

## PEC3 - Efectos Visuales y sonoros

### Implementación del sistema de juego

Para la realización de la práctica se ha generado un mapa de 10x10 casillas siguiendo el ejemplo de la documentación. En dicho mapa, se dispone de 5 tipos de casillas distintas: Empty, Player, Monster, Wall y Exit. Cada una de estas forma parte de un enum llamado Grid que sirve para montar el mapa. Así pues, con tal de crear el tablero de juego, se genera un mapa de tamaño 10x10 y se llenan los elementos del array con el valor correspondiente del grid.

```
Grid map[SIZE][SIZE] = { // 10x10 matrix
    { Empty, Empty, Empty, WALL, Empty, Empty, Empty, Empty, Empty, Empty },
    { Empty, Empty, Empty, WALL, Empty, Empty, Empty, Empty, Empty, Empty },
    { Empty, WALL, Empty, WALL, Empty, Empty, Empty, Empty, Empty, Empty },
    { Empty, Empty, Empty, Empty, Empty, Empty, Empty, Empty, WALL, Empty },
    { Empty, Empty, Empty, Empty, Empty, Empty, Empty, Empty, WALL, Empty },
    { Empty, Empty, Empty, Empty, Empty, Empty, Empty, Empty, WALL, Empty },
    { Empty, Empty, Empty, Empty, Empty, Empty, Empty, Empty, WALL, Empty },
    { Empty, Empty, Empty, Empty, Empty, Empty, Empty, Empty, WALL, Empty },
    { Empty, Empty, WALL, Empty, Empty, Empty, Empty, Empty, WALL, Empty },
    { Empty, Empty, WALL, Empty, Empty, Empty, Empty, Empty, WALL, Empty },
    { Empty, Empty, WALL, Empty, Empty, Empty, Empty, Empty, WALL, Empty },
};
```

Este grid nos muestra los estados de las casillas sin tener en cuenta la posición del jugador, el monstruo o la salida de la cueva. Para tener constancia de ello, nada más empezar la partida, se inicializan las posiciones y se muestra el mapa por consola mediante las funciones InitPositions() y PrintMap():

- **Init Positions:** Calcula la distancia máxima que puede haber entre una punta y otra del mapa, sabiendo que tenemos un ancho y un alto de 10 casillas. Para ello se calcula la hipotenusa del triángulo que forman las esquinas y se asigna a la variable MaxSoundDistance.

Por otro lado, inicializa las posiciones en x y en y de los elementos que hay en la partida. En nuestro caso, se asigna al jugador la posición (1, 8), al monstruo la (4, 3) y a la salida la (9, 9).

```

void InitPositions()
{
    MaxSoundDistance = abs(sqrt(pow(SIZE - 1, 2) + pow(SIZE - 1, 2)));

    // Init player object
    playerObj.pos.x = 1;
    playerObj.pos.y = 8;
    playerObj.dir = DIR_UP;
    map[playerObj.pos.y][playerObj.pos.x] = PLAYER;

    // Init monster object
    monsterObj.pos.x = 4;
    monsterObj.pos.y = 3;
    monsterObj.dir = DIR_DOWN;
    map[monsterObj.pos.y][monsterObj.pos.x] = MONSTER;

    // Init exit object
    exitObj.pos.x = 9;
    exitObj.pos.y = 9;
    exitObj.dir = DIR_DOWN;
    map[exitObj.pos.y][exitObj.pos.x] = EXIT;

    updatePositions();
}

```

Para terminar, nos aseguramos de llamar a `updatePositions` para configurar el audio en el estado inicial. Dentro de esta función, calculamos el ángulo que forman tanto el monstruo como la salida con el jugador; así como la distancia a la que estos se encuentran. Para ello, extraemos un ángulo a partir de los delta en x e y para posteriormente modificarlo en función de la dirección en la que está mirando el jugador. Además, nos aseguramos de que dicho ángulo siempre esté comprendido entre 0 y 360. En cuanto a la distancia, calculamos de nuevo la hipotenusa del triángulo que forman los dos objetos y le pasamos el ángulo. Una vez sabemos la hipotenusa, en función del ángulo recibido, se incrementa la distancia al objetivo en caso de que esté en la espalda del jugador.

