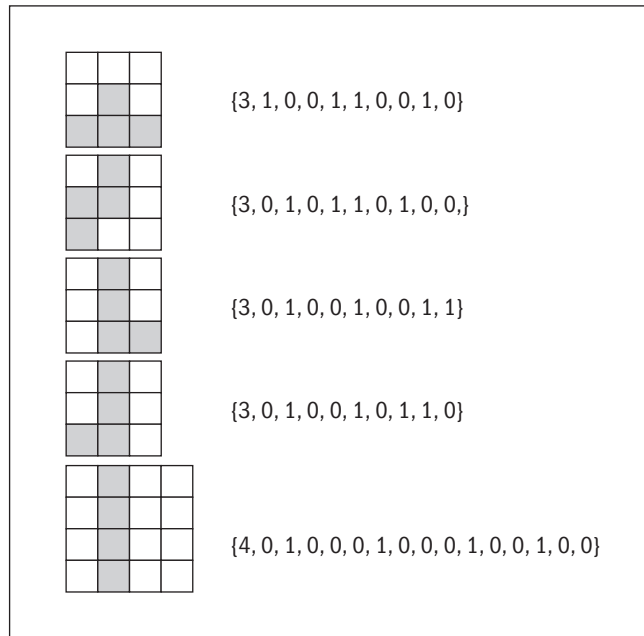
{3,0,0,0,0,1,0,1,1,1}

El 3 indicará que esta pieza posee tres casilleros por lado. Luego, del mismo modo que en el ejemplo anterior, los subsiguientes ceros y unos representarán la forma en sí misma, como nos muestra la **Figura 6**.



**Figura 6.** La figura estará representada por un arreglo cuadrado de  $N \times N$  elementos donde el primer valor indicará el tamaño por lado de la figura, y el resto codificará su forma.

Definiremos el resto de las figuras de manera análoga, como indica la **Figura 7**.



**Figura 7.** Aquí vemos como se definen las figuras restantes.

Como vemos, siempre trabajamos con matrices cuadradas de  $N \times N$  elementos. Esto seguramente nos trae a la mente la pregunta ¿por qué utilizar una matriz cuadrada cuando hay toda una fila o columna en cero? Se debe a la manera en que desarrollaremos la rotación que veremos más adelante en este capítulo.

Observemos el archivo cabecera de la clase **Piece**:

```
#pragma once

#include "zakengine/zak.h"

using namespace zak;

#define MAX_PIECES          7
#define MAX_PIECE_SIZE     4

class Piece : public Sprite {
public:
    void SetRandomType();
    void SetType(int type, DWORD color);
    int     GetType() { return _type; }
```

```

void RotateCW();
void RotateCCW();
void MoveLeft() { _col--; }
void MoveRight() { _col++; }
void MoveUp() { _row--; }
void MoveDown() { _row++; }

DWORD GetColorByType(int type);

void SetStartPos(float x, float y) { _startPosX = x; _startPosY = y;}

void Update(float dt);
void Draw();

Piece();
~Piece();

private:
    int         _size;
    DWORD      _data[MAX_PIECE_SIZE*MAX_PIECE_SIZE];
    DWORD      _color;
    int         _row;
    int         _col;
    float      _startPosX;
    float      _startPosY;
    int         _type;

    friend class BoardMap;
};

```

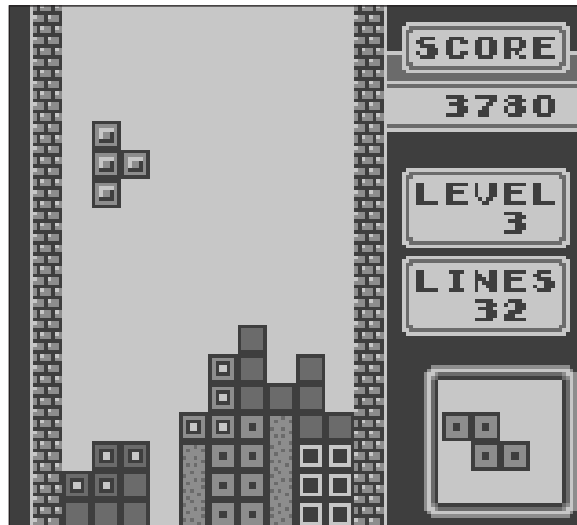
Comencemos por las constantes de preprocesador:

```

#define MAX_PIECES          7
#define MAX_PIECE_SIZE    4

```

La constante **MAX\_PIECES** indica la cantidad máxima de tipos de piezas que tendremos. En el caso del Tetris, este valor está definido por la cantidad de combinaciones posibles entre cuatro cuadrados, que da como resultado 7.



**Figura 8.** Existen innumerables clones de este clásico, incluso para GameBoy.

Luego tenemos la constante **MAX\_PIECE\_SIZE**, que identificará el tamaño máximo de un lado de la matriz cuadrada más grande que podrá tener una pieza; en este caso, será de 4x4.

Pasemos ahora a los métodos:

```
void SetRandomType();
```

Este método seleccionará de manera aleatoria un tipo de pieza.

```
void SetType(int type, DWORD color);
```

El método **SetType** permitirá la selección de un tipo específico de pieza y podremos pasarle, además, el color.

```
int GetType() { return _type; }
```

Devolverá el tipo de pieza.

```
void RotateCW();
void RotateCCW();
```

Los métodos **RotateCW** (rotación *clockwise* o según las agujas del reloj) y **RotateCCW** (rotación *counter clockwise* o contra el sentido de las agujas del reloj) serán invocados para rotar la pieza según corresponda.

```
void MoveLeft() { _col--; }
void MoveRight() { _col++; }
void MoveUp() { _row--; }
void MoveDown() { _row++; }
```

Con estos métodos, moveremos la pieza a derecha, izquierda, arriba y abajo según necesitemos.

```
DWORD GetColorByType(int type);
```

Esto devolverá el color que corresponda, a partir de un tipo de pieza pasado por parámetro.

```
void SetStartPos(float x, float y) { _startPosX = x; _startPosY = y;}
```

Permitirá seleccionar la posición desde la cual se comienza a dibujar la pieza para que coincida con el trazado del mapa.

```
void Update(float dt);
```

Actualizará la posición y, si la tuviera, la animación de la pieza.

```
void Draw();
```

El método **Draw**, como siempre, dibuja la pieza.

Veamos ahora las propiedades:

```
int          _size;
```

Tamaño del lado de la matriz correspondiente a la pieza actual.

```
DWORD _data[MAX_PIECE_SIZE*MAX_PIECE_SIZE];
```

Arreglo que contendrá la codificación de la pieza actual.

```
DWORD _color;
```

Color de la pieza actual.

```
int _row;
int _col;
```

Columna y fila en la que se encuentra la pieza actual dentro del mapa.

```
float _startPosX;
float _startPosY;
```

Posición en coordenadas de mundo a partir de la cual se comienza a dibujar la pieza.

```
int _type;
```

Tipo de pieza que se muestra.

```
friend class BoardMap;
```

Por último, indicamos que la clase **BoardMap** tendrá acceso a las propiedades privadas de la clase.

