Interacción persona maquina
Curso 2013/2014

Tetris en OpenGL

[Alejandro Morán Jiménez bj0606
David Moncó Jiménez bj0604
Javier Herreros Tomé bj0573]

Mecánicas de juego

Pintar tableros

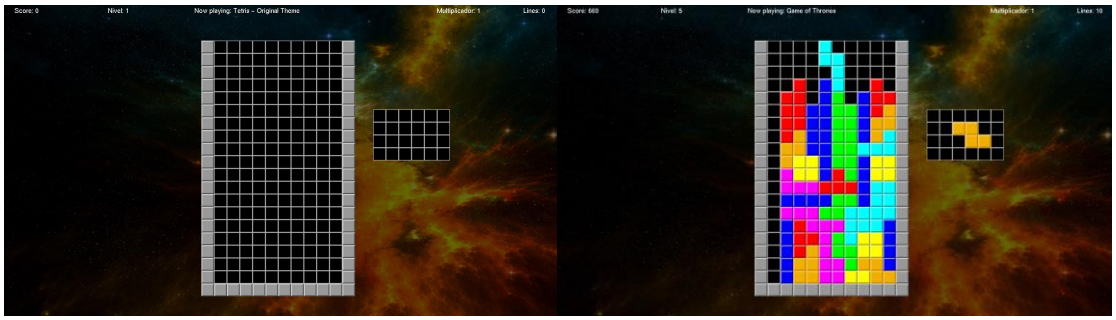
La mecánica del Tetris en principio es bastante sencilla, tenemos que llevar el control de una matriz que es donde se van a encontrar las fichas y para ello y gracias a la sugerencia del profesor decidimos realizar una matriz de enteros para los que cada número significa un color y una ficha en concreto.

Siguiendo esa idea pintar tableros en un juego Tetris es bastante sencillo, únicamente tenemos que recorrer la matriz entera y pintar todo aquello que sea necesario del color adecuado. Para ello contamos con el procedimiento *pintarCuadricula()* que se ejecutara tantas veces como se refresque la pantalla (cincuenta veces por lo general) o siempre que se ejecute el procedimiento *display()*.

```
void pintarCuadricula()
{
   int i,j;
   float sideblock,medio;
   int selectedTexture;

   sideblock = alto/24;
   medio = ancho/3;

   for(i=1;i<ALTO_T+1;i++)
   {
      for(j=1;j<ANCHO_T+3;j++)
      {
         switch (c_p1[i][j])
         {
         case 1:                 // azul
            glColor3ub(0,171,255);
            selectedTexture = 1;
            break;

        . . .

         default:                // blanco
             glColor4f(255, 255, 255, 0.0);
            selectedTexture = 8;
           break;
         }
      // Pintamos los cuadros

         glBindTexture(GL_TEXTURE_2D, blockTextures[selectedTexture]);

         glColor3ub(0, 0, 0);
         glEnable(GL_TEXTURE_2D);


            glBegin(GL_QUADS);
            glColor3ub(200, 200, 200);
            glColor3ub(255, 255, 255);
```

```
            //Abajo Izquierda
            glTexCoord2f(0.0, 0.0);
            glVertex2i(medio+(sideblock*j),20+(sideblock*i));
            //Abajo Derecha
            glTexCoord2f(1.0, 0.0);
            glVertex2i((medio+sideblock)+(sideblock*j),20+(sideblock*i));
            //Arriba Derecha
            glTexCoord2f(1.0, 1.0);
            glVertex2i((medio+sideblock)+(sideblock*j),(20+sideblock)+(sideblock*i));
            //Arriba Izquierda
            glTexCoord2f(0.0, 1.0);
            glVertex2i(medio+(sideblock*j),(20+sideblock)+(sideblock*i));
        glEnd();
        glDisable(GL_TEXTURE_2D);

    }
  }
```

El tamaño de los bloques esta adaptado para que se ajuste a cualquier pantalla, puesto que toma los valores de alto y ancho de la pantalla. De esta forma y puesto que un cuadrado tiene todos los lados iguales tomamos el lado mínimo y empezamos a pintar con ello. Esta función es exactamente igual para la matriz que muestra la siguiente pieza, obviamente adaptada a las dimensiones de la misma.



Generación de piezas

Aunque podríamos realizar una estructura o una clase ficha, por comodidad, las fichas son un array de tres enteros en los que guardamos primero la posición Y, luego la posición X y por ultimo hacia donde mira la ficha.

```
int centro_ficha[3]; // pieza del tablero
int next_piece[3]; // siguiente pieza
```

Una vez teniendo esto claro, para generar piezas de forma aleatoria hacemos uso del random no sin antes alimentarlo con una semilla puesto que de no ser así, solamente obtendríamos números pseudoaleatorios.

```
void generateNextPiece()
{
        srand (time(NULL));
        int pieza = rand()%8;

        if (nextpiece[next_piece[0]][next_piece[1]]!=0) pieza =
```

```
        nextpiece[next_piece[0]][next_piece[1]];
        . . .
}
```