Con los cálculos de ángulo y distancia realizados, podemos llamar al método `SetPosition()` de la clase `SoundEffect`, que se encarga de llamar al método `Mix_SetPosition` de la librería `SDL_Mixer`. Esto nos permite, en función del ángulo y la distancia, determinar dónde colocar el sonido en el espacio.

```

void updatePositions()
{
    MonsterAngle = GetSoundAngleFromPositions(playerObj, monsterObj);
    MonsterDistance = GetSoundDistanceFromPositions(playerObj, monsterObj, MonsterAngle);
    WaterfallAngle = GetSoundAngleFromPositions(playerObj, exitObj);
    WaterfallDistance = GetSoundDistanceFromPositions(playerObj, exitObj, WaterfallAngle);
    monster.SetPosition(MonsterAngle, MonsterDistance);
    waterfall.SetPosition(WaterfallAngle, WaterfallDistance);
}

void SoundEffect::SetPosition(Sint16 angle, Uint8 distance) const
{
{
    Mix_SetPosition(channel, angle, distance);
}

```

- **Print Map:** Tanto al inicializar valores como cada vez que se mueve el jugador, se muestra por consola de comandos una actualización del estado del mapa. Esto nos permite hacer

seguimiento del estado en que se encuentra la partida y obtener información adicional sobre la distancia a la que se encuentran los objetos unos de otros.

```
0 0 0 3 0 0 0 0 0 0  
0 0 0 3 0 0 0 0 0 0  
0 3 0 3 0 0 0 0 0 0  
0 0 0 0 2 0 0 0 3 0  
0 0 0 0 0 0 0 0 3 0  
0 0 0 0 0 0 0 0 3 0  
0 0 0 0 0 0 0 0 3 0  
0 0 3 0 0 0 0 0 3 0  
0 1 3 0 0 0 0 0 3 0  
0 0 3 0 0 0 0 0 3 4  
  
[Monster] SDL angle 30 SDL distance 116  
[Waterfall] SDL angle 97 SDL distance 161  
[Player] direction LOOKING UP
```

Sólo queda una cosa más por ver antes de empezar con el loop principal del juego. La carga y reproducción de sonidos:

- **LoadSounds:** Encargado de llamar al método init de la clase SoundEffect. Sencillamente guarda un nuevo chunk perteneciente al archivo de audio que recibe por parámetro.

```
void LoadSounds()  
{  
    death.Init("PEC3_Sounds/Monster_Eating.wav");  
    gameOver.Init("gameOver.wav");  
    wall.Init("PEC3_Sounds/Wall_Impact.wav");  
    wall.SetVolume(100);  
    step.Init("PEC3_Sounds/Step.wav");  
    step.SetVolume(100);  
    victory.Init("victory.wav");  
    monster.Init("PEC3_Sounds/Monster_Snoring.wav");  
    monsterStep.Init("PEC3_Sounds/Monster_Step.wav");  
    waterfall.Init("PEC3_Sounds/Waterfall.wav");  
}
```

```
void SoundEffect::Init(const char* filename)
{
    chunk = Mix_LoadWAV(filename);
    if(chunk == NULL)
    {
        std::cerr << "*** No se encuentra: " << filename << std::endl;
    }
}
```

- **Play:** Únicamente para los sonidos que deben reproducirse en loop durante toda la partida, se llama la función PlayLoop() de la clase SoundEffect. En nuestro caso, sólo se usa para el ronquido del monstruo y la cascada. Este método sencillamente se encarga de reproducir en un canal distinto cada uno de los sonidos.

```
int SoundEffect::PlayLoop()
{
    channel = Mix_PlayChannel(-1, chunk, -1);
    return channel;
}
```

Tal y como se puede observar, llamamos a Mix\_PlayChannel de SDL\_Mixer y le pasamos como primer y tercer parámetro un -1. Según la documentación, asignar un -1 al primer parámetro permite usar el primer canal que se encuentre disponible para una asignación automática. En cuanto al tercer parámetro, al recibir el -1 asegura que el audio sonará en loop de forma constante.