Veamos ahora el código fuente de la clase:

```
#include "piece.h"
unsigned char pieceType[][1+(MAX_PIECE_SIZE*MAX_PIECE_SIZE)] = {
    {2,
     1, 1,
     1, 1},
```

```
        {3,
          0, 1, 0,
          1, 1, 1,
          0, 0, 0},
        {3,
          1, 0, 0,
          1, 1, 0,
          0, 1, 0},
        {3,
          0, 1, 0,
          1, 1, 0,
          1, 0, 0},
        {3,
          0, 1, 0,
          0, 1, 0,
          0, 1, 1},
        {3,
          0, 1, 0,
          0, 1, 0,
          1, 1, 0},
        {4,
          0, 1, 0, 0,
          0, 1, 0, 0,
          0, 1, 0, 0,
          0, 1, 0, 0}
    };

    Piece::Piece() {
        _size = 0;
        _color = 0xFF00FF00;
        _col = 0;
        _row = 0;
        _startPosX = 0;
        _startPosY = 0;
    }

    Piece::~Piece() {
    }
```

```
DWORD Piece::GetColorByType(int type) {
    DWORD color = 0xFF000000;

    switch(type) {
        case 0:
            color = 0xFFFF0000;
            break;

        case 1:
            color = 0xFF00FF00;
            break;

        case 2:
            color = 0xFF0000FF;
            break;

        case 3:
            color = 0xFFFFFFFF00;
            break;

        case 4:
            color = 0xFF00FFFF;
            break;

        case 5:
            color = 0xFFFF00FF;
            break;

        case 6:
            color = 0xFF601060;
            break;
    }
    return color;
}

void Piece::SetRandomType() {
    int rnd = rand()%MAX_PIECES;

    SetType(rnd, GetColorByType(rnd));
}

void Piece::SetType(int type, DWORD color) {

    // Almaceno el tamaño de lado de la pieza
    _size = pieceType[type][0];
}
```

```
    _type = type;

    _color = color;

    for (int row=0; row<MAX_PIECE_SIZE; row++)
        for (int col=0; col<MAX_PIECE_SIZE; col++)
            _data[col+row*_size] = pieceType[type] [
                (col+row*_size)+1]*_color;

    _col = 0;
    _row = 0;

    SetVisible(true);
}

void Piece::RotateCW() {
    int ncol, nrow;
    DWORD newPiece[MAX_PIECE_SIZE*MAX_PIECE_SIZE];

    // Roto la pieza
    for (int row=0; row<_size; row++)
        for (int col=0; col<_size; col++)
        {
            ncol = _size-1-row;
            nrow = col;
            newPiece[ncol + nrow * _size] = _data[col + row * _size];
        }

    // Copio la pieza temporal a la definitiva
    memcpy((void *) _data, (const void *) newPiece, _size * _size *
        sizeof(DWORD));
}

void Piece::RotateCCW() {
    int ncol, nrow;
    DWORD newPiece[MAX_PIECE_SIZE*MAX_PIECE_SIZE];

    // Roto la pieza
    for (int row=0; row<_size; row++)
        for (int col=0; col<_size; col++)
```

```

        {
            ncol = row;
            nrow = _size-1-col;
            newPiece[ncol + nrow * _size] = _data[col + row * _size];
        }

// Copio la pieza temporal a la definitiva
memcpy((void *) _data, (const void *) newPiece, _size * _size *
        sizeof(DWORD));
}

void Piece::Update(float dt) {
    float x = GetPosX();
    float y = GetPosY();

    Sprite::Update(dt);

    SetPos(_startPosX+_col*GetWidth()*GetScaleX(), _startPosY-
        _row*GetHeight()*GetScaleY());
}

void Piece::Draw() {
    float x = GetPosX();
    float y = GetPosY();

    if (GetVisible()) {
        for (int row=0; row<_size; row++) {
            for (int col=0; col<_size; col++) {
                if (_data[col+row*_size] != 0) {
                    g_renderer.EnableModulate();
                    g_renderer.SetModulationColor
                        (_data[col+row*_size]);
                    SetPos(x+col*GetWidth()*GetScaleX(),
                        y-row*GetHeight()*GetScaleY());
                    Sprite::Draw();
                    g_renderer.DisableModulate();
                }
            }
        }
    }
}

```

```
    SetPos(x,y);  
}
```

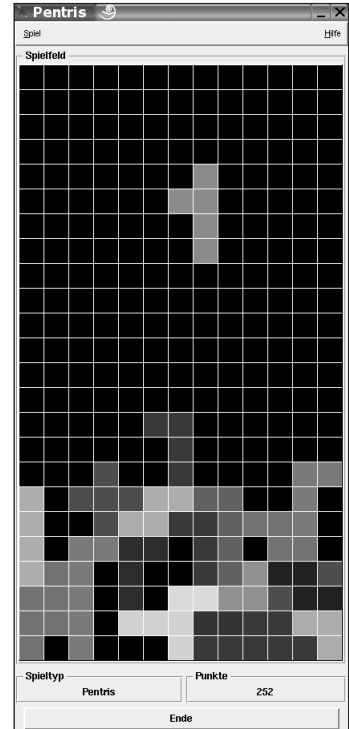
Analicemos detalladamente cada parte del código fuente comenzando por la definición del arreglo constante de piezas:

```
unsigned char pieceType[][1+(MAX_PIECE_SIZE*MAX_PIECE_SIZE)] = {  
    {2,  
     1, 1,  
     1, 1},  
    {3,  
     0, 1, 0,  
     1, 1, 1,  
     0, 0, 0},  
    {3,  
     1, 0, 0,  
     1, 1, 0,  
     0, 1, 0},  
    {3,  
     0, 1, 0,  
     1, 1, 0,  
     1, 0, 0},  
    {3,  
     0, 1, 0,  
     0, 1, 0,  
     0, 1, 1},  
    {3,  
     0, 1, 0,  
     0, 1, 0,  
     1, 1, 0},  
    {4,  
     0, 1, 0, 0,  
     0, 1, 0, 0,  
     0, 1, 0, 0,  
     0, 1, 0, 0}  
};
```

Definimos una matriz constante que hará las veces de repositorio de piezas. El arreglo es de dos dimensiones, donde la primera componente indicará la pieza por

mostrar mientras que la segunda alojará la información de la pieza. El tamaño de la segunda componente estará dado por el tamaño del lado de la matriz definido por **MAX\_PIECE\_SIZE** elevado al cuadrado (por eso, lo multiplicamos por sí mismo **MAX\_PIECE\_SIZE \* MAX\_PIECE\_SIZE**) sumándole uno dado. Como ya vimos, deberemos guardar como primer elemento el tamaño del lado de la matriz que codifica la pieza.

**Figura 9.** Modificando los valores del arreglo, podríamos crear una versión del Tetris con piezas formadas por cinco cuadrados en vez de cuatro.



Pasemos ahora a analizar los métodos de la clase:

```

DWORD Piece::GetColorByType(int type) {
    DWORD color = 0xFF000000;
    switch(type) {
        case 0:
            color = 0xFFFF0000;
            break;

        case 1:
            color = 0xFF00FF00;
            break;

        case 2:
            color = 0xFF0000FF;
            break;

        case 3:
            color = 0xFFFFFFFF;
            break;

        case 4:
    
```

```

        color = 0xFF00FFFF;
        break;
    case 5:
        color = 0xFFFF00FF;
        break;
    case 6:
        color = 0xFF601060;
        break;
}
return color;
}

```

Como podemos ver, simplemente, decidimos el color según el tipo de pieza.

```

void Piece::SetType(int type, DWORD color) {

    // Almaceno el tamaño de lado de la pieza
    _size = pieceType[type][0];
}

```

Almacenamos el tamaño de la pieza que será el primer elemento del arreglo que contiene su codificación.

```

    _type = type;
    _color = color;
}

```

Guardamos el tipo y el color de la pieza.

```

for (int row=0; row<MAX_PIECE_SIZE; row++)
    for (int col=0; col<MAX_PIECE_SIZE; col++)
        _data[col+row*_size] = pieceType[type]
            [(col+row*_size)+1]*_color;
}

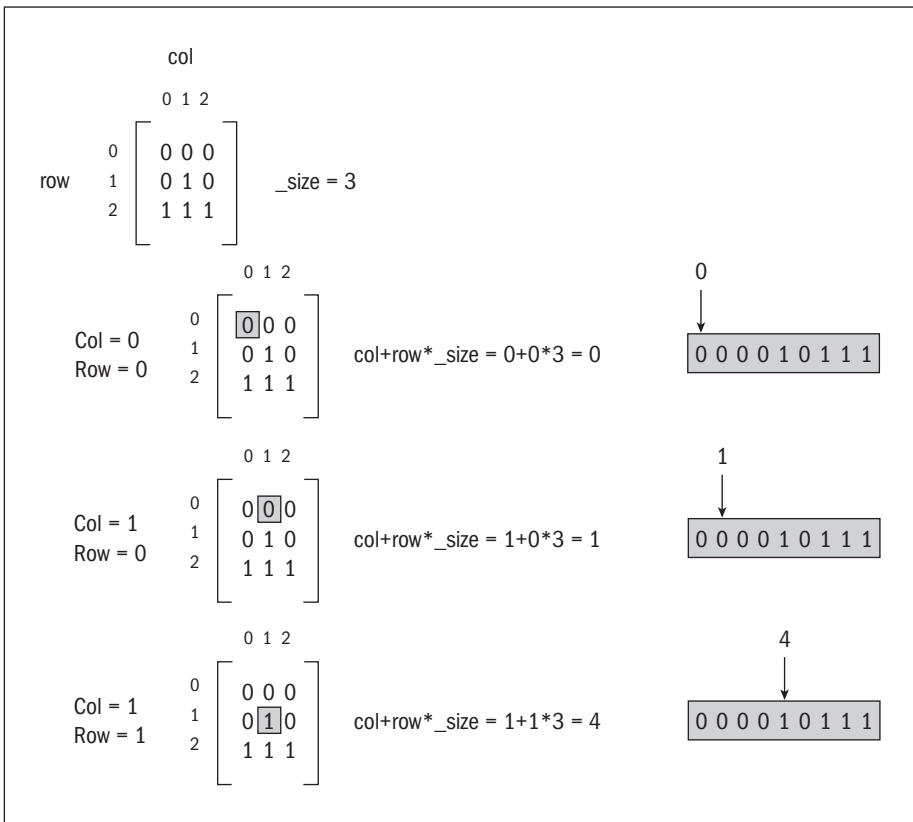
```

Copiamos la pieza del arreglo constante **pieceType** a nuestro arreglo **\_data** que alojará su información y su color para luego permitir modificarlo en cada rotación. Para esto, recorreremos el arreglo constante de la pieza y almacenamos cada uno de los valores multiplicados por el color (porque la información de la pieza en el arreglo constante está definido por ceros y unos).

```
_data[col+row*_size] = pieceType[type][(col+row*_size)+1]*_color;
```

Si analizamos mejor la forma en que recorreremos los arreglos, veremos que utilizamos la expresión `col+row*_size`. Esto nos permitirá manipular un arreglo unidimensional como si fuera una matriz bidimensional por medio de columnas y filas, como indica la **Figura 10**.