Cada pieza tiene un número, por tanto en función del número que salga, pintamos una ficha u otra en la posición inicial del tablero o dela matriz de la siguiente pieza.

```
void generateNextPiece()
{
        . . .
        switch(pieza)
        {
              case 0:
              case 1:
              centro_ficha[0]=22;
              centro_ficha[1]=5;
              centro_ficha[2]=1;

              c_p1[centro_ficha[0]][centro_ficha[1]]=1;
              c_p1[centro_ficha[0]][centro_ficha[1]-1]=1;
              c_p1[centro_ficha[0]][centro_ficha[1]+1]=1;
              c_p1[centro_ficha[0]][centro_ficha[1]+2]=1;

              break;
        . . .

        // Para pintar la siguiente ficha

          pieza = rand()%8;

           switch(pieza)
           {
              case 0:
              case 1:
              next_piece[0]=2;
              next_piece[1]=2;
              next_piece[2]=3;

              nextpiece[next_piece[0]][next_piece[1]]=1;
              nextpiece[next_piece[0]][next_piece[1]-1]=1;
              nextpiece[next_piece[0]][next_piece[1]+1]=1;
              nextpiece[next_piece[0]][next_piece[1]+2]=1;

              break;
}
```

Rotaciones y choques

Lo primero de todo es identificar la ficha de manera inequívoca, esto lo hacemos a través de un array llamado centro_ficha, el cual tiene tres enteros, los dos primeros identifican una posición de la malla(centro_ficha[1] son las X y centro_ficha[0] son las Y), esa posición será el

centro de nuestra ficha, después vamos a esa posición de la malla y miramos que numero hay escrito en ella, con esto ya tenemos que ficha es y su centro ahora solo hace falta su posición, para esto usamos la última posición del array centro_ficha que tendrá un entero de valor 1 a 4 que determina en que giro esta la ficha, el valor de que numero corresponde a que ficha o cual es el centro de una ficha o su número de giro lo especificamos en una hoja antes y fue totalmente arbitrario lo único que se cumple es que una misma ficha no mueve centro por mucho que gire o baje así pues una vez designado el centro este será siempre la misma posición con relación a la ficha.
Ahora que sabemos que ficha es y su posición hay una función para girar, mover derecha, mover izquierda y bajada.

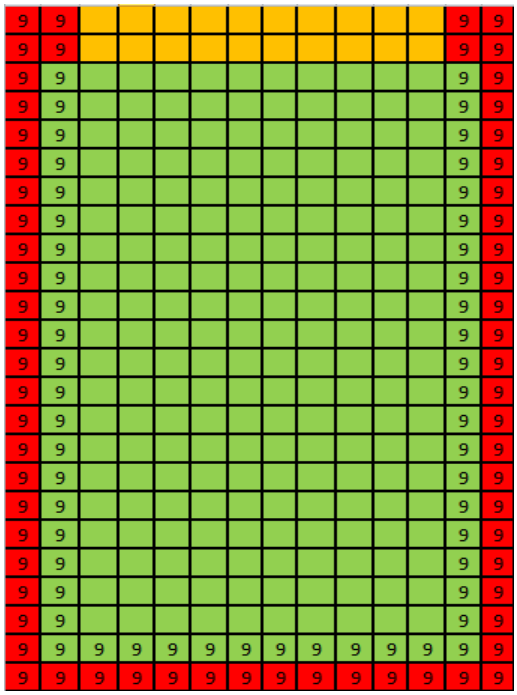
En cada una de ellas hay un switch para filtrar la pieza y dentro de cada case hasta cuatro if anidados que identifican en que giro esta y entonces se intenta hacer la operación que toque en tres fases, la primera comprobar si el movimiento a dónde quiere ir la pieza es posible (el array está a 0 en esas posiciones) si esto se cumple la segunda fase es borrar la pieza anterior y la tercera fase es redibujarla en el nuevo sitio, después la malla se dibuja de nuevo.

La función de bajada tiene una ligera excepción y es que en el momento en que no puede bajar (la primera fase no se cumple) llama a la función *generateNextPiece()* pues una pieza que no puede bajar más está colocada y hay que empezar a mover la siguiente, y también llama *limpiarmalla()* pues cuando se coloca una pieza puede ser que se hagan varias líneas y hay que eliminarlas del juego.

Si a bajar se le llama con un 0 bajara una posición y si se la llama con un 1 bajara hasta que se produzca un choque.

En bajar también se encuentra una comprobación por la cual si un ficha al chocar permanece en las dos últimas líneas se cambia el estado a GameOver. Esto es de esta manera dado que las piezas se generan dos cuadrados por encima de la zona visible de la malla así da la sensación de que la ficha cae y no se genera de la nada, así pues si en estas dos filas se queda alguna parte de una pieza el jugador a perdido.

El control para que una pieza no se salga por los lados o por abajo se ha implementado rellanando 2 filas de la izquierda, dos de la derecha y dos de abajo con un numero diferente a las piezas, en nuestro caso el nueve, esto hace que las funciones de comprobar vean algo distinto de 0 y por tanto no dejen atravesar estos límites a la pieza.