En cuanto al loop principal del juego, estamos constantemente alerta del input que se puede recibir por parte del usuario para actualizar el estado actual. Mientras que no se pulse la tecla ESC, comprobaremos si alguna de las flechas del teclado se ha pulsado para determinar la acción a realizar. El mapeado de teclas es el siguiente:

- **Up arrow:** La única tecla encargada de cambiar la posición del jugador. Segundo la dirección en la que estemos mirando, se moverá a una casilla u otra. Si miramos hacia arriba, se moverá hacia arriba; si miramos hacia la derecha, se moverá hacia la derecha; y así sucesivamente.

```
if (key == SDL_SCANCODE_UP)
{
    if (playerObj.dir == DIR_UP) {
        obj->pos.y -= 1;
    }
    else if (playerObj.dir == DIR_RIGHT) {
        obj->pos.x += 1;
    }
    else if (playerObj.dir == DIR_LEFT) {
        obj->pos.x -= 1;
    }
    else {
        obj->pos.y += 1;
    }
}
```

- **Left/Right Arrow:** Encargadas de cambiar la dirección a la que mira el jugador. Para ello se llama a la función Turn(), que recibe por parámetro un booleano que determina si giramos a un lado o al otro.

```
else if (key == SDL_SCANCODE_LEFT)
{
    Turn(false);
}
else if (key == SDL_SCANCODE_RIGHT)
{
    Turn(true);
}
```

El método turn es bastante sencillo, comprueba el booleano y actualiza la dirección del jugador en función del lado al que estamos girando. Las posibles direcciones son DIR\_UP, DIR\_RIGHT, DIR\_LEFT y DIR\_DOWN. Sabiendo esto, nos queda el siguiente roadmap:

```

void Turn(bool turnRight) {
    if (turnRight) {
        if (playerObj.dir == DIR_UP) {
            playerObj.dir = DIR_RIGHT;
        }
        else if (playerObj.dir == DIR_RIGHT) {
            playerObj.dir = DIR_DOWN;
        }
        else if (playerObj.dir == DIR_DOWN) {
            playerObj.dir = DIR_LEFT;
        }
        else {
            playerObj.dir = DIR_UP;
        }
    }
    else {
        if (playerObj.dir == DIR_UP) {
            playerObj.dir = DIR_LEFT;
        }
        else if (playerObj.dir == DIR_RIGHT) {
            playerObj.dir = DIR_UP;
        }
        else if (playerObj.dir == DIR_DOWN) {
            playerObj.dir = DIR_RIGHT;
        }
        else {
            playerObj.dir = DIR_DOWN;
        }
    }
    step.Play();
    updatePositions();
}

```

A cada paso que da nuestro jugador, se llama al método Play() de SoundEffect para que lance el sonido del Step correspondiente. En este caso, el tercer parámetro será 0, puesto que sólo queremos que se reproduzca una vez.

```

int SoundEffect::Play()
{
    channel = Mix_PlayChannel(-1, chunk, 0);
    return channel;
}

```

Antes de actualizar la posición del jugador, debemos comprobar a qué casilla se ha desplazado para determinar el output de nuestro juego. Hay varias posibilidades:

- **Casilla vacía:** Simplemente actualiza la posición del jugador marcando la casilla anterior como Empty y la nueva como Player, reproduce el sonido de un paso dado y recalcula el ángulo y la distancia a la que se encuentra el monstruo y la salida (representada como una cascada).

```

case Empty:
    UpdatePlayerPosition(newObj);
    step.Play();
    monster.SetPosition(MonsterAngle, MonsterDistance);
    waterfall.SetPosition(WaterfallAngle, WaterfallDistance);
    break;