Notemos que la expresión en el arreglo `pieceType` difiere de la de `_data`, dado que se le suma uno. Esto se debe a que `pieceType` contiene como primer componente el tamaño del arreglo que no copiaremos dentro de `_data` y, por lo tanto, debemos saltar.



**Figura 10.** Utilizando la expresión `col+row*_size`, podemos recorrer un arreglo unidimensional por medio de componentes de columna y fila, como si fuera una matriz bidimensional.

```
_col = 0;
_row = 0;
```

Inicializamos la posición en el mapa a cero.

```
    SetVisible(true);  
}
```

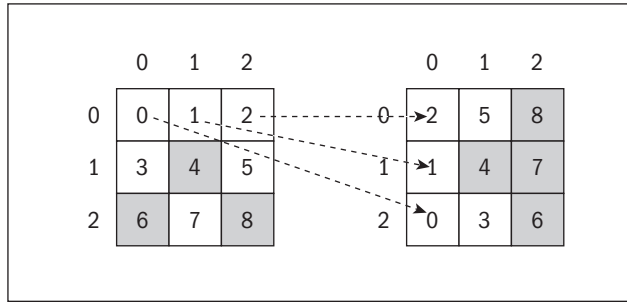
Hacemos visible la pieza.

```
void Piece::SetRandomType() {  
    int rnd = rand()%MAX_PIECES;  
  
    SetType(rnd, GetColorByType(rnd));  
}
```

Como vemos, el método **SetRandomType** toma un número aleatorio entre 0 y **MAX\_PIECES-1** por medio de la expresión **rand()%MAX\_PIECES** y luego la inicializa con su color haciendo uso del método **SetType** visto anteriormente.

```
void Piece::RotateCCW() {  
    int ncol, nrow;  
    DWORD newPiece[MAX_PIECE_SIZE*MAX_PIECE_SIZE];  
  
    // Roto la pieza  
    for (int row=0; row<_size; row++)  
        for (int col=0; col<_size; col++)  
        {  
            ncol = row;  
            nrow = _size-1-col;  
            newPiece[ncol + nrow * _size] = _data[col + row * _size];  
        }  
  
    // Copio la pieza temporal a la definitiva  
    memcpy((void *) _data, (const void *) newPiece, _size * _size  
        * sizeof(DWORD));  
}
```

Para implementar la rotación, deberemos trabajar sobre el arreglo de la pieza modificándolo según el tipo de giro deseado (horario o antihorario).



**Figura 11.** La rotación se desarrolla sobre el arreglo de la pieza.

Ahora debemos determinar qué columna/fila pasará a qué columna/fila. Para esto, tenemos que analizar la tabla que vemos en la **Figura 12**.

(COLUMNA, FILA) DESDE	(COLUMNA, FILA) HACIA
(0, 0)	(0, 2)
(0, 1)	(1, 2)
(0, 2)	(2, 2)
(1, 0)	(0, 1)
(1, 1)	(1, 1)
(1, 2)	(2, 1)
(2, 0)	(0, 0)
(2, 1)	(1, 0)
(2, 2)	(2, 0)

**Figura 12.** Analizando la tabla, podemos determinar qué columna/fila pasará a qué columna/fila.

Como se puede apreciar en la tabla, lo que en la primera columna son filas, en la segunda, terminan siendo columnas. Por lo tanto, obtenemos la primera relación:

```
ncol = row;
```

Nos resta calcular cuál es la fila en la cual termina cada elemento. Si analizamos nuevamente la tabla, inferimos que de 0 pasa a ser 2, de 1 a 1 y de 2 a 0. Por lo tanto, deducimos que:

```
nrow = 2 - col;
```

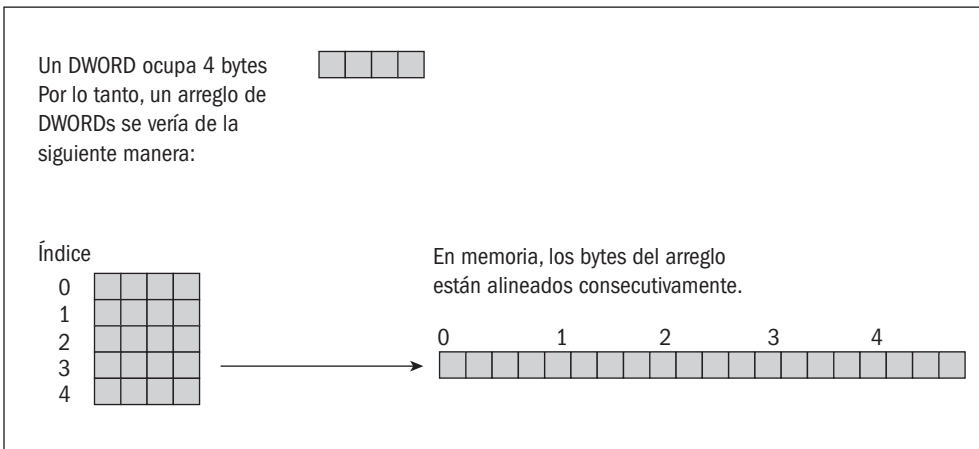
Luego determinamos un método general que sirva para todos los casos. Por lo tanto, analizamos de dónde sale el 2. Al tratarse de una pieza cuyo tamaño es 3, deducimos que el 2 es el tamaño menos uno. Por lo tanto, nos quedará:

```
nrow = _size - 1 - col;
```

Por último, debemos copiar la pieza obtenida en **newPiece** de nuevo a su arreglo original **\_data**:

```
// Copio la pieza temporal a la definitiva
memcpy((void *) _data, (const void *) newPiece, _size *
       _size * sizeof(DWORD));
```

Para ello, utilizamos la función **memcpy** que copiará la cantidad de bytes total del nuevo arreglo **newPiece**. Como indica la **Figura 13**, los bytes de memoria reservada por un arreglo (no importa si es unidimensional o de más de una dimensión) están alineados. Por lo tanto, debemos pasarle por parámetro a la función el tamaño del arreglo en bytes que, en este caso, estará dado por la expresión **\_size \* \_size \* sizeof(DWORD)** desde la posición de memoria a la que apunta el arreglo original **\_data**.



**Figura 13.** La memoria de un arreglo (unidimensional o bidimensional) se encuentra alojada de manera secuencial.

## III CONSTANTES

Modificando las constantes **MAX\_PIECES**, **MAX\_PIECE\_SIZE** y **pieceType** convenientemente, podemos crear un nuevo juego con menos piezas grandes o más piezas más pequeñas. Por ejemplo, podríamos crear un **Pentrix** formando piezas con la combinación de cinco cuadrados.

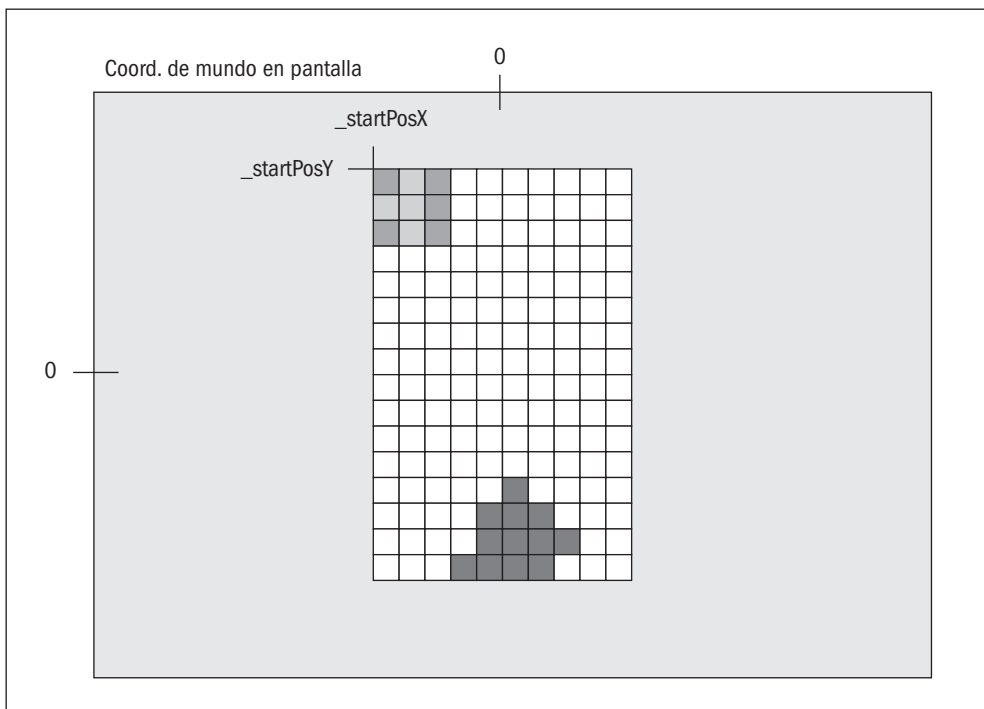
Análogamente, podemos rotar la pieza en sentido contrario por el método **RotateCW**.