	Zona de generacionde piezas
	Zona no visible
	Zona visible
	Los 9 visible se rellenan con una textura metalizada
	que ayuda a ver las dimensiones del tetris

Limpieza de líneas

Una vez que se llama esta función se hace una comprobación para buscar si en las líneas hay algún hueco con 0, si no lo hay se pone toda esa línea a 0 teniendo cuidado de no poner a 0 los límites que tienen un 9 escrito y se llama a una función llamada recolocar que pasándole la Y donde se ha puesto todo a 0 a partir de ese punto baja todas las demás filas un nivel abajo, después de esto se prosigue la búsqueda hasta completar la malla pues se puede eliminar más de una línea en un movimiento, y se llamaría tantas veces a recolocar como líneas se borren.

Menús

El desplazamiento entre menús es bastante sencillo, partimos de un enumerado con los valores que queríamos de los menús, así es mucho más sencillo trabajar con ello puesto que no tienes que acordarte del valor sino que con usar la palabra asociada es más que suficiente.

```
int estado;
enum estados {Menu=0,P1=1,GameOver=2,P2=3, MusicScreen = 4, About = 5};
```

Como hemos comentado en varios apartados la función display se encarga de pintar lo que aparece en pantalla, la idea de los menús es sencilla, en función del menú que este seleccionado pintamos una cosa u otra, por eso, en nuestro procedimiento display, tenemos un switch que en función del valor de la variable estado, que es la que almacena el estado actual, se pinta por pantalla una cosa u otra.

```
void display ( void )
{
    . . .
   switch (estado)
   {
      case Menu:
        . . .
        . . .
         break;

      case P1:
        . . .
        . . .
         break;
      case GameOver:
        . . .
        . . .
         break;
      case MusicScreen:
        . . .
        . . .
         break;
      case About:
        . . .
        . . .
         break;
   }

}
```

Los valores del estado cambian según unos eventos u otros, por ejemplo, en el menú principal esto cambia en función de la tecla que pulsemos, pero en el caso de estar jugando y perder la partida, este evento que se produce cambia el estado pasando del player 1 a la pantalla de GameOver.

Además hay que destacar que tanto al inicio del juego, como cuando se cambia de un menú a otro, suelen utilizarse ciertas funciones de limpieza de la matriz de juego, de puntuaciones, niveles, etcétera, puesto que si terminamos una partida o nos salimos de la que estamos jugando deberíamos empezar de nuevo. Esas funciones son *initializeAll()* y *inicializarArray().*

```
void initializeAll()
{
   cont_bajada=0;
   nivel=1;
   multiplier=1;
   lines=0;
   score=0;
   currentVolume=20;
}

void inicializarArray()
{
   int i,j;

   for(i=0;i<24;i++)
   {
      for(j=0;j<14;j++)
      {
         if(i<2) c_p1[i][j]=9;
         else if(j<2 || j>11) c_p1[i][j]=9;
         else c_p1[i][j]=0;
      }
   }
}
```

Gestión de los controles

Para la gestión de controles hemos utilizado los siguientes procedimientos *keyboard(unsigned char key, int x, int y)* y *keyboardSpecial(int key, int x, int y).* En la función main hemos asignado a los eventos de teclado normal y especial dichas funciones que posteriormente codificaremos.

```
int main(int argc, char** argv)
{
        . . .
        glutKeyboardFunc(keyboard);
        glutSpecialFunc(keyboardSpecial);
        . . .
}
```

En nuestro caso concreto, teníamos un problema, puesto que dependiendo de la pantalla mostrada deberían funcionar unos botones u otros, incluso los mismos botones en diferentes

pantallas (o estados de la aplicación, como puede ser Menu, Player1, GameOver, etc), por eso mismo delegamos estas gestiones a procedimientos propios de cada estado de la aplicación, de la siguiente forma.

```
void keyboard ( unsigned char key, int x, int y )
{
   switch (estado)
   {
      case Menu:
         handleMenuKeyboard(key);
         break;
      case P1:
         handleGameKeyboard(key);
         break;
      case MusicScreen:
         handleMusicKeyboard(key);
         break;
      case GameOver:
         handleGameOverKeyboard(key);
         break;
      case About:
         handleAboutKeyboard(key);
         break;
   }

   glutPostRedisplay();
}
```

Cada uno de esos procedimientos se encarga de realizar unas funciones u otras, debido a que el switch en función del estado en el que se encontrase el juego iba a llamar a un *handler* o a otro. En el caso de las teclas especiales, puesto que solo las íbamos a utilizar para manejar el Tetris, decidimos no encapsularlo tanto y directamente realizarlo en el propio procedimiento asociado a dichas teclas, eso si, comprobando que solo se ejecutasen dichos métodos si nos encontrábamos en la pantalla de player1.

```
void keyboardSpecial (int key, int x, int y)
{
   if(estado = P1){
      switch (key)
      {
      case GLUT_KEY_RIGHT:
         desplazamientoDerecha();
         break;

      case GLUT_KEY_LEFT:
         desplazamientoIzquierda();
         break;

      case GLUT_KEY_UP:
         giro();
         break;
```

```
      case GLUT_KEY_DOWN:
         desplazamientoAbajo(0);
         break;

      case GLUT_KEY_F10:
         break;
      }
   }
   glutPostRedisplay();
}
```