```

- **Muro:** En el caso de chocar contra un muro, reproducimos el sonido de choque y actualizamos la posición del monstruo (nunca la del jugador, puesto que no podemos atravesar muros). Para determinar a qué casilla debe moverse el monstruo, lo primero que debemos saber es si la casilla a la que nos estamos moviendo es una casilla válida (no es un muro). Así pues, se aleatoriza un número del 0 al 7 y para cada caso nos movemos en una dirección. Si la nueva posición a la que se ha movido es un tile vacío, simplemente se desplaza hacia ella. Si la posición corresponde con el tile en que se encuentra el jugador, llamamos al método **SetGameOver()**\*. Por el contrario, si damos con una casilla no válida, se vuelve a repetir el proceso hasta que consideremos que la casilla es buena.

```

case WALL:
    wall.Play();
    ChangeMonsterPosition();
    monsterStep.PlayAtPosition(MonsterAngle, MonsterDistance);
    monster.SetPosition(MonsterAngle, MonsterDistance);
    break;

```

\* El método **SetGameOver()** se encarga de parar todos los sonidos, y reproducir el sonido de muerte y el de game over o el de victoria según el resultado de la partida. Por último, asigna el booleano **ExitGame** a true para que se cierre la ventana y se termine la partida.

```

void SetGameOver(bool isDead)
{
    monster.Stop();
    waterfall.Stop();
    if (isDead) {
        death.PlayAndWait();
        gameOver.PlayAndWait();
    }
    else {
        victory.PlayAndWait();
        GameResult = true;
    }
    ExitGame = true;
}

```

*El método Stop de SoundEffect se encarga de llamar a Mix\_HaltChannel de SDL\_Mixer. Esto sencillamente para la reproducción de audio del canal especificado. Por otro lado, los PlayAndWait reproducen un canal y esperan a que haya terminado la reproducción completa del sonido, sin ser interrumpido por nada.*

```
void SoundEffect::PlayAndWait()
{
    channel = Play();
    while (Mix_Playing(channel) != 0) {
        SDL_Delay(200);
    }
}

void SoundEffect::Stop() const
{
    Mix_HaltChannel(channel);
}
```

Por último, una vez determinada la nueva casilla del monstruo, reproducimos en la nueva posición el sonido de paso del monstruo mediante **PlayAtPosition()**\* y llamamos a SetPosition para recalcular la distancia y el ángulo al que se encuentra del jugador.

*\*PlayAtPosition se encarga de llamar al método Play para reproducir el sonido del canal especificado a una cierta distancia y destacando el ángulo entre los objetos.*

```
void SoundEffect::PlayAtPosition(Sint16 angle, Uint8 distance)
{
    Mix_SetPosition(Play(), angle, distance);
}

int SoundEffect::Play()
{
    channel = Mix_PlayChannel(-1, chunk, 0);
    return channel;
}
```

- **Monstruo:** En caso de que la casilla a la que nos hemos movido sea la del monstruo, se llama al método SetGameOver(), cuyo comportamiento ya se ha visto en el punto anterior.
- **Salida:** Si nos movemos a la casilla de salida, quiere decir que se ha completado el juego con éxito y, por lo tanto, se paran todos los sonidos y se reproduce la música de victoria. Para eso, simplemente llamamos a la función SetGameOver con el parámetro de entrada *isDead = false*.

```
case MONSTER:
    SetGameOver(true);
    break;
case EXIT:
    SetGameOver(false);
    break;
```