Veamos ahora el método **Update**:

```
void Piece::Update(float dt) {
    Sprite::Update(dt);

    SetPos(_startPosX+_col*GetWidth()*GetScaleX(), _startPosY-
        _row*GetHeight()*GetScaleY());
}
```

Lo que haremos será tomar las variables **\_startPosX** y **\_startPosY** que indicarán la posición donde deberá comenzar a dibujarse la pieza en coordenadas de mundo. Luego sumaremos la columna (**\_col**) y la fila (**\_row**) dadas en coordenadas locales de la matriz, multiplicadas por el ancho y el alto, respectivamente, y por la escala del sprite. De esta manera, transformaremos la posición dentro de la matriz en coordenadas de mundo para poder mostrar cada cuadrado de la pieza en la posición adecuada respecto de su tamaño y de su escala. Los valores de **\_startPosX** y **\_startPosY** deberán coincidir con la posición del mapa.



**Figura 14.** Debemos indicar en qué posición de mundo se comenzará a dibujar la pieza.

Por último, tenemos el método **Draw**:

```
void Piece::Draw() {  
    float x = GetPosX();  
    float y = GetPosY();
```

Tomamos la posición actual de la pieza en coordenadas de mundo.

```
    if (GetVisible()) {  
        for (int row=0; row<_size; row++) {  
            for (int col=0; col<_size; col++) {  
                if (_data[col+row*_size] != 0) {
```

Si la pieza es visible, entonces, recorreremos el arreglo buscando algún componente diferente de cero, el cual deberá ser mostrado en pantalla.

```
                g_renderer.EnableModulate();  
                g_renderer.SetModulationColor(_data  
                    [col+row*_size]);
```

Activamos la modulación del color y el color de la pieza.

```
                SetPos(x+col*GetWidth()*GetScaleX(),y-row*GetHeight()*GetScaleY());
```

Luego realizamos el cálculo de posición análogo al del método **Update** con la diferencia de que éste se realizará respecto de la posición actual de la pieza y no respecto de su posición inicial, porque deberá mostrar cada cuadradito que compone la pieza por separado.

```
                Sprite::Draw();  
                g_renderer.DisableModulate();  
            }  
        }  
    }  
}
```

Dibujamos el cuadrado correspondiente y deshabilitamos la modulación del color.

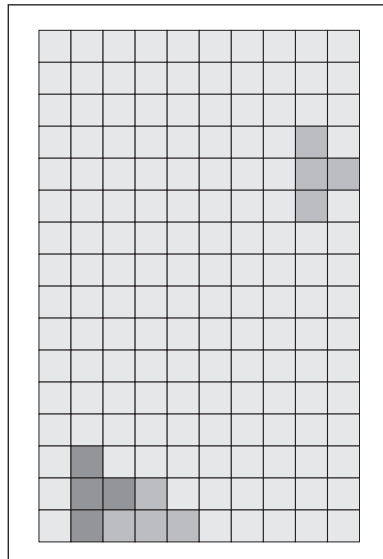
```
SetPos(x,y);  
}
```

Por último, volvemos la pieza a su posición original, dado que fue modificada anteriormente para mostrar cada uno de los recuadros.

Quedará como ejercicio probar la pieza para comprender mejor su funcionamiento.

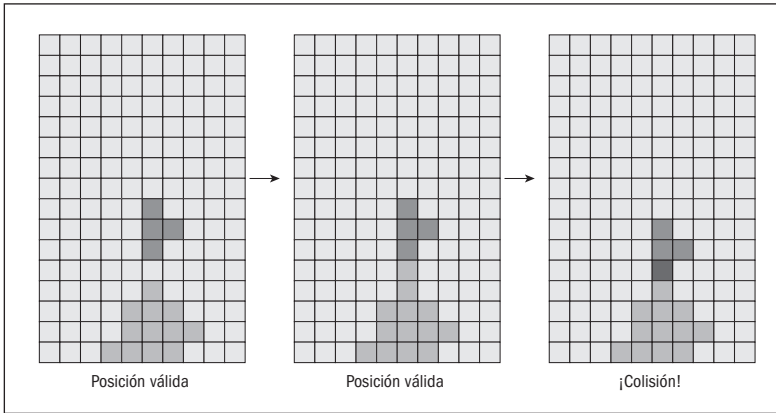
## LA CLASE BOARDMAP

La clase **BoardMap** será la encargada de manipular el tablero o mapa del juego y de las verificaciones de colisión. Cada vez que seleccionamos una pieza y la lanzamos desde el borde superior, verificaremos en cada chequeo de reglas si colisiona o no con alguna otra pieza o con algún extremo del mapa.



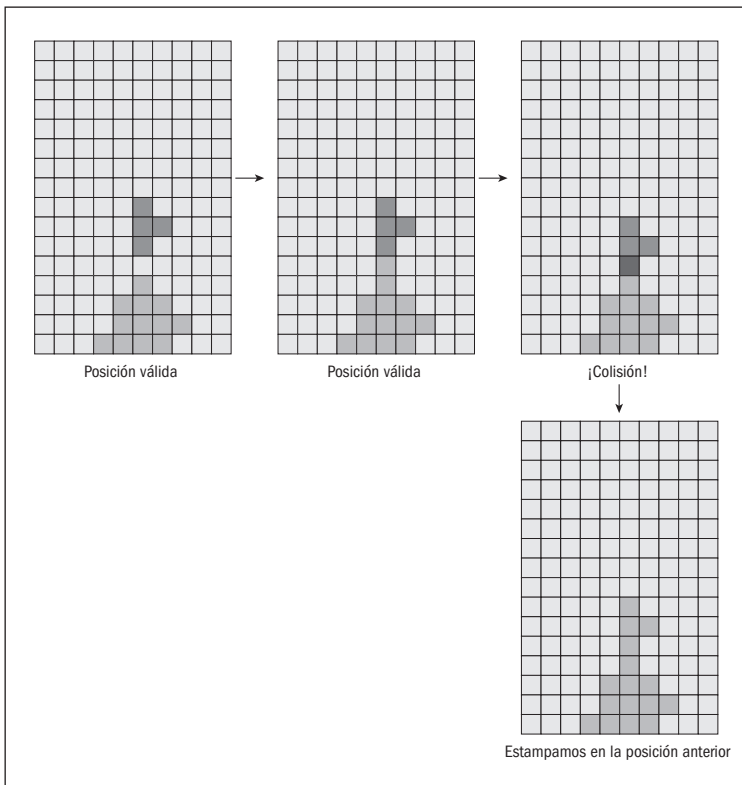
**Figura 15.** Verificaremos si la pieza colisiona con alguna otra pieza o con algún extremo del tablero.

A grandes rasgos, podríamos establecer que, cuando la pieza no colisiona contra nada dentro del mapa, la posición hacia la cual deseamos moverla es válida y podremos actualizar definitivamente su posición.



**Figura 16.** Si la pieza no colisionó, actualizamos su posición definitivamente.

En un determinado momento, la pieza colisionará contra otra o contra el borde inferior del mapa; llegada esta situación, se procederá a estamparla, es decir, se copiará la información que describe la pieza en el arreglo que representa el mapa (en la última posición válida en que quedó), como vemos en la **Figura 17**.



**Figura 17.** Si la pieza colisionó, la estampamos en su posición inmediatamente anterior al evento.

Luego el objeto **pieza** se reutilizará, seleccionando otra cualquiera (al azar) y volviéndola a soltar desde el borde superior del mapa.

Veamos, entonces, en qué consiste la clase **BoardMap**, comenzando por su archivo de cabecera:

```
#pragma once

#include "zakengine/zak.h"

#include "piece.h"

using namespace zak;

class BoardMap : public Sprite {

public:
    static const int COLS = 10;
    static const int ROWS = 20;

    BoardMap(void);
    ~BoardMap(void);

    // Dibuja el mapa
    void Draw();

    // Actualiza
    void Update(float dt);

    // Verifica si la pieza colisiona contra el tablero
    bool CollideWith(Piece & piece);

    // Copia la pieza en el tablero
    void StampPiece(Piece & piece);

    // Limpia una línea específica del tablero
    void ClearLine(int row);

    // Busca líneas formadas en el tablero
```

```
int CheckForLines();

// Reseteo el tablero
void Reset();

private:
    DWORD _map[COLS][ROWS];
};
```

Como podemos ver, la clase **BoardMap** hereda de la clase **Sprite** puesto que deberá dibujar en pantalla cada bloque estampado en él.

```
static const int COLS = 10;
static const int ROWS = 20;
```

Tendrá dos constantes estáticas llamadas **COLS** y **ROWS** que incluirán la cantidad de columnas y filas que tendrá el mapa. Cambiando estos valores, podremos modificar su tamaño.

```
// Dibuja el mapa
void Draw();

// Actualiza
void Update(float dt);
```

Además sobrecargaremos los métodos **Draw** y **Update** puesto que, como veremos más adelante, debemos dibujar el mapa recorriendo cada elemento del arreglo.

```
// Verifica si la pieza colisiona contra el tablero
bool CollideWith(Piece & piece);
```

Dada una pieza, esta clase chequeará si colisiona o no con el mapa. En caso afirmativo, devolverá **true**, y **false** en caso contrario.

```
// Copia la pieza en el tablero
void StampPiece(Piece & piece);
```

En el caso de que la pieza hubiera colisionado con el borde inferior o, al bajar, con otra pieza, deberá ser estampada por medio de este método.

```
// Limpia una línea específica del tablero
void ClearLine(int row);
```

Como veremos más adelante, una vez que una línea horizontal fue completada, ésta deberá ser eliminada para evitar llegar al extremo superior del mapa.

```
// Busca líneas formadas en el tablero
int CheckForLines();
```

Este método chequeará si hubo líneas horizontales completadas. En ese caso, serán eliminadas por medio del método **ClearLine**, y devolverá la cantidad encontrada. De lo contrario, devolverá **0**.

```
// Reseteo el tablero
void Reset();
```

Llenará la matriz que define el mapa de ceros.