Registro de mejores puntuaciones

Para el registro de mejores puntuaciones nos hemos basado en un fichero llamado *max.dat* , el cual almacena la mejor puntuación realizada hasta el momento. Para ello nos basamos en un procedimiento *void meterMAX(int MAX)* y una función *int sacarMAX().*La función para sacar el máximo es muy sencilla, simplemente abre el fichero y lee del mismo la puntuación máxima almacenada, una vez leída la retorna.

```
int sacarMAX()
{
   int sacado;
   FILE * pFile;
   pFile = fopen ("max.dat","r");
   fscanf(pFile,"%d",&sacado);
   fclose(pFile);
   return sacado;
}
```

El procedimiento para meter la máxima puntación es bastante sencillo también, primero sacamos la puntuación máxima y si la que hemos obtenido en ese momento es la mayor, actualiza el fichero con ese valor.

```
void meterMAX(int MAX)
{
   if(sacarMAX()<MAX)
   {
      FILE * pFile;
      pFile = fopen ("max.dat","w");
      fprintf(pFile,"%d",MAX);
      fclose(pFile);
   }
}
```

Por falta de tiempo, no implementamos en su momento las mejores diez puntuaciones, pero la mecánica es la misma. En vez de leer un único valor, deberíamos leer mientras tengamos valores en el fichero y esto introducirlo en un array. Posteriormente ya que la puntuación ya está ordenada podemos hacer una inserción ordenada, encontrando el hueco de la puntuación y desplazando los demás una posición hacia atrás. A la hora de mostrarlo sería igual, pero en vez de mostrar una, tendríamos que mostrar las diez mejores recorriendo el array.

Aunque lo óptimo sería trabajar con una estructura que contenga el valor de la puntuación obtenida y el nombre de quien lo consiguió, pero eso si necesitaría muchísima mas implementación, puesto que deberíamos realizar una nueva pantalla para ingresar el nombre o las tres primeras letras del mismo como se realiza en muchos arcades.

Puntuación

En todo momento tendremos la puntuación actual en la pantalla de juego e incluso la puntuación final y la máxima conseguida en la pantalla de GameOver.

A nuestro procedimiento *void incrementScore(int numberlines)* le llega el número de líneas que se han hecho en ese momento, una, dos, tres o como máximo cuatro debido al tretrimino recto. Para la puntuación hemos decidido dar un número de puntos según el nivel, el número de líneas que se hagan y un multiplicador, que lo único que realiza es que si en diez segundos nos hacemos otra línea, esto influya en nuestra puntuación, descendiendo con el tiempo el bonus de puntuación que se nos dará. Para la puntuación hemos utilizado el siguiente procedimiento

```
void incrementScore(int numberlines)
{
        score = score + (10*(numberlines*numberlines)* (multiplier/100 +1 )*nivel);
        if (multiplier>=1) multiplier=500;
}
```

El score aumentara en base a la fórmula que arriba se describe, es decir, diez por el número de líneas al cuadrado a su vez multiplicado por el multiplicador (le sumamos uno, puesto que como estamos trabajando con enteros, la división del multiplicador por un número menor de cien, dará como resultado un cero y por tanto no se sumaría ninguna puntuación) y por ultimo multiplicado por el nivel, para que a mas nivel mas puntuación.

Como hablábamos antes, este multiplicador ira bajando en función en función de los frames que se pinten, es decir, si lo queremos para que dure diez segundos, tendremos que darle un valor de quinientos, puesto que 500/50 = 10 segundos. Como podemos observar, en la pantalla para jugar y cincuenta veces por segundo, puesto que son las veces que se pinta por segundo, se va decrementando el multiplicador mientras sea mayor que uno.

```
void display ( void )
{
    . . .
   switch (estado)
   {
      . . .
      case P1:
                if (multiplier>1) multiplier--;
      break;
     . . .

}
```

Niveles

Para los niveles se ha realizado una formula bastante sencilla, tenemos una constante llamada *SALTO_NIVEL* la cual determina las líneas para poder pasar de nivel. Como comentamos en la presentación del juego en un principio lo pondríamos a tres para que se vea claramente como aumentaba la dificultad de forma progresiva. Normalmente no tendrá este paso y será de diez, para que se aumente de nivel de diez en diez líneas.

```
Int nivel;
. . .
void incrementLevel()
{
  if(lines>SALTO_NIVEL) nivel=(lines/SALTO_NIVEL);
}
. . .
```