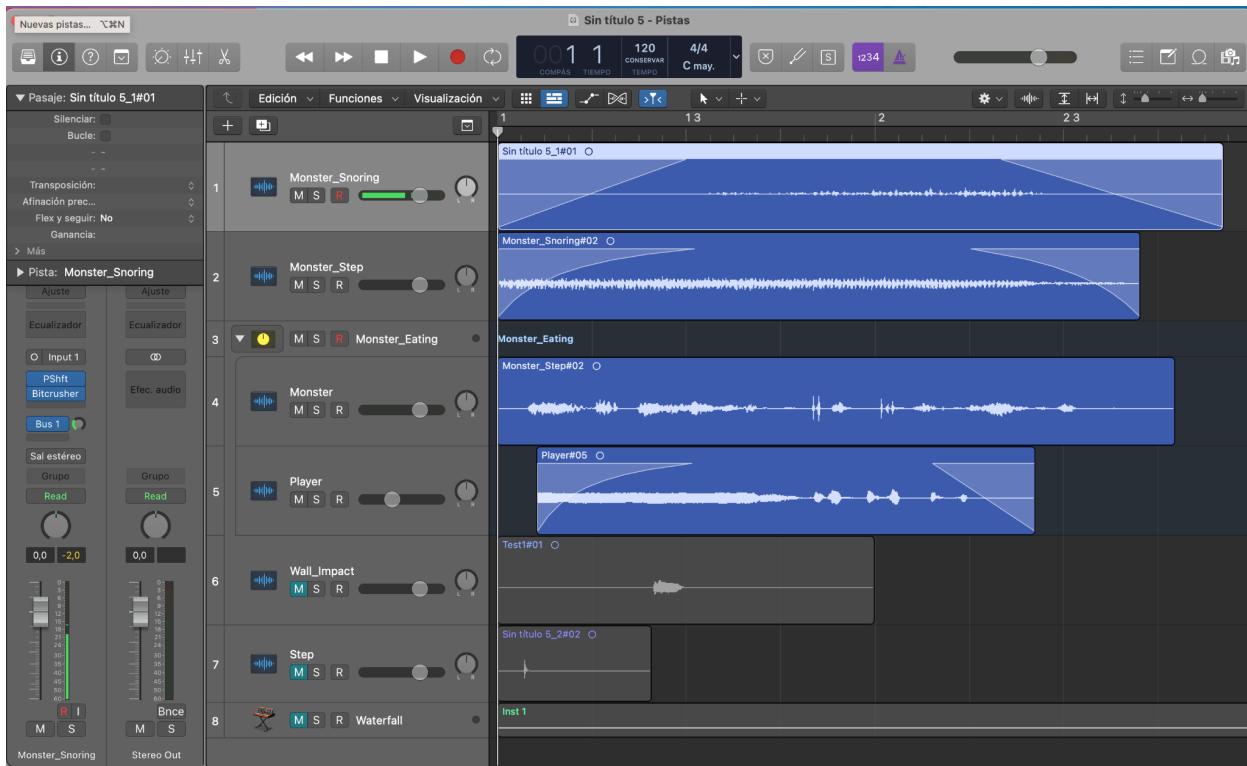
## Creación de sonidos

Se han generado sonidos propios para todos los casos, excepto la música de derrota y de victoria. Esto se ha realizado mediante el uso de Logic Pro X. Como buen músico, dispongo de equipo suficiente en casa como para grabar mis propios sonidos y conocimientos de edición y mezcla para efectuar un postprocesado suficientemente creíble.