Pasemos ahora a analizar el código fuente de la clase:

```
#include "boardmap.h"

BoardMap::BoardMap(void) {
    Reset();
}
```

## III CONSTANTES

Por medio de la utilización de las constantes **COLS** y **ROWS**, tenemos la posibilidad de definir la cantidad de columnas y de filas que contiene el mapa del juego.

```
BoardMap::~BoardMap(void) {

}

// Dibuja el mapa
void BoardMap::Draw() {
    float x = GetPosX();
    float y = GetPosY();

    g_renderer.EnableModulate();

    for (int i=0; i<COLS; i++) {
        for (int j=0; j<ROWS; j++) {
            if (_map[i][j] != 0) {
                g_renderer.SetModulationColor(_map[i][j]);
            } else {
                g_renderer.SetModulationColor(0xFF606060);
            }
            SetPos(x+i*GetWidth()*GetScaleX(),y-j*GetHeight()
                *GetScaleY());
            Sprite::Draw();
        }
    }
    g_renderer.DisableModulate();

    SetPos(x,y);
}

void BoardMap::Update(float dt){
    Sprite::Update(dt);
}

// Verifica si la pieza colisiona contra el tablero
bool BoardMap::CollideWith(Piece & piece) {
    for (int row=0; row<piece._size; row++)
        for (int col=0; col<piece._size; col++)
        {
            // Verifico que la pos. del mapa por afectar
            esté dentro de él
            if (piece._col+col >= COLS || piece._col+col < 0)
```

```

        {
            // Me estoy yendo del mapa por la derecha o
            // por la izquierda
            if (piece._data[col + row * piece._size] != 0)
                return true;
        }
        else if (piece._row+row >= ROWS || piece._row+row < 0)
        {
            // Estoy tocando el piso del mapa o
            // estoy por encima del mapa -> ¿Este es un
            // caso posible?
            if (piece._data[col + row * piece._size] != 0)
                return true;
        }
        else if (_map[piece._col+col][piece._row+row] != 0 &&
            piece._data[col + row * piece._size] != 0)
            // Una parte de la pieza toca otra pieza del mapa
            return true;
    }

    return false;
}

// Copia la pieza en el tablero
void BoardMap::StampPiece(Piece & piece) {
    for (int row=0; row<piece._size; row++)
        for (int col=0; col<piece._size; col++)
        {
            if (piece._col+col >= 0 && piece._col+col < COLS &&
                piece._row+row >= 0 && piece._row+row < ROWS &&
                piece._data[col + row * piece._size] != 0)
                _map[piece._col+col][piece._row+row] = piece.
                    _data[col + row * piece._size];
        }

    piece.SetVisible(false);
}

// Limpia una línea específica del tablero
void BoardMap::ClearLine(int row) {

```

```
// Copio las líneas hacia abajo
for (int i=row; i>0; i-)
    for (int col=0; col<COLS; col++)
        _map[col][i] = _map[col][i-1];

// Elimino la primera línea
for (int col=0; col<COLS; col++)
    _map[col][0] = 0;
}
```

```
// Busca líneas formadas en el tablero
int BoardMap::CheckForLines() {
    bool isALine;
    int    lines=0;

    for (int row=ROWS; row>=0; row-)
    {
        isALine = true;
        for (int col=0; col<COLS; col++)
        {
            if (_map[col][row] == 0)
            {
                isALine = false;
                break;
            }
        }

        if (isALine)
        {
            lines++;
            ClearLine(row);
            row++;
        }
    }
    return lines;
}
```

```
// Reseteo el tablero
void BoardMap::Reset() {
    for (int i=0; i<COLS; i++)
```

```

        for (int j=0; j<ROWS; j++)
            _map[i][j] = 0;
    }

```

Analizamos los métodos más importantes:

```

// Verifica si la pieza colisiona contra el tablero
bool BoardMap::CollideWith(Piece & piece) {
    for (int row=0; row<piece._size; row++)
        for (int col=0; col<piece._size; col++)
        {
            // Verifico que la pos. del mapa por afectar esté
            dentro de él
            if (piece._col+col >= COLS || piece._col+col < 0)
            {
                // Me estoy yendo del mapa por la derecha o
                por la izquierda
                if (piece._data[col + row * piece._size] != 0)
                    return true;
            }
            else if (piece._row+row >= ROWS || piece._row+row < 0)
            {
                // Estoy tocando el piso del mapa o
                // estoy por encima del mapa -> ¿Éste es un
                caso posible?
                if (piece._data[col + row * piece._size] != 0)
                    return true;
            }
            else if (_map[piece._col+col][piece._row+row] !=
                0 && piece._data[col + row * piece._size] != 0)
                // Una parte de la pieza toca otra pieza del mapa
                return true;
        }
    return false;
}

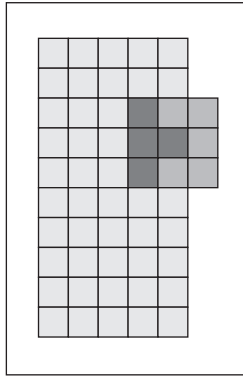
```

Como se puede apreciar en el listado, realizamos una iteración en todas las posiciones de la pieza por medio de dos bucles anidados. Notemos que no recorremos el mapa completo, puesto que sería más lento.

Por cada posición de la pieza verificaremos:

- No quedar fuera del mapa por izquierda ni por derecha.
- No quedar fuera del mapa por abajo ni por arriba.
- No quedar superpuesto a otra pieza ya estampada en el mapa.

Sin embargo, vale una aclaración muy importante: nuestra pieza, trabajada como si fuese un arreglo de dos dimensiones cuadrado, posee una forma irregular. Por lo tanto, no todos los elementos del arreglo son necesariamente parte de la pieza. Como vemos en la **Figura 17**, la pieza parece quedar fuera del mapa, pero no es así dado que la parte que quedó afuera no contiene ninguna porción sólida.



**Figura 18.** No todos los elementos del arreglo son necesariamente parte de la pieza.

```
// Copia la pieza en el tablero
void BoardMap::StampPiece(Piece & piece) {
    for (int row=0; row<piece._size; row++)
        for (int col=0; col<piece._size; col++)
        {
            if (piece._col+col >= 0 && piece._col+col < COLS &&
                piece._row+row >= 0 && piece._row+row < ROWS &&
                piece._data[col + row * piece._size] != 0)
                _map[piece._col+col][piece._row+row] =
                    piece._data[col + row * piece._size];
        }
    piece.SetVisible(false);
}
```

Estampar una pieza en el mapa es bastante simple. Bastará con recorrer el arreglo con dos bucles anidados (de manera análoga a la que desarrollamos anteriormente) en busca de algún elemento diferente de cero (es decir, sólido).

```
_map[piece._col+col][piece._row+row] = piece._data[col + row * piece._size];
```

En el caso de tratarse de un elemento sólido, lo copiamos en la posición correspondiente del mapa que estará dada por la columna y la fila en las que se encuentra la pieza (**piece.\_col** y **piece.\_row**) dentro del mapa, sumándole la columna y la fila correspondiente que estemos leyendo dentro del arreglo (variables locales **col** y **row** declaradas en los bucles **for**).

```
// Busca líneas formadas en el tablero
int BoardMap::CheckForLines() {
    bool isALine;
    int    lines=0;
```

Buscar si una línea fue completada resulta trivial. Recorremos la matriz del mapa. Mientras no haya un cero en alguno de los elementos, seguimos recorriendo.

```
for (int row=ROWS; row>=0; row-)
{
    isALine = true;
```

Inicializamos en **true** la bandera (o *flag*) **isALine**, suponiendo que hay línea para cambiarla a **false** en caso contrario.

```
for (int col=0; col<COLS; col++)
{
    if (_map[col][row] == 0)
    {
        isALine = false;
        break;
    }
}
```

Si encontramos al menos un cero, no existe línea y, por lo tanto, cortamos la ejecución del bucle por medio de una sentencia **break**.

```
}
if (isALine)
```

```

    {
        lines++;
        ClearLine(row);
        row++;
    }

```

Si hubo alguna línea (es decir que **isALine** en este momento es igual a **true**), incrementamos en uno la variable **lines** para almacenar la cantidad encontrada, eliminamos esa línea por medio del método **ClearLines** que veremos a continuación y saltamos a la fila siguiente forzando a **row** a incrementarse.

```

    }
    return lines;
}