Esta variable nivel será determinante para aumentar la caída. Nuestra función display se ejecutara cincuenta veces por segundo, así que en un principio para un nivel 1 hemos determinado que la pieza caiga en aproximadamente un segundo. Puesto que lo que le descontamos a las cincuenta veces por segundo es el valor del nivel multiplicado por cinco a más nivel, mas rápido cae. Para ello realizamos el siguiente código:

```
void display ( void )
{
. . .
                if(cont_bajada>=(50-nivel*5)) {
                        desplazamientoAbajo(0);
                        cont_bajada=0;
. . .
}
```

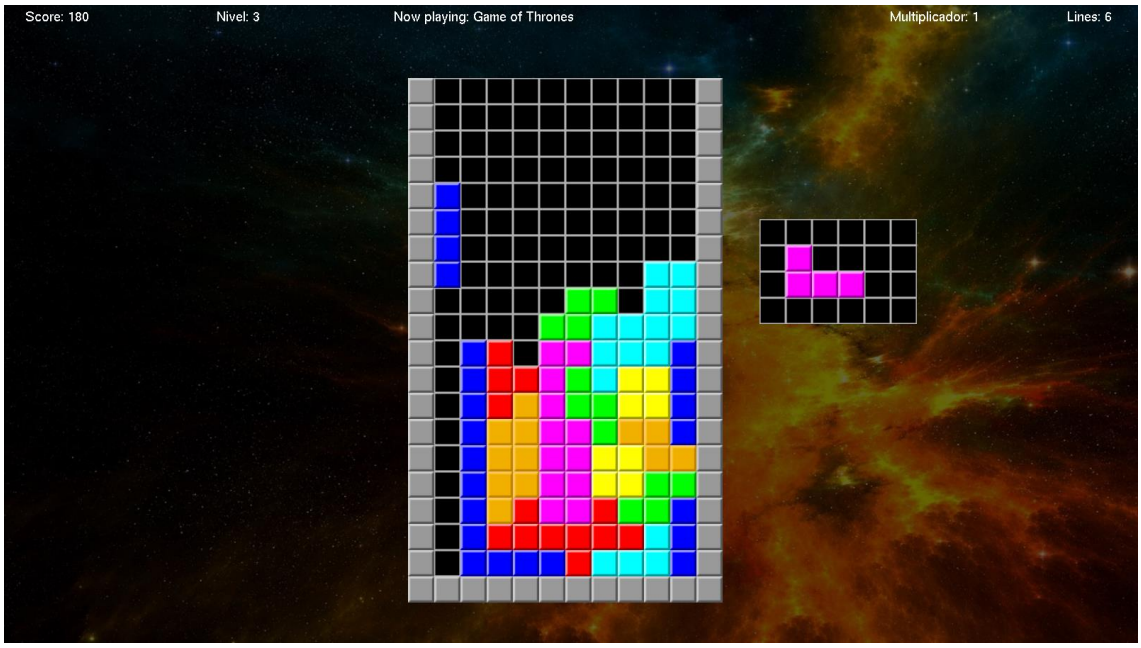
Sonido

Para el sonido hemos implementado algo parecido un JukeBox que nos permite seleccionar canciones tanto antes de empezar el juego, en el menú de selección de canción como dentro del mismo, usando para eso las teclas 1,2,3,4 para seleccionar canciones y M para parar las mismas en cualquier momento. En todo momento, si estamos en un menú o hemos terminado la partida la música se parara puesto que en nuestro caso solo reproducimos sonido cuando estamos jugando.

Aparte de incluir un sonido de fondo, hemos introducido un sonido cada vez que una pieza cae, al igual que cada vez que se realiza una línea. Para ello nos hemos apoyado en las siguientes variables.

```
Mix_Music *musica;
Mix_Chunk *collisionSound;
Mix_Chunk *clearRowSound;
Mix_Chunk *gameOverSound;

const char* selectedMusic[4];
char* musicTitles[4];
int selectedTrack = 0;
```

El array *musicTitles[4]* lo utilizaremos para mostrar el nombre de las canciones que se están reproduciendo mientras estamos jugando como se puede apreciar en la captura en la parte superior, *"Now playing: Game of Thrones".*



Nada mas iniciar el juego, en el procedimiento *init()*, cargamos los Chunk , las canciones en el array *selectedMusic[4]* y les completamos el título de la siguiente forma.

```
Void init(void)
{
        . . .
          selectedMusic[0] = "music/bso1.wav";
          selectedMusic[1] = "music/bso2.wav";
          selectedMusic[2] = "music/bso3.wav";
          selectedMusic[3] = "music/bso4.wav";

          collisionSound = Mix_LoadWAV("music/collisionSound.wav");
          clearRowSound = Mix_LoadWAV("music/clearRowSound.wav");
          gameOverSound = Mix_LoadWAV("music/gameOverSound.wav");

          musicTitles[0] = "Tetris - Original Theme";
          musicTitles[1] = "Start Wars BSO";
          musicTitles[2] = "Lord Of The Rings";
          musicTitles[3] = "Game of Thrones";
         . . .
}
```