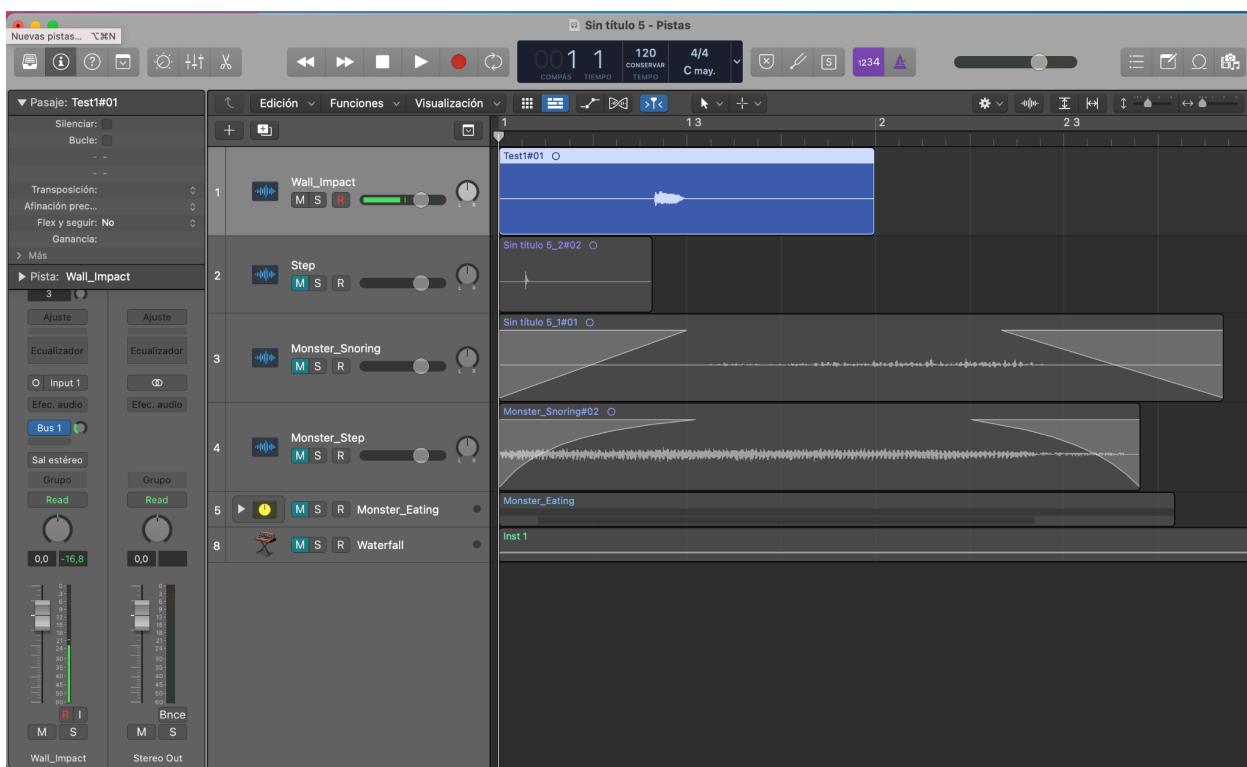
Así pues, para los sonidos relacionados con el monstruo, me he grabado a mi mismo realizando los sonidos y se le ha aplicado un pitch shift al 75% de mix y con 6 semitonos menos para simular el efecto

gutural del monstruo. Para romper todavía más el sonido, se le ha añadido un bitcrusher con un porcentaje de envío menor para romper lo suficiente la señal y dar la sensación de monstruosidad. Además, para el sonido relacionado con la muerte del personaje, se ha creado una pista suplementaria con quejidos humanos (también grabados por mí) que complementan los gruñidos del monstruo comiendo.





Para los pasos humanos se han chocado dos chanclas que generan ese feel de respuesta impulsional que tiene un solo paso. En cuanto al choque contra la pared, se ha grabado un quejido junto con un pequeño golpe de puño contra el pecho (a lo King Kong, si se me permite la expresión).



Por último, para la cascada se ha utilizado un generador de ruido blanco, junto a una ecualización para deshacernos de las altas y bajas frecuencias molestas y dar la sensación de caída del agua.



A todos los efectos se les ha aplicado la misma reverberación para obtener la sensación de estar en un mismo espacio. Se trata de una reverberación de tipo hall con un tamaño de sala medio-grande y un pequeño ajuste en el filtro pasa altos y el pasa bajos. El único efecto con una reverberación distinta son los footsteps, que simulan una sala un poco más pequeña para que el sonido no sea tan disperso y lo escuchemos "más cerca".



En la carpeta PEC3\_Sounds incluida en la entrega hay una versión sin procesar de todos los sonidos y una versión con las cadenas de efectos correspondientes aplicadas.