```

Por último, retornamos la cantidad de filas encontradas.

```

// Limpia una línea específica del tablero
void BoardMap::ClearLine(int row) {
    // Copio las líneas hacia abajo
    for (int i=row; i>0; i-)
        for (int col=0; col<COLS; col++)
            _map[col][i] = _map[col][i-1];
    // Elimino la primera línea
    for (int col=0; col<COLS; col++)
        _map[col][0] = 0;
}

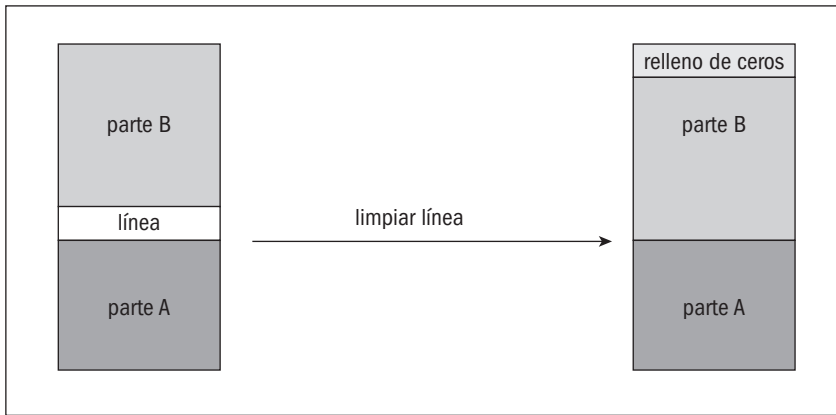
```

Borrar una línea es un poco más complicado. Para darnos una idea de qué se trata, veamos el ejemplo de la **Figura 18**.



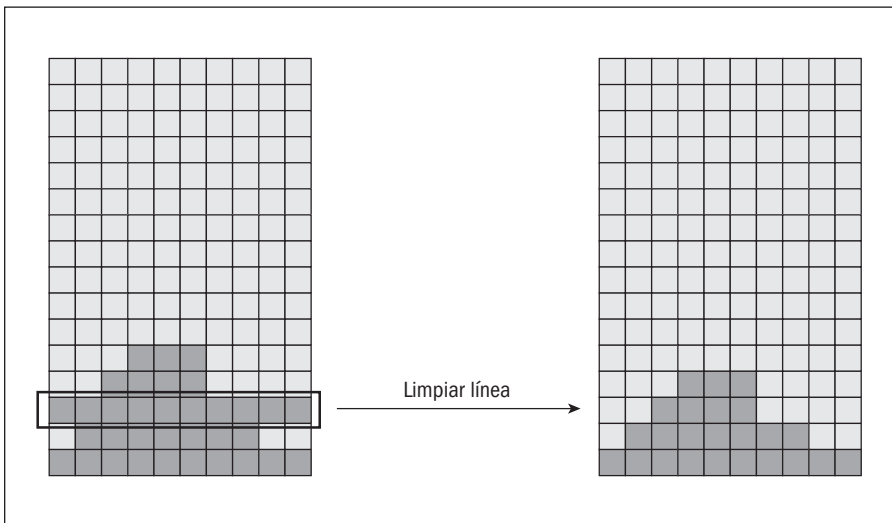
## HISTORIA DEL TETRIS

El Tetris es un juego muy popular desarrollado originalmente por **Alexey Pajitnov** en junio de 1985, mientras trabajaba en Moscú en el Centro de Computación Dorodnicyn de la Academia de Ciencia de la Unión Soviética.



**Figura 19.** Borrar una línea consiste en copiar parte del mapa hacia otra posición.

Como podemos ver, detectada la línea, debemos copiar parte del tablero a otra posición dentro de él. También observamos que parte del tablero permanece inalterado y que debemos modificar la línea superior y fijarla en cero (valor que indica vacío en nuestra implementación). Veamos la **Figura 19** para aclarar el concepto.



**Figura 20.** Viendo un ejemplo práctico, comprenderemos mejor de qué se trata.

Para esto, bastarán dos iteraciones:

```
// Copio las líneas hacia abajo
for (int i=row; i>0; i-)
    for (int col=0; col<COLS; col++)
        _map[col][i] = _map[col][i-1];
```

En esta iteración, hacemos la copia de cada elemento de la fila superior sobre la actual.

```
// Elimino la primera línea
for (int col=0; col<COLS; col++)
    _map[col][0] = 0;
```

En la segunda iteración, llenamos la última fila de ceros.

En cuanto al trazado del mapa, es muy similar al de la pieza:

```
// Dibuja el mapa
void BoardMap::Draw() {
    float x = GetPosX();
    float y = GetPosY();
```

Tomamos la posición actual del mapa y lo guardamos en las variables temporales **x** e **y**.

```
g_renderer.EnableModulate();
```

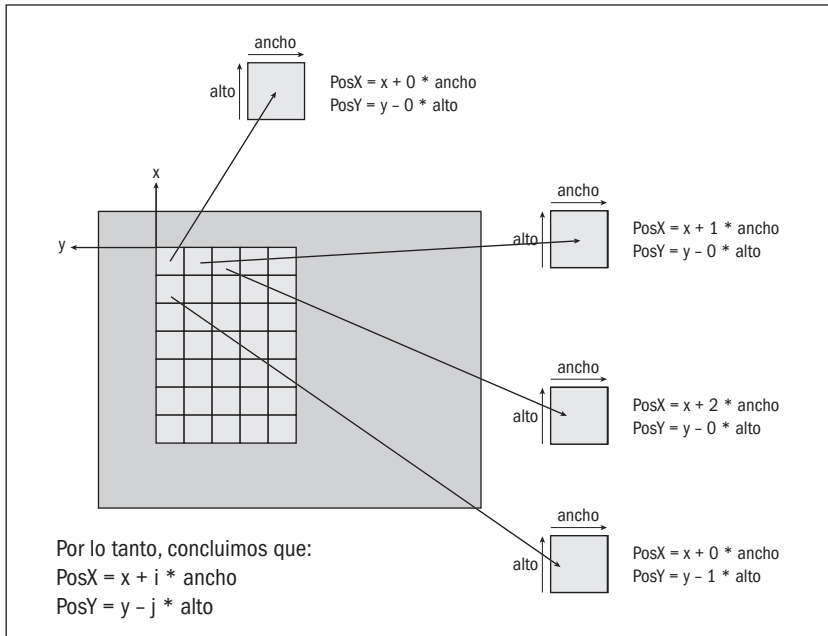
Activamos la modulación del color puesto que dibujaremos cada trozo del mapa con el color correspondiente a la pieza que se encuentre estampada o al vacío.

```
for (int i=0; i<COLS; i++) {
    for (int j=0; j<ROWS; j++) {
        if (_map[i][j] != 0) {
            g_renderer.SetModulationColor(_map[i][j]);
        } else {
            g_renderer.SetModulationColor(0xFF606060);
        }
    }
}
```

Recorremos la matriz del mapa verificando si el elemento actual es vacío (igual a cero) o no. En el caso de no ser vacío, indicamos el almacenado en el mapa. En caso contrario, elegimos un color arbitrario (en este caso **0xFF606060**).

```
SetPos(x+i*GetWidth()*GetScaleX(),y-j*GetHeight()
    *GetScaleY());
```

Indicamos la posición donde debe dibujarse el elemento del mapa. Para esto, tomamos la posición que almacenamos anteriormente en **x** y **y** y le sumamos los contadores **j** e **i** del bucle multiplicados por el ancho y el alto, respectivamente, y luego por la escala que, si bien por defecto es igual a **1.0f**, debemos tenerla en cuenta por si más adelante deseamos modificarla.



**Figura 21.** En el diagrama, suprimimos la escala para simplificar la explicación.

```
Sprite::Draw();
```

Ahora dibujamos el elemento con el color determinado y en su posición.

```
    }
}
g_renderer.DisableModulate();
```

Deshabilitamos la modulación de color, puesto que ya finalizamos el bucle y, por tanto, el trazado.

```
    SetPos(x,y);
}
```

Por último, volvemos a indicar la posición original almacenada en **x** e **y**.

Queda como ejercitación integrar el mapa en el juego para entender, de manera práctica, cada uno de los métodos.

## LA CLASE PRINCIPAL INGAME

Habiendo implementado las funciones más importantes del juego (rotaciones, verificaciones de colisión, etcétera), sólo nos resta analizar cómo debe funcionar el juego en sí mismo.



**Figura 22.** Mientras programamos un clon, nunca debemos perder de vista el juego original para mantener su esencia.

El objetivo aquí será mover la ficha activa hacia el borde inferior del tablero e invocar la verificación de colisiones, estampa al tablero y verificación de líneas armadas.