La carga de los ficheros de sonido se hace en la función *sonido()* en ella inicializamos las opciones de sonido de la librería SDL, el buffer lo ponemos a 1024 ya que al ponerlo a 4096 se producía un retardo en su reproducción. En este momento se pone en funcionamiento la música de fondo del juego.

```
void sonido(void){
	SDL_Init(SDL_INIT_AUDIO);
	int frecuencia = 22050;
	int canales = 2;

	int buffer = 1024;
	Uint16 formato = AUDIO_S16;

	Mix_OpenAudio ( frecuencia, formato, canales, buffer );

	musica = Mix_LoadMUS(selectedMusic[selectedTrack]);


	if(musica != NULL){

		Mix_VolumeMusic(currentVolume);
		Mix_PlayMusic(musica,2);
	}
	else{
		cout << "ERROR reproduciendo cancion..." << endl;
	}
}
```

Como habíamos comentado anteriormente, cuando se producían ciertos eventos reproduciríamos unos Chunk en función de que evento sea, para ello, utilizamos el siguiente procedimiento que nos hemos creado, *playEffect(int effectType).* A este procedimiento le llega un entero en el cual se reflejara el sonido que se va a reproducir en ese momento, para ello, los procedimientos que se encargan de comprobar y validar las líneas, las colisiones, etc llamaran a este procedimiento con el entero adecuado.

```
void playEffect(int effectType)
{
  switch (effectType)
  {
    case 1:
      Mix_Volume(-1,5);
      Mix_PlayChannel(-1, collisionSound, 0);
      break;
    case 2:
      Mix_Volume(-1,50);
      Mix_PlayChannel(-1, clearRowSound, 0);
      break;
    case 3:
      Mix_Volume(-1,100);
      Mix_PlayChannel(-1, gameOverSound, 0);
      break;
  }
}
```

Debido a que los sonidos no estaban balanceados en tema de volumen y que el sonido de golpear pieza sonaba bastante mas alto, regulamos el volumen con el que se van a ejecutar dichos sonidos mediante *Mix_Volume().*

Por último, como habíamos comentado al principio, en todo momento podremos parar la música. Para ello hemos creado el procedimiento *stopSound()* que ejecuta un *Mix_HaltMusic()*.

```
void stopSound(void)
{
   Mix_HaltMusic();
}
```

Hemos valorado que los sonidos del juego, es decir, los Chunk sigan sonando y que para eso haya que bajar el volumen del dispositivo donde se este reproduciendo el juego.

Texturas

Inicialmente, hemos diseñado el juego sin texturas, puesto que primaba la funcionalidad a la calidad visual. Una vez realizado el juego pasamos a ponerle textura a todo aquello que estaba pintado de determinado color en el juego sin texturas. Las texturas son las siguientes

Para la carga de las texturas se utiliza la función *loadImage()* que recibe un nombre de fichero con la imagen que se quiere cargar, y que devuelve el id de la textura cargada. Para eso lo realizamos de la forma más larga, sin utilizar DevIL.

```
// Funcion que carga una imagen, la convierte en una textura, y devuelve el ID de la textura
GLuint loadImage(const char* nombreFichero)
{
	ILuint imagenID;
	GLuint texturaID;
	ILboolean exito;
	ILenum error;
	ilGenImages(1, &imagenID);
	ilBindImage(imagenID);
	cout << nombreFichero << endl;
	exito = ilLoadImage(nombreFichero);	// Cargamos la imagen


	if (exito)
	{
		// If the image is flipped (i.e. upside-down and mirrored, flip it the right way
up!)
		ILinfo ImagenInfo;
		iluGetImageInfo(&ImagenInfo);
		if (ImagenInfo.Origin == IL_ORIGIN_UPPER_LEFT)
```

```
		{
			iluFlipImage();
		}
		exito = ilConvertImage(IL_RGB, IL_UNSIGNED_BYTE);
		if (!exito)
		{
			error = ilGetError();
			cout << "Error en la conversion de la imagen: " << error << endl;
			exit(-1);
		}
		// Generamos una textura
		glGenTextures(1, &texturaID);
		// Asignamos la textura a un nombre
		glBindTexture(GL_TEXTURE_2D, texturaID);
		// Seleccionamos el metodo de clamping de la textura
		glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
		glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

		// Seleccionamos el metodo de interpolacion de la textura
		glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
		glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
		// Especificamos informacion de la textura
		glTexImage2D(GL_TEXTURE_2D,                                    // Tipo de
textura
                                        0,
                                        ilGetInteger(IL_IMAGE_BPP),
                                        ilGetInteger(IL_IMAGE_WIDTH),
                                        ilGetInteger(IL_IMAGE_HEIGHT),
                                        0,
                                        ilGetInteger(IL_IMAGE_FORMAT),
                                        GL_UNSIGNED_BYTE,          // Tipo de datos de la
imagen
                                        ilGetData());
	}
	else
	{
		error = ilGetError();
		cout << "Fallo al cargar la imagen: " << error << endl;
		exit(-1);
	}
	ilDeleteImages(1, &imagenID); // Borramos la imagen, porque ya hemos copiado los
datos en la textura.
	cout << "Textura creada correctamente." << endl;
	return texturaID;
}
```