Veamos entonces el archivo cabecera de la clase **InGame**:

```
#pragma once

#include "zakengine/zak.h"

#include "boardMap.h"
```

```
#include "piece.h"

using namespace zak;

#define GAME_STATE_NONE          0
#define GAME_STATE_EXIT         1
#define GAME_STATE_NEXTLEVEL 2

#define START_POS_X              -100.0f
#define START_POS_Y              100.0f

#define NEXT_PIECE_POS_X        -200.0f
#define NEXT_PIECE_POS_Y        100.0f

#define FALL_INTERVAL            1000.0f
#define MOVE_INTERVAL            80.0f

class InGame {
public:
    bool Initialize(int level);
    bool Shutdown();

    void SetState(int state) { _state = state; }
    int GetState() { return _state; }

    void Update(float dt);
    void Draw();

    InGame();
    ~InGame();

private:
    int          _state;
    Piece        _piece;
    Piece        _nextPiece;
    BoardMap     _boardMap;
    float        _fallInterval;
    int          _score;
};
```

Como vemos, los métodos son los mismos que en los casos desarrollados en capítulos anteriores. Por lo tanto, prestemos atención a las nuevas constantes y propiedades.

En cuanto a las constantes, tendremos las siguientes:

```
#define START_POS_X          -100.0f
#define START_POS_Y          100.0f
```

Las constantes **START\_POS\_X** y **START\_POS\_Y** identificarán la posición del mapa en pantalla.

```
#define NEXT_PIECE_POS_X     -200.0f
#define NEXT_PIECE_POS_Y     100.0f
```

Luego, **NEXT\_PIECE\_POS\_X** y **NEXT\_PIECE\_POS\_Y** tendrán la posición en pantalla donde deberá dibujarse la pieza que utilizaremos luego de estampar la actual.

```
#define FALL_INTERVAL        1000.0f
#define MOVE_INTERVAL        80.0f
```

Por último, vemos **FALL\_INTERVAL** y **MOVE\_INTERVAL**, que serán los intervalos de tiempo para el descenso y el movimiento de la pieza por parte del usuario, respectivamente.

Veamos ahora las propiedades:

```
Piece      _piece;
```



## CONSTANTES

Modificando la constante **FALL\_INTERVAL**, podemos acelerar o desacelerar el intervalo de descenso de la pieza. Además, si cambiamos la constante **MOVE\_INTERVAL**, modificaremos la velocidad de reacción de la pieza respecto de las teclas.

El objeto **\_piece** alojará la pieza que cae actualmente y que manipulará el jugador.

```
Piece          _nextPiece;
```

La instancia **\_nextPiece** mostrará (como su nombre lo indica) la pieza que aparecerá luego de estampar la actual.

```
BoardMap      _boardMap;
```

El objeto **\_boardMap** identificará el mapa actual.

```
float         _fallInterval;
```

La variable **\_fallInterval** determinará el intervalo de tiempo al que irá descendiendo la pieza.

```
int           _score;
```

Por último, tenemos la variable **\_score**, que contendrá el puntaje adquirido hasta el momento.

Ahora que conocemos de qué se tratan las constantes y las propiedades de la clase, pasemos al código fuente:

```
#include "InGame.h"

InGame::InGame() {
}

InGame::~InGame() {
}

bool InGame::Initialize(int level) {

    // Inicializamos el estado
```

```
    _state = GAME_STATE_NONE;

    if (!_piece.LoadIni("data/graphics/piece.spr"))
        return false;

    if (!_nextPiece.LoadIni("data/graphics/piece.spr"))
        return false;

    if (!_boardMap.LoadIni("data/graphics/piece.spr"))
        return false;

    _boardMap.Reset();
    _piece.SetRandomType();

    _piece.SetStartPos(START_POS_X, START_POS_Y);
    _nextPiece.SetStartPos(NEXT_PIECE_POS_X, NEXT_PIECE_POS_Y);

    _boardMap.SetPos(START_POS_X, START_POS_Y);

    _nextPiece.SetRandomType();

    _score = 0;

    return true;
}

bool InGame::Shutdown() {
    return true;
}

void InGame::Update(float dt) {
    static float fallAccumTime = 0;
    static float moveAccumTime = 0;

    _fallInterval = FALL_INTERVAL;

    if (KeyDown(DIK_UP)) {
        _piece.RotateCCW();
        if (_boardMap.CollideWith(_piece))
            _piece.RotateCW();
    }
}
```

```
    }

    if (KeyPressed(DIK_DOWN)) {
        _fallInterval = MOVE_INTERVAL;
    }

    moveAccumTime += dt;

    if (moveAccumTime >= MOVE_INTERVAL) {
        moveAccumTime = MOVE_INTERVAL;

        if (KeyPressed(DIK_LEFT)) {
            moveAccumTime -= MOVE_INTERVAL;
            _piece.MoveLeft();
            if (_boardMap.CollideWith(_piece))
                _piece.MoveRight();
        }
        if (KeyPressed(DIK_RIGHT)) {
            moveAccumTime -= MOVE_INTERVAL;
            _piece.MoveRight();
            if (_boardMap.CollideWith(_piece))
                _piece.MoveLeft();
        }
    }

    fallAccumTime += dt;

    if (fallAccumTime >= _fallInterval) {
        fallAccumTime = 0;

        _piece.MoveDown();
        if (_boardMap.CollideWith(_piece)) {
            int lines=0;

            _piece.MoveUp();
            _boardMap.StampPiece(_piece);

            lines = _boardMap.CheckForLines();

            if (lines == 1) {
```

```
        _score += 100;
    } else if (lines == 2) {
        _score += 300;
    } else if (lines == 3) {
        _score += 600;
    } else if (lines == 4) {
        _score += 1000;
    }

    _piece.SetType(_nextPiece.GetType(), _piece.GetColorByType
        (_nextPiece.GetType()));

    if (_boardMap.CollideWith(_piece)) {
        _state = GAME_STATE_EXIT;
    }

    _nextPiece.SetRandomType();
}
}

// Si presionamos escape, salimos al menu
if (KeyDown(DIK_ESCAPE))
    _state = GAME_STATE_EXIT;

_piece.Update(dt);
_nextPiece.Update(dt);
_boardMap.Update(dt);
}

void InGame::Draw() {
    wstringstream ss;

    _boardMap.Draw();
    _piece.Draw();
    _nextPiece.Draw();

    ss << "Score: " << _score;

    g_renderer.DrawString(ss.str(), 10, 10, 780, 580, ZAK_TEXT_RIGHT);
}
```

¿Mucho código? No tanto, pero mejor vayamos por partes para entenderlo, comenzando por el método **Initialize**:

```
bool InGame::Initialize(int level) {

    // Inicializamos el estado
    _state = GAME_STATE_NONE;

    if (!_piece.LoadIni("data/graphics/piece.spr"))
        return false;

    if (!_nextPiece.LoadIni("data/graphics/piece.spr"))
        return false;

    if (!_boardMap.LoadIni("data/graphics/piece.spr"))
        return false;
```

Comenzamos por inicializar la propiedad `_state` en `GAME_STATE_NONE` para indicar al bucle principal que estamos jugando. Luego intentamos cargar los sprites correspondientes a la pieza, la pieza siguiente y el mapa. Si algo saliera mal, retornamos **false**.

```
_boardMap.Reset();
```

Vaciamos el mapa de cualquier elemento que contenga.

```
_piece.SetRandomType();
```

Tomamos una pieza al azar por medio del método **SetRandomType**.

```
_piece.SetStartPos(START_POS_X, START_POS_Y);
_boardMap.SetPos(START_POS_X, START_POS_Y);
```

Inicializamos la posición inicial de la pieza y del mapa según las constantes `START_POS_X` y `START_POS_Y` para que, de esta forma, coincidan los valores lógicos dentro de los arreglos y lo que se ve en pantalla.

```
_nextPiece.SetRandomType();
_nextPiece.SetStartPos(NEXT_PIECE_POS_X, NEXT_PIECE_POS_Y);
```

Tomamos para la pieza siguiente una al azar e inicializamos su posición en la pantalla según las constantes **NEXT\_PIECE\_POS\_X** y **NEXT\_PIECE\_POS\_Y**.

```
_score = 0;

return true;
}
```

Por último, inicializamos el puntaje a cero y retornamos **true** si todo salió bien.

Analicemos, ahora, el método **Update**:

```
void InGame::Update(float dt) {
    static float fallAccumTime = 0;
    static float moveAccumTime = 0;
```

Declaramos dos variables estáticas que acumularán el tiempo transcurrido para determinar cuándo deberá moverse la pieza. Una será **fallAccumTime**, de la que dependerá la caída de la pieza. La otra, llamada **moveAccumTime**, medirá el tiempo para el movimiento dependiente del usuario.

```
_fallInterval = FALL_INTERVAL;
```

Por cada iteración, inicializamos la propiedad **\_fallInterval** según la constante **FALL\_INTERVAL**, puesto que sólo cambiará este valor si el usuario presionar la tecla correspondiente a la caída rápida.

```
if (KeyDown(DIK_UP)) {
    _piece.RotateCCW();
```

Si el usuario pulsó la tecla determinada por la constante **DIK\_UP** (flecha arriba), la pieza rotará en el contrasentido de las agujas del reloj por medio del método **RotateCCW**.

```

        if (_boardMap.CollideWith(_piece))
            _piece.RotateCW();
    }

```

Luego de haber rotado, verificaremos si esta acción provocó un choque, por medio del método **CollideWith** del objeto **\_boardMap** pasándole por parámetro una referencia a la pieza. De ser afirmativo, volvemos a la posición original rotando en el sentido contrario por medio del método **RotateCW**.

```

    if (KeyPressed(DIK_DOWN)) {
        _fallInterval = MOVE_INTERVAL;
    }

```

Si el usuario presionó la tecla definida por la constante **DIK\_DOWN** (tecla abajo), reduciremos el intervalo de caída dado por la propiedad **\_fallInterval** al valor indicado por la constante **MOVE\_INTERVAL**. De esta forma, el usuario podrá acelerar el descenso de la pieza.

```

    moveAccumTime += dt;

```

Acumulamos el tiempo transcurrido entre iteraciones en **moveAccumTime**.

```

    if (moveAccumTime >= MOVE_INTERVAL) {
        moveAccumTime = MOVE_INTERVAL;
    }

```

Luego si el valor del tiempo acumulado en **moveAccumTime** es mayor o igual que lo que indica **MOVE\_INTERVAL**, forzamos el valor al indicado por dicha constante.



## EL JUGADOR ESTÁ CONDENADO A PERDER

¿Es posible jugar al Tetris por siempre? Esta pregunta se encontró en una tesis de **John Brzustowski** en 1988 y fue investigada más recientemente por **Walter Kusters**. La conclusión obtenida fue que el jugador, inevitablemente, está condenado a perder. La razón se debe a que, si éste recibe una secuencia prolongada de piezas **Z** y **S**, será forzado a dejar un hueco en una esquina.

```

    if (KeyPressed(DIK_LEFT)) {
        moveAccumTime -= MOVE_INTERVAL;
        _piece.MoveLeft();
        if (_boardMap.CollideWith(_piece))
            _piece.MoveRight();
    }
    if (KeyPressed(DIK_RIGHT)) {

        moveAccumTime -= MOVE_INTERVAL;
        _piece.MoveRight();
        if (_boardMap.CollideWith(_piece))
            _piece.MoveLeft();
    }
}

```

Luego, si el usuario presionó las teclas de cursor derecha o izquierda, decrementamos el tiempo acumulado por la constante **MOVE\_INTERVAL**, movemos según la dirección indicada por medio de los métodos **MoveRight** y **MoveLeft**, respectivamente. Por último, chequeamos si el movimiento provocó alguna colisión con el mapa. En caso afirmativo, realizamos la acción contraria.

```
fallAccumTime += dt;
```

Acumulamos el tiempo transcurrido entre iteraciones en la variable **fallAccumTime**.

```

if (fallAccumTime >= _fallInterval) {
    fallAccumTime = 0;

    _piece.MoveDown();
}

```



## EL CREADOR DEL TETRIS

Alexey Pajitnov, nacido en 1956 en Rusia, es un ingeniero en computación que, con la ayuda de **Dmitry Pavlovsky** y **Vadim Gerasimov**, creó el Tetris en 1985. Además, desarrolló la secuela (no tan conocida) llamada **Welltris**, que tiene el mismo principio, pero en tres dimensiones.

Verificamos si el tiempo acumulado en **fallAccumTime** supera el intervalo indicado por la propiedad **\_fallInterval**. De ser así, igualamos a cero el contador y permitimos que la pieza descienda invocando a su método **MoveDown**.

```
if (_boardMap.CollideWith(_piece)) {
```

Si hubo una colisión, quiere decir que debemos estampar la pieza, dado que como descendía, significa que colisionó contra el borde inferior del mapa o contra otra pieza.

```
int lines=0;
```

Creamos e inicializamos la variable **lines** en cero. Esta variable alojará la cantidad de líneas que fueron completadas con esta acción.

```
_piece.MoveUp();
```

Dado que la pieza descendió y, por tanto, provocó una colisión, debemos moverla a su última posición válida. Para esto, invocamos al método **MoveUp**, que produce el resultado contrario.

```
_boardMap.StampPiece(_piece);
```

Ahora sí, estampamos la pieza en la ubicación correcta llamando al método **Stamp Piece** del mapa pasándole la pieza por parámetro.

```
lines = _boardMap.CheckForLines();
```

Chequeamos si la acción completó alguna línea.

```
if (lines == 1) {
    _score += 100;
} else if (lines == 2) {
    _score += 300;
} else if (lines == 3) {
    _score += 600;
```

```

    } else if (lines == 4) {
        _score += 1000;
    }

```

Verificamos la cantidad de líneas completadas e incrementamos el puntaje según sea conveniente.

```

    _piece.SetType(_nextPiece.GetType(),
        _piece.GetColorByType(_nextPiece.GetType()));

```

Debemos lanzar una nueva pieza según indica **\_nextPiece**. Para esto, invocamos al método **SetType** pasándole por parámetro el tipo y color de la siguiente pieza.

```

    if (_boardMap.CollideWith(_piece)) {
        _state = GAME_STATE_EXIT;
    }

```

Verificamos si la nueva pieza (que ahora se encuentra en el extremo superior, dado que el método **SetType** vuelve a la pieza a su posición inicial) provoca una colisión con el mapa. Esto significaría que las piezas estampadas en el mapa han llegado al extremo superior y, por tanto, el juego ha finalizado.

```

        _nextPiece.SetRandomType();
    }
}

```

Inicializamos la nueva pieza por otra aleatoria utilizando el método **SetRandomType**.

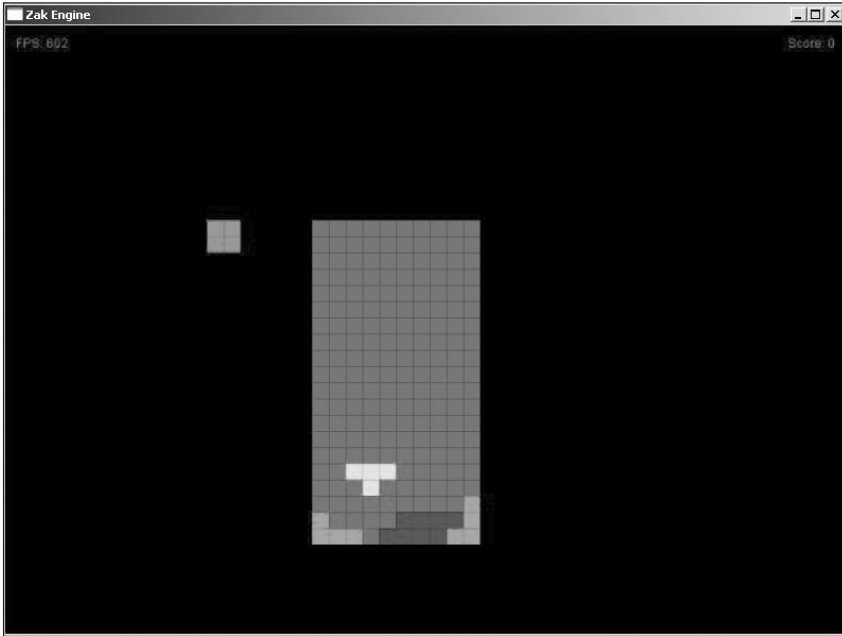
```

// Si presionamos escape, salimos al menu
if (KeyDown(DIK_ESCAPE))
    _state = GAME_STATE_EXIT;

    _piece.Update(dt);
    _nextPiece.Update(dt);
    _boardMap.Update(dt);
}

```

Por último verificamos si se presionó la tecla **Escape** para salir del juego y actualizamos todos los objetos de la escena.



**Figura 23.** Al correr el juego, debemos ver una pantalla como ésta.

¡Genial! Ya tenemos un juego estilo Tetris completamente funcional. En los próximos capítulos, avanzaremos en complejidad y aprenderemos nuevos conceptos.

## RESUMEN

En este capítulo, hemos aprendido el concepto de mapa y cómo se relaciona lo que está almacenado lógicamente en él y lo que se ve en pantalla. En los próximos capítulos, avanzaremos un poco más sobre este tema, analizando los mapas de mosaicos. ¡Además, veremos cómo agregar sonido y música, textos con fuentes creados a partir de bitmaps y mucho más!



## ACTIVIDADES

- 1** ¿Qué elementos contiene el arreglo que aloja las piezas?  
\_\_\_\_\_
- 2** ¿Por qué la matriz que aloja las piezas debe ser cuadrada?  
\_\_\_\_\_
- 3** Al chequear la colisión de la pieza con el mapa, ¿por qué el bucle recorre la pieza y no el mapa?  
\_\_\_\_\_
- 4** ¿En qué consiste imprimir una pieza en el mapa?  
\_\_\_\_\_
- 5** ¿En qué consiste eliminar una línea en el mapa?  
\_\_\_\_\_
- 6** Modificar el código para que, cada cierta cantidad de puntos, cambie de nivel modificando en cada uno la velocidad de descenso de las piezas.  
\_\_\_\_\_
- 7** Modificar el código para que permita dos jugadores con dos mapas.  
\_\_\_\_\_
- 8** Modificar las constantes MAX\_PIECES, MAX\_PIECE\_SIZE y pieceType de manera tal de crear un juego estilo Tetris, pero con piezas formadas por la combinación de cinco cuadrados (Pentris).  
\_\_\_\_\_