La carga de texturas la realizaremos directamente en el procedimiento que pinta la matriz en pantalla, como se aprecia en el siguiente código, es igual tanto para la matriz de juego como para la matriz de siguiente pieza.

```
void pintarCuadricula()
{
   . . .
   for(i=1;i<ALTO_T+1;i++)
   {
      for(j=1;j<ANCHO_T+3;j++)
      {
         switch (c_p1[i][j])
         {
         case 1:                 // azul
            glColor3ub(0,171,255);
            selectedTexture = 1;
            break;

. . .

         default:                // blanco
             glColor4f(255, 255, 255, 0.0);
            selectedTexture = 8;
           break;
         }
      // Pintamos los cuadros

         glBindTexture(GL_TEXTURE_2D, blockTextures[selectedTexture]);

         glColor3ub(0, 0, 0);
         glEnable(GL_TEXTURE_2D);


            glBegin(GL_QUADS);
            glColor3ub(200, 200, 200);
            glColor3ub(255, 255, 255);
            //Abajo Izquierda
            glTexCoord2f(0.0, 0.0);
            glVertex2i(medio+(sideblock*j),20+(sideblock*i));
            //Abajo Derecha
            glTexCoord2f(1.0, 0.0);
            glVertex2i((medio+sideblock)+(sideblock*j),20+(sideblock*i));
            //Arriba Derecha
            glTexCoord2f(1.0, 1.0);
            glVertex2i((medio+sideblock)+(sideblock*j),(20+sideblock)+(sideblock*i));
            //Arriba Izquierda
            glTexCoord2f(0.0, 1.0);
            glVertex2i(medio+(sideblock*j),(20+sideblock)+(sideblock*i));
         glEnd();
         glDisable(GL_TEXTURE_2D);

      }
   }
```

Para aquellos casos en que la textura se carga en toda la pantalla se utiliza la función *dibujarFondo()*.

```
void dibujarFondo(GLuint textura){
	// Clear the screen
	//glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	// Select the texture to use
	glBindTexture(GL_TEXTURE_2D, textura);

	//glColor3ub(255, 255, 255);

	// Draw our texture
	glEnable(GL_TEXTURE_2D);
	glBegin(GL_QUADS);
		// Abajo Izquierda
		glTexCoord2f(0.0, 0.0);
		glVertex2i(0, 0);

		// Arriba Izquierda
		glTexCoord2f(0.0, 1.0);
		glVertex2i(0, alto);

		// Arriba Derecha
		glTexCoord2f(1.0, 1.0);
		glVertex2i(ancho, alto);

		// Abajo Derecha
		glTexCoord2f(1.0, 0.0);
		glVertex2i(ancho, 0);
	glEnd();
	glDisable(GL_TEXTURE_2D);
}
```

Conclusiones

Desde nuestro punto de vista ha sido una de las prácticas más amenas realizadas en la universidad y creemos que no estaría mal que se le dedicase algo más de tiempo para poder profundizar en openGL o incluso realizar un breve vistazo a motores gráficos actuales para desarrollo de juegos. En general, creo que este juego, aparte del esfuerzo realizado y sin tener que ser personas aficionadas a estos, nos ha resultado muy interesante de realizar a todos.

También me gustaría resaltar, que lo mismo este año debido a que la semana santa ha caído tan tarde la falta de tiempo para realizar el mismo. Muchas de las cosas que se nos quedaron en el tintero son las siguientes:

- Multijugador local: fácilmente implementable una vez tienes el modo un jugador completamente pulido, controlando todo tipo de errores y casuísticas, simplemente necesitamos pintar otro tablero y otra ficha siguiente, pudiendo aprovechar los métodos ya creados para un solo jugador, en el momento de que uno pierda, pintar la pantalla de GameOver y con una simple condición comprobar quien ha sido el perdedor y ganador.

- Puntuación más alta: aparte de realizarlo de forma local, la posibilidad de subir la puntuación a una base de datos online, con el fin de poder comprobar rankings no solamente en tu máquina, sino en todas las que tengan el juego y dispongan de conexión a internet.

- Powerups: con los cuales recibir más puntuación, eliminar líneas, eliminar piezas aleatorias de la pantalla, etcétera. Esto además daría mucho más juego en el juego multijugador pudiendo estorbar al contrario, mandándole líneas, impidiéndole la visión (algo como un efecto niebla) y un largo etcétera.

- Multijugador en red: utilizando los sockets en C/C++, puesto que lo que tiene que viajar por el socket es lo mismo para ambos jugadores, cada uno tiene su matriz y lo que llega a la matriz contraria es justamente lo que tiene el otro en su matriz, esta información seria relativamente sencilla de enviar mediante estos.

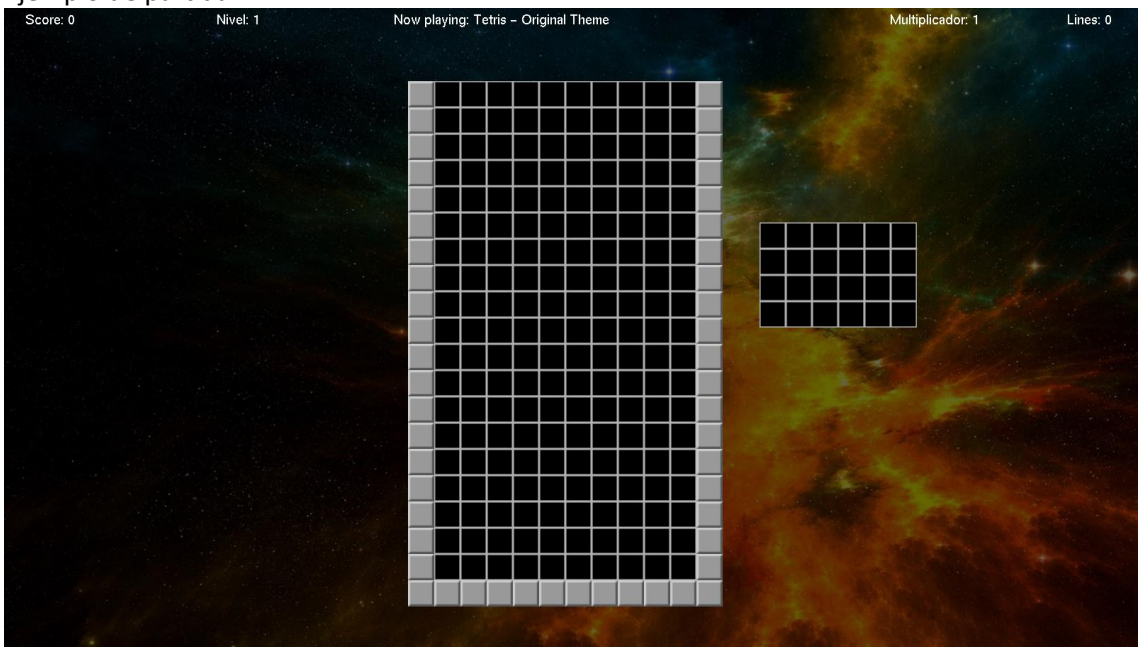
Algunas capturas del juego

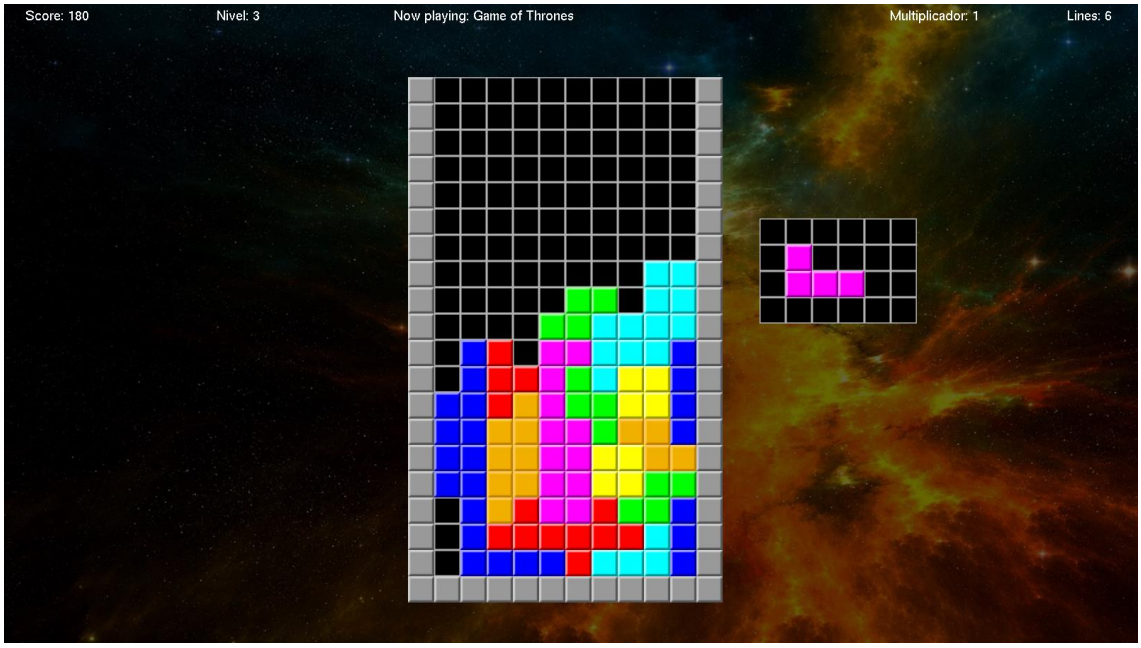
Menú principal

Menú selección de música

Menú About

Ejemplo de partida





GameOver